Vapor SIMD: Auto-Vectorize Once, Run Everywhere

Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams,
David Yuste, Albert Cohen, Ayal Zaks

Vapor SIMD: Auto-Vectorize Once, Run Everywhere

Dorit Nuzman*, Sergei Dyshel*, Erven Rohou†, Ira Rosen*,
Kevin Williams†, David Yuste†, Albert Cohen‡, and Ayal Zaks*
*IBM Haifa Research Lab, Haifa, Israel – HiPEAC member, Email: (dorit,sergeid,irar,zaks)@il.ibm.com
†INRIA Rennes – Bretagne Atlantique, Rennes, France – HiPEAC member, Email: (erven.rohou,kevin.williams,david.yuste)@inria.fr
‡INRIA Saclay – Île-de-France, Orsay, France – HiPEAC member, Email: albert.cohen@inria.fr

***Abstract*—Just-in-Time (JIT) compiler technology offers portability while facilitating target- and context-specific specialization. Single-Instruction-Multiple-Data (SIMD) hardware is ubiquitous and markedly diverse, but can be difficult for JIT compilers to efficiently target due to resource and budget constraints. We present our design for a synergistic auto-vectorizing compilation scheme. The scheme is composed of an aggressive, generic offline stage coupled with a lightweight, target-specific online stage. Our method leverages the optimized intermediate results provided by the first stage across disparate SIMD architectures from different vendors, having distinct characteristics ranging from different vector sizes, memory alignment and access constraints, to special computational idioms. We demonstrate the effectiveness of our design using a set of kernels that exercise innermost loop, outer loop, as well as straight-line code vectorization, all automatically extracted by the common offline compilation stage. This results in performance comparable to that provided by specialized monolithic offline compilers. Our framework is implemented using open-source tools and standards, thereby promoting interoperability and extendibility.**

I. Motivation

Generating code for SIMD hardware has traditionally relied on target-specific manual optimization, using a plethora of intrinsic functions or aggressive offline compiler optimizations. Both approaches suffer from a lack of portability — a painful deficiency in view of the diversity and constantly evolving nature of SIMD architectures. On the other hand, virtual machines are becoming ubiquitous, providing portability while keeping the underlying physical machines hidden from programmers and offline compilers. This creates a problem for sophisticated code optimizations that depend on the details of the underlying machine.

Just-in-time compilation technology holds the promise of efficiently supporting diverse architectures, but is typically constrained by the amount of resources available. We propose a compound compilation technology, as depicted in Figure 1, capable of not only targeting diverse SIMD hardware, but doing so efficiently. This is accomplished by leveraging and complementing traditional offline compilers, thereby yielding a performance-portable and interoperable solution.

Our main goal is to facilitate portable vectorization across diverse SIMD targets of different vector sizes. Our approach is based on the notion of split-compilation [6], in which source code undergoes two (or more) separate yet synergistic compilation stages before finally being translated into machine code. In our design, optimizations are carefully coordinated and distributed across these compilation stages. Our split vectorization framework simultaneously achieves the following four sub-goals: performance competitive with native compilation, negligible JIT compilation time, low overhead for scalar execution, and bytecode compaction.

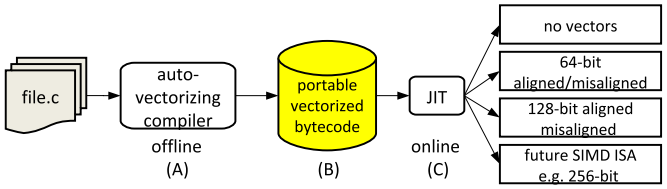


Figure 1. Split compilation flows

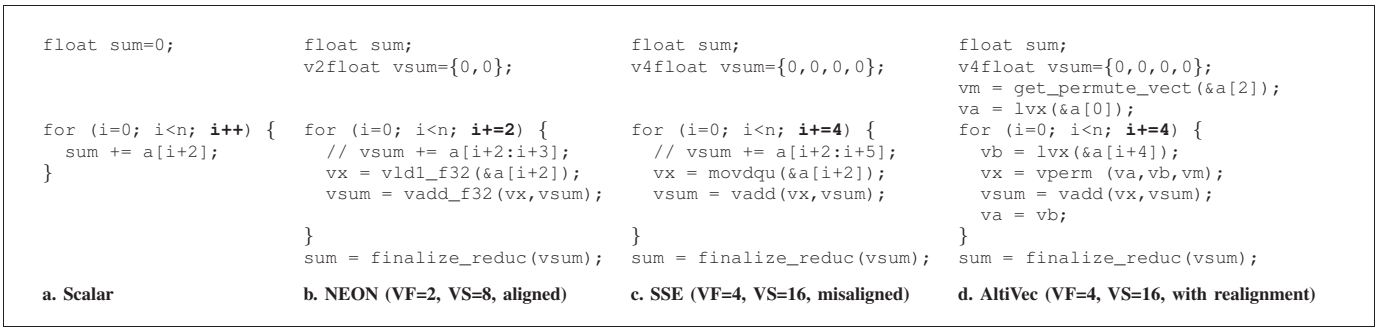
Our work contributes the following:

1) Design of a split vectorization framework capable of automatically vectorizing scalar source code, encompassing state of the art vectorization capabilities, including strided accesses, outer loops, as well as within loops or straight-line code.
2) An open, interoperable framework to mix-and-match offline and online compilation tools, leveraging the advantages of both in the context of automatic vectorization.
3) A study of how split-compilation vectorization compares with monolithic, fixed-target, fully offline compilation, on a variety of kernels exercising diverse SIMD capabilities and evaluated on a range of targets with differing vector sizes and memory alignment restrictions.

The rest of this paper is organized as follows: Section II provides relevant background on vectorization. We then present split-vectorization and related design choices in Section III. In Section IV we describe our experimental environment, and then we evaluate our experimental results in Section V. In Section VI we compare our work with prior art and present our conclusions in Section VII.

II. Background

The goal of vectorization is to replace multiple (dynamic) occurrences of the same scalar instruction operating repeatedly on different data elements with a Single vector Instruction that



```
float sum=0;

for (i=0; i<n; i++) {
  sum += a[i+2];
}
```

a. Scalar

```
float sum;
v2float vsum={0,0};

for (i=0; i<n; i+=2) {
  // vsum += a[i+2:i+3];
  vx = vld1_f32(&a[i+2]);
  vsum = vadd_f32(vx,vsum);

}
sum = finalize_reduc(vsum);
```

b. NEON (VF=2, VS=8, aligned)

```
float sum;
v4float vsum={0,0,0,0};

for (i=0; i<n; i+=4) {
  // vsum += a[i+2:i+5];
  vx = movdqu(&a[i+2]);
  vsum = vadd(vx,vsum);

}
sum = finalize_reduc(vsum);
```

c. SSE (VF=4, VS=16, misaligned)

```
float sum;
v4float vsum={0,0,0,0};
vm = get_permute_vect(&a[2]);
va = lvx(&a[0]);
for (i=0; i<n; i+=4) {
  vb = lvx(&a[i+4]);
  vx = vperm (va,vb,vm);
  vsum = vadd(vx,vsum);
  va = vb;
}
sum = finalize_reduc(vsum);
```

d. AltiVec (VF=4, VS=16, with realignment)

Figure 2. Vectorizing for different platforms

operates on Multiple Data elements simultaneously (SIMD), usually after first packing these data elements into a vector register. Such occurrences can be found, for example, across iterations of a loop, as shown in Figure 2a. (We mostly consider loop-centric cases, however our method is not limited to loops.) The number of elements that are operated upon in parallel is the vectorization factor (VF). Vectorizing a loop with iteration count n can be compared to unrolling the loop by VF, and then replacing the VF occurrences of each instruction in the unrolled iteration with a single vector instruction. The vectorization factor is determined by the vector size (VS) supported by the target architecture, and by the sizes of the data-types in a given computation. For example, the 16-byte vector registers of AltiVec and SSE ($\mathtt{VS} = 16$) can accommodate four float elements (so $\mathtt{VF} = 4$); whereas the 8-byte vector registers of NEON ($\mathtt{VS} = 8$, see Section IV-A) can accommodate two float elements (so $\mathtt{VF} = 2$). Figures 2b,c,d illustrate how VF determines the unrolling factor of the vectorized loop, i.e., the increment of the loop induction variable. The VF and VS also determine the minimum distance admissible for loop-carried dependences, and often the alignment restrictions, as explained below. With the basic vectorization terms defined, we now review the main analysis tasks that vectorizers need to perform.

a) Dependences: For vectorization to be correct, operations on individual data elements must be independent. When vectorizing a loop, the total distance of any cycle made of dependences carried by the loop must be greater than or equal to VF [1]. One must therefore compute a dependence graph with distance abstraction. The complexity of such a computation is quadratic in the number of array references and relies on many compilation passes to be effective, including interprocedural pointer and side-effect analyses, loop nest normalization, and induction variable recognition [1], [17]. In addition, advanced vectorizers also target outer loops and involve loop nest transformations to enable further vectorization opportunities [18], [25]. These extensions require building dependence graphs with distance vectors or polyhedral abstractions and solving combinatorial optimization problems. The burden of such a compilation infrastructure and its algorithmic complexity are currently impractical for JIT compilation. This motivates a split compilation approach in which the identification of vectorization opportunities and their safety conditions are prepared offline.

Some dependence cycles are induced by operations on a value computed by the previous iteration (e.g., reductions). A special vectorization technique can be employed in many cases of reduction operations: partial results are accumulated in a temporary vector and reduced into a single scalar result at the end of the loop (see Figure 2). Detection of reduction cycles is more efficient than general data-dependence analysis, but does require loop-level def-use analysis, and as such is not always suitable for lightweight JIT compilation.

b) Alignment: Memory architectures usually require that data be aligned on certain boundaries (typically on a VS boundary) for vector memory accesses to be efficient or supported at all. Some targets support misaligned accesses (such as the `movdqu` instruction of SSE, see Figure 2c), but these are usually considerably less efficient than aligned accesses. Other targets (like AltiVec) support only aligned accesses, and provide mechanisms for data reorganization: first data is fetched from neighboring aligned addresses (`a[i]` and `a[i+4]` in the example); then, the desired data elements (`a[i+2:i+5]`) are extracted using a realignment idiom (e.g., permute, shuffle, or shift). This involves an increased number of memory accesses as well as increased realignment overhead in each iteration, with potentially significant impact on performance. Static realignment techniques have been a topic of recent work [26], [8], [11], [16], [2].

Multiple memory accesses due to alignment considerations can be optimized together by reusing data loaded in a previous iteration (see Figure 2d), however other realignment overheads remain. Because of these overheads, vectorizing compilers try to compute properties of accesses to determine whether they are aligned or not, potentially forcing their alignment using padding or loop peeling. While in general, the actual alignment may be determined only at run-time, hints about static alignment properties, as well as static transformations to force alignment can be crucial for performance, but are typically unaffordable for JIT compilers [16], [25].

c) Loop selection and cost model: Handling alignment is one example of the overheads incurred by vectorization. Other overheads are associated with runtime dependence testing, loop peeling (e.g., when the number of iterations is unknown or not divisible by VS), data reorganization to deal with strided accesses and type conversions, and more. Because of these overheads, vectorization may not always be profitable. A cost

model is needed to determine when to vectorize. For nested loops, the model should also determine which loop in the nest to vectorize. Loop-counts, strides of data-accesses, locality and alignment behavior of each loop in the nest, all affect this decision [18], [25]. This determination requires complex static analyses, which may be impractical for a JIT compiler.

Indeed, much of the complexity of loop vectorization is associated with proving properties and performing transformations in the scope of loops. This may suggest straight-line code vectorization (also known as SLP [10]) as an appealing choice for JIT vectorization. SLP detects groups of isomorphic instructions in basic blocks, ignoring possible enclosing loop context; it results in a more lightweight vectorization approach. However, this approach cannot optimize across loop iterations (including optimizing realignment) and cannot vectorize loop-carried idioms (like reductions). Furthermore, it is even more dependent on a cost model, because it applies to short sequences of code, which may result in many vector-scalar transitions. Finally, even SLP requires a range of pre-processing analyses and transformations, including alignment [11], if-conversion [24], data-reuse, and careful selection of unrolling factors [23]. SLP itself involves an analysis of quadratic complexity in the number of data-references considered.

The complexity of vectorization, at any scope (basic blocks, loops and loop-nests) calls for a split vectorization design, in which the time-consuming analyses and transformations are carried out by an offline compiler, encoding its decisions and recommendations in such a way that allows a subsequent JIT compiler to produce high quality vectorized code at minimum penalty. We describe our proposed design and its implementation in the next section.

III. Split Vectorization

The three levels that constitute the split-vectorization framework are: (1) defining the "split layer", which is the abstraction layer between the static and dynamic compilation passes, containing vectorized bytecode (Figure 1(B)); (2) generating the vectorized bytecode by the first offline compilation pass (Figure 1(A)), and finally (3) generating actual machine code by the second, online JIT compilation pass (Figure 1(C)).

A. Split Abstraction Layer

Table 1 lists the abstract idioms that constitute our split layer. Defining the vectorized abstraction layer between the offline and online compilers is a similar challenge to that of designing the vectorized intermediate representation (IR) in a static multi-platform vectorizer [16]. In both cases the goal is to introduce idioms that are on one hand translatable to any SIMD platform, and are therefore as high-level and generic as possible, targeting the greatest common denominator of SIMD platforms [21], but that can, on the other hand, result in the best performance for each individual platform.

In a split compilation framework, whose main purpose is to facilitate separate and lightweight JIT compilation, additional considerations come into play. (1) The split layer should facilitate interoperability among different toolchains, and should therefore be incorporated into a standard representation (without breaking it). (2) Everything that depends on machine-specific information (e.g., the `VS`) needs to be abstracted away and parameterized. Pointer increments, loop unrolling factors, misalignment computations, are all expressed using idioms like `get_VF` and `get_align_limit` to abstract this information at the bytecode level and are materialized only during the second compilation pass. Similarly, vector initialization idioms (e.g., `init_*`) manifest the semantics of initialization without being sensitive to the actual `VS`. (3) All information and metadata (e.g., about alignment properties, cost model metrics) need to be encoded in the split layer. Examples include misalignment hints (the `mis`,`mod` arguments to the realignment idioms) and the `loop_bound` and `version_guard_COND` hints, as explained in the following sections. (4) Finally, the split layer should facilitate a JIT vectorization whose complexity is linear in the code size.

One of the surprising consequences of the above requirements is that, in some cases, having the offline compiler generate already optimized bytecode is better, even if only some SIMD platforms can benefit from it. This is true provided the online compiler can easily recover the simpler vectorization scheme that can be supported by the target at hand. We took this approach with respect to realignment. Generating optimized realignment is similar to applying predictive commoning (or software-pipelining) and requires loop-level cross-iteration data-reuse analysis — tasks that are more suitable for the offline compiler. However, this requires support for realignment idioms that are available on some SIMD platforms (like AltiVec, VSX, SPU), but not all. The vectorized bytecode, therefore, has to express the optimized realignment in such a way that allows the online compiler to easily and seamlessly revert back to regular realignment (e.g., using misaligned accesses directly if supported), or even revert to aligned code or scalar code, as appropriate, without performance or compile time penalties. We explain how this is carried out, facilitated by the abstraction layer, in Section III-C. This process differs from our previous approach [22], in which the burden of realignment is left for the JIT compiler to handle.

Lastly, although the idioms of our split layer could be exposed as programming intrinsics, our design focuses on widening the semantic flow across compilation stages, devising SIMD idioms for efficient use by auto-vectorizing compilers. This differs from a programming layer approach that would prioritize simplicity and generality [3]. Our approach borrows from many of the idioms used by current state-of-the-art auto-vectorizing compilers, and as such, it supports all advanced vectorization features of modern vectorizers. Such features include reductions (using for example, `init_reduc` and `reduc_plus/max/min`), special idioms (such as `dot_product` and `widen_mult`), multiple data-types and type conversions (using `pack/unpack` and `cvt_intfp`), strided accesses (using `extract` and `interleave`), and optimized realignment (using `align_load`, `get_rt`, and `realign_load`), leveraging optimizations such as loop peeling and versioning,

Prototype	Description
`T, int m`	`T` is the scalar data type operated on by an operation; `m` denotes how many elements of type `T` can fit in a vector, and is equal to `VS/sizeof(T)`. If `T` is the smallest type operated on in the loop, then $m = VF$.
`int get_VF(T)`	Return the number of elements of type `T` that can fit in a vector register.
`vector init_uniform(T,val)`	Return a vector initialized to `m` copies of `val`.
`vector init_affine(T,val,inc)`	Return a vector initialized to (`val`,`val` + `inc`,`val` + 2`inc`,. . .,`val` + $(m-1)$`inc`).
`vector init_reduc(T,val,default)`	Return a vector initialized to (`val`, `default`, `default`,. . .,`default`), where the first vector element is set to `val`, and the remaining $m-1$ elements are set to `default`.
`scalar reduc_plus/max/min(T,v1)`	Compute the sum/maximum/minimum of the elements in `v1`, and return the (scalar) result.
`vector dot_product(T,v1,v2,v3)`	Elementwise widening multiplication of `v1`,`v2` and add the vector product to `v3`.
`vector widen_mult_hi/lo(T,v1,v2)`	Elementwise widening multiplication of the high/low halves of `v1`,`v2` to a vector of `m`/2 elements whose type size is 2`sizeof(T)`.
`vector pack(T,v1,v2)`	Copy the 2`m` elements in the concatenated `v1`,`v2` to a single vector, demoting these to a type whose size is `sizeof(T)`/2.
`vector unpack_hi/lo(T,v1)`	Copy the high/low half of `v1` to a vector of `m`/2 elements, promoting these to a type whose size is 2`sizeof(T)`.
`vector cvt_int2fp/fp2int(T,v1)`	Convert the `m` elements of `v1` from int (resp. float) to float (resp. int), and return the new vector.
`vector shift_right/left(T,v1,v2,val)`	If `val`=0, then the shift amounts for each element of `v1` are given in the respective element of `v2`. If `val` $\neq 0$, then `val` can be used as the same shift amount for all elements of `v1`.
`vector add/sub/mul/min/max(T,v1,v2)`	Elementwise addition/subtraction/multiplication/minimum/maximum of `v1`,`v2` and return the vector result.
`vector or/xor/and(T,v1,v2)`	Elementwise or/xor/and of `v1`,`v2` and return the vector result.
`vector extract(T,s,off,v1,v2,...)`	Extract the elements at strided locations (`off`+`s`,`off` + 2`s`,. . .,`off` + (`m` − 1)`s`) from the stream of elements formed by the concatenation of `v1`,`v2`,. . .
`vector interleave_hi/lo(T,v1,v2)`	Interleave the high/low half of `v1`,`v2`.
`vector aload(addr)`	Generate an aligned load from the address `addr` (that is guaranteed to be aligned).
`vector align_load(addr)`	Generate a load from the aligned address obtained from floor rounding `addr`. Used together with `realign_load`.
`get_rt(addr,mis,mod)`	Return a "realignment token" `rt`, be it a permutation vector, or bit mask, or a shift amount, etc., that is a function of the misalignment of `addr`, to be used by `realign_load` as explained below. `mis` and `mod` are as explained below.
`vector realign_ld(v1,v2,rt,addr,mis,mod)`	Extract `VF` elements from `v1`,`v2` as specified by `rt` (from `get_rt`). If realignment idioms are not supported, or if the load is aligned, values are loaded from `addr`, ignoring `v1`,`v2`,`rt`. The maximum alignment guaranteed is `mis%mod`.
`int get_align_limit(T)`	Return the alignment requirements (in number of elements of type `T`) for a vector of elements of type `T`.
`int loop_bound(vect_bound,scalar_bound)`	Return `vect_bound` (resp. `scalar_bound`) when the second compilation pass generates vectorized (resp. scalarized) code; tells whether a scalar peel loop should execute only `vect_bound` iterations or the entire `scalar_bound` iterations.
`bool version_guard_COND()`	A condition to control which vectorized version of a loop should be executed.

Table 1
Vector idioms

as well as outer-loop vectorization and straight-line code vectorization (SLP). Figure 3a demonstrates how the example in Figure 2 would be represented in our vectorized bytecode.

```
1. int vf = get_VF(fp);
2. float sum;
3. vfloat vsum=
      init_uniform(fp,0);
4. rt=get_rt(&a[2],8,32);
5. vfloat va= align_load(&a[0]);
6. for (i=0; i<n; i+=vf) {
7.   vb=align_load(&a[i+4]);
8.   vx=realign_load(va,vb,rt,
                &a[i+2],8,32);
9.   vsum=vadd(vx,vsum);
10.  va=vb;
11.}
12.sum= reduc_plus(vsum);
```

a. Vectorized bytecode

```
1. int vf = 1;
2. float sum;
3. float vsum;
4.
5.
6. for(i=0;i<n;i+=vf){
7.
8.   vx=load(&a[i+2]);
9.   vsum+=vx;
10.
11.}
12. sum = vsum;
```

b. Scalarized vector bytecode

Figure 3. Split vectorization scheme

B. First Offline Compilation Stage

We use GCC as the offline compiler, adjusting its multi-platform auto-vectorizer to generate the vectorized bytecode. This section reviews the main changes to an offline compiler required for this process.

a) Vector size: Auto-vectorizing compilers generally rely on the actual `VS` for their analysis and transformation. Some compilers are able to delay that to later stages of compilation keeping the early compilation stages entirely target-independent, with the `VS` abstracted [27]. GCC, however, relies on the value of `VS` early on, and has to be changed to express its transformation in terms of parametric `VS` and `VF`.

b) Dependences: Recall that loops with general loop-carried dependences can be vectorized only if the distance of these dependences is greater than or equal to `VF`. Therefore, the offline compiler cannot safely vectorize such loops without knowing the `VF`. In practice, most dependence distances are either very small (most commonly the distance is 1), in which case the loop cannot be vectorized on any platform, or the distance is practically infinite. A simple conservative approach is to refrain from (offline) vectorizing a loop with loop-carried dependences, regardless of their distances. This guarantees that the JIT compiler will always produce safe vectorized code for any `VS`, at the cost of potentially losing vectorization opportunities. Another approach is for the offline compiler to create two versions of the code: one scalar and one vectorized, controlled by a comparison of the dependence distance against the actual `VF`, much like the run-time aliasing checks that auto-vectorizing compilers already use when non-aliasing cannot be determined at compile time [2]. Passing dependence distance information to the JIT compiler is also possible by using a specialized idiom associated with the loop. This idiom indicates the largest admissible `VF` and relies on JIT scalarization (see Section III-C), rather than versioning. We implemented the former conservative approach, but we could easily incorporate dependence hints in our design.

c) Alignment: One of the simplest and most useful optimizations is to ensure that array bases are aligned on a VS boundary, which helps generate more efficient aligned accesses. However, in a split-compilation framework, the offline compiler does not know the VS and cannot assume that the online compiler or target platform can respect various alignment requirements it may request. The offline compiler must therefore assume that all array bases (and all memory accesses in general) are unaligned, and generate misaligned vector accesses accordingly. The goal is to generate the (unaligned) vectorized bytecode in a way that would allow the JIT compiler to generate aligned vector accesses in those cases in which the JIT compiler can align array addresses or determine (non-array) accesses to be aligned. Aligned accesses are generally more efficient; a split-compilation toolchain would not be competitive with native compilation if it could not facilitate the generation of aligned code.

Our framework incorporates loop-versioning and misalignment hints to solve this problem. The offline compiler computes misalignment information relative to a large modulo (currently set to 32 bytes, the largest SIMD width available today). This information is encoded in the realignment idiom `realign_load`, as illustrated in Figure 3a (which depicts the vectorized bytecode for the scalar C program in Figure 2a). The last two arguments are set to `mis`=8 bytes and `mod`=32 bytes. If the actual VS is no larger than 32 bytes, the misalignment information can be used. If further `mis` is divisible by VS, the JIT compiler can generate an aligned access. This example assumes that the base of array `a` is aligned, or can be forced to be aligned. (GCC indeed forces the alignment of global and local arrays when compiling natively for suitable targets.) When creating portable vectorized bytecode, however, the static compiler cannot assume that the target platform or JIT compiler can align arrays. Such code for aligned accesses must be guarded by an alignment check, coupled with a fall-back version in case the JIT compiler cannot guarantee the alignment of the array. In the fall-back version, the misalignment hints are nulled by setting `mod` to zero. When the offline compiler concludes that an access is aligned conditionally on its base being aligned, it can use the same loop versioning approach to generate aligned accesses. This way, one version of the loop could be vectorized with aligned accesses and/or misaligned accesses with misalignment hints, and the fall-back version of the loop would be vectorized with misaligned accesses and no hints.

There is an alternative approach that avoids creating two versions of the loop. As we show in the next section, our JIT compiler is able to seamlessly convert misaligned accesses into aligned accesses when possible. One additional hint is required, indicating whether the `mis` hint is valid only on the condition that the JIT compiler aligns arrays on the VS boundary. With this approach, the offline compiler always generates misaligned accesses that carry the extra hint, allowing JIT compilers capable of aligning arrays to convert the misaligned accesses into aligned ones.

One final alignment hint relates to loop peeling. Vectorizing compilers often peel loops to align memory accesses (most commonly stores). As a result, the original single scalar loop is transformed by the vectorizer into three loops: the scalar peel loop that iterates between 0 and VF-1 times — until the access becomes aligned, the main vector loop, and potentially another final scalar loop to handle remaining iterations. If the JIT compiler compiles vectorized bytecode for a target with no SIMD support, it would scalarize the vector loop. The overall effect amounts to replacing a single scalar loop by three loops, each with an unknown number of iterations, resulting in an overall performance degradation. To avoid this, we provide the `loop_bound` idiom, and use it to set the loop count of each of the three loops, such that in case of scalarization for non-SIMD targets, only one loop is executed. The semantics of the idiom are simple: the JIT compiler should use the `scalar_bound` argument when generating scalarized code and the `vect_bound` argument when generating vector code.

d) Loop selection and cost model: Our implementation supports cost-model guarded loop versioning to help the JIT compiler make efficient vectorization decisions, specifically in the context of doubly-nested loops. Suppose both inner and outer loops can be vectorized, and the offline compiler cannot decide which one is preferable without knowing the actual target. In such cases, the offline compiler will generate two versions: one vectorizing the inner loop and another vectorizing the outer loop. It also generates guard code (encoded in the idioms `version_guard_COND`) to test the actual features that should affect the final decision (e.g., availability of vector support for certain data-types or operations).

In summary, the analyses employed in the first offline stage, and especially those of vectorization safety, consist of standard offline methods of vectorizing compilers (dependence analysis, alias analysis). The novelty in the offline compiler lies in the interface to the back end, and specifically in (1) the special annotations and hints that are part of this interface to facilitate efficient code-generation by the JIT, and (2) the careful choices of how to vectorize the code, such that the JIT could then easily and quickly be able to adapt it to optimal code for each target, including non-SIMD targets. Leveraging a static offline auto-vectorization engine to enable JIT vectorization across multiple diverse SIMD platforms is the key innovation of the split approach.

C. Last, Online Compilation Stage

The online compiler is responsible for handling all target-specific transformations, including scalarizing when no (suitable) SIMD capabilities are available. The optimizations and hints encoded by the offline compiler aim to facilitate straightforward lightweight translation of vectorized bytecode into high-performance machine code, without further loop-level or data-access analyses at this stage. In this section we demonstrate how this is carried out by translating the vectorized code in Figure 3a to AltiVec, SSE, NEON, and a target without SIMD support.

a) Translating into vectorized code with explicit realignment: An example of assembly code with explicit realignment

is shown in Figure 2d. In this case the translation is a straightforward 1-to-1 mapping of each idiom with the respective target instruction, since the vectorized bytecode follows this approach exactly. For example, on AltiVec the `realign_load` is mapped to a `vperm` instruction, which takes two data vectors (`va,vb`) and a permutation vector (`rt`) generated by the `get_rt` idiom, which in turn is mapped to an AltiVec `lvsr` instruction. The fourth argument, `addr`, is ignored.

b) Translating into vectorized code with implicit realignment: Figure 2c shows an example of assembly code translated with implicit realignment. For targets that do not support explicit realignment, no code is generated for alignment-related idioms, except for `realign_ld`, which is mapped to a single misaligned load instruction (or sequence of instructions that implements a misaligned load), as supported by the target. If a target does not support misaligned loads, the code will be scalarized.

Translating the vectorized bytecode shown in Figure 3a to SSE is carried out as follows: when reaching instruction 4, the JIT compiler processes the misalignment hints and realizes it needs to generate unaligned code, because 8B%`VS`$\neq$ 0 having `VS`=16B. No code is generated for idioms `get_rt` and `align_load` (instructions 4,5,7). Then `realign_load` (instruction 8) is mapped to a misaligned vector load from address `addr`, the fourth argument of `realign_load`. (Other arguments are ignored.) Finally `get_VF` is materialized to 4 and propagated. Remaining idioms are translated into the respective vector instructions, as supported by the target.

c) Translating into vectorized aligned code: This refers to generating the assembly code shown in Figure 2b. When the JIT compiler can arrange for the arrays in question to be aligned, it can generate aligned vector accesses. It will then ignore explicit realignment idioms, as in the previous case (b), and will map the `realign_ld` to an aligned vector load instruction.

Translating the vectorized bytecode shown in Figure 3a into NEON is carried out as follows: when reaching instruction 4, the JIT compiler processes the misalignment hints and realizes that it can generate aligned code, because 8B%`VS`=0 (having `VS`=8B). It does not generate code for idioms `get_rt` and `align_load` (instructions 4,5,7). The `realign_load` (instruction 8) is mapped to an aligned vector load from address `addr`. Finally, `get_VF` is materialized to 2. The remaining idioms are translated into the respective vector instructions, as supported by the target.

d) Translating into scalarized code: Translating into scalar code, as shown in Figure 3b, is usually a straightforward mapping. Effectively using `VF=1`, each vector operation is converted into a corresponding scalar instruction. The challenge arises when the vectorized code involves overheads, such as data reorganization instructions and misalignment calculations. When translating to scalar code such overheads might incur penalties which are redundant for the scalar computation. The JIT compiler can remove some of this code by recognizing dead code. But in some cases, including that of realignment, the vectorization optimization introduces

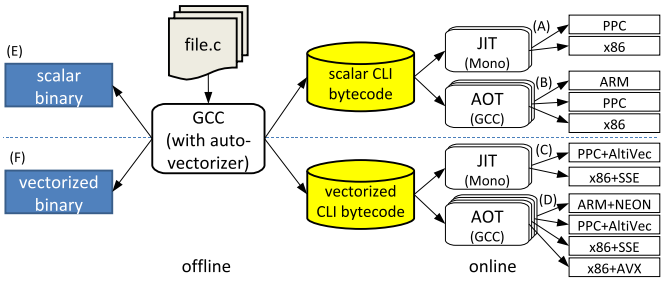


Figure 4. Interoperable compilation flows

cross-iteration dependencies (to minimize redundant loads and exploit data reuse). This may prevent some JIT compilers that are lacking the required loop analyses from removing all redundant code. Our design helps the compiler recognize such unneeded code and ignore it when generating scalar code.

Mapping the example bytecode shown in Figure 3a to a target that has no SIMD support is carried out in multiple steps. Idioms `get_rt` and `align_load` generate no code (instructions 4,5,7). The `realign_load` idiom (instruction 8) is mapped to a scalar load from address `addr`, its fourth argument. The compiler ignores the remaining arguments of `realign_load`. Finally, the compiler materializes `get_VF` to 1 and expands the remaining idioms to scalar code. This mapping is lightweight, resulting in high-quality scalar code, without introducing new overheads due to vectorization and scalarization.

IV. Experimental Environment

We prototyped our split-vectorization technique using GCC and Mono [14]. Figure 4 outlines several compilation flows enabled by the synergistic combination of these compilers and virtual machines. We used GCC's auto-vectorizer [16] to perform offline auto-vectorization of C source code and to emit CLI compliant bytecode with the gcc4cli CLI back-end [7]. Translating C to CLI notably results in no loss of semantic or metadata information, as CLI is a strongly typed format that retains high level information. The choice of compiler for the offline stage is not a key point, as one of the important advantages of split-vectorization is inter-operability, and as such, it can embrace auto-vectorizing compilers other than GCC. The empirical evaluation is of course limited by the capabilities of the offline compiler used, but our approach is more widely applicable. GCC has a recognizably advanced vectorization engine and can therefore facilitate a strong proof of concept for our approach, across a wide variety of advanced vectorization features.

We used Mono as the execution environment. Mono is a CLI-compliant virtual machine with a JIT compiler targeting x86 and PowerPC platforms, among others. We extended it to recognize our vector idioms and map them to equivalent SSE and AltiVec instructions. Mono is a significantly less mature compilation platform than GCC. Despite considerable effort spent improving it (which was not the focus of our work), the

generated code still suffers from poor register allocation, constant propagation, addressing, and other lacking optimizations. This is not inherent to JIT compilation, but relates specifically to Mono's current capabilities. Unfortunately, in some kernels, this offsets the vectorization impact.

To compensate for these phenomena, we also experimented with feeding the CLI bytecode to gcc4cli's experimental front end. This has several important benefits: (1) demonstrating the interoperability of our approach across different online compilers; (2) experimenting with four different platforms (rather than only two) and three different `VS`s; and (3) facilitating an apples-to-apples comparison using a state-of-the-art backend compiler for the online pass.

The CLI back end is based on GCC4.4, the front end is based on version 4.5, and Mono is based on a development branch (version 2.7). For AVX, we used a development branch of GCC4.6, which specifically targets 256-bit vectors.

A. Targets

a) SSE: Intel Core2 Duo E6850 @ 3 GHz: SSE is the SIMD instruction set provided by the x86 family of processors. It supports operations on 8-bit to 64-bit data types. SSE load instructions support misaligned memory accesses. Our target supports SSE, SSE2, SSE3, and SSSE3.

b) AltiVec: PowerPC G5 @ 2.3 GHz: AltiVec is the SIMD instruction set provided by the PowerPC family of processors. It supports operations on 8-bit to 32-bit data types — it does not support 64-bit operations. AltiVec memory accesses are limited to loading and storing from aligned memory addresses only.

c) NEON: ARM Cortex A8 @ 720 MHz: NEON is the SIMD instruction set designed for ARM processors. It provides both 64-bit and 128-bit arithmetic operations; our experiments use 64-bit mode to demonstrate the portability of our approach across distinct vector sizes. NEON supports both aligned and unaligned data accesses.

d) AVX: Intel Software Development Emulator version 3.09 and Intel Architecture Code Analyzer version 1.1.3: AVX is a new SIMD instruction set supporting 256-bit floating point vectors (single and double precision). Currently, no hardware is publicly available. To test and analyze our AVX code, we use Intel's Software Development Emulator (SDE) and Intel Architecture Code Analyzer (IACA) [9]. SDE emulates AVX code and validates program correctness. IACA computes a static evaluation of the cycles spent in a basic block, such as a loop body. In Table 3 we report the total throughput computed by IACA, which corresponds to the asymptotic number of cycles consumed by executing one iteration of the vectorized loop.

B. Benchmark Suites

We study two benchmark suites. The first is a collection of kernels that exercise automatic vectorization capabilities and highlight specific phenomena. The kernels were gathered from various application domains and benchmark suites, and include state-of-the-art vectorization features (see Table 2). In addition, we used the Polybench1.0 suite [20], consisting of 18 medium-size loop nests, showing the impact of our techniques on realistic algorithms from diverse application domains. We configured the Polybench suite to use floating point vectors and matrices of size 128 and 128^2, respectively, except for a 3-dimensional array of size 32^3 in `doitgen`. We manually applied some loop transformations to the baseline code to expose automatic vectorization opportunities: loop interchange and distribution, array layout transposition, and scalar promotion; `lu`, `ludcmp` and `seidel` require loop skewing to be vectorized, which unfortunately results in a control flow incompatible with the current auto-vectorizer.

Name	Description and features
dissolve_s8	Video image dissolve (widening multiplication)
sad_s8	Sum of absolute differences (abs pattern, reduction)
sfir_s16	Single sample finite impulse response (dot-product)
interp_s16	Rate 2 interpolation (strided access, dot-product)
mix_streams_s16	Mix four audio channels (SLP vectorization)
convolve_s32	2D convolution (reduction)
alvinn_s32fp	Weight-update for neural-nets training (outer-loop)
dct_s32fp	8X8 discrete cosine transform [12] (outer-loop)
dissolve_fp	Video image dissolve (constant)
sfir_fp	Single sample finite impulse response (reduction)
interp_fp	Rate 2 interpolation (strided access, reduction)
MMM_fp	Matrix multiplication
dscal_fp	Scale elements by constant (from BLAS)
saxpy_fp	Constant times a vector plus a vector (from BLAS)
dscal_dp	Scale elements by constant (from BLAS)
saxpy_dp	Constant times a vector plus a vector (from BLAS)
correlation, covariance	Datamining
2mm, 3mm, atax, gesummv, doitgen, gemm, gemver, bicg	Linear-algebra kernels
gramschmidt, lu, ludcmp	Linear-algebra solvers
adi, jacobi, seidel	Stencils

Table 2
Auto-vectorization kernels. Kernel names are suffixed by their data type, for example, s8 for signed chars, s32fp for kernels operating on both ints and single-precision floats

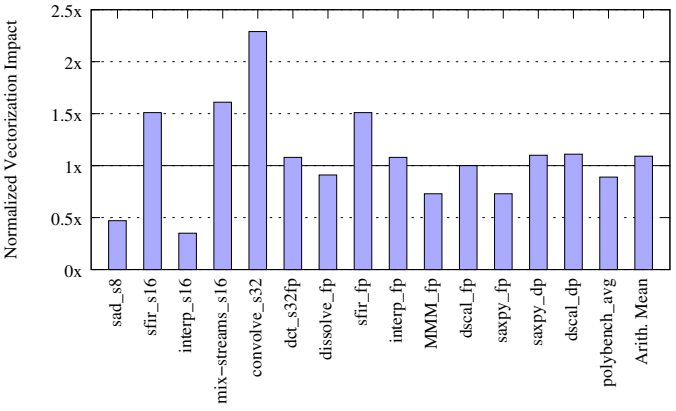
We did not experiment with benchmarks in which the vectorization potential was low (as in SPEC), since that would neither confirm nor invalidate the viability of our approach. Our goal was not to demonstrate the strength of GCC's vectorization capabilities per-se, but rather to demonstrate that our split approach can provide portability while exploiting the available hardware and providing the level of performance achieved by the native approach.

V. Experimental Evaluation

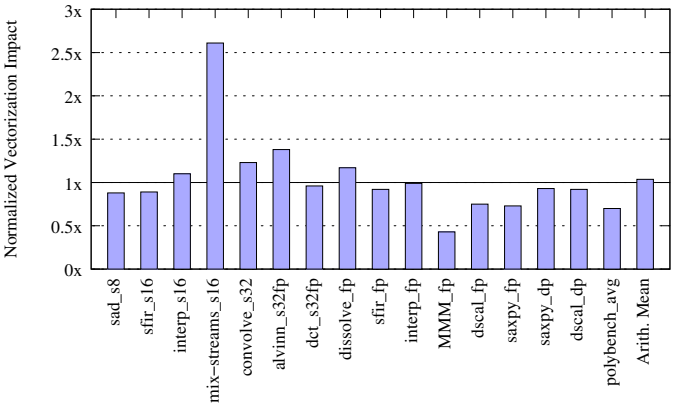
This section presents the experimental results we obtained using the two toolchains described above. We believe that this extensive experimentation across a diverse set of platforms, wide range of kernels and vectorization features, provides a robust proof of concept for split-vectorization in general, and as a means to achieve competitive JIT vectorization in particular.

A. Split vectorization using the Mono JIT

This section evaluates how our split-compilation approach supports lightweight and high-quality JIT vectorization.



(a) SSE (128-bit)



(b) AltiVec (128-bit)

Figure 5. Mono: normalized vectorization impact, Figure 4 ratio of (A/C)/(E/F), (higher is better)

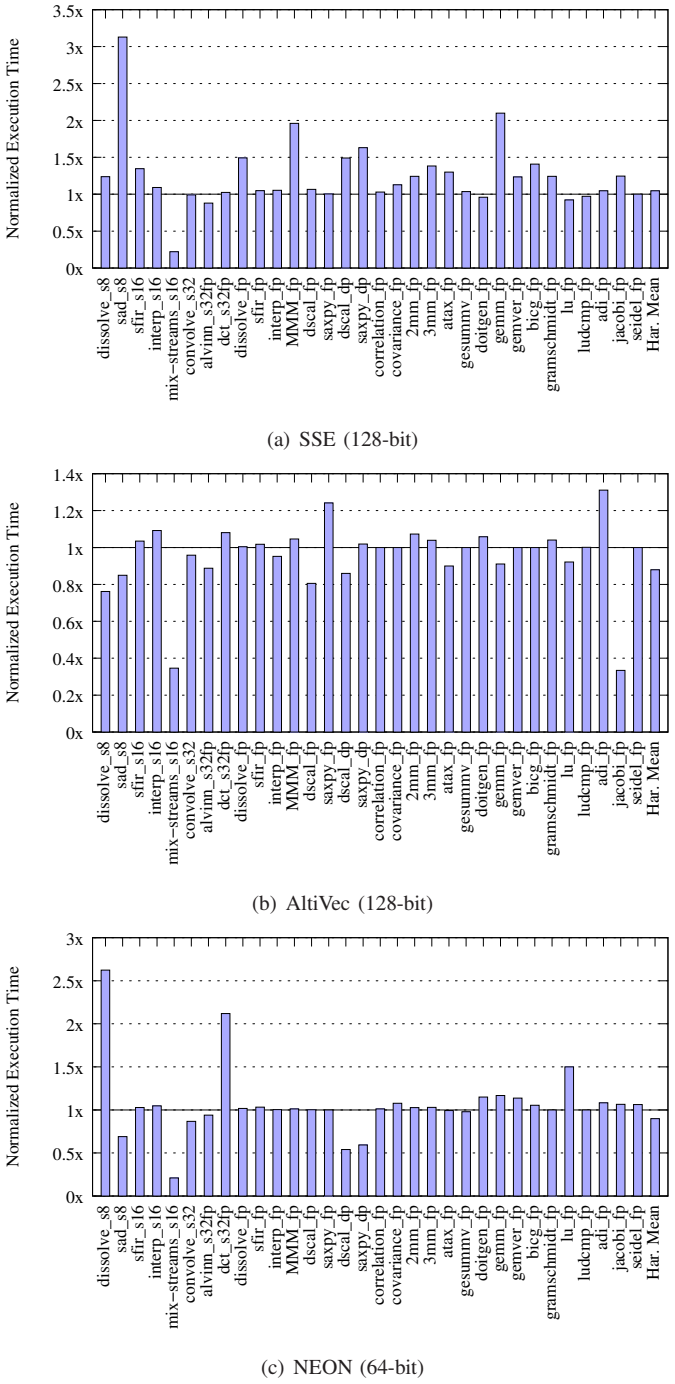
a) JIT vectorization impact: The performance improvements of split-vectorization using Mono on SSE and AltiVec are shown in Figure 5, relative to the improvements of native vectorization. The normalized vectorization impacts are calculated as $\frac{A/C}{E/F}$ (see A, C, E, F in Figure 4). All polybench kernels have a similar behavior, and we show the average across the entire suite. Two major factors affect the vectorization speedups obtained with Mono on x86: (1) lack of proper global register allocation, which results in frequent spills, and (2) use of the x87 floating point unit, which Mono does not optimize. These two factors often affect scalar code more than vectorized code (observe the overly high vectorization speedups in Figure 5a), but occasionally the vectorized code suffers more than the scalar code, resulting in lower vectorization speedups. Lack of global register allocation affects PowerPC code as well, but to a lesser degree: in Figure 5b, we observe a more homogeneous behavior in which most speedups are on average within 15% of native vectorization speedups. Two exceptions relate to the versioning technique of the split-compiler: in *MMM_fp*, Mono is unable to fold constants across a nested loop, so the alignment test is not resolved at compiled time and executed in each iteration of the outer loop. In *mix-streams*, versioning for alignment allows the JIT to generate only the aligned version, resulting in much better speedup than the unaligned natively-vectorized code.

b) Impact of compiler optimizations: Our framework devotes considerable attention to alignment optimizations and conveying hints to the JIT compiler (see Section III-B). To evaluate the importance of these optimizations, we repeated the above experiment with these optimizations and hints disabled. The impact was dramatic: lack of alignment optimizations resulted in misaligned accesses being generated by the JIT, and in some cases even resulted in falling back to scalar code when misaligned accesses are not supported. The average degradation factor is 2.5× across all benchmarks.

c) JIT compilation time: Vectorization is known to incur significant code size increase. The main reasons are: loop peeling to align memory accesses and number of iterations, extra code for realignment and other data reorganization dealing with strided accesses and type conversions, and loop-versioning for runtime aliasing or alignment checks. We observed a bytecode size increase of about 5×, on average, compared to unvectorized code across all kernels in both platforms. We observed a similar increase of 4.85×/5.37× in compile time on x86/PowerPC, respectively, confirming that JIT compilation time is proportional to the bytecode size. Overall, the JIT compile time remained negligible, and was of-course included in the execution times reported in Figure 5. The fact that the split/JIT vectorization impact is comparable to native vectorization impact implies that the overheads associated with dynamically-generating vectorized code are not much higher than the overheads associated with dynamically-generating scalar code, and in any case are not significant enough to offset the benefits from vectorization (JIT compile times are indeed very small, in the microsecond range).

B. Split vectorization using gcc4cli

We now demonstrate the portability of our split vectorization technique by experimenting with a broader range of targets. In Figure 6 we present the performance of split-vectorized code normalized according to the performance of natively-vectorized code compiled for SSE, AltiVec, and NEON. For all targets, we obtain harmonic means in the range of 0.8× to 1×. For these targets the majority of normalized execution times are close to 1×, with a few exceptions including *sad* and *mix-streams*, as explained below. These results show that the overall effect of split-compilation does not cause inherent slowdowns to generated vectorized executables and proves the viability of the split-vectorization approach. Of particular note, *dscal_dp* and *saxpy_dp* are scalarized on AltiVec as it lacks support for doubles. Scalarization hardly degrades performance on both toolchains, and with gcc4cli, the scalarized *dscal_dp* is even faster than the original natively compiled scalar code (due to improved addressing choices). The GCC back end for NEON is less mature than for SSE and AltiVec, and several kernels are not fully supported: *dissolve*



(a) SSE (128-bit)

(b) AltiVec (128-bit)

(c) NEON (64-bit)

Figure 6. gcc4cli: normalized vectorization times, Figure 4 ratio of (D)/(F), (lower is better)

and *dct*. These kernels fall back to library support for correct execution of vector idioms.

The small performance differences in native versus split vectorization are due to the continued GCC optimization in the online stages. Typical differences between the two toolchains are generally related to small code-generation details, such as addressing modes, loop induction variables, initializing vector constants inside or outside of loops, and register promotion. A couple of exceptions are *mix-streams* and *sad*. In *mix-streams*, the split-vectorized version is particularly improved by the versioning that takes place in the split vectorization toolchain. This results in executing an aligned loop version, compared to the native compiler which generates a misaligned version only. Usually, the potential adverse effect of versioning is either negligible or can be altogether avoided in cases when the online compiler is able to resolve the condition and generate the single appropriate version. When that is not the case (e.g., *sad*), performance is degraded.

kernel	Cycles per iteration	
	native	split
dissolve_fp	2	3
sfir_fp	2	4
interp_fp	4	6
MMM_fp	1	2
saxpy_fp	2	2
dscal_fp	2	3
saxpy_dp	2	3
dscal_dp	2	3

Table 3
IACA simulation for AVX

Table 3 presents simulated AVX results for a subset of our kernel suite. The numbers, reported by IACA, measure straight-line pieces of code (the kernel loop). All kernels have a `VF` consistent with the data type (eight for single precision computations, four for double precision). Table 3 shows the number of cycles per loop iteration. We observed similar differences as with other targets using the GCC toolchain, including the choice of induction variables, the addressing modes, and the lack of register promotions of the accumulator in reduction kernels. This discrepancy is further explained by the different versions of GCC used for the native AVX flow: for native compilation of AVX, we used the AVX development branch of GCC (best GCC-based AVX code-generator available to us), which is different than the GCC we used to generate split code. This is the only case in which we could not use the same compiler for both sides of the experiment, and it is also the main reason for the degradations shown in Table 3. The code generators of the two compilers make different decisions, in particular for induction variables and addressing modes. These differences are not related to the split compilation approach.

VI. Related Work

Our recent work [22] describes the use of a combined static- dynamic infrastructure for vectorization, focusing on the ability to revert efficiently and seamlessly to generate scalar instructions when the JIT compiler or target platform do not support SIMD capabilities. In contrast, the present work focuses on providing an infrastructure capable of supporting diverse SIMD targets, across a wide range of vectorizable kernels, with performance comparable to monolithic compiler vectorization. Clark et al. take a different approach in their Liquid SIMD method [5]. There, a static compiler auto-vectorizes the code, but then scalarizes it to emit standard scalar instructions. The scalar patterns are later detected by

the hardware, if equipped with suitable SIMD capabilities, resurrecting vector instructions and executing them. Pajuelo et al. [19] propose a purely hardware solution that detects strided loads in scalar code and speculatively generates vector instructions. We also use a static auto-vectorizing compiler, but instead of confining it to a scalar ISA (thereby limiting the vectorizable patterns), we expand the IR to convey additional, explicit information from the static compiler to the JIT.

Bocchino and Avde in their work Vector LLVA [3] propose a common vector format that can be automatically translated to different platforms, focusing on manual programming using their format, and demonstrating the translation of a few kernels to AltiVec, SSE, and RSVP. As in LLVA, Intel's Array Building Blocks [4] also introduces general vector idioms to support manual vectorization. While these works focus on target-agnostic manual vectorization, our work focuses on a target-agnostic auto-vectorization engine. In that sense, the two approaches are orthogonal and complementary. Our approach is especially advantageous over the manual approach in situations involving several alternatives for vectorizing the code, each preferable for a different SIMD taregt. A programmer will pre-determine one vectorization alternative when vectorizing the code manually, whereas using the offline auto-vectorizer, as in our approach, the generated bytecode will convey enough information to allow the JIT to choose the right vectorization scheme for a given SIMD platform.

Attempting to apply binary translation technology to migrate assembly code, including SIMD instructions, over to a markedly different architecture at runtime suffers from several difficulties stemming from lack of type information [13]. Our aim is to leverage higher abstractions to achieve portability of SIMD code, coupled with harnessing static vectorizing compilers. Recent work integrates initial automatic vectorization capabilities in JIT compilers for Java [15]. Our focus is to provide portable advanced vectorization of static programs across many targets, leveraging aggressive offline processing.

VII. Conclusions

We present a split auto-vectorization framework based on interoperable tools and a standard intermediate language (CLI), targeting diverse SIMD architectures with different vector sizes and idioms. To our knowledge, this is the first framework combining sophisticated auto-vectorization with JIT compiler technology competitive with native optimization. We adapt state-of-the-art vectorization algorithms, operating static and dynamic analysis, program transformation, and decision heuristics synergistically, across offline and online compilation stages. Our experiments with GCC demonstrate that split vectorization is competitive with native, offline compilation, while offering transparent performance-portability of a single bytecode representation. Our experiments with GCC and Mono demonstrate that strong speedups can be obtained from auto-vectorization in a resource-constrained JIT compiler. In the future, we wish to extend our framework to take full advantage of online compilation, leveraging dynamic context and workload information for improved specialization.

Acknowledgements

This work was partly supported by the European Commission through the HiPEAC network of excellence, and by the French Ministry of Industry and STMicroelectronics through the Mediacom grant. Our work benefitted from numerous discussions with Sami Yehia, and stems from the pioneering work and vision of Marco Cornero and Benoît Dupont de Dinechin. We are also thankful to Andrea Ornstein for supporting the gcc4cli backend.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[2] A. Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE*, pages 46–56, 2006.

[4] C. J. Newburn et al. Intel array building blocks: a retargetable, dynamic compilation framework. In *CGO*, 2011.

[5] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA'07*, pages 216–227, Washington, DC, USA, 2007.

[6] A. Cohen and E. Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *DAC*, pages 102–107, June 2010. Special session on Embedded Virtualization.

[7] R. Costa, A. C. Ornstein, and E. Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.

[8] A. E. Eichenberger, P. Wu, and K. O'brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI*, June 2004.

[9] Intel Corporation. *Intel Architecture Code Analyzer – User's Guide*, 1.1.2 edition, 2009.

[10] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.

[11] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *PACT*, Sept. 2002.

[12] C. G. Lee. UTDSP benchmarks. http://www.eecg.-toronto.edu/ corinna/DSP/infrastructure/UTDSP.html, 1998.

[13] J. Li, Q. Zhang, S. Xu, and B. Huang. Optimizing dynamic binary translation for SIMD instructions. In *CGO*, 2006.

[14] The Mono Project. `http://www.mono-project.com`.

[15] J. Nie, B. Cheng, S. Li, L. Wang, and X.-F. Li. Vectorization for Java. In *NPC*, pages 3–17, 2010.

[16] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, 2006.

[17] D. Nuzman and A. Zaks. Autovectorization in GCC – two years later. In *GCC Developer's summit*, June 2006.

[18] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short SIMD architectures. In *PACT*, Oct. 2008.

[19] A. Pajuelo, A. González, and M. Valero. Speculative dynamic vectorization. In *ISCA*, pages 271–280, 2002.

[20] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC*, 2010.

[21] G. Ren, P. Wu, and D. Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *LCPC*, Oct. 2003.

[22] E. Rohou, S. Dyshel, D. Nuzman, I. Rosen, K. Williams, A. Cohen, and A. Zaks. Speculatively vectorized bytecode. In *HiPEAC*, Heraklion, Greece, Jan. 2011.

[23] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, Sept. 2002.

[24] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, Mar. 2005.

[25] K. Trifunović, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, Sept. 2009.

[26] P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment. In *CGO*, Mar. 2005.

[27] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *ICS*, June 2005.