Compiler-Assisted Compaction/Restoration of SIMD Instructions

Juan M. Cebrian, Thibaud Balem, Adrián Barredo,
Marc Casas, Miquel Moretó, Alberto Ros *Senior member*, and Alexandra Jimborean

Abstract—Vector processors (e.g., SIMD or GPUs) are ubiquitous in high performance systems. All the supercomputers in the world exploit data-level parallelism (DLP), for example by using single instructions to operate over several data elements. Improving vector processing is therefore key for exascale computing. However, despite its potential, vector code generation and execution have significant challenges. Among these challenges, control flow divergence is one of the main performance limiting factors. Most modern vector instruction sets, including SIMD, rely on predication to support divergence control. Nevertheless, the performance and energy consumption in predicated codes is usually insensitive to the number of active elements in a predicated mask. Since the trend is that vector register size increases, the energy efficiency of exascale computing systems will become sub-optimal.
This paper proposes a novel approach to improve execution efficiency in predicated vector codes, the Compiler-Assisted Compaction/Restoration (CACR) technique. Baseline CR delays predicated SIMD instructions with inactive elements, compacting active elements from instances of the same instruction of consecutive loop iterations. Compacted elements form an equivalent *dense* vector instruction. After executing the dense instructions, their results are restored to the original instructions. However, CR has a significant performance and energy penalty when it fails to find active elements, either due to lack of resources when unrolling or because of inter-loop dependencies. In CACR, the compiler analyzes the code looking for key information required to configure CR. Then, it passes this information to the processor via new instructions inserted in the code. This prevents CR from waiting for active elements on scenarios when it would fail to form dense instructions. Simulated results (gem5) show that CACR improves performance by up to 29% and reduces dynamic energy by up to 24.2% on average, for a a set of applications with predicated execution. The baseline CR only achieves 18.6% performance and 14% energy improvements for the same configuration and applications.

Index Terms—SIMD, predication, LLVM, density-time performance.

1 Introduction

The end of Dennard's scaling supposed the stagnation of CPU clock frequency. In this scenario, computer architects and software developers are forced to exploit parallelism to achieve an exascale level of 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 being extensively studied, there are still many unexplored opportunities to achieve significant performance and energy improvements from data-level parallelism (DLP).

Developers usually rely on vector computations in order to expose DLP to the hardware [1], [2]. A vector instruction is a single instruction that operates over multiple data streams (SIMD). Early vector machines exploited DLP with long vectors of thousands of bits. They appeared in the early 1970s and dominated supercomputer designs for two decades [2], [3], [4], [5], [6]. The late 1990s brought the introduction of SIMD extensions to scalar instruction set architectures (ISA). Their goal was to improve the efficiency of multimedia applications, using short 128-bit vectors [7], [8]. Such SIMD extensions have become ubiquitous in today's computer architectures [9], [10], [11], [12]. Indeed, all the supercomputers in the Top 500 exploit DLP in some way.

Processors with longer SIMD vector lengths appeared in the last years, such as the 512-bit SIMD implementations from Intel [9], [13] and Fujitsu [14]. Wider vector designs are less popular nowadays, although NEC's SX-Aurora processor was recently released and uses 16,384-bit vectors [15]. 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.

Vectorized applications offer better performance, higher energy efficiency and greater resource utilization [16]. However, the code vectorization process has several obstacles to overcome, such as horizontal operations[1], data structure conversion, or divergence control, the most challenging being the latter one [17]. Ultimately, the effectiveness of a vector architecture depends on its ability to vectorize large quantities of code [18].

While there are many ways to implement divergence control [19], 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

- *Juan M. Cebrian, A. Ros, and A. Jimborean are with the Computer Engineering Department, University of Murcia, Spain.*
E-mail: jcebrian@um.es, aros@ditec.um.es, alexandra.jimborean@um.es
- *T. Balem is with ENS Rennes.*
- *A. Barredo, M. Casas and M. Moreto are with Barcelona Supercomputing Center.*

1. Horizontal instructions, such as *shuffles*, that move or compute a value from a particular vector register lane to/with another one.

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. This means that current SIMD implementations have *VL-time* performance, wasting a significant portion of energy on unnecessary computations and increasing contention of vector functional units (VFUs). 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 increasing the register size [16], SIMD implementations with VL-time performance will become extremely energy inefficient when executing predicated instructions. Thus, there is an urgent need in exascale computations towards SIMD implementations with density-time performance for predicated executions.

In [20] we proposed a novel hardware mechanism, Compaction/Restoration (CR). CR achieves density-time performance and energy in SIMD codes without programmer intervention. CR identifies code sections with SIMD instructions guarded by a mask, defined from now on as *compactable* instructions. CR delays the execution of compactable instructions with inactive elements. Active elements are extracted and copied into a single equivalent *dense* instruction. The dense instruction is filled up with active elements from instances of the same instruction from later loop iterations that are naturally unrolled by the out-of-order processor. Dense instructions will have, in the best case, all the elements active. Therefore, compactable instructions are executed efficiently with density-time performance, via a single dense instruction. Then, dense results are restored to the original predicated SIMD instructions.

However, delaying the execution of instructions can have a performance and energy penalty if the hardware cannot make up for the time invested in the compaction process. This happens when CR fails to find active elements to form dense instructions, either due to lack of resources when unrolling or because of inter-loop dependencies. In this work, we improve the effectiveness of CR with compiler support. Our compiler analyzes predicated loops of compactable instructions and provides the hardware with critical information in order to re-configure CR timeouts for optimal performance. This prevents CR from waiting for active elements when it would fail to form dense instructions.

CR improves energy consumption by requiring less accesses to the VFU than the baseline. Moreover, performance improves 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 [21]).

2 The Divergence Control Flow Problem

On a scalar code, whenever there is a code section that needs to be guarded, the developer can use a conditional statement. However, in a vector architecture, conditional statements do not have a binary nature, since each element of the vector register can provide a different result to the conditional test. This is known as the divergence control problem, and appears frequently in vectorized codes [17]. Previous studies indicate that at least 10% of the most common vectorizable loops have divergence control issues [22]. Different mechanisms to implement divergence control in the context of SIMD instruction sets have been proposed [19]. From all the proposals, predicated execution is considered the most effective and compiler-friendly.

The main issue with predicated execution in SIMD architectures is its low energy efficiency. Measured mask density[2] is between 18-20% on typical benchmarks [23], [24], [25]. This means that sparse predicated masks are common on modern codes. Therefore, the nature of SIMD architectures translates into a waste of energy in unnecessary computations (up to 80%) and increases contention in the VFU, which can hurt performance. 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.

3 The CR Mechanism

3.1 Overview

CR targets SIMD extensions available in current processors, such as AVX-512 [9]), where the vector length (VL) equals the VFU width. CR creates a *dense* version of each predicated instruction (*compactable* instructions) within the vectorized loop. The loop is naturally unrolled by the out-of-order processor. The active elements from several dynamic instances of the same compactable instruction are extracted and *compacted* into a dense instruction (one dense instruction per program counter, or PC). Compactable instructions are no longer executed unless the CR mechanism fails to compact. Dense instructions delay execution until dense registers are full or a timeout happens.

In the best scenario, dense instructions have source registers with all elements active and are executed instead of the original instructions. As a result, the number of accesses to the VFU and their contention decreases. This is crucial for performance and energy efficiency. We have measured the package power (whole processor) using RAPL in a Skylake-X i7-7820X CPU (8-Cores), using both FIRESTARTER[3] and likwid-bench[4] (peakflops). We set the frequency to 3.3Ghz for both scalar and AVX512, running in "single" mode and making sure power measurements are taken at the same temperature (45C). FIRESTARTER reports 120W for AVX512 and 103W for Scalar, with an idle power of 15W. This means an increment of 16.5% (or 36,3% if idle power is subtracted). Likwid-bench reports an increment from 62W to 73W when going from scalar to AVX512, meaning an increment of 17.7% (55.3% if the 15W are subtracted). Our processor does not allow to measure core/uncore power (PP0/PP1 RAPL), but Intel's Chief Architect MIC Processor stated in 2011 that VFU can add up to 75% of the power dissipated by the core [26]. Once the dense instructions are 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 [27] and arm *Scalable Vector*

2. Percentage of active elements in the mask register.
3. https://tu-dresden.de/zih/forschung/projekte/firestarter
4. https://hpc.fau.de/research/tools/likwid/

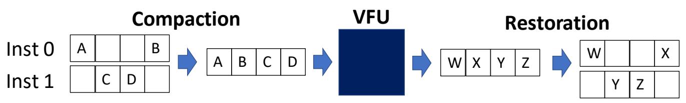

Figure 1. CR basic functionality

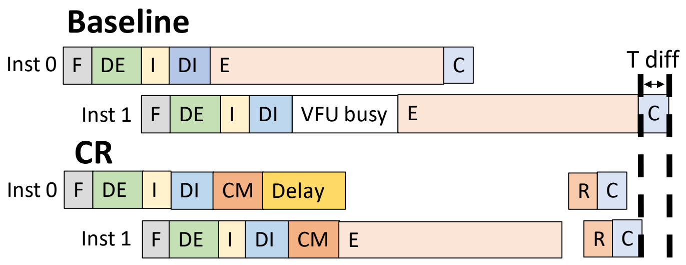

Figure 2. Baseline/CR time diagram. Fetch (F), Decode (DE), Issue (I), Dispatch (DI), Execute (E), Commit (C), Compact (CM), Restore (R)

Extension (*SVE*) [28] 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 tested CR in a simulated out-of-order processor with 512-bit VFUs. Section 3.2 describes the new hardware components to support CR, while Section 3.3 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 3.4), ii) compaction of dense instructions (Section 3.6), iii) execution of dense instructions (Section 3.7), and iv) restoration of compacted instructions (Section 3.8). Next, we present a case study with CR (Section 3.10). Finally, we discuss other considerations (Section 3.11).

3.1.1 Basic Functionality Example

CR basic functionality is shown in Figure 1. 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 2 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.

3.2 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 dense instructions and their compactable instructions. It is needed to perform the compaction (Section 3.6) and restoration (Section 3.8) phases. Table 1 defines the functionality and size of every CIT entry. In this case, we target double-precision operations although smaller types 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 $ReorderBuffer(ROB)Entries/2$ although we did not exceed half of its capacity.

Table 1
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	Reorder Buffer 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. Section 3.6 describes the compaction phase.

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. Section 3.8 describes the restoration phase.

3.3 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 3 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 3.4). 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

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

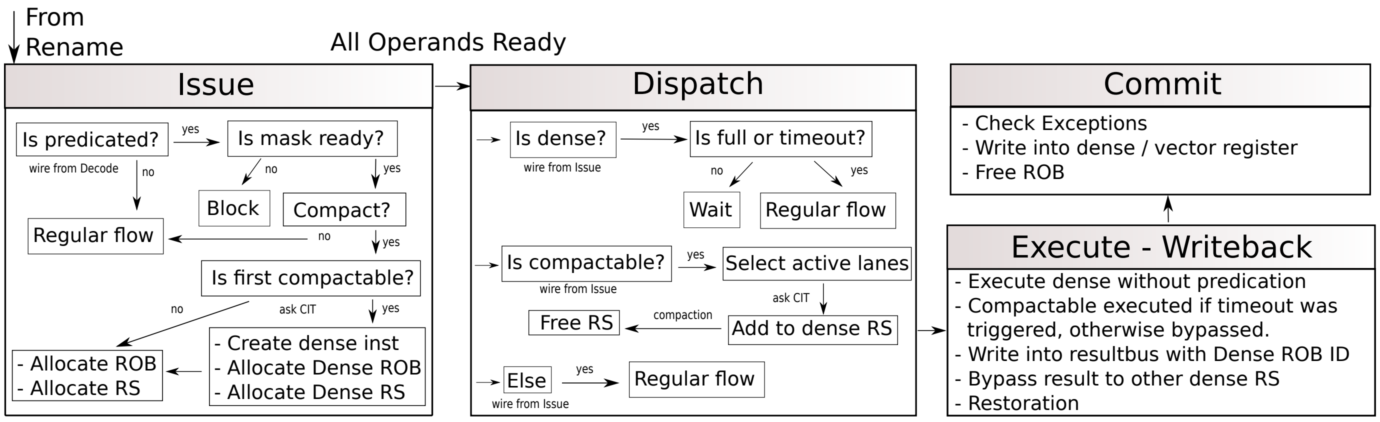

Figure 3. CR overview when incorporated to an out-of-order processor

occupied. In case a new dense instruction is required, a dense instruction is created and its operands are renamed. To keep renaming logic simple, dense registers belong to a specific physical register range. 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 and the *alloc occupancy*, *mask*, *dest reg idx* and *allocate* CIT fields are updated.

3) **Dispatch**: As compactable operands become ready, the compaction phase triggers, their reservation station (RS) are freed and 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 3.7). If the dense destination register is used by subsequent dense instructions, it is forwarded (Section 3.9).

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 3.8).

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

3.4 Detecting Compactable Instructions

Our base CR implementation considers all predicated loops as compaction candidates. However, in a preliminary analysis (Section 6) and in the evaluation (Section 7) we observe that several factors should be considered to enable an more efficient CR: i) predicated instruction latency, ii) number of instructions per iteration, iii) inter-loop dependencies and horizontal instructions[6], iv) arithmetic intensity (AI)[7], v) mask densities and vi) processor events that hide latencies.

The first four factors can be statically determined and have important effects on performance. For instance, inter-loop dependencies cause an execution serialization. On the compiler-assisted CR version, the compiler inserts new instructions that pass this information to the CR hardware. When CR is not provided with compiler information it tries to adapt dynamically to the code requirements. On the other hand, mask densities are fundamental and input-dependent (see Section 6). Finally, some processor events, such as cache misses, pause the core backend, hiding CR latencies.

Compaction candidate SIMD instructions 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 3.9).

3.5 Populating Dense Instructions

In order to populate a dense register, compactable instructions delay execution until the register 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).

3.6 Compaction Phase

In this phase, active elements from compactable instructions in an RS are moved into the RS belonging to the dense instruction. 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 since the VFU already reads all inputs from the RS simultaneously. When compaction finalizes, compactable instructions are called *compacted*.

3.7 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:

6. This refers to dependencies after vectorization, that may come from a reduction or "clever" data reuse between loop iterations (e.g., shuffle or a permute of a register to reuse data).

7. Simplified to the ratio of memory μops to total μops.

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 VFU 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.

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) **Hard timeout**. A dense instruction could have allocated but not inserted compacted instructions waiting for dependencies to be freed or data to be available due to a cache miss. 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 regularly. Section 6.1 studies the impact of the timeout policies.

3.8 Restoration Phase

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

3.9 Dense Register Forwarding

A dense register can be forwarded if it is fully occupied or if subsequent dense instruction share the same inserted compacted instructions positions. The $insert_c$ CIT entry bit provides this information for every allocated compactable

```
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 4. SIMD loop in Intel's pseudo-assembly

instruction. If not, the remaining compactable instructions will be compacted. An efficient dense register forwarding reduces CR latencies and hides the restoration process.

3.10 CR Case Study

To illustrate how the CR mechanism works, we refer to the code from Figure 4. 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 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 5 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 instruction, since the mask registers are ready and the rename stage has been previously accessed. A ROB and an RS entry are allocated for each dense instruction. 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 5. 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

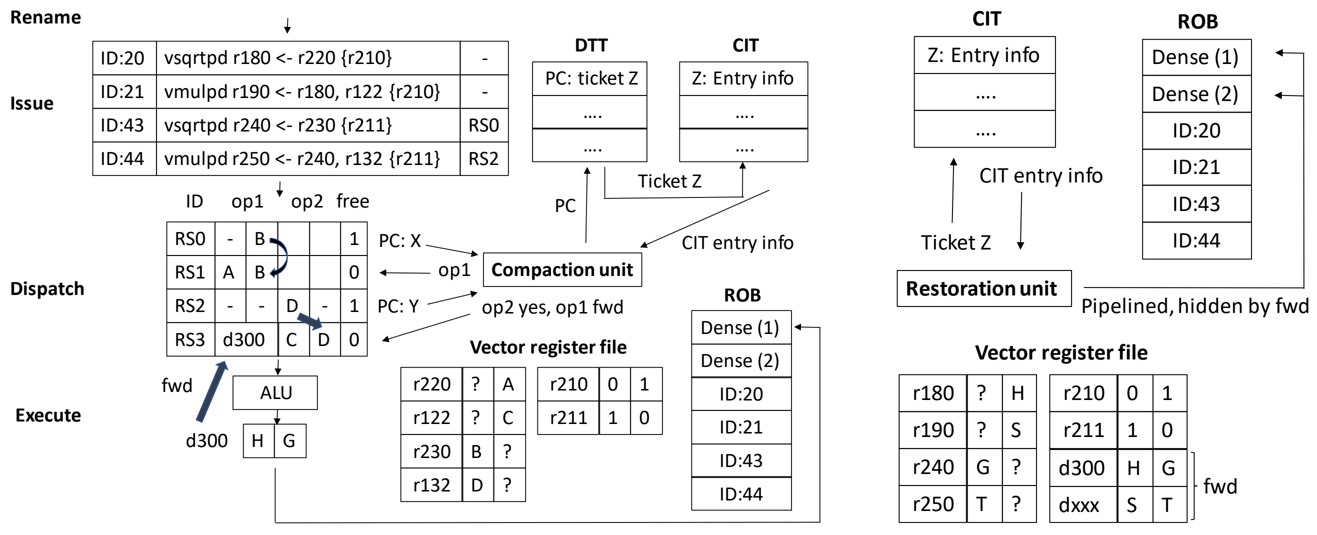

Figure 5. Example of the Compaction (left) and Restoration (right) phases

situation and skips its compaction, notifying that a dense register forwarding is going to happen. In particular, *d300*.

Execution Phase. The dense *vsqrtpd* instruction executes as compaction finishes and its destination register *d300* is forwarded to the dense *vmulpd*, which will then execute.

Restoration Phase. After execution, the restoration phase occurs for the dense instructions *vsqrtpd* and *vmulpd*. A brief overview is depicted in Figure 5. 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.

3.11 Other Considerations

The CIT is squashed in the event of a branch misprediction. Two scenarios must be considered: a) mispredicted 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 mispredicted instructions operands. The first scenario is handled by making the CIT aware of mispredictions. The second scenario is not critical because results are written into mispredicted ROB entries in the restoration phase, but 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. At the moment a dense register is created, the original element positions are lost so the operation cannot be done.

4 Compiler-Assisted CR

To close the gap to density-time performance, we provide the hardware with relevant information to guide the decision on whether to compact or not. This information is also critical to configure the aggressiveness of the CR timeouts. Indeed, the arithmetic intensity, the presence of high-latency instructions (e.g., div, sqrt, exp, log), the μop count and the presence of horizontal operations are application characteristics retrievable at compile-time. To this end, we extended the compiler to analyze the applications and convey such information to the hardware, by inserting new instructions before each target loop. We use one instruction for each piece of information, but this can be optimized by inserting a single instruction with a 32-bit immediate that contains all the information.

The compiler support to assist CR is implemented in LLVM [29]. LLVM is a compiler framework for program analysis and transformation, which facilitates retrieving high-level information at compile-time.

We implemented the compiler support as an LLVM Loop Pass, which is executed on each loop independently. We parse all instructions of the loop, checking their type. For vector instructions, we further analyze the following in order to provide useful information to the CR:

- We check if the vector instruction is a memory operation (load or store), which is then used to compute the arithmetic intensity.
- We check if the vector instruction is a high-latency operation (e.g. div, sqrt, log), to help CR estimate the predicated instruction cost.

Table 2
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	8 insts/cycle
Branch predictor, Branch target buffer	L-TAGE 64KB, 8K+8K entries
Fetch Buffer, Decode Buffer	16B, 56-μops
Fetch, Load and Store Queues	32 entries, 128 entries, 72 entries
Physical Registers	200 integer + 360 floating point
Issue Queue, Re-order Buffer	128 entries, 352 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 (4c.), mul (4c.), 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	1MB, 16-way, 14 cycle access latency
L3 unified cache	16MB, 16-way, 36 cycle access latency
Prefetcher	IPCP++
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

- We check if the vector instruction is a horizontal operation (shuffle, permute, reductions, horizontal addition and horizontal subtraction), which would disable CR.

Next, we compute the total number of static loop μops as follows: pointer loads and stores count as three μops, arithmetic instructions that have one operand in memory count as two μops. The rest of instructions count as one μop. We focus on the Intel's X86_64 target only, since identifying arithmetic instructions with one operand in memory is target dependent. Adding support for other target architectures is straightforward. μops count is used to determine resource utilization (e.g., reorder buffer), and get an approximation of how many loop iterations can we unroll for the given architecture. Finally we detect the LLVM intrinsics and instructions corresponding to the vector horizontal, shuffle, permute and reduction operations. The presence of any such operation would usually disable the CR mechanism.

Once the analysis is complete, we precede the loop with four instructions to convey to the hardware the information retrieved by the compiler (arithmetic intensity, number of high-latency instructions, total number of μops, and the presence of horizontal operations). The hardware can then take a better informed decision on how to configure CR, leveraging this data. Details about how this information is used is discussed in Section 6.2.

5 Experimental Methodology

5.1 Full-System Simulation Infrastructure

We employ *gem5* [30] 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 2 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 1 lists all the fields in the CIT). As explained in Section 3, 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 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 Icelake (ICE) [21] and the Knights Landing (KNL) [13]. 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 [21].

Power consumption is evaluated with McPAT [31] using a process technology of 22nm, a voltage of 0.6V and the default clock gating scheme for the core. We do not measure power for uncore (memory controller, network, etc). We incorporate the changes suggested by Xi Vaidya *et al.* [32] to improve the accuracy of the models. The CIT structure is modeled in CACTI 6.5 [33], adding the appropriate counters in gem5 to measure the extra power introduced by it. The CR units have been modeled in RTL [34], [35]. Results for a 22nm technology show area requirements of 5000μm^2. It is almost three orders of magnitude smaller than a 512-bit VFU 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 VFU computed by McPAT (0.92W).

5.2 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 [36], a quadratic equation (Quadr), a Box-Muller number generator (RNG) [37], a sound distorter (S-Distort) and a raytracing kernel (Ray) [25].
Section 6 makes use of a SIMD micro-benchmark to explore the design space. This micro-benchmark 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.

Fung and Vaidya *et al.* studied the mask densities in several applications [23] [24]. They show that they are usually input-dependent and range between 15-60%. Since input selection for these applications 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 work on SIMD

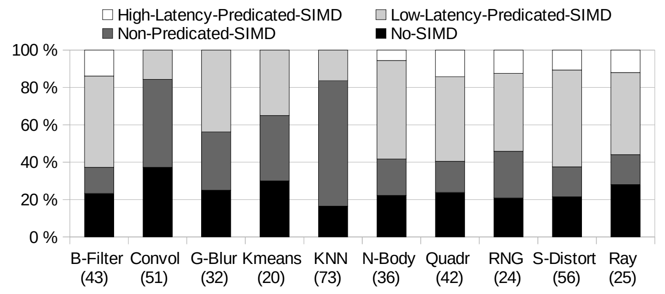

Figure 6. Dominant loop iteration breakdown. In the X axis, the applications name and their number of instructions per iteration. In the Y axis, the instruction types in every iteration

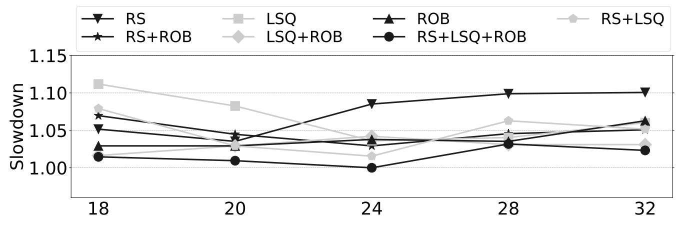

Figure 7. Timeout policy combinations impact on performance, normalized to the best scenario. The hard timeout policy is implicit in every scenario. In the x-axis the number of cycles for each timeout policy

control flow divergence [23], [24] does. Moreover, Vaidya *et al.* [24] 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 combination possibilities of the true-value positions inside the mask.

Figure 6 shows the instruction breakdown of the main loop in the ROI of each benchmark. Loops contain between 20 and 73 μops. The predicated instruction percentage is between 17% (KNN) and 62% (B-Filter).

6 Timeout Policies

6.1 Without Compiler Support

Next, we measure the impact of the timeout policies discussed in Section 3.7. In this case, the micro-benchmark is used with different timeout policies. Figure 7 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 the hard timeout, since it is required for the 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.

6.2 With Compiler Support

This section describes how to utilize the information provided by the compiler in order to optimize the CR hardware. As discussed in Section 4, the compiler provides the CR hardware with four key pieces of information: a) the arithmetic intensity, b) the presence of high-latency instructions (e.g., div, sqrt, exp, log), c) the total μop count and d) the presence of inter-loop dependencies or horizontal operations. We define the unroll factor (UF) as the ROB size divided by μop count. Given the huge amount of combinations and for the sake of clarity, we omit micro-benchmark raw data and focus on key insights from our design space exploration.

Our analysis reveals the following key insights: When the arithmetic intensity is high or medium, there are no loop dependencies and there are high-latency operations, Icelake needs an UF of at least four times for CR to improve performance and energy, while an UF of two suffices for KNL. Under the same conditions, but with loop dependencies, both platforms require an UF of four to break even, while eight provides energy and performance gains. On the other hand, with medium or high AI, loop dependencies and no high-latency instructions, CR only provides energy improvements when it can unroll at least 8 times, but it provides no performance improvement.

The hard timeout should be set to 100+ cycles for high AI and the loop UF mentioned in each scenario can be achieved. Resource utilization timeout should be set to 95% occupancy. For medium and low AI and matching UF, the hard timeout should be set around 40 cycles, and resource utilization timeout to 85% usage. When loop cannot be matched CR should be disabled. Finally, when AI is low, and there are loop dependencies, CR should be disabled.

7 Evaluation

7.1 Predicated SIMD Applications

First we will compare the best per-benchmark statically selected CR configuration against the compiler-assisted CR and the baseline conservative CR. Our previous work did not use the best statically selected configuration, but an adaptive approach that was quite conservative [20]. CR is evaluated with ten different applications. We will measure the performance, energy efficiency and VFU access reduction on each application. As described in Section 5, we explore two different mask densities (25% and 50%) and two processor configurations with different instruction latencies.

Results are normalized to a regular no-CR execution. In all the experiments, the KNL-like configuration provides more optimization opportunities for CR, since there is more contention in the VFU. Moreover, we can see that the performance and energy effects of CR greatly depend on the presence of high-latency instructions that block the VFU. In addition, lower mask densities (i.e. 25%) lead to more compaction opportunities. The CR hardware decided to disable compaction for Convol and G-Blur, given the information provided by the compiler.

Figure 8 depicts the results in terms of speedup. Significant speedups are obtained for some of the evaluated benchmarks with CACR. This is the case of Quadr, RNG and Ray in KNL. All of them contain a high percentage of long latency SIMD instructions per loop iteration (as shown in Figure 6). They achieve performance improvements up to 100%, 93% and 64% respectively. CACR outperforms the original CR, even doubling the performance for Quadr. B-Filter and S-Distort also contain long latency SIMD instructions. However, their higher number of instructions per loop iteration prevents CR from unrolling more than four

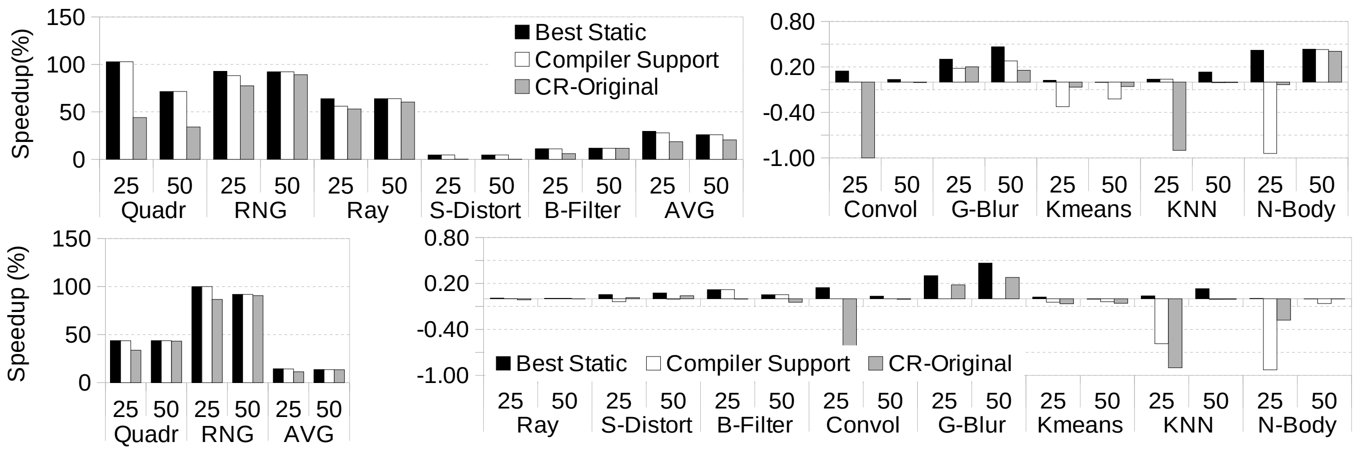

Figure 8. Performance improvement (%) for KNL (top) and ICE (bottom). Normalized to a non-CR scenario

iterations. This means that we can use CR to improve energy efficiency, but we miss the opportunity to extract instruction level parallelism between dense instructions. Nevertheless CACR improves performance for these applications by 4.5% and 11.1% respectively. When running with ICE latencies, VFU contention is only present for Quadr and RNG. Indeed, CACR is able to improve performance by 45% and 100% respectively. For all other applications, the compiler-assisted CR does not have a significant effect on performance.

Figure 9 shows the instruction reduction factor for CACR and CR. We can see how CACR is able to reduce the number of VFU accesses by a factor of around 3.6x for Quadr, Ray, S-Distord and B-Filter. RNG and Kmeans instruction reduction is limited to around 2x. This is due to the fact that these applications have loop dependencies and non-predicated instructions.

The application memory access pattern is important for both approaches CR, since it can hide the dense register compaction/restoration. Kmeans and KNN have a contiguous memory access pattern and no long latency predicated instructions. However, the large amount of instructions and the low percentage of predicated instructions in KNN prevent both CRs from achieving their potential, limiting compaction factor to 27%. KNN also contains horizontal operations, blocking dense register forwarding. CACR disables CR for Convol and G-Blur, so no instruction reduction.

For all the applications, the long latencies of the KNL configuration enable higher VFU access reductions that lead to better dynamic energy results (Figure 10). Energy reduction ranges from 3% to 80% for KNL and a 25% mask density, and between 2% and 68% for ICE. In KNL there is a higher contention in the VFU than in ICE. 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, the compaction phase can be avoided for that lane, reducing latency and energy consumption. For instance, 72% of dense lanes can be forwarded in B-Filter, 65% in S-Distort, 73% in Kmeans, 46% in KNN, 77% in RNG and 46% in G-Blur.

Finally, Figure 10 shows the energy improvements of CACR and CR. We can see that the CACR is able to improve energy for all the applications, without any performance impact. Energy improvements can reach up to 80% for Quadr in KNL and 70% for RNG in ICE. The huge difference with benchmarks like N-Body come from the performance improvements, that reduce leakage energy significantly. The more predicated instructions (especially those that benefit from dense register forwarding), the better the energy.

7.2 Comparison with Other Proposals

This section compares CACR with disable inactive lanes (DIL) [38], 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. Interestingly, CR and DIL can be combined to further reduce the power consumption of CR when a timeout avoids compaction.

As expected, DIL and CR+DIL do not improve performance over the baseline and CR, respectively. Figure 11 shows the average energy reduction of the three techniques over a baseline without CR. DIL reduces energy between 8% and 16% 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, and thus, a reduction on leakage power. However, in benchmarks in which CR provides no performance benefits, DIL achieves better energy reduction. Thus, CR+DIL provides the best energy results with a reduction between 25% and 55%.

8 Related Work

Sparse to dense transformations have been broadly studied from the software standpoint for some time. Harrison *et al.* and Pichon *et al.* proposed reordering techniques and implemented math libraries to mitigate the sparsity problem [39] [40]. However, our approach is a hardware mechanism to improve execution efficiency in the context of sparse predicated SIMD instructions.

Smith *et al.* [19] explored several alternatives to implement conditional operations in vector ISAs. One of the proposals, *Register compress/expand*, 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

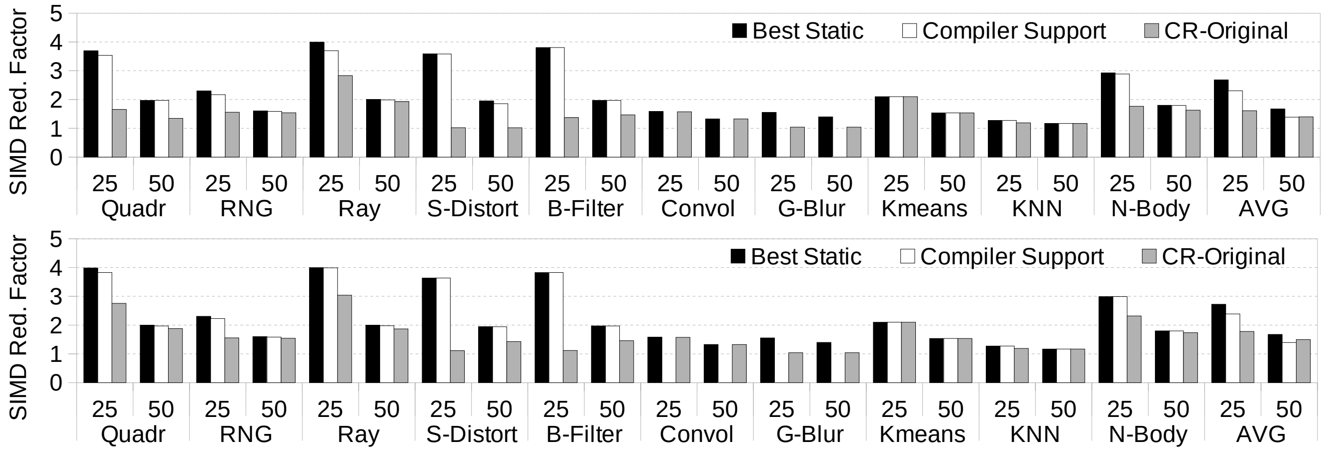

Figure 9. VFU reduction factor (n times) for KNL (top) and ICE (bottom). Normalized to a non-CR scenario

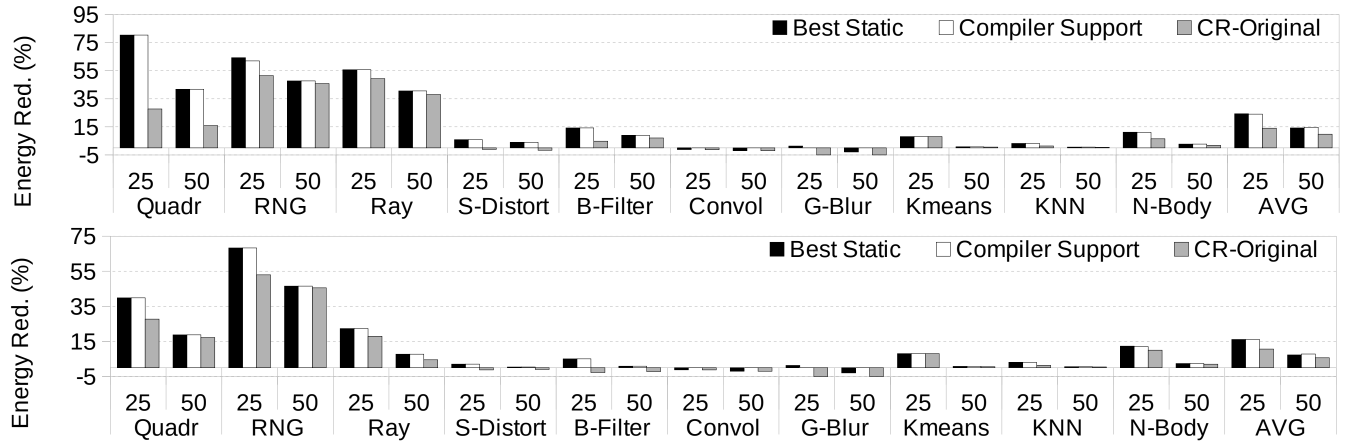

Figure 10. Energy reduction (%) for KNL (top) and ICE (bottom). Normalized to a non-CR scenario

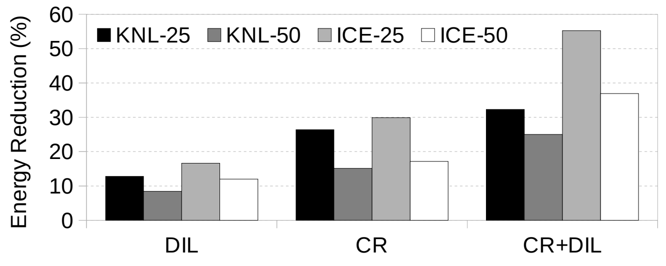

Figure 11. Average energy reduction of DIL, CR and CR+DIL normalized over a non-CR scenario

in multiple vector supercomputers [3], [4], [41]. A similar instruction, *vcompress*, is available in AVX-512 [9]. The key differences with CR are: i) CR does not require new instructions; ii) CR is transparent to the programmer; and iii) the authors assume a traditional vector architecture with a vector length (VL) much longer than the VFU width. In contrast, CR targets short vectors available in modern processors with SIMD extensions. Shorter vector registers limit the impact of the compress/expand approach.

Some divergence control proposals assume an architecture with several *scalar* datapaths [42], [43], [44]. 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.* [24] 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 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 is a hardware version of *Register compress/expand*. Instead of pushing the active lanes to the beginning of a register, SCC reorders them in a single instruction to obtain blocks of dead lanes padded to the ALU-width and suppresses them. Performance is limited by the complexity of the shuffling algorithm. This proposal does not benefit from active elements from different loop iterations and it does not perform *dense forwarding*.

Other proposals for GPUs [45] 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.* [46] propose the execution of two instructions from different disjoint paths (similar to multiple "scalar" datapaths). Khorasani *et al.* [47] 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* [48], 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.

9 Conclusions

In this paper we propose the Compiler-Assisted Compaction/Restoration (CACR) 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 can be achieved without programmer intervention. However, CR induces a performance and energy penalty when it fails to find active elements, either due to lack of resources when unrolling or because of inter-loop dependencies. In order to optimize CR's timeout strategies, in CACR the compiler gathers critical information regarding arithmetic intensity, the presence of high-latency instructions (e.g., div, sqrt), μop count and the presence of inter-loop dependencies/horizontal operations. Then, it passes this information to the hardware via new instructions.

Our evaluation in a simulated environment (gem5) shows that CACR improves average performance by up to 29% and reduces average dynamic energy by up to 24.2% on real unmodified predicated applications. Our original CR implementation (without compiler support) only achieves 18.6% performance and 14% energy improvements. This is because it uses an adaptive timeout policy, that needs to be very conservative to prevent performance degradation.

Acknowledgments

This work has been partially supported by the Spanish Government (SEV2015-0493, BES-2017-080635), the Spanish Ministry of Science and Innovation (PID2019-107255GB-C21/AEI/10.13039/501100011033, RTI2018-098156-B-C53), the ECHO and RoMoL ERC projects (819134, 321253), the European HiPEAC Network and the Mont-Blanc 2020 project (EU-FP7-610402 and EU-H2020-779877). and the Spanish Ministry of Economy, Industry and Competitiveness (RYC-2016-21104, RYC-2017-23269 and RYC-2018-025200-I).

References

[1] K. Asanoviĉ, "Vector Microprocessors," Ph.D. dissertation, 1998.

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

[3] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

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

[5] G. H. Barnes, R. M. Brown *et al.*, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, 1968.

[6] 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)*, 1972, pp. 221–228.

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

[8] S. Fuller, "Motorola AltiVec Technology." Motorola, 1998.

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

[10] "ARM NEON Technology," ARM, Ltd.

[11] N. Stephens, S. Biles *et al.*, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

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

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

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

[15] NEC, "Vector Supercomputer SX Series: SX-Aurora," 2017.

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

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

[18] N. Satish, C. Kim *et al.*, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 440–451.

[19] 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)*, 2000, pp. 260–269.

[20] A. Barredo, J. M. Cebrian *et al.*, "Improving predication efficiency through compaction/restoration of SIMD instructions," in *Proceedings of the 2020 IEEE 26th International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

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

[22] 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)*, 1988, pp. 98–105.

[23] 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)*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 25–36.

[24] A. S. Vaidya, A. Shayesteh *et al.*, "SIMD Divergence Optimization Through Intra-warp Compaction," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

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

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

[27] A. Waterman, Y. Lee *et al.*, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

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

[29] C. Lattner and V. S. Adve, "Llvm: A compilation framework for lifelong program analysis and transformation." in *CGO*. IEEE Computer Society, 2004, pp. 75–88.
[30] N. Binkert, B. Beckmann *et al.*, "The gem5 simulator," vol. 39, pp. 1–7, 08 2011.
[31] S. Li, J. H. Ahn *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.
[32] S. Xi, H. Jacobson *et al.*, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.
[33] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A Tool to Understand Large Caches," Available at https://github.com/HewlettPackard/cacti.
[34] Mentor, "Precision RTL Plus," Available at https://www.mentor.com/products/fpga/synthesis/precision_rtl_plus/.
[35] Cadence, "Genus Synthesis Solution," Available at https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis.html.
[36] J. Davies, "A 2D N-body solver, with OpenMP, MPI and AVX," 2015, Available at https://github.com/jodavies/nbody.
[37] M. Yip, "Normally Distributed Random Number Generator Benchmark," 2015, Available at https://github.com/miloyip/normaldist-benchmark.
[38] 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 HPCC 2013, Zhangjiajie, China, November 13-15, 2013*, 2013, pp. 518–525.
[39] A. Harrison and D. Joseph, "High performance rearrangement and multiplication routines for sparse tensor arithmetic," *SIAM Journal on Scientific Computing*, 2018.
[40] G. Pichon, M. Faverge *et al.*, "Reordering strategy for blocking optimization in sparse linear solvers," *SIAM Journal on Matrix Analysis and Applications*, 2017.
[41] K. Uchida and N. Kasuya, "FACOM Vector Processor," pp. 69–71, 01 1983.
[42] Y. Wang, S. Chen *et al.*, "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)*, Feb 2013, pp. 603–614.
[43] R. Krashinsky, C. Batten *et al.*, "The Vector-Thread Architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 52–.
[44] Y. Lee, R. Avizienis *et al.*, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 129–140.
[45] W. W. L. Fung, I. Sham *et al.*, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, pp. 7:1–7:37, Jul. 2009.
[46] 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)*, June 2012, pp. 49–60.
[47] 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)*, 2015, pp. 204–215.
[48] Y. Park, S. Seo *et al.*, "SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 47, March 2012.

Juan M. Cebrian Received his Bsc, Msc and PhD in computer science from the University of Murcia (UM) in 2006, 2007 and 2011 respectively. Postdoctoral researcher in the Department of Computer and Information Science at NTNU (2012 to 2014), at the Barcelona Supercomputing Center (2015 to 2018) and the UM (current). His research interests include the design of energy efficient multicores and SIMD technologies.

Thibaud Balem received his BSc degree in Computer Science at the University of Rennes in 2018. He is a student in a research-oriented MSc in computer science at University of Rennes and École Normale Supérieure (ENS) Rennes, France. His research interests include compilation and low level optimization and more recently data science and pattern mining.

Adrián Barredo Born in Santander, Cantabria (Spain) in 1993. He received his BSc degree in Telecommunications Engineering from Universidad de Cantabria (UC) in 2015. In 2017, he obtained his MSc in Innovation and Research in Informatics from Universitat Politècnica de Catalunya (UPC). Obtained his PhD at the Barcelona Supercomputing Center in 2020 and he is currently working for Huawei.

Miquel Moretó Senior researcher at the Barcelona Supercomputing Center (BSC). He spent 15 months as a postdoctoral fellow at the International Computer Science Institute (ICSI), affiliated with UC Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from the Universitat Politecnica de Catalunya (UPC), Spain. His research interests include studying shared resources in multithreaded architectures and hardware-software codesign for future systems.

Marc Casas Senior researcher at the Barcelona Supercomputing Center. Prior to this, he spent 3 years as a post-doctoral fellow at the Lawrence Livermore National Laboratory (LLNL). He received his B.Sc. and M.Sc. degrees in mathematics in 2004 from the UPC and the PhD in Computer Science in 2010 from the Computer Architecture Department of UPC. His research interests are high performance computing, runtime systems and parallel algorithms.

Alberto Ros is associate professor in the Computer Engineering Department at the University of Murcia, Spain. He hold postdoctoral positions at the Universitat Politècnica de València and at Uppsala University. He received an European Research Council Consolidator Grant in 2018. Working on cache coherence, memory hierarchy designs, memory consistency, and processor microarchitecture, he has co-authored more than 70 peer-reviewed articles.

Alexandra Jimborean is a Ramón y Cajal researcher at the University of Murcia, since 2020. Her research focuses on compile-time and runtime code analysis and optimization for performance, energy efficiency, and security and on software-hardware co-designs. She obtained her PhD from the University of Strasbourg, France in 2012, held a postdoctoral fellowship in Uppsala University, Sweden and continued as Assistant Professor and Associate Professor.