Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions

Adrián Barredo*†, Juan M. Cebrian*‡, Miquel Moretó*†, Marc Casas* and Mateo Valero*†
Barcelona Supercomputing Center, Universitat Politècnica de Catalunya†, Universidad de Murcia‡*
firstname.lastname@bsc.es

***Abstract*—Vector processors offer a wide range of unexplored opportunities to improve performance and energy efficiency. However, despite its potential, vector code generation and execution have significant challenges, the most relevant ones being control flow divergence. Most modern processors including SIMD extensions (such as AVX) rely on predication to support divergence control. In predicated codes, performance and energy consumption are usually insensitive to the number of true values in a predicated mask. This implies that the system efficiency becomes sub-optimal as vector length increases.**

In this paper we focus on SIMD extensions and propose a novel approach to improve execution efficiency in predicated SIMD instructions, the Compaction/Restoration (CR) technique. CR delays predicated SIMD instructions with inactive elements and compacts them with instances of the same instruction from different loop iterations to form an equivalent *dense* vector instruction, where, in the best case, all the elements are active. After executing such dense instructions, their results are restored to the original instructions. Our evaluation shows that CR improves performance by up to 25% and reduces dynamic energy consumption by up to 43% on real unmodified applications with predicated execution. Moreover, CR allows executing unmodified legacy code with short vector instructions (AVX-2) on newer architectures with wider vectors (AVX-512), achieving up to 56% performance benefits.

I. Introduction

Since the end of Dennard's scaling and the subsequent stagnation of CPU clock frequency, both computer architects and software developers are forced to exploit parallelism to improve performance. Parallelism can be extracted either at the instruction, data or thread levels. While both Instruction-Level Parallelism (ILP) and Thread-Level Parallelism (TLP) are extensively studied, there are still many unexplored opportunities to achieve significant performance and energy improvements from Data-Level Parallelism (DLP).

DLP can be exposed to the hardware by means of vector computations [3], [12], where a Single Instruction operates over Multiple Data streams (SIMD). Vector machines appeared in the early 1970s and dominated supercomputer designs for two decades [4], [10], [12], [33], [46]. These designs exploited DLP with long vectors of thousands of bits. Such vector designs are less popular nowadays, although NEC's SX-Aurora processor has been recently announced with 16,384-bit vectors [29].

SIMD extensions to scalar Instruction Set Architectures (ISA) appeared in the late 1990s to improve the efficiency of multimedia applications, using short vectors of 128 bits [14], [20]. Such SIMD extensions have become ubiquitous in today's computer architectures [1], [2], [21], [39]. Processors with longer SIMD vector lengths have appeared in the last years, such as the 512-bit SIMD implementations from Intel [21], [37] and Fujitsu [49]. Nowadays, DLP exploitation is not limited to SIMD extensions. GPUs are alternative architecture designs that benefit from DLP with a massive amount of threads executing the same instruction in a lock-step model.

Many applications can potentially benefit from vectorized execution for better performance, higher energy efficiency and greater resource utilization [18]. Ultimately, the effectiveness of a vector architecture depends on its ability to vectorize large quantities of code [34]. However, the code vectorization process incurs in several obstacles to overcome, such as horizontal operations, data structure conversion or divergence control, the most challenging being the latter one [19]. While there are many ways to implement divergence control [35], predicated execution is the most common in current vector architectures. The predicated execution model consists in guarding instructions by predicates instead of branches. The predicates, or mask operands, are used to store the correct combined results back to memory.

However, current SIMD extensions to scalar ISAs execute all elements in a predicated instruction independently of the values stored in the mask operand. As such, the execution time of the predicated instruction just depends on the architecture vector length (VL) and it is independent of the percentage of active elements in the mask register. As a result, current SIMD implementations have VL-time performance, waste a significant portion of energy on unnecessary computations and increase contention in the Vector Functional Unit (VFU). Ideally, the execution time and energy consumption of predicated instructions should be proportional to the mask density (i.e., fraction of true/false values). Such an implementation would have *density-time* performance and energy efficiency.

With the current trend of doubling the register size every four years [18], SIMD implementations with VL-time performance will become extremely energy inefficient when executing predicated instructions. Thus, there is an urgent need towards SIMD implementations with density-time performance for predicated executions.

In this paper we propose a novel hardware mechanism, the Compaction/Restoration (CR) design. CR achieves density-time performance and energy efficiency in SIMD extensions to scalar ISAs without requiring any programmer intervention. CR identifies code sections with SIMD instructions guarded by a mask, which we call *compactable* instructions. Active elements in compactable instructions belonging to different loop iterations are compacted into a single *dense* instruction. Such dense instructions are executed efficiently with density-time performance. Then, their results are restored to the original predicated SIMD instructions.

CR aims at improving energy consumption by requiring less accesses to the VFU than the baseline. Moreover, performance can be improved for applications that suffer from contention in the VFU (e.g. due to partially pipelined instructions, such as divisions or square roots, which block the VFU [13]). Finally, CR improves the performance of unmodified legacy code by dynamically and transparently compacting several vector instructions into a wider register ISA.

Next, we list the main contributions of this paper:

- The CR hardware design to enable density-time performance and energy efficiency for predicated SIMD instructions. CR requires minimal hardware support to compact predicated instructions. A detailed design space exploration is performed to properly size the CR hardware structures.
- An exhaustive evaluation with a full system cycle-accurate simulator considering real applications. Our evaluation shows that CR achieves an average of 11% speedups, while reducing dynamic energy consumption by an average of 16%.
- CR transparently executes unmodified legacy code with 256-bit Advanced Vector Extensions (AVX-2) [20] on a newer architecture with twice longer vectors. By using the 512-bit registers and VFUs in AVX-512 [21], CR achieves an average of 17% speedups on unmodified AVX-2 applications.

II. The Divergence Control Flow Problem in SIMD Extensions

The divergence control problem appears frequently when executing vectorized codes [19]. Previous studies indicate that at least 10% of the most common vectorizable loops have divergence control issues [8]. Different mechanisms to implement divergence control in the context of SIMD instruction sets have been proposed [35]. From all the proposals, predicated execution is considered the most effective and compiler-friendly.

However, predicated execution models still face multiple issues. For example, the latency of vector/SIMD instructions guarded by masks depends on the architectural vector length (VL), not on the number of elements to be executed. Sparse predicated masks are common on modern codes. Measured mask densities[1] are between 18-20% on typical benchmarks [9], [15], [41]. Thus, vector processors and SIMD extensions with a VL-time performance implementation waste a significant portion of energy on unnecessary computations

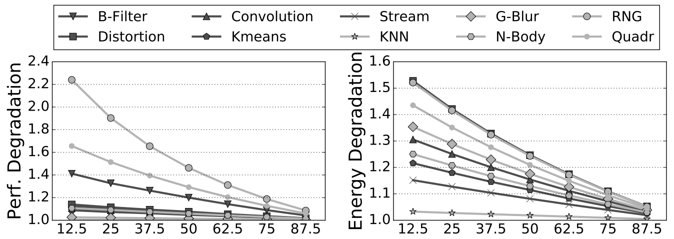


Figure 1. Performance and dynamic energy degradation for predicated SIMD applications with different mask densities.

and increase contention in the VFU, which can hurt performance.

To illustrate the divergence control problem for predicated SIMD instructions, we analyze the performance degradation and energy waste in a set of benchmarks[2] with different mask densities. The selected representative benchmarks contain AVX-512 instructions (VL=512 bits), with a wide range of SIMD instructions types including different percentages of predicated instructions.

For the selected benchmarks, Figure 1 shows the potential performance degradation and dynamic energy waste with several percentages of active elements in the masks. Mask densities range from 12.5% to 87.5% in increments of 12.5%. We use a baseline processor configuration similar to Intel's Knights Landing (KNL) [37]. Performance and energy degradations are estimated with respect to an ideal processor with density-time performance and energy efficiency. Such implementation is ideal in the sense that it provides an upper bound of the potential performance and energy benefits.

Results with several mask densities show no significant difference in time for the evaluated configurations. Indeed, AVX-512 has a *VL-time performance* in the evaluated architecture, which is not optimal. As a result, performance significantly degrades with respect to an ideal density-time SIMD implementation. All benchmarks are sensitive to mask density, with performance degradations ranging from 25% (Convolution) to 2.8× (B-Filter) with 12.5% mask densities.

In the case of dynamic energy, a density-time implementation reduces VFU energy linearly with the mask density. In benchmarks with a high percentage of predicated SIMD instructions such as S-Distortion or B-Filter, this translates into a significant waste in dynamic energy (up to 54% with 12.5% mask densities). On average, dynamic energy waste is 35% with 12.5% densities.

The results shown in Figure 1 make clear that significant fractions of energy and performance are wasted in current SIMD implementations with VL-time performance and energy efficiency. In the next section we introduce CR, a hardware proposal that achieves *density-time* performance and energy efficiency in predicated SIMD instructions without any code transformations.

[1]Percentage of active elements in the mask register.

[2]Section IV describes in detail the experimental methodology.

III. The CR Mechanism

The CR approach aims at achieving *density-time*[3] performance and energy efficiency in predicated instructions in SIMD extensions without user mediation.

A. Overview

CR targets SIMD extensions available in current processors (such as AVX [21]), where vector length (VL) is equal to the VFU width. CR creates a *dense* version of several dynamic predicated SIMD instructions for a certain PC. The active elements[4] of these instructions are selected and *compacted* into a dense instruction. Candidates for compaction delay execution until dense registers are full. In the best scenario, this dense instruction has source registers with all elements active and is executed instead of the original instructions. As a result, contention and the number of accesses to the VFU decreases. This is crucial for performance and energy efficiency, since VFU can add up to 75% of the total power dissipated by the core [36]. Once the dense instruction is executed, results are *restored* back to the original destination registers.

CR can be implemented in any architecture with predication support. Modern SIMD architectures with variable-length vectors, such as RISC-V [45] and Arm *Scalable Vector Extension* (*SVE*) [38] can also benefit from CR. These processors know the register length at run-time and CR needs the same information. In this paper, we have deployed CR in an out-of-order processor with 512-bit VFUs. Section III-B describes the new hardware components to support CR, while Section III-C contains a detailed description of the changes required to an out-of-order pipeline to implement CR. Afterwards, we describe the different phases in the CR mechanism: i) detection of compactable instructions (Section III-D), ii) compaction of dense instructions (Section III-F), iii) execution of dense instructions (Section III-G), and iv) restoration of compacted instructions (Section III-H). Next, we present a case study with CR (Section III-J). Finally, we describe how CR can be used to execute SIMD legacy code on newer and wider SIMD extensions (Section III-K) and discuss other considerations (Section III-L).

1) Basic Functionality Example: CR basic functionality is shown in Figure 2. In this case, two predicated instructions with 50% mask densities, corresponding to two loop iterations for the same PC, are compacted. After compaction, they are executed and their results are restored to the original registers. Figure 3 shows a time diagram of the same example comparing the baseline to CR. In the baseline, the second predicated instruction cannot access the VFU as it is busy executing operations of the same type. In CR, the execution of the first instruction is delayed until the dense registers become full (best case scenario). After compaction, only one instruction is executed reducing VFU contention. Finally, a pipelined restoration phase happens and results are committed.

[3]Results are relative to the mask densities.

[4]Elements whose corresponding mask bits are true.

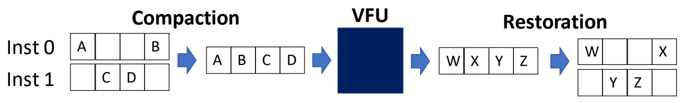
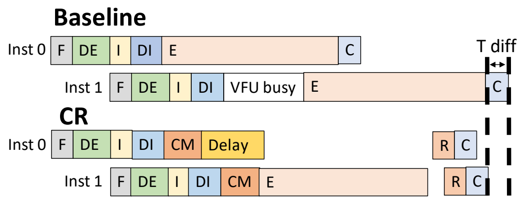


Figure 2. CR basic functionality.



Baseline
Inst 0 F DE I DI E C
T diff
Inst 1 F DE I DI VFU busy E C
CR
Inst 0 F DE I DI CM Delay R C
Inst 1 F DE I DI CM E R C

Figure 3. Baseline/CR time diagram. Fetch (F), Decode (DE), Issue (I), Dispatch (DI), Execute (E), Commit (C), Compact (CM), Restore (R).

B. CR Hardware Components

CR hardware components are described below.

1) The **Compactable Instruction Table** (CIT) is a direct-mapped table which contains the information regarding to dense instructions and their compactable instructions. It is needed to perform the Compaction (Section III-F) and Restoration (Section III-H) phases. Table I defines the functionality and size of every CIT entry. In this case, we target double-precision operations although finer-grained ones could be supported (e.g. machine learning). It would require more bits per entry but the chances of finding a non-true element would be higher, increasing CR efficiency. The number of CIT entries should be smaller than the maximum amount of in-flight instructions. In our design, CIT entries must be filled with at least one compactable. Thus, the maximum number of entries is $ROBEntries/2$ although we did not exceed half of its capacity.

Table I
CIT entry fields, size in bits.

Dense instruction information		
Capacity	Number of elements the dense instruction may handle	4
Alloc Occupancy	Number of elements allocated by compactable instructions	4
Insert Occupancy	Number of elements inserted by compactable instructions	4
Last Insertion	Cycle the latest compacted instruction was inserted	6
isSquash/isTimeout	Whether dense instruction was squashed/timeout triggered	1
$Insert_d$	Whether dense instruction was inserted	1
Compactable Instruction Information		
Mask	Instruction mask bits	8
Dest Reg Idx	ROB entry where instruction is stored	8
Allocate	Whether instruction is allocated	1
$Insert_c$	Whether instruction is inserted	1

2) The **Dense Ticket Table** (DTT) is a direct-mapped table which keeps track of the latest created dense instruction for every PC. It facilitates the accesses to the CIT, since there can be multiple dense instructions for the same PC waiting to be executed. The DTT holds a set of unique keys or *tickets*, representing CIT entry identifiers. Every dense and compactable instruction keeps a ticket to access the CIT. The number of DTT entries is limited by the number of instructions in every loop iteration, a maximum of 60 in our applications. By indexing DTT entries using the 10 lowest PC bits, we avoid conflicts. If no entry exists for a particular PC, a new one is created and a new ticket is chosen from the DTT. If a new dense instruction is created, the existing DTT entry for that PC

gets a new ticket. Tickets are restored as the associated dense instructions commit. The ticket size is limited by the number of in-flight dense instructions (i.e., $\log_2 ROBEntries/2$ bits).

3) The **Compaction Unit** moves active lanes[5] from source vector registers in compactable instructions to the assigned dense registers. It happens separately for every source register as they become ready. It receives a vector register and a mask as inputs and a dense register as output. Section III-F describes the Compaction phase and Section V-A explores its design space.

4) The **Restoration Unit** restores the results of an executed dense instruction back to the original destination registers. The dense destination register elements are moved to the corresponding active lanes of the destination registers. It receives a dense vector register and a mask as inputs and a vector register as output. Section III-H describes the Restoration phase, while Section V-A performs a design space exploration to size it.

C. CR in an Out-of-Order Processor

Next, the main functional changes to incorporate CR into a classic out-of-order processor are described. Figure 4 depicts the whole process in a state-diagram style.

1) **Decode**: In case a predicated instruction is found a signal is sent to Issue stage.

2) **Issue**: If the signal from Decode is active and the mask register is ready, a logic decides whether the instruction has to be compacted or not (see Section III-D). If so, it is marked as compactable. Then, the DTT and CIT are accessed to know if it is the first compactable instruction for that PC or if existing dense instructions for that PC are already fully occupied. In case a new dense instruction is required, a dense instruction is created and its operands are renamed. The DTT creates and stores a new ticket, which is provided to the compactable instruction and employed to create a new CIT entry. In the CIT entry, the *Capacity* field is updated with the total number of lanes in the dense register. A reservation station (RS) and a re-order buffer (ROB) entry are allocated for the dense instruction. Also, a dense destination register is reserved in the Register Alias Table (RAT) to allow operand forwarding. Candidates to be compacted on it are given the DTT ticket after their mask operand becomes ready. Finally, the *Alloc Occupancy*, *Mask*, *Dest Reg Idx* and *Allocate* CIT fields are updated with the compactable instruction information.

3) **Dispatch**: As compactable operands become ready, the compaction occurs independently for every compactable instruction and their RS are freed. The *Insert Occupancy*, *Insert*$_c$ and *Last Insertion* fields in the CIT are updated. Once dense operands are full, a timeout occurs, or a squash happens, the instruction becomes ready to execute. If dense operands are not ready (*Insert*$_d$=false), the instruction will not execute.

4) **Execution**: The dense instruction is executed and compacted instructions are bypassed (Section III-G). If the dense destination register is used by subsequent dense instructions, it is forwarded (Section III-I).

[5]Position in a vector register that contains an element.

5) **Writeback**: The dense instruction is written in the ROB and the restoration is performed to copy the results to the original destination registers (Section III-H).

6) **Commit**: Dense and compacted instructions commit sequentially, ensuring speculation and exception handling are performed in-order.

D. Detecting Compactable Instructions

To have a simple CR implementation, we currently consider all loops as compaction candidates. However, in a preliminary analysis (Section V) and in the evaluation (Section VI) we observe that several factors should be considered to enable an efficient CR mechanism: i) predicated instruction latency, ii) number of instructions per iteration, iii) inter-loop dependencies, iv) mask densities and v) processor events that hide CR latencies.

The first three factors can be statically determined and have important effects on performance. For instance, inter-loop dependencies cause an execution serialization. On the other hand, mask densities are fundamental and input-dependent (see Section V). Finally, some processor events, such as cache misses, pause the core backend hiding CR latencies. A compiler may analyze the first three static factors and produce a hint to enable CR, if the two latter factors happen at runtime, for every loop (e.g. using a memory-mapped register).

Predicated SIMD instructions that fulfill all these factors cause a CIT allocation, becoming *compactable*. CR distinguishes between CIT *allocation* and *insertion*. Allocation is done in program order, while insertion may happen out of order. Allocation reserves the CIT entries which will be later filled in the insertion step. Insertion is performed as compactable instructions become ready. Ensuring program order in insertion is critical to enable *dense register forwarding* (described in Section III-I).

E. Populating Dense Instructions

In order to populate a dense register, compactable instructions delay execution until it is full or a timeout triggers. The ROB is used as a *buffer* to obtain candidates for compaction. Some events, such as cache misses, pause the core backend until they are resolved. For this reason, regular processor behavior may hide the delayed execution and it may not affect performance in many situations (e.g. irregular memory accesses).

F. Compaction Phase

In this phase, active elements from compactable instructions in an RS are moved into the RS belonging to the dense. The CIT is accessed to obtain information about the compaction. It occurs as source operands of compactable instructions become ready, and after CIT insertion is done. The compaction phase does not require extra ports or buffers as the VFU already reads all inputs from the RS simultaneously. When compaction finalizes, compactable instructions are called *compacted*.

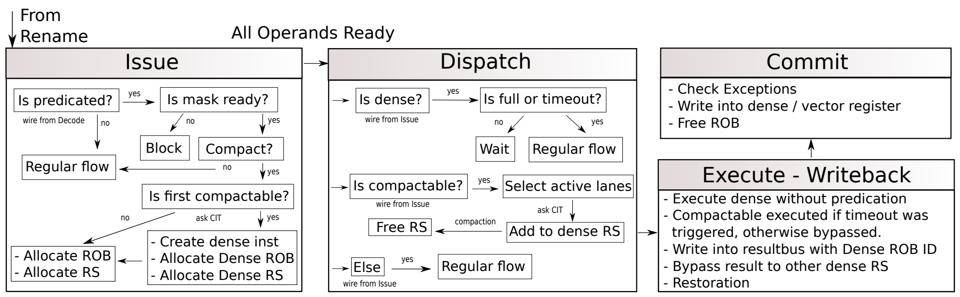


Figure 4. CR overview when incorporated to an out-of-order processor.

G. Execution of Compacted Instructions

Once the dense instruction is ready in the RS, it is executed. Dense instructions can be ready due to three reasons: i) dense operands are completely populated; ii) a squash happens; or iii) a timeout is triggered.

The first case is the ideal scenario for CR, minimizing the number of SIMD ALU accesses as a result. In this case, compacted instructions are not executed (they are bypassed to the next pipeline stages). It also facilitates *dense register forwarding* to dependent instructions.

When a squash happens, CR removes allocated, but not yet inserted, compacted instructions from the CIT entry, forcing the dense to become ready to execute.

Finally, multiple timeout policies are incorporated into CR to avoid delaying too much the execution of predicated SIMD instructions. Postponing the execution of predicated instructions increases the utilization of internal processor resources, potentially stalling the pipeline and slowing down the whole application. For this reason, two timeout policies are created. They stop the allocation/insertion of new CIT entries and trigger the dense instruction execution.

1) **Resource occupancy**. The lack of free hardware resources prevents instructions from entering into the pipeline, and thus, it may not allow dense operands to be completely populated. This situation may lead to performance degradation. For this reason, if resources are occupied above a certain threshold, the CIT forces the execution of dense instructions whose *Last Insertion* field is higher than a timeout. CR considers the occupancy in the reservation station (RS), the ROB, and the Load-Store Queue (LSQ).

2) **Circular dependencies**. A dense instruction could have allocated but not inserted compacted instructions waiting for dependencies to be freed. If the dependency is associated to another dense instruction, execution is blocked. For this reason, if the dense maximum commit time is exceeded execution is forced.

If a timeout is triggered, the remaining allocated but not yet inserted compactable instructions referring to that CIT entry will execute the ordinary way. Section V-B studies the impact of the timeout policies.

H. Restoration Phase

In the Restoration phase, the elements from dense destination registers are moved into the active lanes of the destination vector registers from the original compacted instructions.

```
1   for (i←0; i≤N_ELEMENT; i+=VL)
2      vmovapd  r2, &B[i]
3      vaddpd r1, r2, <imm>
4      vmovapd r3, &C[i]
5      vmovapd r4, &D[i]
6      vcmppd k1, r3, <zero>, <NE>
7      vsqrtpd r5 {k1}, r4
8      vmulpd r5 {k1}, r5, r3
9      vsubpd r1 {k1}, r1, r5
10     vmovapd &A[i], r1
```

Figure 5. SIMD loop in Intel's assembly.

Restoration is performed in the Writeback stage, after the dense instruction is executed and after its result is placed on its ROB entry. It happens in parallel with the *dense register forwarding*. Restoration can be done in parallel for every compacted instruction. The CIT is accessed to get the information of every compacted instruction. The dense instruction keeps the ticket provided in the Compaction phase to know its corresponding CIT entry.

In the Restoration phase, multiple data values must be written to the ROB. This phase is usually out of the critical path of execution, as the dense version of the instruction continues executing. Thus, this phase can be handled by buffering writes to the ROB not requiring extra ports.

I. Dense Register Forwarding

A dense register can be forwarded if it is fully occupied or if the dense instruction and its dependent ones share the same inserted compacted instructions positions. The $Insert_c$ CIT entry bit provides this information for every allocated compactable instruction. If not, the remaining uninserted compactable instructions will be compacted. An efficient dense register forwarding reduces CR latencies and hides the restoration process.

J. CR Case Study

To illustrate how the CR mechanism works, we refer to the code from Figure 5. It is used to describe the different phases in CR: activation, compaction, execution, and restoration. For the sake of simplicity, in this particular example, we assume a 128-bit vector length architecture. Thus, each vector register may hold 2 double precision elements. In this case, a vector multiplication (*vmulpd*, line 8), a subtraction (*vsubpd*, line 9), and a square root (*vsqrtpd*, line 7) represent the 3 predicated instructions in this loop. They are guarded by a mask register *k1* created in line 6. This mask is built by comparing each

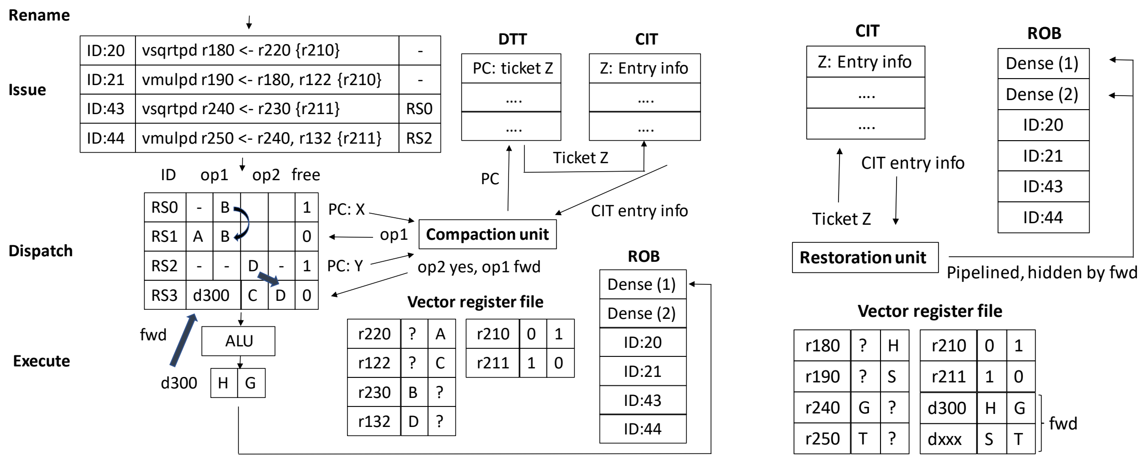


Figure 6. Example of the Compaction (left) and Restoration (right) phases.

element in array *C* to a zero-filled vector. In this case, we assume that the compiler marks this loop as suitable for CR. Figure 6 shows the compaction and restoration processes for the instructions *vsqrtpd* and *vmulpd*.

Activation Phase. In the Issue stage, there are two instances of these instructions (with identifiers 20, 21, 43 and 44). Mask registers *r210* and *r211* are read as they become ready. Since their mask density is low (50%), CR is enabled for this loop. Then, two dense instructions for these PCs are created and the CIT allocation is performed, allocating two CIT entries with *Capacity* 2. The *Alloc Occupancy*, *Mask*, *Dest Reg Idx* and *Allocate* fields are updated for every compactable, since the mask registers for every dynamic instrucion are ready and the Rename stage has been previously accessed. A ROB and an RS entry are allocated for each dense. Two tickets are created and stored in DTT.

Compaction Phase. As operands become ready, the instructions are moved to the Dispatch stage. The CIT insertion is performed, updating the corresponding *Insert Occupancy*, *Insert*$_c$ and *Last Insertion* CIT fields. After that, the compaction for the dense *vsqrtpd* instruction starts. This process is shown in Figure 6. In this case, the active element in register *r220* (A) is moved to the dense RS entry (RS1) using the CIT information. After that, the RS belonging to ID:20 is released. Similarly, in the next loop iteration, CR compacts the active element from register *r230* (B) into RS1. This dense instruction is ready for execution. The same process is done with instruction *vmulpd*, where the second operand is compacted moving the active lane in *r122* (C) and *r132* (D) to the dense instruction in RS3. However, the first operand in the compactable instruction is dependent of *vsqrtpd*, an already compacted one. The CIT notices this situation and skips its compaction, notifying that a dense register forwarding is going to happen. In particular, the register *d300*.

Execution Phase. The dense *vsqrtpd* instruction is executed as compaction is finished and its destination register *d300* is forwarded to the dense *vmulpd*, which will also be executed afterwards.

Restoration Phase. After execution, the restoration phase occurs for the dense instructions *vsqrtpd* and *vmulpd*. A brief overview is depicted in Figure 6. The CIT contains the information regarding every inserted compactable instruction for every dense. In *vsqrtpd*, the restoration unit reads the dense output *d300* and the original mask values from the instructions with ID 20 and 43, inserted in the CIT. Then, the restoration unit moves the *d300* elements to the destination entries in the ROB, performing an offset calculation depending on the mask values and the compacted instruction insertion order. For example, the register *r180* (instruction ID:20) receives the first element from the dense register *d300* (H) and it is placed in the second lane, where the mask register *r210* contains a true element. The register *r240* gets the second element (G), as the accumulated capacity is one, and it is placed in the first lane, specified by mask *r211*. The same process is done with the dense *vmulpd*, moving S and T to the second and first lanes of registers *r190* and *r250* respectively.

K. Optimizing SIMD Legacy Code

CR hardware can also be employed to optimize legacy SIMD codes on modern and wider processors. Many applications make use of hand-coded programs with SIMD intrinsics. Porting such codes to modern SIMD architectures is costly and time consuming. For this reason, many 256-bit or 128-bit SIMD codes are executed on 512-bit VFUs, underutilizing hardware capabilities. The CR mechanism can be employed to dynamically create dense instructions that compact two AVX-2 instructions into a single AVX-512 instruction.

This way, every SIMD instruction is a candidate for compaction as the CR mechanism is not restricted to predicated instructions. In this case, the mask density and the active element positions are known before-hand, as they are defined by the architecture. In such scenario, the compaction/restoration units complexity is reduced, enabling lower CR latencies than in the general CR case, and enhancing the CR mechanism efficiency.

This approach is transparent to the programmer and only requires a compiler to analyze the static factors described in Section III-D to determine if CR could improve performance. Section VI-B evaluates the AVX-2 instruction compaction over AVX-512 instructions using CR.

Table II
CONFIGURATION OF THE GEM5 SIMULATIONS.

Chip details	
Core	1 out-of-order core, single threaded, 2.0GHz
Core details	
Fetch, decode, rename bandwidth	4 insts/cycle
Dispatch, issue, commit bandwidth	4 insts/cycle
Branch Target Buffer	1 way, 2048 entries
Branch predictor, Branch target buffer	Bimode, 8K+8K entries
Fetch Buffer, Decode Buffer	16B, 56-μops
Fetch, Load and Store Queues	32 entries, 90 entries, 72 entries
Physical Registers	200 integer + 360 floating point
Issue Queue, Re-order Buffer	196 entries, 320 entries
Functional Units	1 Int ALU + 3 Int/FP/SIMD ALU
Instruction Latencies (Int)	add (1c.), mul (4c.), div (22c.)
Instruction Latencies (FP)	add (5c.), mul (5c.), div (22c.)
Instruction Latencies (Icelake SIMD)	add (3c.), mul (5c.), div (14c., 8c. issue), sqrt (16c., 10c. issue)
Instruction Latencies (KNL SIMD)	add (6c.), mul (10c.), div (30c., 16c. issue), sqrt (40c., 20c. issue)
L1 instruction cache	32KB, 8-way, 1 cycle access latency
L1 data cache	32KB, 8-way, 4 cycle access latency
L2 unified cache	4MB, 16-way, 12 cycle access latency
CR structures	
Compaction Unit	1 pipelined unit, 2 stages
Restoration Unit	1 pipelined unit, 2 stages
Dense Ticket Table	64 entries, 8 bits per entry
Compactable Instruction Table	160 entries, 170 bits per entry

L. Discussion

The CIT is squashed in the event of a branch miss-prediction. Two scenarios must be considered: a) miss-predicted instructions created an entry within a dense instruction, but operands were not ready and thus, not compacted; and b) operands were ready and compacted. In the first case, the CIT would be waiting forever for this instruction. In the second case, a false version of the dense register would be created, since some lanes belong to miss-predicted instructions operands. The first scenario is handled by making the CIT aware of miss-predictions. The second scenario is not critical because results are written into miss-predicted ROB entries in the Restoration phase, but these results never commit.

Page faults need a special handling as they are attended at commit but a dense instruction may be blocking its attendance. A timeout is required to force the dense execution and of every instruction prior to it.

Precise exceptions are also feasible with CR. If an exception occurs while a dense instruction is executing, such as arithmetic overflow, the exception is *restored* to the corresponding compacted instruction to be handled.

A challenge to be faced in the future is the implementation of dense horizontal instructions. Horizontal instructions, such as *shuffles*, move a value from a particular vector register lane to another one. At the moment a dense register is created, the original element positions are lost so the operation cannot be done.

IV. EXPERIMENTAL METHODOLOGY

A. Full-System Simulation Infrastructure

We employ *gem5* [5] to simulate an x86 full-system environment that models the application, the operating system and the architecture in detail. We simulate a one-core processor using the detailed out-of-order CPU and memory models of gem5, extended with the proposed architectural support for CR. Table II summarizes the main simulation parameters, including the selected size of the CIT and the compaction/restoration hardware configurations. The ticket size and the number of entries in the CIT and in the DTT are defined by the ROB size. We have a CIT design supporting AVX-512 instructions, double precision elements, and up to eight compactable instructions per entry. Thus, each CIT entry requires a total of 170 bits: 26 bits for the dense instruction information; and 8 times 18 bits for the compactable instruction information (Table I lists all the fields in the CIT). Section V performs a detailed design space exploration to justify the compaction/restoration unit configurations. As explained in Section III, the CR mechanism requires accessing to the corresponding hardware structures several times. These latencies are modeled in detail in our simulator.

The simulated system is a 16.04 Ubuntu with a 4.9.4 Linux kernel. The ISA is extended to support Intel's SSE, AVX-2 and AVX-512 instructions. These extensions have been developed to simulate an x86 Icelake processor. Concerning the SIMD units, two micro architectures are modeled: a latency and a throughput-oriented implementation based on the Icelake (ICE) [42] and the Knights Landing (KNL) [37]. They represent two scenarios with different VFU contention and employ pipelined VFUs with different execution and issue latencies as measured on real hardware by A. Fog [13].

Power consumption is evaluated with McPAT [26] using a process technology of 22nm, a voltage of 0.6V and the default clock gating scheme. We incorporate the changes suggested by Xi Vaidya *et al.* [47] to improve the accuracy of the models. The CIT structure is modeled in CACTI 6.5 [28], adding the appropriate counters in gem5 to measure the extra power introduced by it. The CR units have been modeled in RTL [7], [27] with the configurations chosen in Section V-A. Results for a 22nm technology show area requirements of 5000μm^2. It is almost three orders of magnitude smaller than a 512-bit ALU modeled in McPAT (4.45 mm^2). In terms of power, every unit consumes 11.25mW of peak power (combined leakage plus dynamic), almost two orders of magnitude smaller than the power of the 512-bit ALU computed by McPAT (0.92W).

B. Benchmarks

To test CR we use ten real *unmodified* predicated AVX-512 applications. We employ an image filter (B-Filter), a signal convolution (Convol), an image processor (G-Blur), a K-means clustering (Kmeans), k-nearest neighbors (KNN), an N-Body (N-Body) application [11], a quadratic equation (Quadr), a Box-Muller number generator (RNG) [48], a sound distorter (S-Distort) and a distance calculator (Stream) [9].
Section V makes use of a SIMD microbenchmark to explore the design space. This microbenchmark is hand-coded using Intel's AVX-512 intrinsics. The mask density, the percentage of costly instructions and the number of instructions in each loop iteration can be changed.

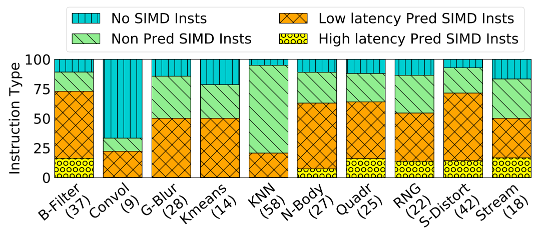


Figure 7. Loop iteration breakdown. In the X axis, the applications name and their number of instructions per iteration. In the Y axis, the instruction types in every iteration.

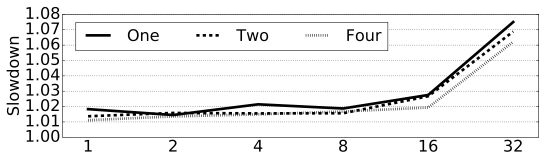


Figure 8. Compaction unit configuration slowdown on performance. Normalized to non-latency CR scenario. In the x-axis the different number of stages. Each line represents a different compaction unit count.

Fung and Vaidya *et al.* studied the mask densities in several applications [15] [41]. They showed they are usually input-dependent and range between 15-60%. Since input selection may strongly impact mask density, we consider values from 25% to 50% for all the codes. These values capture almost entirely the mask density range from the representative applications. Also, we apply static masks (25% and 50%) during the whole simulation as the most relevant previous works on SIMD control flow divergence do [15], [41]. Moreover, Vaidya *et al.* [41] also demonstrate that the true-value position inside the mask register leads to no variability in performance. For the sake of clarity we will omit the combinational possibilities of the true-value positions inside the mask.

Figure 7 shows the instruction breakdown of the main loop in the ROI of each benchmark. Loops contain between 9 and 58 instructions. The predicated instruction percentage is between 21% (KNN) and 72% (B-Filter).

V. Design Space Exploration

Next, a design space exploration is done to size the CR harware and to study the application impact on CR.

A. Compaction and Restoration Latencies

As explained in Section III, CR requires four hardware components. The DTT and CIT sizes are defined by the ROB size. The compaction and restoration units are sized in this section. First, we start with the compaction unit design. Figure 8 shows the performance slowdown obtained when varying its number and operation latency. Performance is normalized to an ideal design with no CR latencies.

Figure 8 depicts the average results for the SIMD micro-benchmark executed with several mask densities. Increasing the number of compaction units from 1 to 4 provides less than 1.3% performance improvements. In contrast, when having more than 8 compaction stages, performance degrades. Thus, we select a compaction configuration with a single unit and two pipeline stages. It provides only a 1.4% performance degradation with respect to an ideal CR mechanism and a simple design.

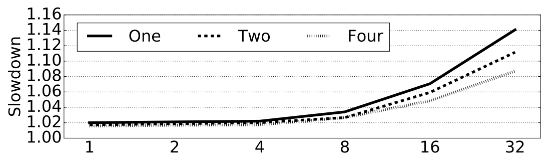


Figure 9. Restoration unit configuration slowdown on performance. One two-stage compaction unit latency considered. Normalized to non-latency CR scenario. In the x-axis the different number of stages. Each line represents a different restoration unit count.

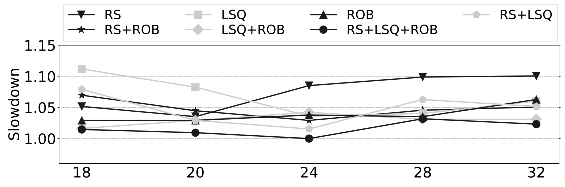


Figure 10. Timeout policy combinations impact on performance, normalized to the best scenario. The circular dependency policy is implicit in every scenario. In the x-axis the number of cycles for each timeout policy changes.

Next, we explore the restoration unit design. Figure 9 shows the performance degradation with different restoration formats. For this experiment, use the selected compaction unit configuration. Results are normalized to an ideal design with no CR latencies. In this case, with 1, 2 and 4 stages, varying the number of units and restoration stages marginally degrades performance (less than 0.5% and 0.2% slowdowns, respectively). However, as 8-stage restoration units are reached, performance degrades drastically. A 14% performance degradation is achieved with 32-stage units, where increasing the unit number from 1 to 4 provides a benefit of up to 6%. As we are interested in reducing energy consumption, the final design is limited to a single two-stage unit. Thus, restoring a dense instruction with four compacted ones takes five cycles. This format combined with the selected compaction unit, has a 1.9% slowdown compared to an ideal scenario.

B. Timeout Policies

Next, we measure the impact of the timeout policies discussed in Section III-G. In this case, the micro-benchmark is used with different timeout policies. Figure 10 depicts the performance degradation obtained by combining the original timeout policies, normalized to the best configuration. The timeout policies consider the occupancy in different resources (RS, LSQ and ROB) and different timeouts (from 18 to 32 cycles). All policies take into account circular dependencies as this is required for the correct execution of the benchmarks.

Selecting the optimal timeout policy is fundamental for CR, preventing the CPU from waiting too much for dense register population. Results show up to a 10% slowdown when only the issue queue is considered. The best outcome is obtained when considering all resources.

C. Costly SIMD Instruction Ratio

Next, we analyze the influence of the predicated instructions latencies to the performance and to the energy reduction. In this case, we are considering the same base micro-benchmark where the ratio between low and high latency instructions increases, from 0% to 100%. All of them have the same memory access pattern and the same number of instructions per

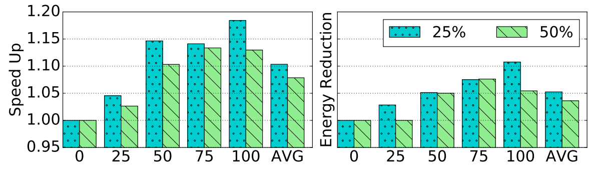


Figure 11. Impact of costly predicated SIMD instructions to performance (left) and dynamic energy (right). Normalized to the no-long latency instruction scenario. In the x-axis, the percentage of costly predicated instructions.

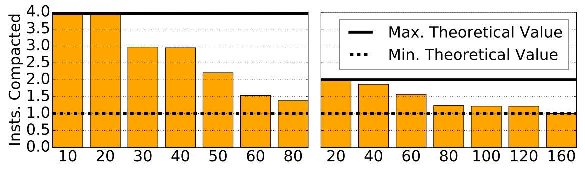


Figure 12. Average number of predicated instructions compacted per dense in CR. In the x-axis, the number of instructions per loop iteration. Masks: 25% (left) and 50% (right).

iteration. The mask density varies between 25-50%. Results are normalized to the 0% long latency instruction scenario.

Figure 11 shows performance and energy results. The higher the costly instruction ratio, the better the speedup and energy reduction. If instructions have a long latency, the dense register population and the CR latencies can be hidden by the execution and even lead to a performance benefit. For instance, in the case of a predicated square root in a 25% mask density scenario, the fact of delaying the execution of instructions from four iterations (50 cycles) and executing only one instruction (20 cycles) would be better than executing four instances of the same dynamic instruction (70 vs 80 cycles). Long latency SIMD instructions also permit higher timeout policy values, allowing more occupied dense registers, reducing the accesses to VFUs, and thus, generating higher dynamic energy benefits.

D. *Effectiveness with Different Loop Lengths*

Finally, we study the sensitivity of the CR mechanism to the loop instruction length. In this case, we consider the same SIMD micro-benchmark as in the previous sections. We use the same mask densities (25%, 50%) and different number of instructions per iteration in a processor with a 320-entry ROB.

Figure 12 shows the average number of predicated instructions compacted per dense instruction. With both mask densities, CR achieves a high number of compacted instructions with loops of 40 or less instructions. An increase in the number of instructions per loop iteration causes a higher ROB occupancy, preventing CR from doing an efficient population of dense registers. For example, moving from 20 to 60 instructions per iteration reduces the average compaction from 4 to 1.5 in a 25% mask density scenario. Also, a higher mask density leads to more pressure on the ROB occupancy as a dense instruction is added more frequently (every 2 compacted instructions with 50% mask density; every 4 instructions with 25%). Consequently, loops with 160 instructions and 50% mask density can not be compacted with CR. In contrast, loops with 80 instructions and 25% mask density can be partially compacted with CR (1.45 instructions are compacted per dense).

VI. Evaluation

This section explains the performance and energy benefits of CR in real applications. We also describe the benefits of using CR to optimize legacy SIMD code.

A. *Predicated SIMD Applications*

The CR proposal is evaluated with ten different applications. As described in Section IV, we explore two different mask densities (25% and 50%) and two processor configurations with different instruction latencies (ICE and KNL). For the CR mechanism, we make use of the configuration determined in Section V.

Figure 13 depicts the results in terms of speedup, VFU access reduction and dynamic energy reductions. Results are normalized to a regular no-CR execution. On average, applications achieve between 3.6% and 10% performance speedups, between 21% and 41% VFU access reductions, and between 6.2% and 13.4% dynamic energy reductions. In all the experiments, the KNL configuration provides more optimization opportunities to the CR mechanism as there is more contention in the VFU. Also, lower mask densities (i.e., 25%) lead to more compaction opportunities.

Significant speedups are obtained for some of the evaluated benchmarks. This is the case of N-Body and RNG, which contain a high percentage of long latency SIMD instructions per loop iteration (as shown in Figure 7). They achieve performance improvements up to 25% and 15%, respectively, and dynamic energy reductions up to 22% and 43%. This reduction in dynamic energy is a result of the significant reduction in VFU accesses (up to 42% and 87%, respectively). In the case of N-Body, the CR phases are hidden by the memory access requests and lead to better performance benefits.

B-Filter and S-Distort also contain long latency SIMD instructions. However, a higher number of instructions per loop iteration prevents an efficient population of dense registers. Only with a VFU contention increase in the KNL configuration, speedups reach a 7%.

The application memory access pattern is important for CR, since it can hide the dense register compaction/restoration. Convol, with an irregular access pattern and low-latency predicated instructions, is able of marginally improving performance and reducing dynamic energy consumption up to 5%. In contrast, Kmeans and KNN have a contiguous memory access pattern and no long latency predicated instructions. Kmeans is capable of reducing VFU accesses up to a 60%. However, the large amount of instructions and the low percentage of predicated instructions in KNN prevent CR from achieving performance benefits. KNN also contains horizontal operations, blocking dense register forwarding.

For all the applications, the long latencies of the KNL configuration enable higher VFU access reductions that lead to better dynamic energy results. In this configuration, there is a higher contention in the VFU than in the ICE one. As a result, a higher occupancy of dense registers is achieved. We have measured the *dense register forwarding*, in particular, at the lane level. If a dense register lane can be forwarded,

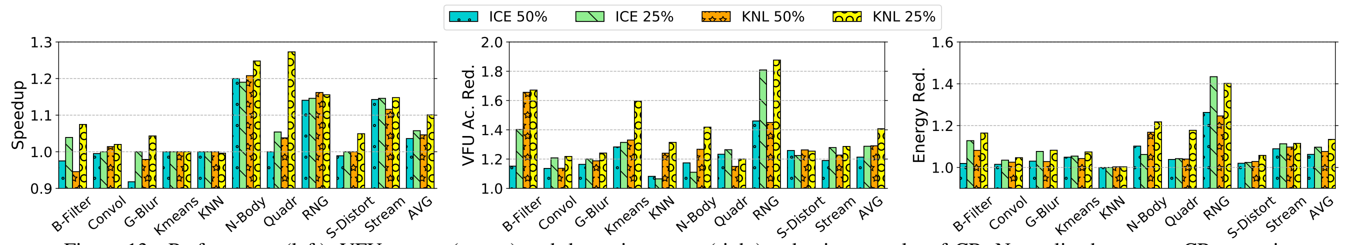


Figure 13. Performance (left), VFU access (center) and dynamic energy (right) reductions results of CR. Normalized to a non-CR scenario.

the compaction phase can be avoided for that lane, reducing latency and energy consumption. For instance, 72% of dense lanes can be forwarded in BF, 65% in S-Distort, 73% in Kmeans, 46% in KNN, 77% in RNG and 46% in G-Blur.

B. Optimizing AVX-2 Legacy Code

The CR mechanism can also be used to optimize SIMD legacy code. Section III-K describes the motivation and the advantages of this approach. In this section, we explain the results of employing CR to compact two AVX-2 instructions into one AVX-512 instruction.

Figure 14 shows the results of CR with real applications compiled with AVX-2 support. Results are normalized to a regular execution without CR. In this case, we limit the original set of evaluated applications to seven, since three of them do not have a memory access behavior or the required percentage of SIMD instructions suitable for CR. A compiler may identify these static application characteristics and notify CR when to compact AVX-2 codes in wider SIMD extensions.

As expected, average results are better than in the scenario with predicated SIMD instructions. In the KNL configuration, speedup and leakage energy reduction reach 17% on average, while dynamic energy reaches 16% reductions. In the ICE configuration, average results are more modest (5% and 12%). Both configurations achieve an average 35% reduction in VFU accesses.

The largest reductions in VFU accesses are achieved with B-Filter and RNG (between 60% and 73%). This translates into significant reductions in dynamic energy. RNG achieves a significant 56% performance improvement. N-body also reduces dynamic energy (between 10% and 12%). In contrast, KNN still suffers from the blocking of dense register forwarding due to horizontal operations and achieves minimal energy savings, even if VFU accesses are reduced by more than 10%.

C. Comparison with Other Proposals

This section compares CR with Disable Inactive Lanes (DIL) [24], an alternative hardware proposal to reduce power consumption in the VFU. DIL reads the mask operands before executing predicated instructions and disables the lanes in the VFU with inactive elements. This solution reduces power consumption at the cost of increasing the complexity of the VFU design. However, DIL does not reduce the contention in the VFU and cannot be used to speedup the execution of AVX-2 legacy codes. Interestingly, CR and DIL can be combined to further reduce the power consumption of CR when a timeout avoids instructions compaction.

The left chart of Figure 15 presents the average speedup of CR, DIL and CR+DIL over a baseline without CR. As expected, DIL and CR+DIL do not improve performance over the baseline and CR, respectively. The right chart of Figure 15 presents the average energy reduction of the three techniques over a baseline without CR. DIL reduces energy between 5% and 8% as it reduces the dynamic power in the VFU. CR achieves higher energy reductions than DIL due to the increased performance in some of the benchmarks. However, in benchmarks in which CR provides no performance benefits (Convol, Kmeans, KNN), DIL achieves up to 18% energy reduction. Thus, CR+DIL provides the best energy results with average energy reductions between 6% and 13%.

VII. Related Work

Sparse to dense transformations have been broadly studied from the software standpoint. Harrison *et al.* and Pichon *et al.* proposed reordering techniques and implemented math libraries to mitigate the sparsity problem [17] [31]. However, CR is a hardware mechanism to improve execution efficiency in the context of sparse predicated SIMD instructions.

Smith *et al.* [35] explored several alternatives to implement conditional operations in vector ISAs. One of the proposals, *Register compress/expand* (RCE), is similar to CR. It compresses the active elements of a long vector into a dense one using new instructions, such as *IOTA*. It is supported in multiple vector supercomputers [10], [33], [40], [44]. RCE can be applied to traditional vector architectures that have vectors with hundreds/thousands of elements and a vector unit that processes a few of them per cycle. Current SIMD extensions contain vector registers and VFUs with the same width (i.e. 512-bit width in AVX-512). In these scenarios, RCE by compressing the register to a vector of N bits (N<512), would not provide performance benefits as it would take the same time to execute (the VFU would still process 512 bits instead of N). RCE does not combine vector instructions from different iterations and thus, it cannot improve performance in SIMD scenarios. CR covers this gap.

Some divergence control proposals assume an architecture with several *scalar* datapaths [23], [25], [43]. These designs can dynamically manipulate the VL of each datapath and execute them optimally, but the ALU design (64-bit) is different from the VFUs (512/1024-bit) in current SIMD extensions where CR may be applied.

Vaidya *et al.* [41] propose two micro-architectural techniques to improve the performance of predicated instructions in GPUs. They rely on the fact that the VL is usually multiple

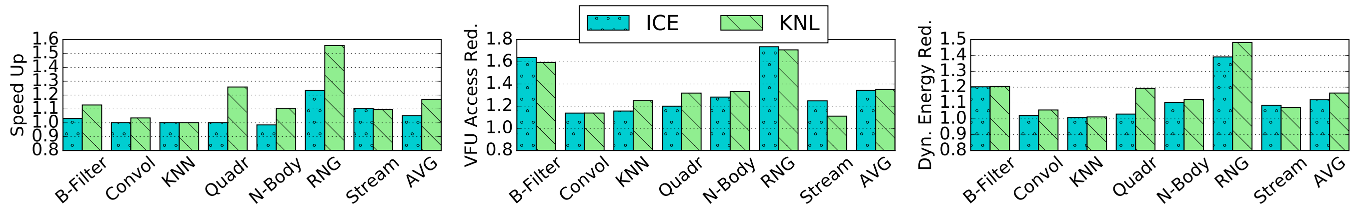


Figure 14. Results of AVX-2 legacy codes compacted into AVX-512 using CR. Normalized to a non-CR scenario. Speedup left, VFU access reduction (center) and dynamic energy reduction (right).

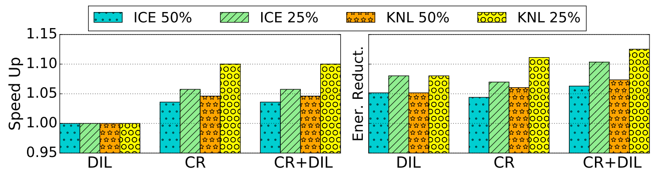


Figure 15. Speedup (left) and total energy reduction (right) of DIL, CR and CR+DIL normalized over a non-CR scenario.

of the number of hardware execution units (or ALU-width). The first idea, called *Basic Cycle Compression* (BCC), detects contiguous blocks of disabled lanes coinciding in the same execution cycle and suppresses the rest of the instruction stages. The slots are used to execute subsequent instructions. It only works when all lanes in a contiguous block are disabled and that is not the case for many divergent applications. The CR proposal does not require zero mask densities to operate and its benefit depends on the percentage of active elements. The second and more complex idea is a generalization of BCC, called *Swizzled Cycle Compression* (SCC). SCC dynamically selects and combines cross-warp threads that are executing the same instruction to the same warp, such that cross-warp threads can now execute on the same SIMD unit, and thereby improve the utilization of the SIMD lanes. CR targets SIMD extensions in CPU architectures, where dynamic instructions for several iterations of the same PC are compacted into one instruction to improve VFU efficiency. Compacting predicated SIMD instructions from different threads is left for future work. Unlike SCC, CR performs *dense operand forwarding*.

Other proposals for GPUs [16], [32] improve the performance and energy efficiency of divergent applications with *Dynamic Warp Formation* and *Variable Warp Sizing* respectively. Both prove these situations with significative benchmarks and motivate the existence of a variable warp size. Also in the GPU domain, Brunie *et al.* [6] propose the execution of two instructions from different disjoint paths (similar to multiple "scalar" datapaths). Khorasani *et al.* [22] introduce the concept of *Collaborative Context Collection* (CCC), a software solution that collects the relevant registers of divergent threads and delays their execution until the best warp lane utilization is obtained. The goal is similar to CR. However, it needs shared memory regions to keep track of the divergent operations for every thread, with significant performance overhead and programmer intervention. GPUs work at a thread level, with individual register information per thread and individual simple ALUs for each "lane" equivalent, so they do not require data movement to optimize ALU energy (what CR does). GPU optimizations work more as a *scheduling optimizer* while CR works as a data defragmenter at a register/RS/ROB level.

Finally, Park *et al.* present *SIMD Defragmenter* [30], a compiler optimization that tries to extract additional DLP by fusing groups of compatible instructions (e.g., two 128-bit additions into a 256-bit addition). This approach does not deal with predication nor extracts DLP from different iterations, but we believe it is complimentary to our proposal and can be used to enable CR in loops with high instruction count by compacting compatible instructions with different PCs.

VIII. Conclusions

Exploiting DLP in current processors with SIMD extensions is critical to improve performance and energy efficiency. When vectorizing applications, divergence control using predication is one of the most challenging obstacles to overcome. Current SIMD extensions execute all elements in a predicated instruction independently of the values in the mask operand, wasting significant fractions of energy and performance.

In this paper we propose the Compaction/Restoration (CR) hardware design, which is capable of achieving density-time performance and energy efficiency with predicated SIMD instructions. CR creates a dense instruction with several dynamic predicated instructions for a certain PC. The active elements of these regular SIMD instructions are *compacted* into a dense instruction. Then, dense instructions are executed and their results are *restored* to the original instructions. This is achieved without programmer intervention.

Our evaluation shows that CR improves performance by up to 25% and reduces dynamic energy consumption by up to 43% on real unmodified predicated applications. Moreover, CR allows executing unmodified legacy code with short SIMD instructions (AVX-2) on newer architectures with wider vectors (AVX-512), achieving up to 56% performance benefits.

IX. Acknowledgements

This work has been partially supported by the RoMoL ERC Advanced Grant (GA 321253), the European HiPEAC Network of Excellence, the Spanish Government (contract TIN2015-65316-P) and the European Union's Horizon 2020 research and innovation program under the Mont-Blanc 2020 project (grant agreement 779877). A. Barredo has been supported by the Spanish Government under Formación del Personal Investigador fellowship number BES-2017-080635. M. Moretó and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and RYC-2017-23269.

References

[1] AMD. 3DNow! Technology Manual. Motorola, 2000.

[2] ARM NEON Technology.

[3] K. Asanoviċ. *Vector Microprocessors*. PhD thesis, 1998.

[4] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, 1968.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib Bin Altaf, N. Vaish, M. Hill, and D. Wood. The gem5 simulator. 39:1–7, 08 2011.

[6] N. Brunie, S. Collange, and G. Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 49–60, June 2012.

[7] Cadence. Genus Synthesis Solution. Available at https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis.html.

[8] D. Callahan, J. Dongarra, and D. Levine. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing)*, pages 98–105, 1988.

[9] J. M. Cebrian, M. Jahre, and L. Natvig. ParVec: Vectorizing the PARSEC Benchmark Suite. *Computing*, pages 1077–1100, 2015.

[10] I. Cray Research. Cray X-MP Series Model 48 Mainframe Reference Manual, 1984.

[11] J. Davies. A 2D N-body solver, with OpenMP, MPI and AVX, 2015. Available at https://github.com/jodavies/nbody.

[12] R. Espasa, M. Valero, and J. E. Smith. Vector Architectures: Past, Present and Future. In *Proceedings of the 12th International Conference on Supercomputing (ICS)*, pages 425–432, 1998.

[13] A. Fog. Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns, 2018. Available at http://www.agner.org/optimize/instruction_tables.pdf.

[14] S. Fuller. Motorola AltiVec Technology. Motorola, 1998.

[15] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

[16] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.*, 6(2):7:1–7:37, July 2009.

[17] A. Harrison and D. Joseph. High Performance Rearrangement and Multiplication Routines for Sparse Tensor Arithmetic. *SIAM Journal on Scientific Computing*, 2018.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

[19] J. N. Huber, O. R. Hernandez, and M. G. Lopez. Effective Vectorization with OpenMP 4.5. 3 2017.

[20] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture., 2012.

[21] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference., 2015.

[22] F. Khorasani, R. Gupta, and L. N. Bhuyan. Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection. In *International Symposium on Microarchitecture (MICRO)*, pages 204–215, 2015.

[23] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanoviċ. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 52–, 2004.

[24] R. Kumar, A. Martínez, and A. González. Vectorizing for wider vector units in a HW/SW co-designed environment. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*, pages 518–525, 2013.

[25] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanoviċ. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 129–140, 2011.

[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.

[27] Mentor. Precision RTL Plus. Available at https://www.mentor.com/products/fpga/synthesis/precision_rtl_plus/.

[28] N. Muralimanohar and R. Balasubramonian. CACTI 6.0: A Tool to Understand Large Caches. Available at https://github.com/HewlettPackard/cacti.

[29] NEC. Vector Supercomputer SX Series: SX-Aurora TSUBASA, 2017.

[30] Y. Park, S. Seo, H. Park, H. Kyu Cho, and S. Mahlke. SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 47, March 2012.

[31] G. Pichon, M. Faverge, P. Ramet, and J. Roman. Reordering Strategy for Blocking Optimization in Sparse Linear Solvers. *SIAM Journal on Matrix Analysis and Applications*, 2017.

[32] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A Variable Warp Size Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 489–501, New York, NY, USA, 2015. ACM.

[33] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, Jan. 1978.

[34] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 440–451, 2012.

[35] J. E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 260–269, 2000.

[36] A. Sodani. Race to Exascale: Opportunities and Challenges. Micro '11 Keynote, 2011.

[37] A. Sodani. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *Hot Chips*, 2015.

[38] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, et al. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[39] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, 2017.

[40] K. Uchida and N. Kasuya. FACOM Vector Processor. 1983.

[41] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi. SIMD Divergence Optimization Through Intra-warp Compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379, 2013.

[42] VentureBeat. Intel confirms Ice Lake Core CPUs with 10nm+ process to followup its 8th-gen chips, 2017. Available at https://venturebeat.com/2017/08/14/intel-confirms-ice-lake-core-cpus-with-10nm-process-to-followup-its-8th-gen-chips/.

[43] Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning. A multiple SIMD, multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614, Feb 2013.

[44] T. Watanabe, T. Furukatsu, R. Kondo, T. Kawamura, and Y. Izutani. The Supercomputer SX System: An Overview. In *Proceedings of the Second International Conference on Supercomputing (ICS)*, pages 51–56, 1987.

[45] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[46] W. J. Watson. The TI ASC: A Highly Modular and Flexible Super Computer Architecture. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I (AFIPS)*, pages 221–228, 1972.

[47] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 577–589, 2015.

[48] M. Yip. Normally Distributed Random Number Generator Benchmark, 2015. Available at https://github.com/miloyip/normaldist-benchmark.

[49] T. Yoshida. Introduction of Fujitsu's HPC processor for the Post-K computer. In *Hot Chips*, 2016.