An Integrated Vector-Scalar Design on an In-order ARM Core

MILAN STANIC, OSCAR PALOMAR, TIMOTHY HAYES, IVAN RATKOVIC, ADRIAN CRISTAL, OSMAN UNSAL, and MATEO VALERO, Barcelona Supercomputing Center

In the low-end mobile processor market, power, energy and area budgets are significantly lower than in the server/desktop/laptop/high-end mobile markets. It has been shown that vector processors are a highly energy-efficient way to increase performance; however adding support for them incurs area and power overheads that would not be acceptable for low-end mobile processors. In this work, we propose an integrated vector-scalar design for the ARM architecture that mostly reuses scalar hardware to support the execution of vector instructions. The key element of the design is our proposed block-based model of execution that groups vector computational instructions together to execute them in a coordinated manner. We implemented a classic vector unit and compare its results against our integrated design. Our integrated design improves the performance (more than 6x) and energy consumption (up to 5x) of a scalar in-order core with negligible area overhead (only 4.7% when using a vector register with 32 elements). In contrast, the area overhead of the classic vector unit can be significant (around 44%) if a dedicated vector floating-point unit is incorporated. Our block-based vector execution outperforms the classic vector unit for all kernels with floating-point data and also consumes less energy. We also complement the integrated design with three energy-performance efficient techniques that further reduce power and increase performance. The first proposal covers the design and implementation of chaining logic that is optimized to work with the cache hierarchy through vector memory instructions, the second proposal reduces number of reads/writes from/to the vector register file while the third idea optimizes complex memory access patterns with the memory shape instruction and unified indexed vector load.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**;

Additional Key Words and Phrases: vector processors, low-power, energy efficiency, mobile processors

ACM Reference format:
Milan Stanic, Oscar Palomar, Timothy Hayes, Ivan Ratkovic, Adrian Cristal, Osman Unsal, and Mateo Valero. 0. An Integrated Vector-Scalar Design on an In-order ARM Core. *ACM Transactions on Architecture and Code Optimization* 0, 0, Article 0 (0), 25 pages.
DOI: 0000001.0000001

The research leading to these results has received funding from the RoMoL ERC Advanced Grant GA no 321253 and is supported in part by the European Union (FEDER funds) under contract TIN2015-65316-P. This research has been also supported the Agency for Management of University and Research Grants (AGAUR - FI-DGR 2014). O. Palomar is funded by a Royal Society Newton International Fellowship.
Current author's affiliations: M. Stanic, ASML; O. Palomar, University of Manchester; T. Hayes, ARM. A. Cristal is also affiliated with CSIC-IIIA and UPC.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0 Copyright held by the owner/author(s). Publication rights licensed to ACM. XXXX-XXXX/0/0-ART0 $15.00
DOI: 0000001.0000001

1 INTRODUCTION

In the last 15 years, power dissipation and energy consumption have become crucial design concerns for almost all computer systems due to several reasons: for example, technology feature size scaling leads to higher power density and therefore to complex and costly cooling. While power dissipation is critical for high-performance systems such as data centers due to large power usage, for mobile systems battery life is the primary concern.

Driven with this goal, researchers have focused on improving performance in an energy-efficient way. Vector processors (Asanovic 1998) are energy efficient architectures that yield high performance whenever there is enough data-level parallelism (DLP) (Lee et al. 2011). Besides the long and successful history of vector processors in supercomputers, vector units have been proposed in microprocessor design (Kozyrakis and Patterson 2003), (Espasa et al. 2002), (Batten 2010). Recent research on vector processors shows that they can be a good match even for applications from domains such as column-store databases (Hayes et al. 2012). Knights Landing (Sodani 2015) is a recent, second generation of the Xeon Phi processor. It is massively parallel x86 microprocessor designed by Intel and based on the Larrabee (Seiler et al. 2009) GPU that contains a 512-bit SIMD vector processing unit in each core. Also, SIMD multimedia extensions (Thakkar and Huff 1999), (Firasta et al. 2008) are often included in modern microprocessors. While vector processors and SIMD extensions both exploit DLP, they differ in the way the data operand elements are handled at the execution stage. While SIMD extensions process all elements at once, vector processors execute elements in a pipelined fashion. Although vector processors are energy efficient, they still have power and area overheads that are too high for mobile processors. This is mostly due to the strict power and area budget of such systems.

This paper contributes a method to increase the performance of the low-power low-end embedded systems in an energy-efficient way. The energy efficiency is attained by modifying a scalar core to execute vector instructions on the existing infrastructure. In particular, we propose an integrated vector-scalar design that combines scalar and vector processing mostly using existing resources of an energy-efficient scalar processor (in our evaluation environment it is based on the ARM Cortex A7). In addition to a design that uses a conventional vector execution model, we also contribute a novel block-based model of execution for vector computational instructions. We present performance, power, area and energy evaluation results of this integrated design. The results show that all vector designs significantly reduce energy over the scalar baseline for most of the considered kernels with a small area overhead. We report up to 5x energy reduction for our block-based execution model over the scalar baseline. Additionally, we found that the block-based execution model provides better results (up to 26% of energy saving) than a conventional vector unit with dedicated units. Regarding performance gains, we report more than a 6x speed-up compared to the scalar baseline. Moreover, our block-based execution model is up to 1.4x faster than the conventional vector unit for floating-point (FP) kernels.

Additionally, we also propose three techniques for an advanced integrated design that improve energy and/or performance: (1) chaining from the memory hierarchy, (2) direct result forwarding and (3) memory shape instructions and unified indexed vector load. We propose and implement two novel techniques that chain from the cache with the goal of further improving the performance of our integrated design. They can be applied to a conventional vector unit as well. We design and implement a novel result forwarding mechanism which complements the block-based execution and does not require writing to the vector register file. We design a vector memory unit with support for complex memory instructions including memory shape and scatter/gather instructions. Our results reveal that additional speed-up (up to 20%) is achieved with our chaining techniques over the integrated design without chaining. Direct forwarding reduces energy/power consumption

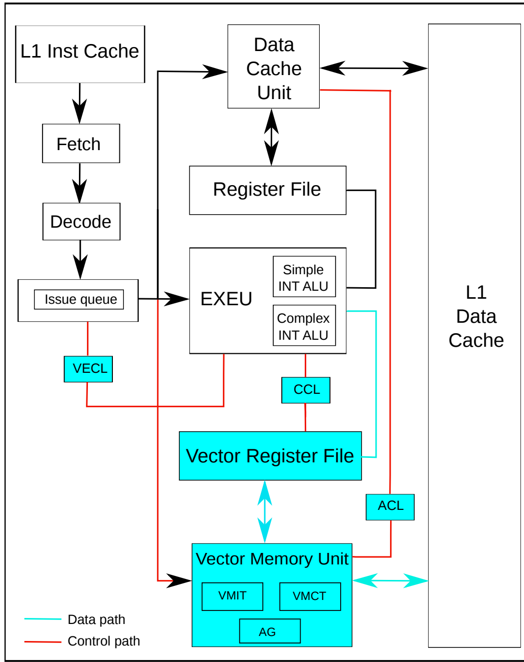

Fig. 1. Block diagram of the integrated design. White boxes are present in the original scalar design, blue boxes indicate the new hardware added in the integrated design.

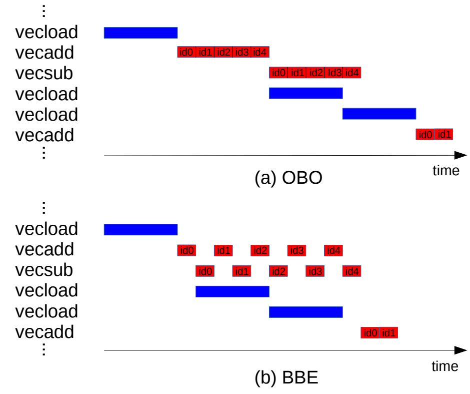

Fig. 2. An example of vector code executed with one ALU assuming the OBO (a) and the BBE (b) model.

of the vector register file by more than 50% for three evaluated kernels while the vector memory shape instruction increases speed-up of a vectorized kernel from 1.77x to 2.66x for the block-based model of execution.

Section 2 describes our integrated design. Section 3 outlines the experimental methodology and evaluates our integrated design regarding performance, area, power and energy. In Section 4 we propose techniques that further improve the integrated design and we evaluate them in Section 5. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 INTEGRATED DESIGN

As a baseline, we use a scalar core based on the highly energy-efficient ARM Cortex-A7. It is an in-order, dual-issue processor that implements the ARM v7 architecture with an 8-stage pipeline (non-highlighted gray blocks in Figure 1).

In our proposed integrated vector-scalar design, we attempt to maximize the reuse of resources already present in the baseline scalar core (white blocks in Figure 1) while adding support for vector instructions. While the front-end of the pipeline is the same (fetch and decode stages, with the obvious extension to the decode logic to support the new instructions), in the back-end we added two structures to support the execution of vector instructions on the scalar core: a vector register file, and a vector memory unit (blue blocks in Figure 1). There is also additional logic that controls the execution of vector instructions. Vector execution control logic (VECL) is added in the issue stage to support the execution of computational vector instructions. Aliasing control logic (ACL) exchanges information between the vector memory and the data cache unit and forces scalar and vector memory instructions to be executed in-order. We implement support for chaining of

computational instructions (Russell 1978), a well-known concept in vector processors. Similar to result forwarding in scalar processors, chaining allows starting the execution of a dependent vector instruction as soon as the first element of the vector is generated by the previous computational instruction. Chaining control logic (CCL) is responsible for the execution of chained dependent computational instructions.

2.1 Execution of Vector Computational Instructions

For executing the vector computational instructions on the existing scalar functional units (FUs), we study two alternatives: 1) the One-By-One model of execution (*OBO*), in essence the classic vector execution model, in which a vector instruction is executed to completion once it starts execution in a functional unit, i.e. for all the operations of the vector; and 2) a novel execution model called Block-Based Execution (*BBE*). In this model, for a block of consecutive vector computational instructions, first all operations on the first element of the vectors are executed, then the operations of the second element, and so on. Figure 2 shows an illustrating example of the difference between the two execution models. For this example, we assume that vector instructions operate on FP data by using a single FP unit and a single data cache port. The first *vecload* instruction is executed in the same way and with exactly the same timing on both models, since the models refer only to computational instructions. Regarding computational vector instructions, in the first case (*OBO*, Figure 2 (a)) all operations of one vector computational instruction (*vecadd*) are executed, and then we move on to the next vector instruction (*vecsub*). In the second case (*BBE*, Figure 2 (b)), several consecutive vector computational instructions form a block of vector instructions, and we execute one operation from each instruction of the block and repeat this for each operation in the block of vector instructions. In the example, we execute one operation from *vecadd* and then one operation from *vecsub*. The process ends once all operations are computed. The next subsection describes the *BBE* model in more detail.

2.1.1 Block-Based Execution. In order to support this model of execution, we added simple control logic and a small table that keeps the information of the instructions of the block. In the design presented in this paper, the blocks of vector computational instructions are formed dynamically in a very simple way: once a computational vector instruction is ready for execution, the control logic examines the next instruction in the issue queue and adds it to the block if it is a vector computational instruction. This process stops when the next instruction in the issue queue is of another type (a scalar or vector memory instruction) or the block table is full.

The number of vector instructions that can be executed in parallel or with chaining using the *OBO* model is restricted by the number of available FUs. *BBE* does not have this limitation, allowing for execution of more vector instructions in parallel. Inherently, more dependent instructions can be chained (scalar bypass logic can be reused) since one vector instruction does not occupy the ALU for all its elements in consecutive cycles, and thus it can be interleaved with other instructions using the same ALU. For example, *vecadd* and *vecsub* are not chained using the *OBO* model in Figure 2 because there is only one FP unit, while they are executed in parallel in *BBE* even though there is only one FP unit. This approach also allows for forwarding/bypassing results to next instruction and therefore reducing number of reads/writes to the vector register file as it is shown in subsection 5.2. An important advantage of *BBE* over *OBO* or a classic vector unit is the following: while a block of vector computational instructions is under execution, *BBE* allows for the execution of subsequent scalar or vector memory instructions if they are ready for execution and there are free functional units that can execute them. In Figure 2 (b), the second *vecload* instruction can start execution just after the *vecsub* started with execution of the first operation. Evaluation of kernels

Table 1. Microarchitectural parameters.

Parameter	Value	Parameter	Value
Instruction Width	32-bits	L1 D-Cache	32KB, 64B/line, 4-cycle latency
L/S Queue	16 entries	L2 Cache	256KB, 64B/line, 12-cycle latency
VMIT Entries	8 entries	ALU units	One simple, one complex int ALU and one FP ALU
VRF	16 registers	L1 D-Cache ports	One read/one write

with FP data in subsection 3.1 also clearly shows the advantage of *BBE* over *OBO* or the classic vector unit.

BBE has some drawbacks: the number of vector computational instructions dynamically included in a block is sensitive to the placement of the vector instructions inside vectorized code. Smaller blocks present less opportunities for forwarding/bypassing that can be achieved with longer blocks. To keep the design simple, in this paper the implementation does not allow overlapping of execution between two consecutive blocks, which is a potential solution to alleviate this problem. This means that the first block needs to finish execution before the second block can start. Another limitation is that dealing with multi-cycle instructions and especially groups of dependent instructions with different latencies inside a block requires additional control logic support. In our model, we support execution of multi-cycle instructions as well as execution of dependent instructions with different latencies, but restricted to the case where they are executed on different ALUs. The benefit of *BBE* can be further increased if we can chain vector memory instructions in an efficient way. The next section describes the vector memory unit of the proposed system, and subsection 4.1 presents our proposal to implement chaining from the memory hierarchy.

2.2 Vector Memory Unit

The vector memory unit holds and controls the execution of vector memory instructions. There are two tables that hold the necessary information to execute vector memory instructions, as shown in Figure 3. The vector memory instruction table (VMIT) keeps information for each instruction (instruction, start address, stride, number of elements, current element and number of completed operations). The vector memory control table (VMCT) controls the exchange of packets with the L1 cache. Each entry in this table has the following fields: instruction ID, packet/request ID and a valid bit. Additional fields are used to identify the corresponding element(s) of the source/destination vector register.

The address generator (AG) performs the address generation for vector memory instructions. The information about the instruction stored in the VMIT, namely the opcode, start address, stride, number of elements and current element, is enough to generate all requests to the cache hierarchy. We can decode the type of vector memory instruction (unit-strided, strided or indexed) and destination/source register from the instruction opcode field. The stride field can hold the stride for strided memory instructions or the ID of a vector register that holds the index vector for indexed memory instructions. We load/store whole cache lines for unit-stride vector memory instructions with a single access.

ACL controls the proper execution of vector and scalar memory instructions. In our model, we support a very simple aliasing policy to limit the complexity of the control logic: scalar loads wait until all older vector stores finish and vector loads wait until all older scalar stores finish. We provide more architecture-level details in the next subsection regarding the additional control logic.

Vector Memory Instruction Table

32 bits	32 bits	32 bits	8 bits	8 bits	8 bits
instruction	start_address	stride	num_elem	curr_elem	completed
⋮		⋮			⋮

Vector Memory Control Table

3 bits	8 bits	1 bit	2 bits	6 bits	6 bits
instID	reqID	valid	FLM	first_elem	last_elem
⋮		⋮			⋮

Fig. 3. Vector memory unit.

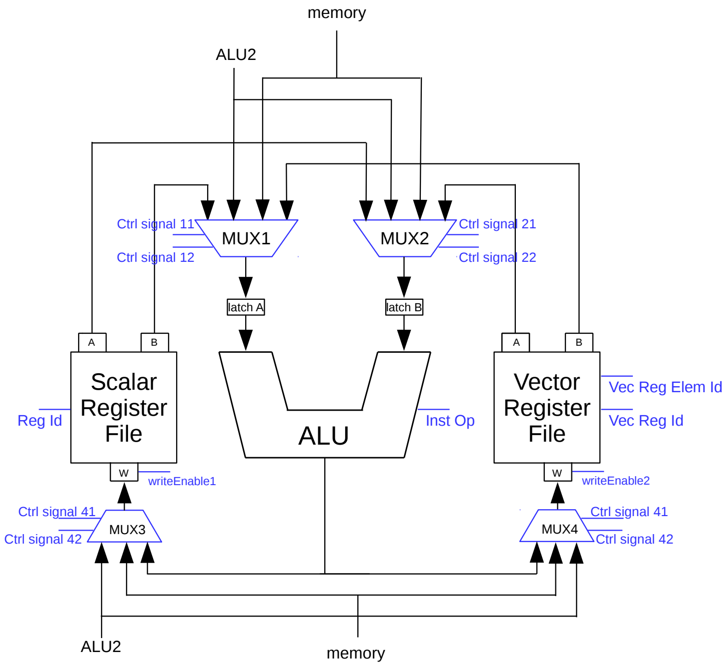

Fig. 4. Required control logic for a single ALU to support execution of vector computational instructions.

2.3 Implementation

As it is explained in Section 2, we added the control logic to support execution of vector instructions (VECL, ACL, CCL). We first describe the hardware implementation to support VECL. We augmented the multiplexers (MUX1 and MUX2) on each input of the ALUs to select the input from the scalar or the vector register files. Each multiplexer has two additional inputs: the first one (labelled as âĂIJALU2âĂİ) comes from the output of the second ALU and is present in the scalar design for bypassing, but we use it also for chaining of vector computational instructions and direct forwarding (see Section 4.2); the second input (âĂIJmemoryâĂİ) is related to the possible implementation of direct forwarding from the vector memory instructions (see Section 4.2); however, it would be present already in the scalar design if it implements bypassing from memory instructions. There are also two multiplexers (MUX3 and MUX4) at the input of the write ports (W) in the scalar and vector registers files in order to write a result from the appropriate source (an output of one of ALUs or data loaded from the memory hierarchy). We need control signals for each of the multiplexers. Figure 4 shows the required hardware for a single ALU. VECL also contains control logic including a counter and a comparator that tracks how many elements of a vector computational instruction have been processed and compares the current counter value with the VL register.

Table 2. Vectorized kernels.

kernel name	benchmark	access pattern	com/mem ratio	# of vec insts per iteration	loop count	data type	vector/scalar op ratio
sphinx-a	Sphinx3	strided	2:1	12	128	FP	56:1
sphinx-b	Sphinx3	unit-stride	4:3	7	32	FP	10:1
sphinx-c	Sphinx3	indexed	1:1	4	20.5	INT	2:1
saxpy	-	unit-stride	1:1	4	512	FP	53:1
h264ref	h264ref	unit-stride	3:1	4	16	INT	1:2
hmmer	hmmer	unit-stride	11:8	38	256	INT	3:1
graph500	graph500	indexed	1:3	4	28	INT	4:1
facerec	Facerec	stride	17:11	56	25.3	FP	6:1

Regarding ACL, we added two registers in LSQ and VMU that point to entries of the oldest scalar and vector loads respectively and one comparator to compare scalar loads with the oldest vector store and vector loads with the oldest scalar store. The hardware design of CCL follows conventional implementations of chaining logic (Hennessy and Patterson 2011).

3 INTEGRATED DESIGN EVALUATION

We extended the gem5 simulator (Binkert et al. 2011) to model an in-order ARM core, a classic vector unit (*CVU*) and our two models of execution: *OBO* and *BBE*. *CVU* can be seen as a co-processor or accelerator to the scalar core (Espasa et al. 2002). The *OBO* model is very similar to the ARM's own VFP mode, which reuses FP registers as short vector registers (Seal 2000), when executing vector computational instructions with FP data. There are still some differences that are discussed in Section 6. Simulations are performed using system call emulation mode.

Table 1 summarizes the micro-architectural parameters used in our experiments. *CVU* has two additional integer ALUs (one simple and one complex) and one FP ALU to execute vector computational instructions. The vector register file has 16 vector registers. We use four different maximum vector lengths (MVLs): 16, 32, 64 and 128 elements, with 32 being the default length. Gem5's model of the bus between the CPU and the L1 data cache is extended to model bandwidth and bus contention. We used 16 bytes for the bus width. L2 bus bandwidth is one cache-line per cycle. The L2 also has a simple strided hardware prefetcher. *OBO* has a similar configuration to *CVU* except it uses scalar ALUs to execute vector computational instructions. Broadcast logic (including the broadcast bus) is also different. The differences between *OBO* and *BBE* are a table that holds vector computational instructions (four instructions) for *BBE* and logic that controls execution of vector computational instructions. We used a latency of one cycle for all instructions that use the simple int ALU, three cycles for the complex int ALU and four cycles for all instructions that use the FP ALU.

We modified McPAT (Li et al. 2009) to evaluate power, energy and area of these micro-architecture variants. We modeled additional structures using the same approach and borrowing the parameters from existing structures in McPAT or CACTI if it is suitable. For example, in order to model a decoder for chaining logic from the memory hierarchy we use a decoder from decode stage with new parameters (e.g. number of bits to compare). We assume a 40nm technology for embedded processor with low operating power for energy, power and area evaluation.

Table 2 lists the kernels that are used to evaluate our design: saxpy microkernel, the three most time consuming kernels from Sphinx3, one from H264ref, one from Hmmer, one from Facerec and one from Graph500. The first three benchmarks are from the SPEC2006 benchmark suite. We have chosen these applications because they represent typical mobile applications: Sphinx3

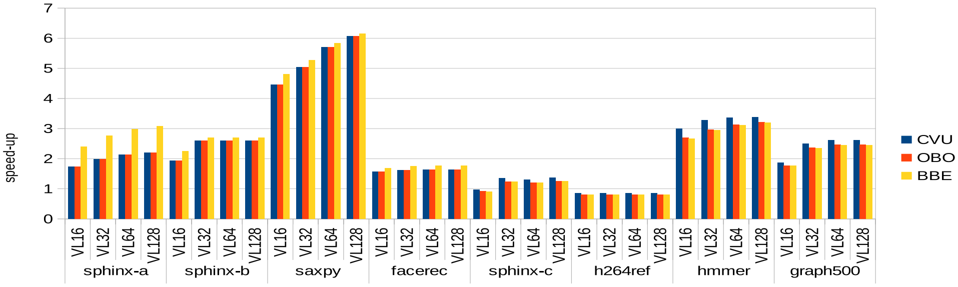

Fig. 5. Speed-up for *CVU*, *OBO* and *BBE* over the scalar baseline.

performs speech recognition, H264ref does video coding while Hmmer feature hidden markov models which are used in machine learning. Facerec is from the SPEC2000 benchmark suite. Even though SPEC benchmarks are typically used to evaluate general purpose processors, applications such as speech recognition or face recognition are widely used in mobiles. For example, Facerec contains FFT computation - a kernel found in EEMBC benchmark suite (Weiss 1999). The `saxpy` microkernel is not as representative of mobile workloads but we chose it as a a paradigmatic example of a vectorized kernel. Moreover, its simplicity and characteristics help to reason about the results. The Graph500 (Murphy et al. 2010) benchmark is a data intensive, high performance graph processing application but we choose this application because it is highly cache unfriendly and it is a good example to evaluate our ideas for cache unfriendly scenarios. The vectorization potential of the Graph500 was evaluated by Stanic at al. (Stanic et al. 2014). `sphinx-a`, `sphinx-b`, `saxpy` and `facerec` operate on FP data while the rest of kernels use integers. Table 2 also presents characteristics for each kernel. Our eight kernels exploit several different memory access patterns. Four kernels have a unit-stride, `sphinx-a` and `facerec` have strided while `sphinx-c` and `graph500` have indexed memory access patterns. The ratio between computational and memory instructions varies across kernels as well as the number of vector instructions inside a vectorized loop. Kernels also have different loop counts. There are kernels with very short loop count (`sphinx-c`[1], `facerec`[2], `h264ref`, `sphinx-b` and `graph500`[3]) and kernels with a longer loop count (`sphinx-a`, `saxpy` and `hmmer`). The last column shows the ratio between vector and scalar operations. Most of the kernels have a high percentage of operations executed from vector instructions (`sphinx-a`, `sphinx-b`, `saxpy`, `sphinx-b` and `facerec`). `sphinx-c`, `hmmer` and `graph500` have between 66% and 80% of all operations executed in vector instructions, while `h264ref` is the only kernel with dominant scalar operations (around 66%). We tried to cover as many scenarios and aspects as possible with these eight kernels. We extracted the input data from the applications when running the ref input data set and used them to initialize the data structures before simulation. We extended the ARM ISA with a set of 27 vector instructions that we used to vectorize the kernels.

3.1 Performance Evaluation

Figure 5 shows the speed-ups for *CVU*, *OBO* and *BBE* over the scalar baseline for all kernels. As expected, *CVU* outperforms the integrated designs (*OBO* and *BBE*) for integer data (`sphinx-c`, `h264ref`, `hmmer` and `graph500`) because it has two additional ALUs for computational vector instructions while ALUs are shared between scalar and vector computational instructions in the

[1]This is the average vector length for all iterations. Most of iterations are short, the maximum count is 166.
[2]This is the average vector length for all iterations. Three vector lengths (4, 18 and 64) repeat cyclically.
[3]This is the average vector length.

Table 3. Total number and number of vector loads that start earlier execution in *BBE* per iteration.

kernel name	# of vector loads per iteration	# of early vector load executions in BBE per iteration for FP kernels.
sphinx-a	27	25
sphinx-b	3	1
saxpy	2	0
facerec	14	4

integrated designs. In the vectorized loops, there are still many scalar instructions for bookkeeping, which then interfere with the execution of vector instructions in the integrated designs. The integrated designs provide good speed-up over the scalar baseline for `hmmer` and `graph500`, while there is little speed-up over the scalar for `sphinx-c`. The main reasons are the use of index memory accesses, the small amount of computational vector instructions and the short vector lengths for the majority of iterations. `h264ref` is an example where vectorization is inefficient. We obtained slow down over the scalar for all vector models. Despite this kernel has a short vector length (only 16), the main reason for slow down is that in order to vectorize this kernel, we require twice the computational operations as the scalar version.

Regarding the FP kernels, the vector models are always faster than the scalar baseline. Speed-ups for `saxpy` are extremely high. It is a very simple kernel with a unit-strided memory access pattern and highly regular DLP. A whole cache line can be accessed with one access (16 elements per access) and for this particular experiment the input data set fits into L1 cache. `sphinx-a` and `sphinx-b` have decent speed-ups. `sphinx-a` uses strided vector memory instructions to load data from the cache, which prevents reaching higher speed-ups. `sphinx-b` uses unit-stride vector memory instructions but it is limited by the number of iterations of the original scalar loop that has been vectorized (only 32 iterations). `facerec` has the smallest speed-up and the main reasons are the presence of strided memory accesses and a short vector length in most of cases (only four or eight). *CVU* and *OBO* have the same execution time (a consequence of having a single FP unit) while *BBE* outperforms them due to the advantage of executing subsequent integer scalar (loop overhead instructions) or vector memory instructions in parallel with the current block of vector computational instructions. In-order execution blocks the vector instructions and subsequent instructions in *CVU* and *OBO*, but this does not happen in *BBE*, as shown above in Figure 2. This difference is especially noticeable for `sphinx-a` where the speed-up for *BBE* over *CVU* and *OBO* is around 1.4x for all MVLs. `sphinx-b`, `saxpy` and `facerec` exploit this advantage from *BBE* execution inside single iteration of a vectorized loop, while `sphinx-a` is a kind of kernel that is able to exploit this across multiple iterations. There is often vector memory store instruction at the end of a vectorized loop (`sphinx-b`, `saxpy` and `facerec`). This instruction needs to wait until all elements of a vector source register are computed before it starts with execution. Then we can proceed with the next iteration. `sphinx-a` does not have a vector memory store instruction in the inner loop and it is able to overlap execution of a vector load instruction in next iteration with vector computational instructions in the previous iteration. For this particular case, vector computational instructions are completely overlapped with vector memory instructions across all iterations of the inner loop, yielding significant speed-up of *BBE* over *CVU* and *OBO*. Table 3 shows how often we are able to start earlier with vector loads execution in *BBE* compared to *OBO* for FP kernels. Almost all vector loads in `sphinx-a` can start execution earlier in *BBE* and we have significant speed-up for `sphinx-a` as a direct consequence. `Sphinx-b` and `facerec` have around one third of all vector loads that can start earlier execution in *BBE* while scalar loop bookkeeping instructions are executed earlier in `saxpy`. We can notice bigger difference for shorter MVLs between *BBE* and *OBO* as a consequence of more loop iterations for shorter MVLs.

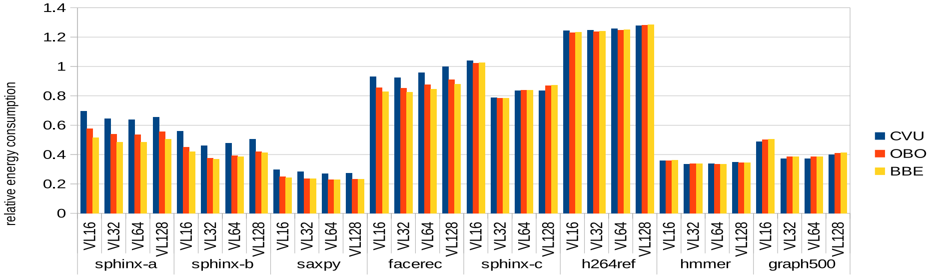

Fig. 6. Normalized energy for *CVU*, *OBO* and *BBE* over the scalar baseline.

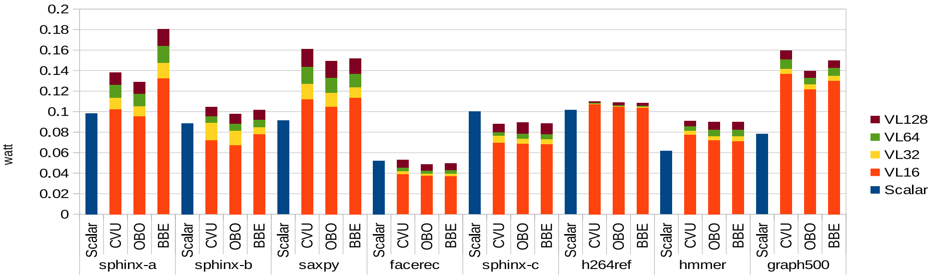

Fig. 7. Dynamic power.

Increasing the MVL from 16 to 32 elements provides better execution time for all kernels except `h264ref` (only 16 iterations in the vectorized loop) and `facerec` (in most cases only four or eight iterations in the vectorized loop). Further increasing MVL to 64 is beneficial for `sphinx-b`, `saxpy`, `hmmer` and `graph500` while vector registers with 128 elements only provide marginal speed-up for `sphinx-a` and `saxpy`. `sphinx-b` cannot exploit the benefits of MVLs beyond 32 because the vectorized loop has only 32 iterations. `sphinx-c` is limited by short loop counts and does not scale with longer MVLs.

We also analyzed the sensitivity to the latency of vector arithmetic instructions. We used four different latencies for vector arithmetic FP instructions: four, five, six and eight cycles. If we increase latency from four to eight cycles, speed-up over scalar baseline is decreases by 5.9% using MVL 16 in `sphinx-a`, while this number is only 1.1% MVL 128. Results are similar for other kernels.

3.2 Area, Power and Energy

Table 4 shows the area and leakage numbers for the scalar baseline and the *CVU*, *OBO* and *BBE* models of execution. We present results for vector registers with 32 elements. The area is computed in mm^2. Leakage is presented in watts.

The area overhead of *OBO* and *BBE* is only 4.66% compared to the scalar baseline (it is around 3.3% for vector registers with 16 elements and it goes up to 12.2% for vector registers with 128 elements). *CVU* without a FP unit increases the area of the baseline for 9.6%. When we include a FP unit, the area overhead of adding *CVU* is significant, around 44% with vector registers of 32 elements.

Energy consumption of *CVU*, *OBO* and *BBE* is shown in Figure 6, normalized to the scalar baseline. It is significantly lower than in the baseline for `sphinx-a`, `sphinx-b`, `saxpy`, `hmmer` and `graph500` (kernels with decent or high speed-ups over the scalar baseline), showing how adding a vector unit is an energy-efficient way to increase performance. As can be expected, the exceptions

Table 4. Area and leakage.

	Scalar	CVU		OBO/BBE
		w/o FP	w/ FP	
Area [mm^2]	2.831	3.065	4.040	2.925
Leakage [W]	0.057	0.069	0.085	0.060

are `facerec`, `h264ref` and `sphinx-c` due to slow-down or small speed-up. Energy consumption for `sphinx-c`, `h264ref`, `hmmer` and `graph500` is very similar for *CVU*, *OBO* and *BBE*. Regarding FP kernels, *OBO* and *BBE* clearly outperform *CVU*. *BBE* is also better than *OBO* for `sphinx-a`, `sphinx-b` and `facerec`. If we consider different MVLs, Figure 6 shows that 32 elements is the optimal size for the vector register with respect to energy consumption.

Figure 7 presents hows dynamic power for scalar, *CVU*, *OBO* and *BBE*. Results are stacked for vector models using four different MVLs. *OBO* and *BBE* have slightly lower dynamic power than *CVU* for integer kernels. For FP kernels, *OBO* has always lower dynamic power than *CVU*, while *BBE* has the highest dynamic power for `sphinx-a`. This is a direct consequence of the speed-up achieved with *BBE* over *CVU* and *OBO* (Figure 5) combined with its not so large reduction of consumed energy (Figure 6). For the rest of kernels this difference is smaller and we observed that the most power-consuming model changes with the MVL. *OBO* and *BBE* also have lower dynamic power than the scalar baseline for `sphinx-c` and `facerec` for all MVLs and `sphinx-b` with shorter MVLs (16 and 32) due to significant savings in the front-end of processor (instruction fetch and dispatch).

This evaluation indicates that if only performance is important and only integer data is used then *CVU* should be chosen. For mobile devices, in which power and area are of the utmost importance, one of our proposed integrated models is a good match. For FP kernels, our *BBE* model is clearly the best choice from the performance, area and energy consumption perspectives.

4 ENHANCEMENT OF INTEGRATED DESIGN

In this Section, we present three techniques that improve the basic integrated design presented and evaluated in previous sections. The techniques have negligible area overhead and increase performance or reduce energy and/or power with respect to the basic integrated design. The first technique covers the design and implementation of chaining logic that is optimized to work with the cache hierarchy through vector memory instructions, the second technique reduces number of reads/writes to/from the vector register file while as a third technique we propose two vector memory instructions: memory shape instruction and unified indexed vector load. We combined techniques in the evaluation whenever it has sense. For example, we studied the interaction of other techniques with the chaining from the memory hierarchy.

4.1 Chaining from the Memory Hierarchy

Chaining from vector memory instructions was a feature typically implemented in classic vector supercomputers (Russell 1978), (Schönauer 1987). In those systems, the vector processor accessed main memory directly. The lack of a cache hierarchy made memory access time completely predictable, so given an instruction it was simple to determine exactly in which cycle every element would arrive from memory. Therefore, additional control logic to support chaining from vector memory instructions did not require substantial resources. The issues arose with vector microprocessors with a cache hierarchy. Due to the nature of the cache hierarchy and cache misses it was difficult to predict when the requested data will be arrive from the cache. Processor architects considered that

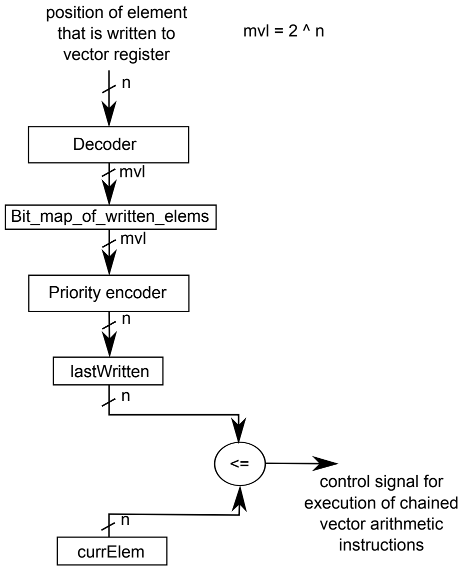

Fig. 8. Chaining from memory hierarchy.

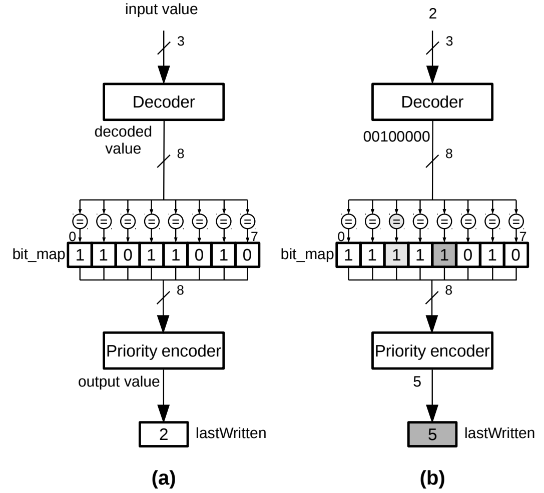

Fig. 9. An example of how to update the *lastWritten* register for chaining from the memory hierarchy.

supporting chaining from the memory hierarchy is expensive and required complex control logic, and did not implement this feature in vector microprocessors (e.g. (Kozyrakis and Patterson 2003)).

We consider that chaining from cache can be particularly fruitful for our *BBE* model due to the opportunity to chain larger number of vector arithmetic instructions in a block. Therefore, we decided to design and evaluate support for this feature. However, this technique works not only for *BBE*, but for *OBO* and *CVU* as well. Figure 8 shows the logic and structures we propose to allow chaining from the memory hierarchy to a dependent instruction. The main idea is to track the last written element (*lastWritten*) of vector register and use it in the VECL (the Vector Execution Control Logic, as explained in Section 2). Then, the VECL can determine if the current vector operation is ready for execution, simply by comparing the last written element and the current operation (*currElem*): if *lastWritten* is greater than or equal to *currElem* then VECL can execute the current vector operation. Otherwise it needs to wait until *currElem* is written in the vector register and *lastWritten* updated. The most complex operation of this design is updating the *lastWritten* register. Due to the presence of caches, the elements of a vector load can arrive out of order to the vector register, so we need to track all elements that have arrived. To this purpose, we added a bitmap. The size of the bitmap is equal to the MVL of the vector registers. In order to decode which element is written, a decoder is added. Subsequently, a priority encoder is used to determine the new value of the *lastWritten* register: the position of an element in the vector register, in which all prior elements are written already, as well as itself, and its immediate successor is not written yet.

Figure 9 illustrates an example of how the *lastWritten* register is updated. Current state is presented in Figure 9 (a). The *lastWritten* register has value two because, as can be seen in the bitmap, elements 0 and 1 of the vector register are already written while element 3 has still not arrived from the cache (i.e. element 0 and element 1 have value one and element 2 has value zero in the bitmap). Figure 9 (b) shows what happens when element 2 in the vector register is written and 2 is the input of the decoder to update the *lastWritten*. The decoded value is *00100000* and the corresponding element in the bitmap is set to one (the light gray box). Then the content of

```
vecloadst  VR2
vecloadst  VR3
vecsubsv   VR4, scalar, VR3
vecmul     VR5, VR4, VR4
vecmul     VR6, VR5, VR2
vecsub     VR7, VR7, VR6
```

Fig. 10. A sequence where writing to the vector register file can be avoided.

the bitmap is send to the input of the priority encoder. Value 5 is generated since it is the last consecutive one in the bitmap (dark gray box) before the first zero. Therefore, 5 is written to the *lastWritten*.

In an initial design, we first assume that each vector register has dedicated chaining logic. This design tracks the last written element for each vector load even though there is no ready dependent computational vector instruction that can be chained. This decision leads to high area, power and energy overheads. Aiming to decrease the number of chaining elements and track the last written element only for vector loads that can be effectively chained from, we also propose a restricted chaining from memory with fewer chaining elements. The chaining control logic needs to know if it will track the last written element for a vector load, i.e. if there is any dependent computational vector instruction that is ready for execution. Since there is a time window to resolve if there is any ready dependent instruction between the moment a vector load starts execution and the moment the first element arrives to the destination vector register, we decided to use this as a requirement to apply chaining from memory. This means that a dependent instruction will be chained only if it is in the issue queue and ready for execution before the first element arrives to the vector register. Otherwise, the dependent instruction stalls until the vector load is completed, just like in the basic design without support for chaining from the memory hierarchy.

4.2 Direct Forwarding

While vectorizing the kernels, we realized that there are cases in which data are computed and then used only once. Figure 10 shows an example from Sphinx3 kernel. Instruction *vecsubsv* stores the result of its computation in vector register *VR4* and only the subsequent instruction *vecmul* uses it as input operand. The scenario is the same for vector registers *VR5* and *VR6*. They are only used once as input operands by subsequent instructions. Therefore, if these instructions are chained and we can take advantage of the fact that they are read only once, we could save three x VL writes and four x VL reads to/from the vector register file in this particular example. We are saving two reads in the first *vecmul* instruction, both for vector register VR4. Moreover, forwarding from vector memory instructions we could avoid writes and reads to vector registers *VR2* and *VR3* as well. It means that we would need only to read data from vector register *VR7* and to write the final result there.

To further analyze the potential applicability for result forwarding without writing to the vector register file (direct forwarding), we examined all vectorized kernels with *BBE* model of execution in mind. Table 5 shows how many vector registers are read or written per iteration (we just counted one access per vector register of each instruction, not VL times). Numbers for vector register reads and writes are counted for the configuration of the functional units presented in the table 1 with enabled chaining for vector computational instructions. They are the same for all three models of execution. Numbers related to direct forwarding are counted for *BBE*. Numbers related to direct forwarding show that we can reduce the number of reads and writes by more than two for `sphinx-a`, `sphinx-b` and `hmmer`, but there are also kernels that can not benefit from this technique.

Table 5. Potential reduction in writes and reads to/from the vector register file.

kernel name	sphinx-a	sphinx-b	sphinx-c	saxpy	h264ref	hmmer	graph500	facerec
#_of_vec_reg_reads	92	7	3	4	3	24	4	72
#_of_vec_reg_writes	82	6	3	4	3	17	2	48
reads with forwarding	42	3	3	3	2	10	4	60
writes with forwarding	41	3	3	3	2	9	2	36

Since there is already logic for scalar result forwarding/bypassing we decided to reuse it with small additional logic that will allow for direct forwarding. The key idea is to somehow identify if the result of an instruction is used in only one subsequent instruction. In many situations, the compiler can relatively easy identify the case when a value is used only once. The second part is annotation of identified instruction. One solution to annotate the instruction is to reserve a bit in encoding for vector computational instructions (e.g. a bit set to one). Forwarding logic detects if the bit is set to one and forwards results to corresponding FU without writing to the vector register. The consequence of this approach is that the ISA must be extended to allow for annotating of vector instructions.

Direct forwarding can be applied to all three models: *CVU*, *OBO* and *BBE* but it is much more suitable for *BBE* because it allows for execution of a larger number of vector computational instructions in parallel and therefore, it increases the benefit of direct forwarding. In *CVU* and *OBO* it is only useful for chained instructions, which depends on the number and types of the instructions executed.

4.3 Vector Memory Shape Instruction

Even though `facerec` is highly vectorizable, we did not obtain high speed-ups (up to 1.77x for *BBE*). The main reason is that it has a complex memory access pattern to a matrix. Elements are loaded from the matrix using three memory access patterns that are repeated in a cyclic fashion. Each memory pattern loads 64 elements. In the first pattern, 64 elements are loaded using a single strided vector memory load instruction with stride two. In the second pattern, we need four strided vector load instructions with stride two to load all 64 elements while in the third pattern we need 16 instructions (each one loads only four elements). We realized that there is a regularity in the accesses for the second and third case that cannot be expressed with strided memory instructions, but it would be possible to load all the elements by providing more complex vector memory instruction that supports this pattern: the "vector memory shape instruction" (vmshape) (Ciricescu et al. 2003). vmshape uses a base address and three scalar values: stride, span and skip to describe a vector. Stride has the same role like in strided vector memory instructions (spacing between each accessed element). Span describes how many elements to access at stride spacing before applying the second-level skip offset. Memory shape instructions can be called 2-D strided vector memory instruction because they are an extension of stride instructions to 2-D patterns.

In order to support the execution of vmshape, we slightly modified the vector memory unit (Figure 3). VMIT is extended with span (8 bits), skip (32 bits) and a field that counts the number of elements before skip is applied (8 bits). An additional circuitry is added to increment the third field and compare with span. AG is also extended to increment an address for stride or skip depending on the output of the previously mentioned comparator.

Table 6. L1 data cache hits and misses for unit-stride and indexed vector loads.

cache size	unit-stride load hits	unit-stride load misses	indexed load hits	indexed load misses
small	2254	4029	65970	1361
medium	2226	4057	56325	11006
big	2223	4060	33776	33555

4.4 Unified Indexed Vector Load

We achieved decent speed-up for `graph500` even though it is data-intensive, cache unfriendly application. In the first part of the BFS kernel, algorithm loads all neighboring nodes of the current node and checks if they are visited. In the current implementation we used three vector instructions to perform this task: 1) unit-stride vector load gets indices, 2) indexed vector load gets neighboring nodes, and 3) compare vector-scalar instruction checks if they are already visited. The indices are used directly by the second load. Since we already implemented chaining from memory instructions, we wanted to study how much would `graph500` benefit from merging the two load instructions. The idea is to have a single instruction and once we receive data from the data cache for indices (index values), automatically initiate corresponding accesses for the indexed vector load instruction (operations that correspond to loaded indices - final values). Loading an array and use it as index vector is also used in other applications (i.e. Sphinx3).

The desired scenario, in which the unified indexed load would be more useful, would be if one of the requests of the unit-stride vector load hits L1 data cache and the corresponding elements of the indexed vector load miss L1 data cache. We profiled `graph500` using 3,000 nodes and 50,000 edges as input parameters and three different cache configurations: small (L1 8KB and L2 128KB), medium (L1 16KB and L2 256KB) and big (L1 32KB and L2 256KB). Our findings were that unit-stride vector load misses a lot for all cache sizes (2/3 of all requests misses L1 data cache) while miss rate for indexed vector load depends on number of nodes and cache size (as table 6 shows). In this particular case we will receive earlier values for indexed vector load compared to the current implementation where the indexed vector load needs to wait until the unit-stride vector load is finished. Figure 11 shows a simplified example of how elements are loaded from the memory system using unit-stride and indexed vector load (a) and unified vector load (b). Requests to load indices (with a sequential pattern) are blue squares and loaded indices (with) are blue cycles. Requests to load final values (with an indexed pattern) are red squares and loaded final values are red cycles. Lets assume that we have the desired scenario for the second element: the request for the index value hits in the L1 while the request for the final value misses in the L1. The basic approach that uses separated unit-stride and indexed vector load (Figure 11 (a)) can not start sending requests for final values until all index values are loaded, while in the case of the unified vector load requests for final values initiate immediately after the index values are received (Figure 11 (b)). Therefore, we can see in the presented example that the second final value is loaded much earlier using the unified vector load as well as all final values. If we assume that a hit to L1 is four and miss twelve cycles, all elements will be loaded in 33 cycles in the first case, while we will need 22 cycles with the unified vector load. We further profiled `graph500` and we found that there is always at least one request inside unit-stride vector load that hits L1 data cache. We can start with indexed load accesses for those cases while waiting for the rest. Our profiling results were promising and we decided to implement unified indexed vector load instruction (see subsection 5.4).

We had several dilemmas regarding the implementation of unified indexed vector load instruction. The initial idea was to add hardware that detects packets with indices (when the unit-stride vector load), buffer them and initiate accesses for indexed vector load at the cache side. We realized that it

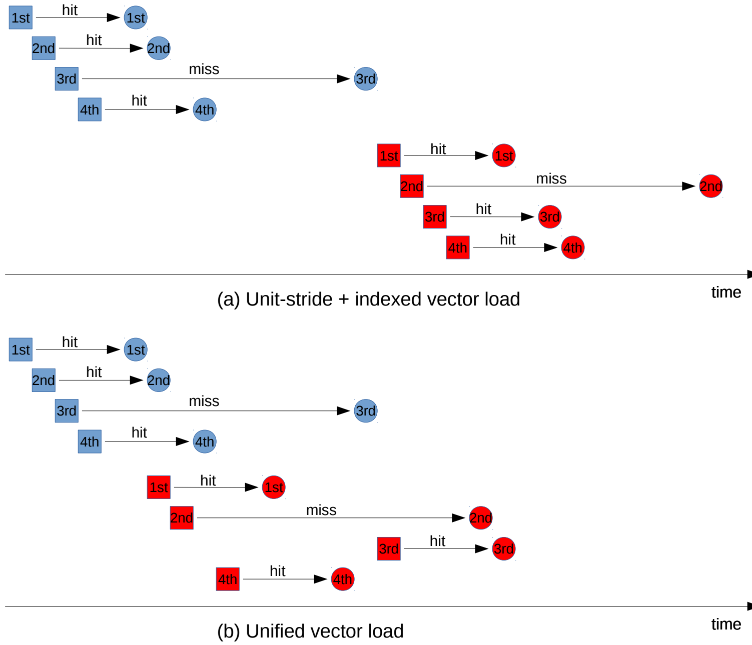

Fig. 11. An example that shows execution of (a) unit-stride and indexed vector load and (b) unified vector load.

would need to communicate with the TLB in some cases (e.g. if the index value is higher than the TLB page size) in order to perform the address translation. We also realized that the indexed vector loaded by unit-stride vector load is often used by several subsequent vector instructions and that we will still need to transfer those values to the vector register file. Therefore, we decided to move the additional hardware to the vector memory unit (see subsection 2.2) where most of the existing hardware can be reused to support execution of unified indexed vector load. If the index values are just used in the unified instruction then moving hardware to the data cache size could be a good design option.

Once the unified indexed vector load instruction is dispatched to the vector memory unit, two entries in VMIT are reserved. The first entry controls loading of index values, while the second entry is responsible for getting final values. As additional hardware we need only two buffers. The first buffer holds index values that are received from the data cache. Since packets that contain index values can return out of order due to cache misses, we need the second buffer where each value indicates the position in vector register for each index value in the first buffer. As it is explained above, index values are stored in a vector register. Once a corresponding element in vector register is selected to be written, the ID number is written in the second buffer. This information is important for the second part of the unified instruction in order to write the final values in the right position in the destination vector register. The size of the buffers is equal to the size of vector registers.

Our current implementation supports forwarding between older stores and subsequent loads based on addresses that are used in load instructions to get data. Since we will not have the address of the final value until we receive the corresponding index value from the data cache, we decided to apply a conservative approach during the execution of the unified instruction. Our unified instruction waits all older vector stores to complete before it starts execution. Particularly, the kernel `graph500`, it is possible that there is a conflict between older vector stores and subsequent

vector loads because both access the same array. It means that subsequent vector loads could access a value that is written by older vector stores. Therefore, our conservative approach is required. If we are sure that there is no collision between stores and loads, a weak ordering mechanism as in the latest NEC SX-ACE machine (Momose et al. 2014) could be applied.

5 EVALUATION OF ENHANCED INTEGRATED DESIGN

In this Section we evaluate and present results for the techniques described in the previous Section.

5.1 Chaining from the Memory Hierarchy

Figure 12 presents the speed-ups achieved with our two proposed approaches over the scalar baseline: full chaining support (FCS) and restricted chaining support (RCS), while Figure 13 presents the speed-ups achieved over the same models of execution without chaining. We used 16 chaining elements for FCS and four for RCS.

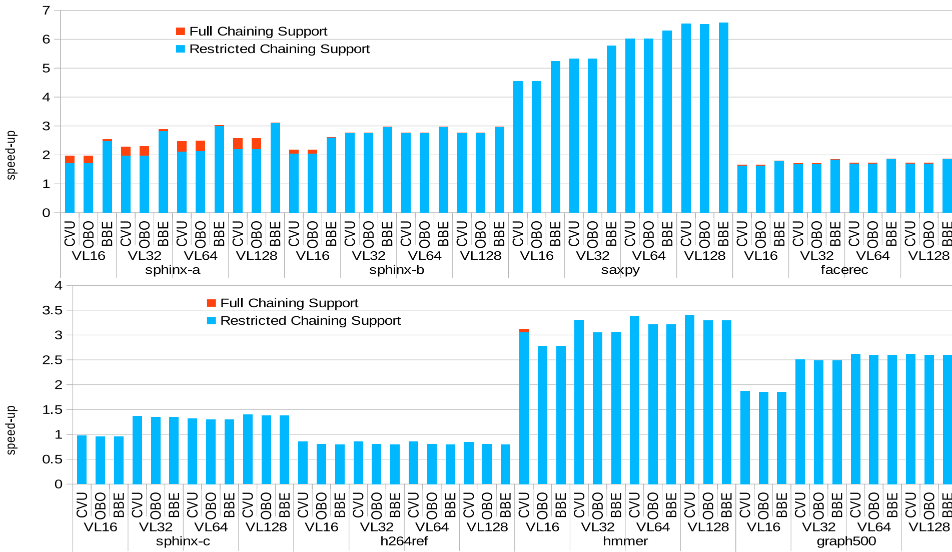

Fig. 12. Speed-up for CVU, OBO and BBE with full (FCS) and restricted (RCS) chaining support from memory hierarchy over the scalar baseline.

There are several interesting points in these figures. The difference in speed-up for FCS and RCS is negligible or inexistent for almost all kernels in all three models, except for `sphinx-a` and `sphinx-b` with MVL 16 for *CVU* and *OBO*. Therefore, for our target design where area and power are the most important constraints, RCS can provide good results for the most of kernels.

Each kernel has different trends and we analyze them case by case. FCS provides up to 17% improvement in `sphinx-a` for *CVU* and *OBO*, while *BBE* does not benefit a lot from chaining for this kernel. *BBE* without chaining already has significant speed-up over *CVU* and *BBE* (see subsection 3.1). This speed-up is mainly due to overlapped execution of vector memory and computational instructions in *BBE*. The single L1 D-Cache port is already almost fully utilized, and therefore chaining is not able to provide additional significant performance improvements. The speed-up of *BBE* over *CVU* and *OBO* is reduced from 40% to 20% with chaining. `Sphinx-a` does not benefit at all from RCS for *CVU* and *OBO*, while there is small speed-up for *BBE* and shorter MVLs. Applications with strided and indexed vector loads can benefit more from chaining because memory instructions

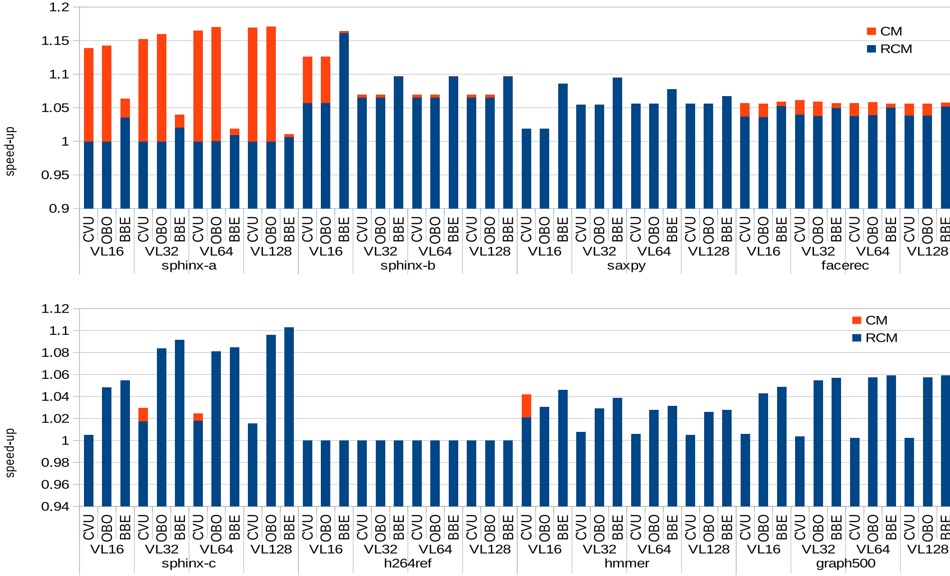

Fig. 13. Speed-up for *CVU*, *OBO* and *BBE* with full (FCS) and restricted (RCS) chaining support from memory hierarchy over the same models without chaining.

have longer execution time. This is the reason why we obtained the highest speed-up with chaining for `sphinx-a`. *BBE* benefits the most from FCS in `sphinx-b`, around 16% for MVL 16. *CVU* and *OBO* also experience improvements higher than 10% and trends are consistent across different MVLs. We knew that unit-stride vector loads are fast (whole cache line per access) but we still expected better results for `saxpy`, speed-up increases around 5% for *CVU* and *OBO*, while *BBE* has slightly better results, up to 9.5% for MVL 64. The data set for `saxpy` is cache resident, so we decided to perform a test to pollute the L1 data cache before `saxpy` starts computation. The results that are then obtained for *CVU* show 20.5% of improvement for MVL 32. This means that applications with poor cache locality can benefit a lot from chaining from memory hierarchy, similarly to the applications with strided and indexed memory instructions. Table 7 shows L1 data cache miss rates for current data input sets and MVL 32. Speed-ups are the same for FCS and RCS in `saxpy`. *BBE* increases the speed-up over *CVU* and *OBO* for `sphinx-b` and `saxpy`. FCS provides around 5% of improvement in `facerec` for all models, while using of RCS slightly decreases the speed-up achieved with FCS.

Even though `sphinx-c` has indexed vector loads, *CVU* does not benefit a lot from chaining mainly due to the use of short vectors, while chaining provides up to 10% improvements for *OBO* and *BBE* with MVL 128. The difference in execution time between *CVU* and *OBO* or *BBE* with chaining is only 1.4% for MVL 128 while it was 9.5% without chaining. `h264ref` does not benefit at all from chaining. The reason is the short vector length of the loop, combined with the unit-stride access. The vector load is able to bring all 16 elements from data cache with one access. Therefore, chaining does not play any role in this case. We also expected higher speed-ups for `hmmer`, but we observed that the execution of consecutive vector computational instructions is serialized. The reason is that a unit-stride vector load is followed by several vector computational instructions that are all of the same type. Since there is only one ALU for that operation that they must share, the first instruction can start execution a few cycles earlier while the rest of instructions still need

Table 7. L1 data cache miss rates for MVL 32.

kernel name	L1 miss rate in %
sphinx-a	8.0
sphinx-b	25.1
sphinx-c	3.0
saxpy	7.8
h264ref	6.0
hmmer	7.2
graph500	9.8
facerec	17.2

Table 8. Total number and number of chained vector loads per iteration.

kernel name	# of vector loads per iteration	# of chained vector loads per iteration
sphinx-a	27	26
sphinx-b	3	2
sphinx-c	2	2
saxpy	2	2
h264ref	1	0
hmmer	14	9
graph500	2	1
facerec	14	4

to wait for the ALU. Adding another ALU could overcome this issue. *CVU* does not benefit from chaining in `graph500` while chaining provides almost 6% of improvement for *OBO* and *BBE*. *OBO* and *BBE* provides almost the same speed-ups with chaining as *CVU* for `graph500`.

Table 8 shows how often chaining is used. The first column lists the kernels. The second column shows the number of vector loads per single iteration in vectorized kernel, while the last column shows how many vector loads are chained. We can observe that the chaining from vector loads is quite often used in most of the vectorized kernels, except in `h264ref` and `facerec`.

The area overhead of adding chaining logic from the memory hierarchy depends on the size of the vector register. Area for FCS ranges from 0.4% for MVL 16 up to 2.6% for MVL 128 in *OBO* or *BBE*. It is around 19% of total area overhead for *OBO* or *BBE* for MVL 128. This percentage is lower for *CVU* since it has a larger total area. RCS contributes four times less area overhead and it is around 5% of total area overhead for *OBO* or *BBE* for MVL 128.

Dynamic power contribution of FCS or RCS to the total dynamic power is always lower than 1% for all kernels. Regarding energy consumption for FCS, there is up to 6% savings for `sphinx-a` and `saxpy` in *CVU*, while the rest of kernels are in the range of 2%. Savings are smaller for RCS in *CVU*, up to 2.8% for `sphinx-b`. Savings are similar for *OBO* and *BBE*. Some kernels have higher energy consumption over models without chaining for higher MVLs and RCS but it is 1.6% in the worst case for `sphinx-a` with MVL 128 in *BBE*.

All these numbers suggest that chaining from memory hierarchy is fruitful for most of the kernels in terms of performance and energy savings with small area overheads. The contribution of chaining logic to the total dynamic power is also negligible.

5.2 Direct Forwarding

We evaluated this technique in terms of energy and power in McPAT and results are presented in Figure 14. Using this technique does not affect the performance results. Obtained results show that we can save more than 50% of energy/power of the vector register file in `sphinx-a`, `sphinx-b` and `hmmer`. `h264ref`, `saxpy` and `facerec` have decent savings between 33% and 16%. Depending on the size of vector register, savings at the CPU level range from 2.5% for short vector registers (16 elements) up to 11% for MVL of 128 elements for `sphinx-b`.

The results presented in this subsection show the benefit of applying this technique between vector arithmetic instructions. It would be possible to achieve even further savings in the vector register file if this technique is combined with chaining from the memory hierarchy to provide direct forward for vector memory instructions as well. This idea is more challenging because it will require more complex logic to implement it and the proposed solution should not affect the performance results. We leave this for future work.

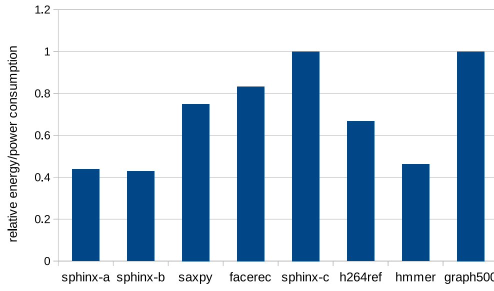

Fig. 14. Normalized energy/power consumption for BBE model with direct forwarding over the same model without direct forwarding.

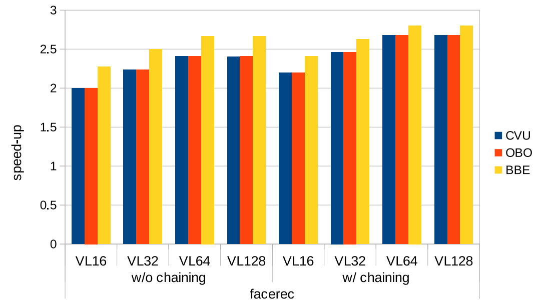

Fig. 15. Speed-up for *CVU*, *OBO* and *BBE* over the scalar baseline when using vmshape.

5.3 vmshape Instruction

Figure 15 shows the speed-ups for *CVU*, *OBO* and *BBE* over the scalar baseline for `facerec` with vmshape. We presented results both with and without chaining from the memory hierarchy. vmshape increases speed-up from 1.63x to 2.4x for *CVU* and *OBO* using MVL 64 while the speed-up is increased from 1.77x to 2.66x for *BBE*. Chaining from the memory hierarchy also additionally improves performance (from 2.4x to 2.68x for *CVU* and *OBO*). It is also interesting that performance scales better with increased MVL from 16 to 64 elements compared to the implementation without support for vmshape (Figure 5). We also modeled the additional hardware in McPAT and results show negligible area overheads (less than 0.2%) while energy savings are almost proportional to the achieved speed-up (from 9% for MVL 16 up to 18% for longer MVLs).

5.4 Unified Indexed Vector Load

Figure 16 shows the speed-ups for the vectorized version of `graph500` using the unified vector load over the version with the indexed vector load. Results are presented with and without chaining from the memory hierarchy for all three models of execution: *CVU*, *OBO* and *BBE*. The unified vector load instruction increases speed-up only up to 1.08x for *CVU* without chaining from memory. Results are similar with chaining from the memory hierarchy. We profiled execution of both vectorized versions and we found that the unified vector load completes execution faster than the indexed vector load. The speed-up per single instruction goes from 1.1x to 1.22x. Since the rest of the kernel is the same, overall speed-up goes only up to 1.08x. We also noticed that loading all neighboring nodes and their current state (visited or not visited) takes a few hundreds cycles (300-400) for 30 neighboring nodes on average. Therefore, the benefit of having a single instruction and saving some cycles in the front-end of the processor is also small in this particular example. The area overhead of the additional hardware is small, less than 0.3% in the worst case when two buffers with 128 elements are added. Energy savings are also small and proportional to the speed-ups, less than 4.1%, while dynamic power increases up to 8%.

Even though current results are not promising, the unified indexed vector load could be further improved. For example, if the index vector is used only in the unified vector load instruction, we could avoid writing to the vector register. We also noticed that a cache line that holds only neighbors of one node (index values) is accessed only once. We could avoid caching these lines in L1 or L2 and provide a small victim cache or streaming buffer for those lines. Therefore, L1 and/or L2 caches would not be polluted with cache lines that are used only once.

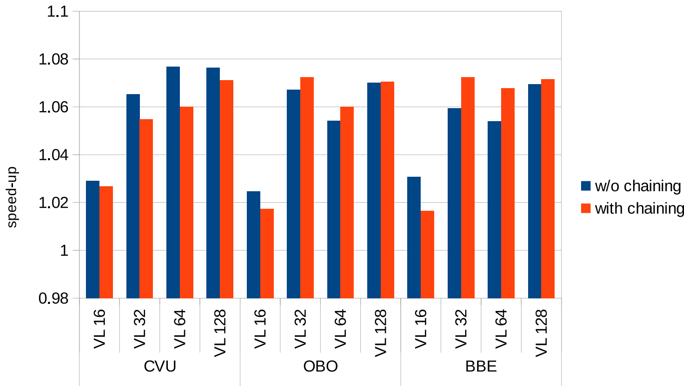

Fig. 16. Speed-up with the unified vector load over the indexed vector load in `graph500` for *CVU*, *OBO* and *BBE*.

6 RELATED WORK

In this section we discuss differences between our integrated design and several works that combine vector and scalar processing. We also discuss alternatives to vectors and justify design decisions.

An important characteristic of most microprocessor vector architectures is that the vector processing unit is designed as an extension or co-processor to a scalar core, but there are few works that combine vector and scalar processing on the same substrate. Quintana et al. (Quintana et al. 1999) added a vector unit to a superscalar core. Since their research focused on high performance, this processor design resembles a classic vector with multiple lanes and direct L2 access which is completely different from our design. This work also does not include any evaluation of power and energy of the proposed design, even though their approach requires additional hardware that is idle most of the time in scalar intensive applications. Gebis et al. (Gebis 2008) proposed the first integrated solution that combines scalar and vector processing (ViVA). However, ViVA adds support only for vector memory instructions while regular scalar instructions are used for computation. We have observed the advantages of using vector computation instructions, e.g. the energy reduction due to reduced front-end activity. Soliman (Soliman 2011) proposed a low-complexity vector core that has a common execution data-path for executing scalar/vector instructions, but using the scalar register file limits the design to support only short vector (up to 8 elements). PTX (Compute 2010) supports vector instructions, but they are transformed to scalar instructions for Nvidia SIMT microarchitectures, which behave like many independent scalar units. This can be seen an instance of using scalar hardware to execute vector instructions.

Combined vector-scalar design is also proposed in CELL and power processors (Gschwind et al. 2006), (Gschwind 2006), (Gschwind 2016). Cell and power processors target high-performance and it is crucial for them to have parallel hardware that can exploit available DLP and provide high performance. Since they realized the importance of coupling closely vector-scalar processing, support for scalar execution is added in the parallel hardware (SIMD vector unit). In our work we are targeting the low-end embedded market, where power, energy and area are important design concerns. So we used *the opposite* approach: we added minimal hardware to support execution of traditional vector instructions (not SIMD) on a scalar in-order core.

The Imagine processor (Khailany et al. 2001) implements a somewhat similar model of execution to our *BBE*. They called it compound stream operations. They perform multiple computational operations on each stream element. Our *BBE* model is more flexible, as instructions in the block

can be independent. The Imagine targets streaming applications with little data reuse, while our approach is more general. Their model of execution is implemented on specific accelerators while *BBE* works on a general purpose core and does not have any specific constraint. It can use regular vectorized code and execute in a block-based fashion without additional support from the compiler. The SCALE processor (Krashinsky et al. 2004) also implements a kind of block-based execution at the level of virtual processor (VP). VP instructions (RISC-like) are grouped into atomic instruction blocks (AIBs), which is the unit of work issued to a VP at one time. Albeit the common use of the term "block", our *BBE* blocks are completely distinct from AIBs. In our case, it is a block of vector instructions that is dynamically formed during execution. Like for the Imagine, *BBE* can execute regular vector code without additional compiler support.

BERET (Gupta et al. 2011) and DySER (Govindaraju et al. 2011) are architectures that provide reconfigurable datapaths to accelerate critical subgraphs of computation within a loop iteration, while XLOOPS (Srinath et al. 2014) deals with inter-iteration loop dependence patterns. Govindaraju et al. (Govindaraju et al. 2013) tuned a compiler to generate optimized code for DySER that can be seen as an alternative to SIMD. DySER, BERET and XLOOPS are accelerators, in which critical subgraphs or loops are executed on dedicated additional hardware that has own functional units, while we use existing functional units in our integrated design. They also extensively modified compiler to provide support for corresponding execution model, while our blocks are formed dynamically during execution. We just needed small support in a compiler for direct forwarding technique.

We would like to emphasize that we use *traditional vector execution* (Cray-I inspired) in our integrated design rather than SIMD implementations found in commodity processors such as Intel/AMD X86, Cell, Power, etc. SIMD processing units operate in parallel requiring multiple functional units and thus they are less efficient solutions in area and energy for embedded systems. Additionally, even when focusing on high-performance, execution of vector instructions in a pipelined form is still a relevant design point. For example the modern NEC SX-ACE vector processor (2013) has 16 vector pipelines in each core; each pipeline can execute up to four operations per cycle but each vector register has 256 elements; thus each instruction is pipelined serially in the functional units unlike SIMD-based architectures.

NEON is a SIMD unit incorporated in ARM processors. We predict that area overhead of NEON unit would be similar or higher to CVU in the case of 128-bit SIMD FP unit. Regarding performance, NEON should provide better performance for cache-friendly applications with unit-stride access and short vectors: NEON is able to process two or four operations in parallel, while we chose to provide a single vector lane. The advantages of our design would be apparent in applications with long vectors (due to the smaller code size and reduced dynamic instruction count, resulting in less front-end activity) and irregular memory access patterns - strided or indexed memory accesses (NEON only supports vectors that are stored consecutively in memory and therefore code with indexed or strided memory access pattern cannot be vectorized). Cache-unfriendly applications would also benefit greatly from using long vectors because vector memory instructions are able to hide long latencies.

OBO mode of execution is similar to the ARM's VFP mode (Seal 2000) for FP computation but there are still some differences. Our *OBO* model supports longer vector lengths while VFP is restricted to shorter vector lengths. Also, we provide support for gather/scatter vector memory instructions. We applied chaining for computational vector instructions as well as chaining from memory hierarchy. Moreover, the execution of vector instructions with integer data is also different, since our model uses scalar ALUs.

Our baseline is an energy-efficient in-order ARM core and we wanted to improve it in terms of energy and performance. Since vector processors are by default energy-efficient (Lemuet et al. 2006), we added support for vector instructions. We did not consider adding multiple lanes due to the following reasons: (1) we wanted to do minimal changes to the existing scalar processor in order to keep the area/power envelope and reuse the existing processor resources in the most efficient way; (2) adding one additional lane increases area of scalar baseline by 44%; adding four lanes will more than double the area of the processor and it is not acceptable in highly constrained low-end devices.

Early vector processors had single lane architectures (CRAY-1) and they were successful compared to others at that era such as IBM370. Efficiency was the factor. By analogy, in-order processors are overwhelmingly used in the embedded domain because they are more energy-efficient compared to bulky out-of-order architectures. Recent work demonstrates that single-lane vector units offer significant performance gains over commodity processors (Hayes et al. 2012). It shows quite clearly that the advantages come from a reduction of fetch/decode/rename/commit and consolidating memory requests together. Our results in the paper also confirmed that we can achieve good speed-ups over scalar baseline.

As we mentioned above, chaining from the memory hierarchy was popular in classic vector machines (Russell 1978), (Schönauer 1987). Since access time to the memory hierarchy was constant, it was not so difficult to implement it. To the best of our knowledge there is no published work that discusses how to implement chaining from memory hierarchy with caches.

Regarding direct forwarding, it can be seen as a extreme case of short-lived registers (Ponomarev et al. 2003), (Hu and Martonosi 2000) because we are using each produced value only once. Short-lived registers or values are used only for a short period of time after they are written, meaning that the destination registers targeted by these values are renamed by the time the results are written back. Writing to the register file is avoided by caching in a small dedicated register file (Ponomarev et al. 2003) or by using a small structure which sits between the functional units and the register file that buffer and filter accesses to the register file (Hu and Martonosi 2000).

7 CONCLUSION

Using a vector processor is one of the most energy efficient ways of achieving high performance for a wide number of applications that contain significant DLP. Power dissipation, energy consumption and area are critical concerns in processor design, especially for embedded systems in the low-end market. In this paper, we propose the integrated vector-scalar design. The integrated design allows for execution of vector computational instructions mostly reusing resources of an ARM in-order core. We implement two models to execute vector computational instructions: one-by-one and block-based execution models. To provide additional performance improvements, we propose and implement chaining from the memory hierarchy with relatively small area and power overheads. We also implement restricted chaining from the memory hierarchy that minimizes area overhead and provides the same performance improvements for the most of the evaluated kernels compared to the first approach. The key advantages are that our integrated design (a) apply vector-scalar processing automatically in hardware, (b) gain the advantages of a vector ISA, and (c) apply several energy-performance efficient techniques to reduce power and increase performance in a mobile CPU.

Our integrated design has several advantages. It has a small area and power overheads (only 4.66% when using a vector register with 32 elements) while at the same time the *BBE* model provides even better FP performance results (up to 1.4x) compared to *CVU*. As a result, not only a significant reduction of energy over the scalar is achieved (up to 5x), as expected due to the energy efficiency

of vector architectures, but the integrated design also consumes less energy than *CVU* (up to 26% of reduction). Direct forwarding is applied to *BBE* and it provides additional power/energy saving in the vector register file (up to 57%). We propose other three energy-performance efficient techniques that reduce power and increase performance in our integrated design. Generally speaking, our three advanced techniques (chaining from the memory hierarchy, vector memory shape instruction and unified vector load) provide improvements for applications with indexed or strided memory access patterns including 2-D strided access (Sphinx3, Graph500 and Facerec). One of them (chaining from the memory hierarchy) also provides improvements for applications with poor temporal cache locality such as saxpy. Applications in which often the result of one instruction is only consumed by one of the subsequent instructions will benefit from direct forwarding.

Obviously, not all aspects of the integrated design are perfect and there are a few open issues. In our future work, we will work on an evaluation of *BBE*'s control logic at the circuit level to accurately study the complexity/power/performance trade-offs of its design. An implementation at the circuit level will determine if the changes to the hardware affect the critical path and maximum operating frequency. Another concern is that *CVU*, *OBO* and *BBE* are sensitive to the order of the instructions in the program because they are all in-order. This could be partially relieved with decoupling (Espasa and Valero 1996) , but careful power evaluation should be performed to ensure it does not compromise the savings achieved with the proposed architecture.

REFERENCES

Krste Asanovic. May, 1998. *Vector Microprocessors.* Ph.D. Dissertation. University of California, Berkeley.

Christopher Francis Batten. 2010. *Simplified vector-thread architectures for flexible and efficient data-parallel accelerators.* Ph.D. Dissertation. Cambridge, MA, USA. Advisor(s) Asanovic, Krste.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7.

Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVP). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 141–150.

NVIDIA Compute. 2010. PTX: Parallel Thread Execution ISA Version 2.3. 1 (2010).

Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaacspecialis Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. 2002. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02).* 281–292.

Roger Espasa and Mateo Valero. 1996. Decoupled vector architectures. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 281–290.

Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. 2008. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. White Paper. (2008).

Joseph James Gebis. 2008. *Low-complexity vector microprocessor extension.* Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Patterson, David A.

Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA).* 503–514.

Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking simd shackles: Liberating accelerators by exposing flexible microarchitectural mechanisms. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT).*

Michael Gschwind. 2006. Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers.* ACM, 1–8.

Michael Gschwind. 2016. Workload acceleration with the IBM POWER vector-scalar architecture. *IBM Journal of Research and Development* 60, 2-3 (2016), 14–1.

Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. 2006. Synergistic processing in Cell's multicore architecture. *IEEE micro* 26, 2 (2006), 10–24.

Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO). 12–23.

Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. 2012. Vector Extensions for Decision Support DBMS Acceleration. In *Proceedings of the 45th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 166–176.

John L. Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach* (5th ed.). Elsevier.

Zhigang Hu and Margaret Martonosi. 2000. Reducing register file power consumption by exploiting value lifetime. In *Proceedings of WCED in conjunction with ISCA*, Vol. 27.

Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. 2001. Imagine: Media Processing with Streams. *IEEE Micro* 21, 2 (2001), 35–46.

Christos Kozyrakis and David Patterson. 2003. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. 399–409.

Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. 2004. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 52–64.

Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. 129–140.

Christophe Lemuet, Jack Sampson, Jean-Francois Collard, and Norm Jouppi. 2006. The potential energy efficiency of vector acceleration. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC)*. Article 77.

Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara. 2014. The brand-new vector supercomputer, SX-ACE. In *Proceedings of the 2014 International Supercomputing Conference (ISC)*. Springer, 199–214.

Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray UserâĂŹs Group (CUG)* (2010).

Dmitry Ponomarev, Gurhan Kucuk, Oguz Ergin, and Kanad Ghose. 2003. Reducing datapath energy through the isolation of short-lived operands. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 258–268.

Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. 1999. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing (ICS)*. 1–10.

Richard M. Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21 (Jan. 1978), 63–72.

Willi Schönauer. 1987. *Scientific computing on vector computers*. Elsevier Science Inc.

David Seal. 2000. *ARM Architecture Reference Manual* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. 2009. Larrabee: A Many-Core x86 Architecture for Visual Computing. *IEEE Micro* 29, 1 (2009), 10–21.

Avinash Sodani. 2015. Knights landing: 2nd generation intel "xeon phi" processor. In *Proceedings of Hot Chips: A Symposium on High Performance Chips*.

Mostafa I. Soliman. 2011. LcVc: Low-complexity vector-core for executing scalar/vector instructions. In *Computer Engineering Conference (ICENCO), 2011 Seventh International*. 19–24.

Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. 2014. Architectural specialization for inter-iteration loop dependence patterns. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 583–595.

Milan Stanic, Oscar Palomar, Ivan Ratkovic, Milovan Duric, Ozan Unsal, Adrian Cristal, and MR Valero. 2014. Evaluation of vectorization potential of Graph500 on Intel's Xeon Phi. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 47–54.

Shreekant Thakkar and Tom Huff. 1999. Internet Streaming SIMD Extensions. *Computer* 32 (December 1999), 26–34. Issue 12.

Alan R. Weiss. 1999. Benchmarking, Selection and Debugging of Microcontrollers. In *Proceedings of the 1999 IEEE International Conference on Computer Design (ICCD '99)*.