Packing and Fusing Narrow-Width Vector Operations for Energy-Efficient SIMD

Miguel Pinho
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa

***Abstract*—Application developers usually decide on the size of each variable data type by either considering its maximum range or simply by comfortably using larger data types. Since these represent maximum values (and not typical), the applications most often do not make full use of the bit-width offered by the processor arithmetic units. This wasted bit-width is especially relevant when using Single Instruction Multiple Data (SIMD) instructions, since the inefficient use of each arithmetic unit is multiplied by the number of vector elements. This (rather frequent) circumstance is exploited by proposing new run-time mechanisms to (i) efficiently handle narrow vector elements, by removing excess sign bits and packing these elements in a smaller vector, and (ii) agglomerate (fuse) multiple vector instructions pending in the execution queue of the processor, to simultaneously execute them on a single SIMD unit. When combined with clock and power gating techniques, the proposed approach provides a very significant reduction of energy consumption in the SIMD units, by dynamically optimizing the execution of narrow-width integer vector computations, with low hardware overhead and no need for any changes in the application executable. Experimental results based on a prototyping implementation supported on an ARM Cortex A76 model show a reduction of the dynamic and leakage energy consumption in the vector units of up to 54.4%, with either a negligible performance reduction or even some slight improvements of the execution time.**

***Index Terms*—Narrow-width, SIMD Units, Clock and Power Gating, Energy Efficiency, General-purpose Processors**

I. Introduction

Single Instruction Multiple Data (SIMD) extensions have become a common architectural feature in General Purpose Processors (GPPs) as they allow for a significant performance increase in regular data-parallel workloads [1], [2]. As vector extensions evolved along the last decades, the vector width has increased to enable higher performance benefits [2], [3]. The increase in vector size is noticeable, for example, in successive Intel x86 architectures, which started at 64-bit (MMX) and scaled to 128-bit (SSE), 256-bit (AVX), and more recently to 512-bit (AVX-512) [2], [4]. ARM SVE extension is already designed to scale up to 2048 bits [3]. Hence, not only has the vector units' width tended to increase but also the number of these units included per core, as is presented in Table I. These additional resources are highly costly in chip area and power consumption [5]. However, energy efficiency has become a central concern in microprocessors architecture design, and is one of most limiting factors in increasing performance.

In this scenario, this paper proposes a new approach to reduce the number of integer SIMD functional units required by a given processor design while maintaining an identical execution throughput. It takes advantage of the fact that most values in a vector operand do not make use of the whole element width that is fixed at compile-time. Hence, vector elements can be narrowed by discarding redundant sign bits, and packed together in smaller width vectors. Furthermore, by packing the operands of narrow vector instructions, a new opportunity is opened up for simultaneously issuing multiple instructions to the same functional unit (fusing), by using the remaining (available) width in the unit. Hence, the unneeded units (or portions) can be either removed from the design, or they can be dynamically turned on and off as required, using clock and power gating techniques [6]–[9].

TABLE I
Number and width of integer SIMD units present in recent General Purpose Processors

Microarchitecture	Domain	Integer SIMD Units
ARM Cortex-A76/A77/A78	Mobile	2 × 128-bit
Apple A12 (Vortex) / A13 (Lightning)	Mobile	3 × 128-bit
ARM Cortex-X1	Mobile	4 × 128-bit
ARM Neoverse N1	Server	2 × 128-bit
AMD Zen/Zen+	Desktop, Server	4 × 128-bit
AMD Zen2	Desktop, Server	4 × 256-bit
Intel Sunny Cove	Desktop, Server	2 × 256-bit, and 1 × 512-bit (Server)

This is particularly advantageous in highly vectorized code regions and kernels, where the processor's front-end is capable of putting enough pressure into the SIMD units to fully utilize the vector processing capabilities of modern processors (see Table I). In these cases, the proposed technique allows reducing the number of active units with no significant performance losses, thus contributing to a reduction of the power consumption. Moreover, it is also advantageous in other code regions, since it amortizes the performance penalty caused by the delay in powering up gated SIMD units, thus facilitating the exploitation of power-gating mechanisms.

Although some previous works have already exploited the use of narrow-width and low-precision integer arithmetic [10]–[14], they focused only on the scalar pipeline. Vector extensions have increased their relevance over the recent years, and while the processors integer scalar unit width has been kept at 64-bit over the last decades, the current design trend is to increase the vector length with each new ISA generation.

However, exploiting narrow-width values in the vector pipeline brings new challenges, as the overhead of detecting the required bit-width for each vector element is multiplied by the number of elements. Moreover, the proposed scheme must

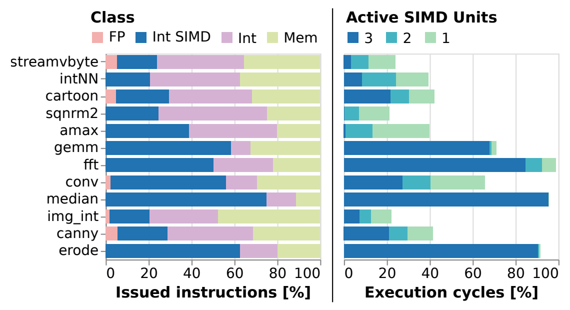

Fig. 1. Profiling and characterization of integer SIMD units' usage in data-parallel benchmarks: percentage of instruction issued by type (left) and number of active SIMD units (right).

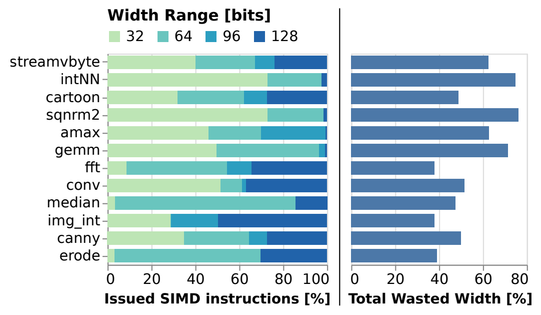

Fig. 2. Profiling of the portion of the vector width that is actually used by the considered benchmarks: width of issued instructions (left) and total fraction of wasted width (right).

be designed to handle the multiple element modes that vector extensions usually support (e.g. 64, 32, or 16-bit elements). In accordance, the main contributions of this paper are:

- A low-overhead mechanism for dynamic detection of the required operands' width for each SIMD lane;
- A new scheme to efficiently pack integer vector operands, by discarding unnecessary sign bits between elements - these packed vectors can execute directly in an available portion of existing units (with minor changes);
- A mechanism to fuse multiple packed instructions for simultaneous execution in a single functional unit;
- A set of architectural modifications to the out-of-order vector execution pipeline of a modern processor to support the proposed packing and fusing mechanisms, as well as to provide the means for a more aggressive application of power and clock gating.

II. Motivation

In data-intensive integer applications, most vector computations make use of only a small fraction of the total element width available on the assigned unit.

To support this statement, we profiled several integer intensive and data-parallel applications from different domains, namely big-data (`streamvbyte`), media (`cartoon`), and machine learning (`integerNN`). This set of applications were also complemented by a broad collection of linear algebra, signal, and image processing kernels (detailed in Section V), which were optimized to take advantage of the SIMD capabilities on modern architectures. The percentage of integer vector instructions issued in these applications and kernels is very high (see Fig. 1), and up to three SIMD units are used at the same time, for a large portion of the execution cycles.

Moreover, as it can be observed in Fig. 2, most of the vector operations in these benchmarks use only a fraction of the total available width (128 bits, in the considered ARMv8 NEON vector extension). Therefore, while several integer SIMD units are kept switched-on at each clock cycle, a large portion of their width is not used by most computations, and represents wasted energy. The unused width of each functional unit, which for the profiled benchmarks is on average 55% of the full width, could be either switched off or reused for other computations. By exploiting this narrow-width opportunity, we aim to more efficiently manage the SIMD execution unit's power-hungry resources, increasing the core's energy efficiency while maintaining the original performance, even on kernels with high SIMD operational intensity.

III. Exploiting narrow width in SIMD operations

Henceforth, the **width of an integer value** shall be defined as the minimum number of bits required to uniquely encode it, while excluding the redundant most significant sign bits (see Fig. 3). Naturally, this compact representation does not result in any precision loss, as the original value can be recovered through sign bit extension.

Decimal	Binary	Width
25	...0000011001 6 5 4 3 2 1	6 bits
-8	...1111111000 4 3 2 1	4 bits
0	...0000000000	1 bit
-1	...1111111111	1 bit

Fig. 3. Narrow-width integer representation.

The main goal of restricting the bit-width of integer values is to reduce the amount of logic that is necessary to perform computations. For example, if the two operands of an integer addition only require half the default width, only half of the functional unit is actually required for the computation. In particular, a typical vector operation consists in computing several sub-operations in parallel, over all the elements of the operand vectors, by following the SIMD execution model. Hence, the useful width of a vector operation may be defined as the sum of the widths of its sub-operations. If each sub-operation is computed over more than one element (e.g. in an addition), its width corresponds to the widest operand value.

A. Detecting narrow-width in SIMD operations

The execution of narrow-width operations is optimized by first detecting the width required by each sub-operation, at run time, i.e. when the actual values are available. Such a procedure is implemented by introducing an additional step in the execution of vector instructions, after their operands are fetched, which is henceforth called by **width encoding** (see also Section IV). To implement this step, the encoded width values are restricted to multiples of a fixed **width-block** (e.g. 4 or 8-bits), rounding values up. This minimum width-block parameter (henceforth denoted by w), which is set at design

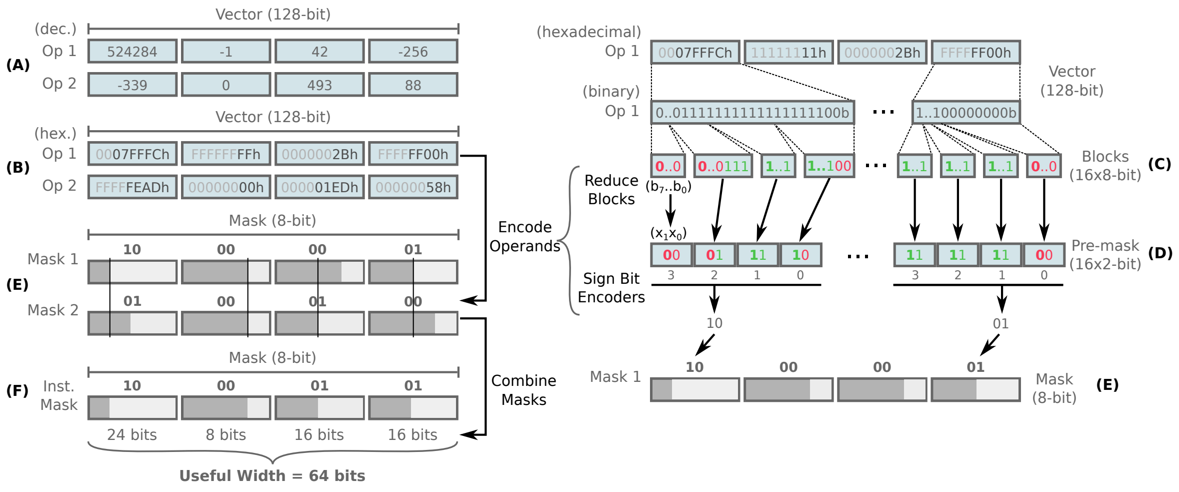

Fig. 4. Width encoding for an example with a 4×32-bit vector operation with two vector operands, considering an 8-bit width-block. In the hexadecimal representation (B), the unneeded bit-blocks are represented in light gray. In the width masks in (E) and (F) the coloring indicates the bit-blocks which are useful in the corresponding operand (light-grey), and those that are wasted (darker grey).

time, allows for a portion of wasted width to be traded for a much simpler implementation.

As an example, consider the two 4×32-bit vector operands in Fig. 4, written both in decimal and hexadecimal formats (see parts (A) and (B)). By considering an 8-bit width-block, each 32-bit lane is represented as four groups of 8 bits (C).

At this step, the width required by each vector element is detected and encoded in a **width mask**. This mask is composed of a group of bits per vector lane, where each bit-set encodes the amount of bits actually used by the corresponding lane, i.e. after removing excess sign bits. To compute this mask, each width-block (composed of w bits, and each bit denoted as b_i) is first reduced to a 2-bit pre-mask (x_1, x_0), where one bit (x_1) corresponds to the block leading bit $(x_1 \leftarrow b_{w-1})$. The other bit (x_0) states whether the remaining bits in the block are equal to that sign bit $(x_0 \leftarrow x_1 \ if \ \forall i < w-1, b_i = b_{w-1})$, or are different $(x_0 \leftarrow \overline{x}_1 \ if \ \exists i < w-1 : b_i \neq b_{w-1})$, as in (D). Then, a priority encoder determines the width-block containing the leading sign bit, outputting the number (in binary) of such width-block, for each group of blocks that corresponds to a lane. Finally, the codes from all lanes are concatenated in a (8-bit) width mask (E).

Hence, this width mask can be computed using an array of leading sign-bit detectors, one for each vector element. Since the widths only have to be determined with the granularity defined by the width-block, only the index of the block with the leading sign bit has to be detected, and not the index of the bit itself. Therefore, the detection can happen after the blocks have been reduced, since 2-bits per block are enough to determine if it contains the leading sign bit, greatly decreasing the number of entries and the complexity of the detectors.

It is worth noting that this reduction step is agnostic to the opted **vector mode** (e.g. 8, 16, 32, or 64-bit) and can be implemented using few logic gates. However, the encoding step depends on the chosen mode, since the size and the number of encoders will depend on it (i.e. on the number and size of the lanes). However, the encoders for different modes can share the same logic, as a larger priority encoder can be implemented using smaller ones.

Finally, an operation width mask can then be obtained by combining the masks of its operands, so that the mask's code in each lane is the maximum code of all sub-operands (F).

Notice that the number of bits per lane of the width mask depends on the considered vector mode, being determined as $\log_2(\frac{m}{w})$, where m is the mode and w is the width-block. For a n-bit vector, the total size of the width mask (all lanes) corresponds to $\frac{n}{m}\log_2(\frac{m}{w})$. The upper-bound for the mask size is $n/(2w)$ when considering an architecture supporting multiple vector modes, but sharing the width bits across all of them, and when n, m, and w are powers of 2. Hence, the overhead of computing and handling this width mask is significantly constrained by the width-block parameter. For example, when a width-block of 8 bits is applied to an architecture with 128-bit vector registers, the width mask of an operation requires 8 bits, corresponding to $\sim$3% of the width of the two operands.

B. Packing narrow vector elements

With the information of the width required by each element in a vector operation already available, a possible optimization is to clock gate the unneeded portions of the functional unit, whenever a vector instruction is issued, to reduce the dynamic power dissipation. To simplify handling vector operations with narrow elements, we propose the **operand packing** mechanism, which consists in compacting the vector elements and discarding unneeded bit-blocks. The elements in a vector are allowed to have different sizes, as encoded by the accompanying width mask, using only the width required by each data element (see Fig. 5). This way, only a contiguous portion of the vector unit width is required, and the remaining can be clock gated using simple control logic.

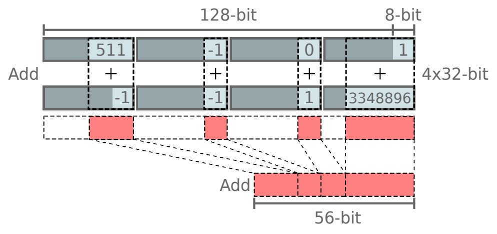

Fig. 5. Packing vector operands in a SIMD Add instruction. By using a width-block of 8 bits, this 128-bit operation can be compressed to use only 56 bits of the SIMD unit.

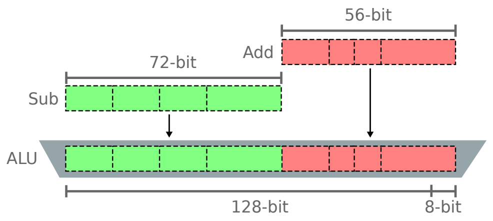

Fig. 6. Fusing two similar vector operations (an addition and a subtraction) that fit in the 128-bit unit.

For instructions with more than one operand, the lanes of the packed operands must be properly aligned, reserving enough space for the widest sub-operand, so that the existing vector units can be directly used to execute the packed instructions. Hence, supporting the execution of packed instructions is a matter of allowing the boundaries of partial operations (e.g. the carry-chains in additions) to be changed with a finer degree of control. Particularly, since conventional SIMD units already support several modes, most of logic is already present.

Considering, as an example, an addition operation, the same adder can be shared for different vector modes (e.g. 32-bit or 16-bit) by inhibiting the carry-chain propagation between carry-lookahead adder blocks at the corresponding element boundaries, as in the 128-bit multi-mode adder from [15]. The carry kill signals in this design are generated from the vector mode, but this can be extended to take into account the width mask, setting the boundaries according to each elements' width. For the multiply (and accumulate) operation, [15] and [16] proposed designs that share the same multiplier circuit for different vector modes, by selecting the appropriate partial products to be added in each case, and discarding products that cross the selected boundaries. Such designs can be easily extended to support irregular element sizes, by adding a finer degree of control over the selection of partial products, according to the vector mode and the width mask.

For correctness, the execution of packed instructions must support the overflow detection in sub-operations with a lower width than the original element size, as this is not an actual overflow. For arithmetic operations, whenever the carry/overflow bit is generated, the corresponding encoding in the width mask must be increased, and the resulting vector expanded to fit the wider resulting elements. For the multiplication, the bits corresponding to the higher half of each element are already generated in the reduction of partial products, and, whenever an overflow is detected, they must be multiplexed to the result vector. Something similar is already provided in current vector extensions for instructions that allow the higher portion of the multiplication to be kept (by increasing the element size of the result).

C. *Fusing vector operations*

The main goal of packing vector instructions is to reduce the consumption of the execution unit, by switching off unneeded portions of the vector unit. However, if several (independent) packed instructions are ready for execution in the processor's issue queues at the same time, whenever two or more of these instructions fit in the full vector width, they can be simultaneously issued to the same arithmetic unit. This mechanism will henceforth be denoted as **operation fusing**. While these instructions share the execution stage, they continue to be regarded as independent instructions, writing to different registers and committing (or even being squashed) independently.

Hence, if a significant number of instructions can be fused for execution in the same unit, the execution throughput can be sustained using fewer SIMD units. The remaining units can be completely clock gated, or, whenever they are not needed for long execution intervals, they can even be power gated or removed from the microprocessor's design. Power gating also reduces leakage dissipation but has the disadvantage of introducing a high state transition overhead and only provides relevant energy savings when it is not interrupted frequently.

One possible implementation of this fusing mechanism is to only allow instructions with the same arithmetic operation to be executed together, which does not add any extra complexity to the existing design and can be seen as an extended type of auto-vectorization. However, it can still be advantageous to allow different (but similar) operation types to be fused (see Fig. 6), in order to increase the number of opportunities for multiple issuing, even if at the cost of a slight increase in the functional unit complexity.

IV. ARCHITECTURAL CHANGES TO EXPLOIT NARROW-WIDTH VECTOR OPERATIONS

Implementing the optimization mechanisms proposed in the previous section only requires some minor architectural changes in the processor execution engine. In the particular case of an out-of-order core, it only requires some adaptations to the vector pipelines and, in particular, in their issue queues, issue schedulers, and SIMD functional units (see Fig. 7).

Fig. 8 depicts the proposed vector execution engine. First, each vector operand is width-encoded, while it is fetched to the issue queue (A), and the result is used to update the width mask for that instruction. The width mask for each vector instruction can be stored using a single additional field in the issue queue (B). As an example, with an 8-bit width-block and 128-bit vectors, this field only has 8 bits, which is minimal when compared to the size of the vector operands that are stored in the queue. Hence, adding support for width encoding is simply a matter of extending the operand fetch mechanism (part (A) in Fig. 8), whether the values are fetched from the

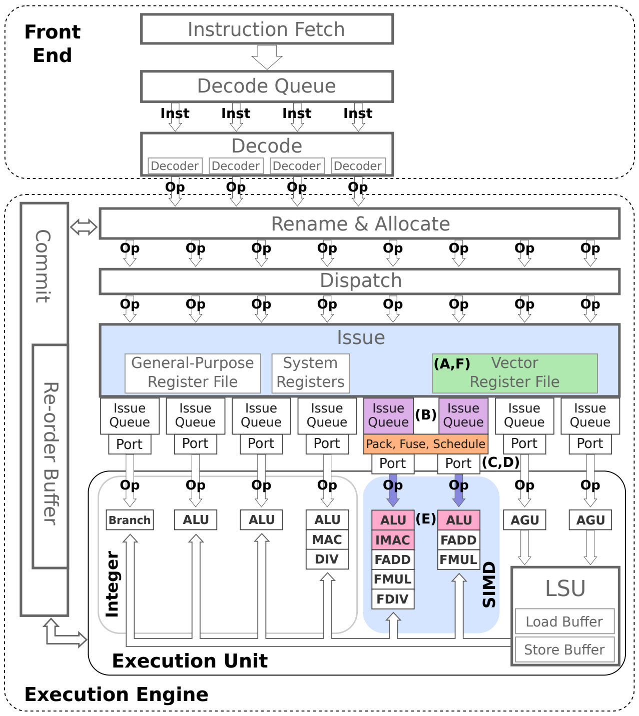

Fig. 7. Overview of the introduced modifications to an out-of-order core (the changes are highlighted in color). Only the vector pipelines need to be changed, by adding an extra stage to implement the new mechanisms - example based on the microarchitecture of the ARM Cortex-A76 processor[2].

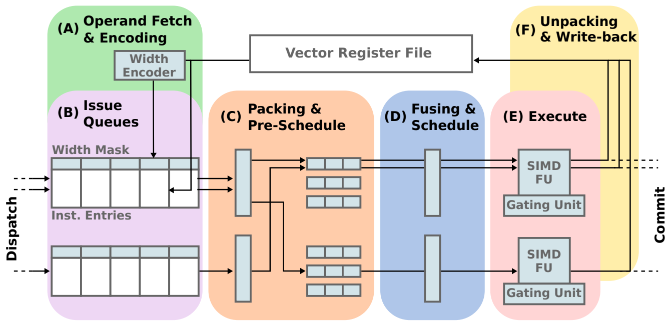

Fig. 8. Detailed changes to the execution pipeline of vector instructions, in an out-of-order execution engine.

physical register file or directly from the common data bus. This should not cause a significant increase in the critical path, as the encoding process is very simple (recall Fig. 4), and can be performed transparently when writing back the previous (dependent) instructions on the common data bus.

The vector operands can only be packed when the width mask for that instruction has been fully updated, i.e. when all the operands have been fetched (and the considered instruction is ready to issue). To avoid any penalization in terms of critical path, with a consequent degradation in the processor performance, a new pipeline stage is added to the vector unit, between the issuing and execution stages (C). In this new stage, the operands for ready instructions are packed, and the instructions are reorganized in buffers, according to their width (and instruction type), so that they can be easily fused and issued. This consequent increase of the vector pipeline latency (by one cycle) is mostly hidden by the out-of-order execution, and the resulting performance penalty is not significant (sometimes it is even mitigated by the fusing mechanism) and is largely outweighed by the consequent energy savings.

Operation fusing is accomplished by extending the issue queue scheduler, by monitoring the width masks of ready instructions, and by trying to issue multiple instructions at the same time, whenever possible (D). The vector operands of these additional instructions are then shifted (or multiplexed), so that there are no overlaps in the functional unit, and the instructions are issued using the remaining bandwidth in the issue port. The width masks of the fused instructions are used to encode the boundaries between elements (in the same instruction or between fused ones) for packed execution in the same unit (E), which is extended to support irregular element sizes. After the instruction is executed, in the write-back stage (F), the resulting vector is unpacked using the information given by the width mask.

Since each individual execution port usually has its independent issue queue, some fusing opportunities might be lost because those instructions are in different pipelines. To circumvent this problem, the additional packing and pre-issue stage is also used to exchange instructions between pipelines, when organizing them in buffers according to their width (C), and thus allowing more pairs of compatible instructions to be found. An essential remark in this modified issuing stage is that some degree of priority should be given to the execution of instructions that were fetched earlier, even if this means missing some opportunities to efficiently compact later instructions. Otherwise, an instruction that does not have a fusing candidate might not find an opportunity for execution, stalling the pipeline.

The clock or power gating mechanisms of the processor are triggered by the gating control units, which are added to each vector functional unit (E). Each control unit monitors the activity of its vector unit and **clock gates** the entire unit (or portions of it) when it is idle. These control units also decide when to trigger **power gating**, whenever the unit is expected to remain idle for a long period of time (for example, by using one of the decision mechanisms designed in [7]–[9]). While a vector unit is in the deep-sleep mode (i.e. power gated), the instructions in its issue queue are handled by the remaining units, thanks to the new shared stage in the issuing step, so that its reactivation is postponed until a significant increase in throughput is required.

V. Evaluation Methodology

The impact of the proposed architectural changes was evaluated using a modified version of the gem5 architectural simulator [17], with two different out-of-order CPU models, whose baseline parameters are presented in Tab. II: a model based on the ARM **Cortex-A76** 4-wide OoO core; and a **High-performance** 8-wide OoO core model, whose front-end and

[2]https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76

TABLE II
BASELINE PARAMETERS OF THE CONSIDERED CPU MODELS

	Cortex-A76	High-performance
Frequency	2.0 GHz	
Fetch Width	4 insts/cycle	8 insts/cycle
Dispatch/Issue Width	8 insts/cycle	12 insts/cycle
Issue Queue	120 entries	180 entries
Load Queue	68 entries	68 entries
Store Queue	72 entries	72 entries
ROB	128 entries	192 entries
Integer Reg. File	256 registers	384 registers
FP/SIMD Reg. File	256 registers	384 registers
Functional Units	3 Int ALUs (1 cycle) 1 Int Mul/Div (2/12 cycles) 2 FP/SIMD units: - SIMD ALU (2 cycles) - SIMD Mul (4 cycles) - FP ALU (2 cycles) - FP Mult (3 cycles)	6 Int ALUs (1 cycle) 2 Int Mul/Div (4/8 cycles) 3 FP/SIMD units: - SIMD ALU (2 cycles) - SIMD Mul (3 cycles) - FP ALU (3 cycles) - FP Mult (4 cycles)
Private L1 ICache	64KB / 4-way (8 MSHRs) 1-cycle latency	
Private L1 DCache	64KB / 4-way (20 MSHRs) 2-cycle latency	
Shared L2 Cache	256KB / 8-way (46 MSHRs) 9-cycle latency	

TABLE III
BENCHMARKED ALGEBRA, SIGNAL, AND IMAGE PROCESSING KERNELS

Kernel	Description	Kernel	Description
`sqnrm2`	Squared vector l2 norm	`median`	2-D Median filter
`amax`	Vector absolute max	`img_int`	Integral image
`gemm`	General matrix multiplication	`canny`	Canny edge detector
`fft`	Fast fourier transform	`erode`	Image erosion
`conv`	2-D Convolution		

backend parameters were scaled up from the Cortex-A76 base model, and its functional unit pool is based on the Apple Vortex 8-wide core.

The simulations were performed in gem5's full-system emulation mode, using a Linux Ubuntu 14.04 LTS distribution. The micro-architecture prototyping was performed by considering the ARM NEON Advanced SIMD architecture extension from ARMv8 64-bits Instruction Set Architecture (ISA). This particular extension was chosen due to the availability of accurate CPU models (e.g. in gem5) supporting it and its relative maturity, as there are several toolchains and libraries available for it. ARM NEON has 128-bit vector registers and supports 64-bit, 32-bit, 16-bit, and 8-bit element modes.

To evaluate the impact of the proposed mechanisms in the processors' energy consumption and execution time, the A76 and the high-performance cores were simulated without any of proposed mechanisms (`Original`), only with operand packing enabled (`Packing`), and with packing and operation fusing enabled (`Fusing`). The proposed packing and fusing modes were also evaluated with one additional clock cycle latency in the SIMD pipeline (`PackingExtraStage` and `FusingExtraStage`, respectively), to model the impact of the extra pipeline stage that was suggested for the implementation of these mechanisms. Each core model and proposed mechanism was evaluated using a different number of integer SIMD units, to measure the impact of power gating or removing units. The different configurations are labeled as $\texttt{x-y-z}$, where $\texttt{x} \in \{\texttt{A76}, \texttt{HP}\}$ denotes the core, $\texttt{y}$ is the mechanism mode, and $\texttt{z} \in \{\texttt{1FU}, \texttt{2FU}, \texttt{3FU}\}$ is the number of considered units. The `A76-Original-2FU` and the `HP-Original-3FU` labels denote the baseline configurations for each core. Unless stated otherwise, the width-block parameter for all configuration is 8 bits.

The set of benchmarks that was considered for the experimental evaluation was divided into two groups:

- a set of computationally intensive and vectorizable **kernels** from the linear algebra, signal, and image processing domains (Table III), by relying on the implementations provided by the Eigen, Arm Compute, and Ne10 libraries, and using randomly generated data as input;
- a collection of **mini-applications**, which exhibit data-level parallelism and are optimized for ARM NEON.

The considered mini-applications are the following:

- `IntegerNeuralNetwork`: An implementation of a feed-forward neural network with one hidden layer using integer weights and inputs[3], modified to use the Eigen 3 library to vectorize the linear algebra operations;
- `StreamVByte`: A fast byte-oriented compression library, optimized using integer SIMD instructions, used in database applications (UpscaleDB) and in information retrieval systems (RediSearch and Trinity) [18].
- `Cartoon`: An application to cartoonify images, by combining a gaussian convolution and canny edge detector kernels, whose implementation was obtained from the Arm Compute library.

These benchmarks were compiled using gcc 7.4.0. The results presented in Section VI correspond to the relevant Region of Interest (ROI) of each application, not considering the code regions where parameters and inputs are read and the resulting output is written.

The leakage and dynamic power consumption for the vector execution unit were estimated using the McPAT modeling framework [19], by considering a 28 nm technology process and an operating temperature of 340 K.

VI. EXPERIMENTAL RESULTS

Figure 9 presents the obtained energy saving results when evaluating the proposed packing and fusing mechanisms with the considered set of benchmarks. The estimated energy consumption is broken down in terms of its dynamic and leakage parcels. They also depict the execution time for each benchmark and configuration, normalized against the baseline configuration for each core model, namely `A76-Original-2FU` for the Cortex-A76 and `HP-Original-3FU` for the `HP` core.

As it can be observed, when maintaining the original number of functional units, the packing mechanism reduces the total energy consumption in the SIMD unit by 22.4% and 21.8% (on average), for the `A76` and `HP` cores, respectively, and by 24.5% and 23.8% when the fusing mechanism is added. As the number of switched-on units is maintained, these savings are mostly provided by a reduction in dynamic power, as idle and unused portions of the functional units are

[3]https://github.com/spolsley/integerNeuralNetwork

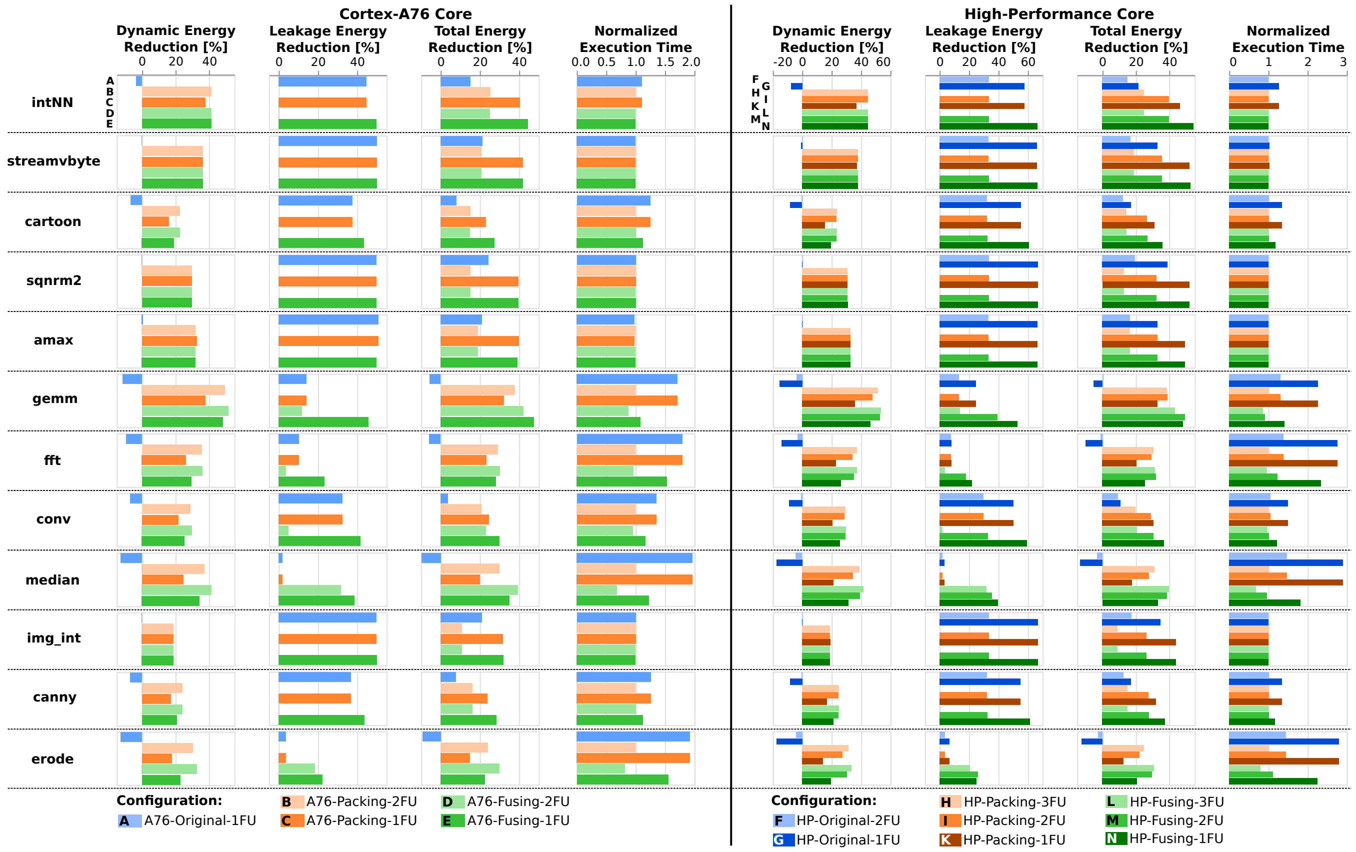

Fig. 9. Energy reduction (broken down in the dynamic and leakage parcels) in the SIMD unit and normalized execution time for each benchmark and configuration. The baseline configurations for each core (i.e. `A76-Original-2FU` and `HP-Original-3FU`) are used as reference and are not included.

clock gated. When both mechanisms are enabled, the average dynamic energy reduction is 33.6% and 34.8%.

An interesting aspect is also observed for the three benchmarks which use the SIMD unit more intensively (`gemm`, `median`, and `erode`). Since they are bounded by its throughput, the fusing mechanism allows for an average speed-up of 26.9% and 29.0%, for the `A76` and `HP` cores, respectively, by executing additional narrow instructions in the same units. This increase in throughput (using the same number of units), with a consequent reduction of the execution time, also contributes to a leakage energy decrease, resulting in a total reduction of 37.4% and 38.1% for the `A76` and `HP` cores.

The leakage power in the SIMD unit can be further reduced by switching off functional units. In fact, by enabling the operation fusing mechanism, more units can be power gated (or removed), while maintaining a sufficient execution throughput. This is particularly interesting both to limit the number of units in the design (thus saving area), and to power off some units on a non-SIMD application phase (where most SIMD units can be turned off). In the later case, there is a costly delay when units need to be powered-on again, which is compensated by the proposed mechanisms. Moreover, the need for reactivating units can be even postponed. As an example, for the most intensive benchmarks (i.e. `gemm`, `median`, and `erode`), when one of the three SIMD units in the `HP` core is removed, the total energy consumption is reduced by 39.7% with an average speed-up of 1%. Without the fusing mechanism, removing one unit leads to a slow-down of 41%. For the remaining benchmarks, two units can be removed with an average slow-down of only 16% (energy reduction of 43.7%), when it would be 29% without fusing.

As it can be observed in Figure 10, in most benchmarks the fusing mechanism allows for a very significant fraction of the vector instructions to be issued and executed together with another instruction. In particular, the fusing effectiveness is very high for the `integerNN`, `gemm`, and `median` benchmarks, where (on average) 76% of the vector instructions are fused.

The average percentage of fused instructions is significantly higher in the `HP` core (42.8% compared to 29.9% in the `A76`, using the baseline number of SIMD units). This increased effectiveness of the fusing mechanism in the wider model can be explained by the larger instruction window used by this core, which allows more vector instructions to be waiting for execution in the same cycle, making fusing opportunities more likely. In fact, there is also a slight average increase in the fusing percentage when the number of units is reduced: from 29.9% to 31.1% in `A76`, and 42.8% to 44.2% in `HP`. A likely explanation is that the reduction in the execution throughput causes the issue queues to become fuller, so there is a higher chance that compatible instructions are waiting for execution

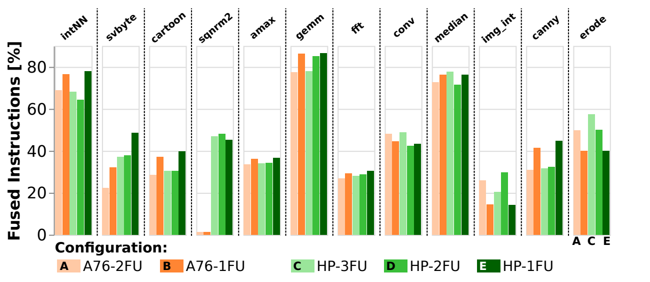

Fig. 10. Percentage of issued instructions that are fused, for the Fusing configurations

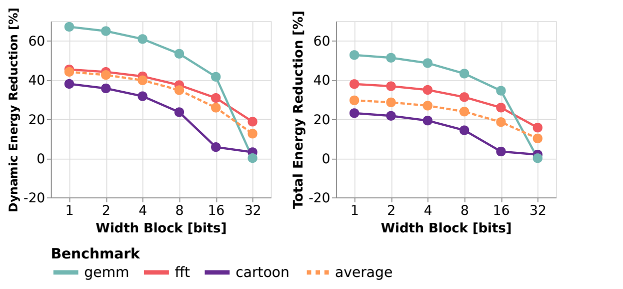

Fig. 12. Comparison of the dynamic and total energy savings with different width-block sizes, for the HP-Fusing-3FU configuration

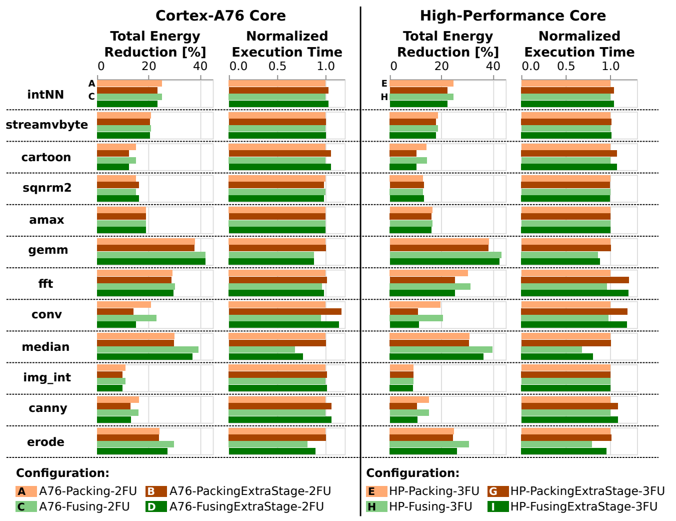

Fig. 11. Impact of the additional pipeline stage in the vector unit in terms of the energy reduction and execution time, for the packing and fusing modes

in the same cycle.

Figure 11 shows the impact of adding an additional pipeline stage in the vector unit, to account for the additional steps in the packing and fusing mechanisms. As it can be observed, the energy savings are mostly unchanged, while the slow-down is not significant for most benchmarks, except for conv (in both cores) and for fft (in HP), where there is (at most) a 21% slow-down. In fact, the fusing mechanism tends to minimize the impact of the additional latency, by allowing an increase in throughput.

Table IV reports the best total energy reduction results obtained when varying the number of available units, for each operation mode and benchmark in the HP core. To limit the the degradation in the execution time, we only considered configurations with a slow-down lower than or equal to 10%, unless none of the configurations satisfies this requisite (in which case the lowest execution time is reported). As it can be seen, the operation packing mechanism allows for significant energy savings in all benchmarks, with an average reduction of 37.8%. The operation fusing mechanism enables an higher energy reduction of 41.6% (on average). In the particular examples of the integerNN and gemm benchmarks, this is achieved by allowing an additional unit to be power gated (or removed) without any performance penalty, and even a slight speed-up in gemm. The execution time was also significantly reduced in median and erode (with speed-ups of 46.8% and 25.9%, respectively).

The presented results also demonstrate that the impact of the extra latency cycle in the vector pipeline is generally not significant. The average energy savings only slightly decrease from 37.8% to 34.6% and from 41.6% to 38.9%, in the packing and fusing modes, respectively, and there is only a relevant execution time increase in two benchmarks, fft and conv (the average slow-down is only 4%, in the FusingExtraStage configuration).

The proposed architecture also introduced the width-block design parameter, which significantly influences the energy reductions that can be obtained, as presented in Fig. 12. A larger width-block minimizes the overhead of implementing the new mechanisms, but implies that more bits will be wasted in each operation, limiting the resulting energy savings. According to the presented chart, a good compromise for the considered processor architecture (i.e. ARM NEON) is an 8-bit block (which was used in all the previous experiments), not only because it allows for significant energy savings, but also because it corresponds to the smallest element size already supported in this vector extension, which means fewer changes are required in the functional units.

VII. Discussion

In the proposed architecture, the operands are packed before each instruction is executed and the result is unpacked afterward (see Fig. 8). However, an interesting extension to this architecture could directly store the packed results in the register file, together with the corresponding width mask, thus simplifying the write-back stage. This approach would reduce the critical path not only in the operand fetch stage, as the width mask is also recovered from the register file, but also in the new issuing stage, as the operands are already packed. In the limit, only immediate operands and memory values would need to be explicitly encoded and packed, but that would not interfere with the critical path of the execution engine, as it could be done during the instruction decoding or even earlier in the memory controller, respectively. This approach would require significantly more modifications in the execution engine, to extend the packing vector format throughout the whole datapath, but it would also allow to exploit narrow-width in other components of the microprocessor. For example,

TABLE IV
SUMMARY OF THE BEST ENERGY SAVINGS OBTAINED IN THE HP CORE, WHILE NOT ALLOWING A SIGNIFICANT PERFORMANCE DEGRADATION

	Original			**Packing**			**Fusing**			**PackingExtraStage**			**FusingExtraStage**		
	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time
integerNN	2	14.9	1.00	2	39.6	1.00	1	54.4	1.00	2	37.6	1.04	1	53.0	1.05
streamvbyte	1	33.0	1.02	1	51.8	1.02	1	52.4	1.00	1	51.5	1.03	1	52.0	1.01
cartoon	2	12.4	1.02	2	26.7	1.02	2	27.0	1.01	2	23.8	1.09	2	24.1	1.08
sqnrm2	1	38.9	1.00	1	51.8	1.00	1	51.9	1.00	1	52.0	1.00	1	52.0	1.00
amax	1	32.8	1.00	1	49.3	1.00	1	49.3	1.00	1	48.8	1.02	1	48.9	1.02
gemm	3	0.0	1.00	3	38.4	1.00	2	49.3	0.91	3	38.2	1.00	2	48.8	0.93
fft	3	0.0	1.00	3	30.4	1.00	3	31.3	0.96	3	25.2	1.21	3	25.3	1.21
conv	2	9.3	1.05	2	29.0	1.05	2	30.5	1.01	3	10.9	1.19	3	11.3	1.19
median	3	0.0	1.00	3	30.9	1.00	3	39.7	0.68	3	30.8	1.01	2	39.1	0.94
img_int	1	34.7	1.00	1	43.7	1.00	1	43.7	1.00	1	43.7	1.00	1	43.7	1.00
canny	2	12.6	1.02	2	27.6	1.02	2	27.9	1.01	3	10.6	1.08	2	24.6	1.09
erode	3	0.0	1.00	3	24.9	1.00	3	30.7	0.79	3	24.5	1.01	3	26.0	0.96
average	—	17.0	1.01	—	37.8	1.01	—	41.6	0.94	—	34.6	1.05	—	38.9	1.04

unneeded portions of vector registers could be gated for an even higher efficiency.

VIII. Conclusions

This paper showed that there is a highly relevant opportunity to exploit narrow-width vector computations in a wide variety of data-parallel and integer intensive applications. To support this claim, it was observed that although wide data types are assigned at compile-time (e.g. 64 or 32-bit) to many application variables, for a large portion of vector operations, the actual values fit in a narrower width (e.g. 16 or 8-bit).

Following this assessment, two complementary mechanisms to exploit narrow-width in vector operations for energy efficiency were proposed: **operand packing**, and **operation fusing**. These mechanisms allow the usage of the SIMD units to be dynamically optimized to the size of the input data, reducing the dynamic and leakage energy consumption while maintaining the same performance levels.

The conducted experimental evaluation considered a prototyping architecture based on an ARM Cortex-A76 out-of-order core (with the ARM NEON vector extension), and demonstrated that energy savings in the SIMD unit as high as 54.4% can be achieved with the proposed implementation. Moreover, the proposed fusing mechanism allowed up to one or two integer SIMD units to be gated (or removed), with none or negligible performance reductions.

Acknowledgment

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 research and innovation programme under grant agreement No. 826647.

References

[1] M. Hassaballah *et al.*, "A review of SIMD multimedia extensions and their usage in scientific and engineering applications," *Comput. J.*, vol. 51, no. 6, pp. 630–649, 2008.

[2] D. Y. Hong *et al.*, "Exploiting longer SIMD lanes in dynamic binary translation," in *Int. Conf. Parallel Distrib. Syst.*, 2016, pp. 853–860.

[3] N. Stephens *et al.*, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[4] D. Habich *et al.*, "Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms," in *Work. Test. Database Syst.* ACM Press, 2018, pp. 1–6.

[5] R. Kumar *et al.*, "Dynamic selective devectorization for efficient power gating of SIMD units in a HW/SW Co-designed environment," in *Int. Symp. Comput. Archit. High Perform. Comput.* IEEE, 2013, pp. 81–88.

[6] J. Kathuria *et al.*, "A Review of Clock Gating Techniques," *MIT Int. J. Electron. Commun. Eng.*, vol. 1, no. 2, pp. 106–114, 2011.

[7] Z. Hu *et al.*, "Microarchitectural Techniques for Power Gating of Execution Units," in *Int. Symp. Low Power Electron. Des.* ACM Press, 2004, pp. 32–37.

[8] A. Youssef *et al.*, "Dynamic standby prediction for leakage tolerant microprocessor functional units," in *Int. Symp. Microarchitecture*. IEEE, 2006, pp. 371–381.

[9] A. Lungu *et al.*, "Dynamic power gating with quality guarantees," in *Int. Symp. Low Power Electron. Des.*, 2009, pp. 377–382.

[10] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Int. Symp. High-Performance Comput.* IEEE, 1999, pp. 13–22.

[11] O. Ergin *et al.*, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *Int. Symp. Microarchitecture*. IEEE, 2004, pp. 304–315.

[12] G. H. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," in *Int. Symp. Microarchitecture*. IEEE, 2002, pp. 395–405.

[13] O. Rochecouste *et al.*, "A case for a complexity-effective, width-partitioned microarchitecture," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 3, pp. 295–326, 2006.

[14] T. T. Hoang and P. Larsson-Edefors, "Data-width-driven power gating of integer arithmetic circuits," in *IEEE Comput. Soc. Annu. Symp. VLSI*. IEEE, 2012, pp. 237–242.

[15] A. Danysh and D. Tan, "Architecture and implementation of a vector/SIMD multiply-accumulate unit," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 284–293, 2005.

[16] S. Krithivasan and M. J. Schulte, "Multiplier architectures for media processing," *Asilomar Conf. Signals, Syst. Comput.*, vol. 2, pp. 2193–2197, 2003.

[17] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] D. Lemire *et al.*, "STREAM VBYTE: Faster byte-oriented integer compression," *Inf. Process. Lett.*, vol. 130, pp. 1–6, 2018.

[19] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Int. Symp. Microarchitecture*. ACM Press, 2009, pp. 469–480.