Runtime Vectorization Transformations of Binary Code

Nabil Hallou, Erven Rohou, Philippe Clauss

Noname manuscript No.
(will be inserted by the editor)

Runtime Vectorization Transformations of Binary Code

Nabil Hallou · Erven Rohou · Philippe Clauss

Received: date / Accepted: date

Abstract In many cases, applications are not optimized for the hardware on which they run. Several reasons contribute to this unsatisfying situation, such as legacy code, commercial code distributed in binary form, or deployment on compute farms. In fact, backward compatibility of ISA guarantees only the functionality, not the best exploitation of the hardware. In this work, we focus on maximizing the CPU efficiency for the SIMD extensions. The first contribution was originally published in the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS XV, Jul 2015, Agios Konstantinos, Greece. It is a binary-to-binary optimization framework where loops vectorized for an older version of the processor SIMD extension are automatically converted to a newer one. It is a lightweight mechanism that does not include a vectorizer, but instead leverages what a static vectorizer previously did. We show that many loops compiled for x86 SSE can be dynamically converted to the more recent and more powerful AVX; as well as, how correctness is maintained with regards to challenges such as data dependencies and reductions. We obtain speedups in line with those of a native compiler targeting AVX. The second contribution is the runtime vectorization of loops in binary codes that were not originally vectorized. For this purpose, we use open source frameworks that we have tuned and integrated to (1) dynamically lift the x86 binary into the Intermediate Representation form of the LLVM compiler, (2) abstract hot loops in the polyhedral model, (3) use the power of this mathematical framework to vectorize them, and (4) finally

N. Hallou and E. Rohou
Inria
Rennes, France
E-mail: {nabil.hallou, erven.rohou}@inria.fr

Ph. Clauss
Inria
ICube lab., University of Strasbourg, France
E-mail: philippe.clauss@inria.fr

compile them back into executable form using the LLVM Just-In-Time compiler. In most cases, the obtained speedups are close to the number of elements that can be simultaneously processed by the SIMD unit. The re-vectorizer and auto-vectorizer are implemented inside a dynamic optimization platform; it is completely transparent to the user, does not require any rewriting of the binaries, and operates during program execution.

Keywords Performance · Runtime optimization · Binary code · Dynamic binary optimization · Vectorization · Polyhedral model

1 Introduction/Motivation

Automatic code optimizations have traditionally focused on source-to-source and compiler Intermediate Representation (IR) transformation tools. Sophisticated techniques have been developed for some classes of programs, and rapid progress is made in the field. However, there is a persistent hiatus between software vendors having to distribute generic programs, and end-users running them on a variety of hardware platforms, with varying levels of optimization opportunities. The next decade may well see an increasing variety of hardware, as it has already started to appear particularly in the embedded systems market. At the same time, one can expect more and more architecture-specific automatic optimization techniques.

Unfortunately, many "old" executables are still being used although they have been originally compiled for now outdated processor chips. Several reasons contribute to this situation:

- commercial software is typically sold without source code (hence no possibility to recompile) and targets slightly old hardware to guarantee a large enough base of compatible machines;
- though not commercial, the same applies to most Linux distributions[1] – for example Fedora 16 (released Nov 2011) is supported by Pentium III (May 1999)[2];
- with the widespread cloud computing and compute servers, users have no guarantee as to where their code runs, forcing them to target the oldest compatible hardware in the pool of available machines.
- some compilers are capable of performing some kind of optimizations that others cannot.

All this argues in favor of binary-to-binary optimizing transformations. Such transformations can be applied either statically, i.e., before executing the target code, or dynamically, i.e., while the target code is running.

Dynamic optimization is mostly addressing adaptability to various architectures and execution environments. If practical, dynamic optimization

[1] with the exception of Gentoo that recompiles every installed package

[2] `http://docs.fedoraproject.org/en-US/Fedora/16/html/Release_Notes/sect-Release_Notes-Welcome_to_Fedora_16.html`

should be preferred because it eliminates several difficulties associated with static optimization. For instance, when deploying an application in the cloud, the executable file may be handled by various processor architectures providing varying levels of optimization opportunities. Providing numerous different adapted binary versions cannot be a general solution. Another point is related to interactions between applications running simultaneously on shared hardware, where adaptation may be required to adjust to the varying availability of the resources. Finally, most code optimizations have a basic cost that has to be recouped by the gain they provide. Depending on the input data processed by the target code, an optimizing transformation may be or not profitable.

In this paper, we target SIMD ISA extensions and in particular the x86 SSE and AVX capabilities. On the one hand, AVX provides wider registers compared to SSE, new instructions, and new addressing formats. AVX has been first supported in 2011 by the Intel Sandy Bridge and AMD Bulldozer architectures. However, most existing applications take advantage only of SSE and miss significant opportunities of performance improvement. On the other hand, even if in theory it could take advantage of vectorization, a loop may be left unvectorized by the compiler. This might happen when using an outdated version of a compiler which does not support vectorization, or when the compiler is unable to analyze the data dependencies or transform the code for ensuring correct vectorization. This paper shows that it is possible, at runtime, to (1) automatically convert SSE-optimized binary code to AVX, as well as (2) auto-vectorize binary code whenever profitable. The key characteristics of our approach are:

- we apply the transformation at runtime, i.e. when the hardware is known;
- we only transform hot loops (detected through very lightweight profiling), thus avoiding useless work and minimizing the overhead;
- for SSE loops, we do *not* implement a vectorization algorithm in a dynamic optimizer. Instead, we recognize already statically vectorized loops, and convert them to a more powerful ISA at low cost;
- for scalar (unvectorized) loops, we integrated some open source frameworks to lift the binary code into the IR form of the LLVM compiler, auto-vectorize them on the fly, and compile them back using the LLVM Just-In-Time (JIT) compiler.

Note that the two presented techniques are not meant to be integrated. The first technique is the closest to a production solution: it is self-contained and relies on a single technology. The second technique is more exploratory. It relies on various existing tools that we adapted to fit our needs. Our goal is to demonstrate its feasibility, and to assess its potential performance.

Section 2 reviews the necessary background on the two key technologies at play: vectorization and dynamic binary optimization. Section 3 presents our first contribution of translating on-the-fly SSE-optimized binary code into AVX. Section 4 addresses our second contribution of runtime auto-vectorizing of originally unvectorized loops. Our experiments are presented in Section 5. Section 6 discusses related work. Section 7 concludes and draws perspectives.

Scalar version

```
int A[], B[], C[];
...
for(i=0; i<n; i++) {
  a = A[i];
  b = B[i];
  c = a+b;
  C[i] = c;
}
```

Vectorized version

```
int A[], B[], C[];
...
/* vectorized loop */
for(i=0; i<n; i+=vf) {
  va = A[i..i+vf[;
  vb = B[i..i+vf[;
  vc = padd(va, vb);
  C[i..i+vf[ = vc;
}
/* epilogue */
for( ; i<n; i++) {
  /* remaining iterations
     if n not multiple of vf */
}
```

Fig. 1 Vector addition (pseudo code)

2 Background

2.1 Vectorization at a glance

The incorporation of vector units into modern CPUs extended the instruction set with SIMD instructions. These instructions operate on vector operands containing a set of independent data elements. They include wide memory accesses as well as so-called *packed* arithmetic. In brief, the same operation is applied in parallel to multiple elements of an array. Figure 1 depicts the pseudo code of sequential and vectorized versions of an addition of arrays C=A+B. Variables `va`, `vb`, and `vc` denote vectors, `padd` stands for *packed add*, a simultaneous addition of several elements. The number of elements processed in parallel is the *vectorization factor* (vf). In the example of Figure 1, elements are of type `int`, i.e. 32-bit wide, SSE vectors are 128 bits, the vectorization factor is $vf = 128/32 = 4$.

The vectorized loop is faster because it executes fewer iterations (the scalar loop iterates n times; meanwhile, the vectorized one iterates $\lfloor \frac{n}{vf} \rfloor$ times), fewer instructions, and fewer memory accesses (accesses are wider but still fit the memory bus width: this is advantageous).

Over time, silicon vendors have often developed several versions of SIMD extensions of their ISAs. The Power ISA provides AltiVec, and the more recent VSX. Sparc defined the four versions VIS 1, VIS 2, VIS 2+, and VIS 3. x86 comes in many flavors: starting with MMX, ranging to different levels of SSE, and now AVX, AVX2, and AVX-512.

Compared to SSE, AVX increases the width of the SIMD register file from 128 bits to 256 bits. This translates into a double vectorization factor, hence, in ideal cases, double asymptotic performance.

2.2 Dynamic Binary Optimization

Dynamic binary optimization aims at applying optimizing transformations either at program load time or during program execution, with no access to source code or any form of intermediate representation. In this study, we use the Padrone platform [22], which integrates several generic and architecture-specific binary analysis and manipulation techniques. Padrone services can be roughly divided into three major categories: 1) profiling, 2) analysis, and 3) optimization, which are briefly described here.

The *profiling* component leverages the Linux performance events subsystem to access hardware performance counters, using the *perf_event* system calls. The included low-cost sampling technique provides a distribution of program counter values, which can then be used to locate *hot spots*.

The *analysis* component accesses the program's code sections, parses binary code to build a control-flow graph (CFG), and locates loops inside functions whenever possible. The analysis is completely static in our case (i.e., does not use dynamic control flow). Re-vectorization happens on this reconstructed CFG. This latter is further lifted into the IR form for auto-vectorization.

Padrone has the same limitations as other binary re-writers working on x86 instruction sets. In particular, disassembling binary code may not always be possible. Reasons include indirect jumps, presence of *foreign bytes*, i.e. data in the middle of code sections, self rewriting code, or exotic code generation schemes, such as those used in obfuscated code. Errors are of two kinds: parsing errors and erroneous disassembly. We easily handle the former by aborting the transformation. The latter is beyond the scope of this paper. Quite a few techniques have been proposed in the literature, interested readers can refer to a survey in [24]. The nature of our target program structures (vectorized loops) are almost never subject to these corner cases.

The *optimization* component provides basic binary code manipulation utilities, as well as a code-cache injection mechanism. In this study, this component was used mainly to regenerate code from the internal representation of a vectorized loop after transformation. This component takes care of subtle x86 idiosyncrasies, such as the size of the offsets and the reachability of jump instructions, the updating of IP-relative addresses, or the addition of VEX prefixes on remaining SSE instructions after upgrading to AVX (to avoid the SSE/AVX transition penalty – see §11.3 of Intel Optimization Manual [1]), to mention a few.

Padrone runs as a separate process, that interacts with its target thanks to `ptrace` system call and other Linux features such as the `/proc` filesystem. The target does not need any change, in particular it may be an unmodified commercial workload. The optimizer also operates on running applications, which do not need to be restarted. It monitors the program's execution, detecting hot spots, selecting SSE-vectorized loop, and providing a corresponding CFG. After re-vectorization, or auto-vectorization, have been performed, Padrone is responsible for installing an updated version and redirecting the execution.

```
1  .L2:
2    movaps  A(rax),xmm0
3    addps   B(rax),xmm0
4    movaps  xmm0,C(rax)
5    addq    $16,rax
6    cmpq    $4096,rax
7    jne     .L2
```

(a) Original SSE

```
.L2:
  vmovaps     A(rax),xmm0
  vinsertf128 1,A(rax,16),ymm0
  vaddps      B(rax),ymm0
  vmovaps     ymm0,C(rax)
  addq        $32,rax
  cmpq        $4096,rax
  jne         .L2
```

(b) Resulting AVX

Fig. 2 Body of vectorized loop for vector addition

3 Re-Vectorization of Binary Code

3.1 Principle of the SSE into AVX translation

Our goal is to transform loops that use the SSE instruction set to benefit from a CPU that supports AVX technology. Since the binary is already vectorized, we are concerned only with the *conversion* of instructions from SSE into AVX, and some *bookkeeping* to guarantee the legality of the transformation.

At this time, our focus is on inner loops, with contiguous memory accesses. Future work will consider outer-loop vectorization and strided accesses.

The main advantage of using AVX over SSE instruction set is that the size of the vector operand doubles from 128 bits into 256 bits. Therefore, the number of data elements that fits into the SSE vector operand doubles in the AVX one. Hence, in the perfect scenario, an AVX loop runs half the number of iterations of a SSE loop.

As we operate in a dynamic environment, we are constrained to lightweight manipulations. The key idea is to leverage the work already done by the static compiler, and to *tweak* the generated SSE code and adjust it to AVX, while we must guarantee that the generated code is correct and semantically equivalent to the original code.

Figure 2 illustrates the loop body of the vector addition (from Figure 1) once translated into SSE assembly, and how we convert it to AVX. Register `rax` serves as a primary induction variable. In SSE, the first instruction (line 2) reads 4 elements of array A into `xmm0`. The second instruction (line 3) adds in parallel 4 elements of B, and the third instruction (line 4) stores the results into 4 elements of C. The induction variable is then incremented by 16 bytes: the width of SSE vectors (line 5).

Converting the body of the loop is relatively straightforward. But to guarantee the legality of the transformation, some bookkeeping is necessary. It is detailed in the enumeration below, items 2 to 6, and further developed in Sections 3.3 to 3.7. The key parts of our techniques are the following:

1. convert SSE instructions into AVX equivalents (cf. Section 3.2);
2. restore the state of `ymm` registers in the (unlikely) case they are alive (cf. Section 3.3);
3. adjust the stride of induction variables (cf. Section 3.4);

4. handle loop trip counts when not a multiple of the new vectorization factor (cf. Section 3.5);
5. enforce data dependencies (cf. Section 3.6);
6. handle alignment constraints (cf. Section 3.7);
7. handle reduction (cf. Section 3.8).

3.2 Converting instructions from SSE into AVX

The optimization consists of translating a packed SSE SIMD instruction into an AVX equivalent, i.e. an instruction (or sequence of instructions) that has the same effect, but applied to a wider vector. When it is impossible to decide whether the conversion maintains the semantics, the translation aborts.

In most practical cases, there is a one-to-one mapping: a given SSE instruction has a direct AVX equivalent. These equivalents operate on the entire 256 bits vector operand. However, in some cases, an SSE instruction needs a tuple of instructions to achieve the same semantics. These equivalents operate on halves of the 256-bit vector operand.

Furthermore, an SSE instruction might have several alternatives of equivalent instructions because of alignment constraints, notably the SSE aligned data movement instruction `movaps`. It moves a 16-byte operand from or into memory that is 16-byte aligned. The conversion of this proposes two alternatives: on one hand, when data to be moved is 16-byte aligned, it is replaced by two instructions. Primary `vmovaps` requires data to be aligned on 16 bytes and moves only the lower half of the vector operand. Secondary, a `vinsertf128` that moves the upper half. Figure 2 shows that the instruction in line 2 in the original SSE is translated into instructions in lines 2 and 3 in the resulting AVX. We assume that array A is 16-byte aligned. On the other hand, when the data to be moved happens to be 32-byte aligned, an aligned instruction `vmovaps` that moves the whole vector operand at once is selected. Assuming that array C, in Figure 2, is 32-byte aligned, the 16-byte memory access (in line 4) is converted into a 32-byte `vmovaps`. Accordingly, the registers are converted from `xmm` to `ymm`. Finally, the translator encodes the translated instructions. Table 1 summarizes our instruction conversion map from SSE to AVX.

Finally, there are also opportunities to map several SSE instructions to a single AVX instruction. For example, AVX provides a fused multiply-add instruction that provides additional benefits (and also semantics issues due to floating point rounding). The new three-operand format also opens the door to better code generation. These elaborated translations are left for future work.

3.3 Register liveness

A register is said to be alive at a given program point if the value it holds is read in the future. Its live range begins when the register is first set with a specific value and ends when the value is last used. Registers can be alive for

Table 1 SSE-to-AVX Instruction Conversion Map

SSE	AVX
movlps	vmovaps (only when combined with movhps, and data are aligned on 16 bytes)
movlps	vmovups (when data are not aligned)
movhps	vinsertf128 (only when combined with movlps, from memory to register)
movhps	vextractf (movement from register to memory)
movaps	vmovaps (when data are aligned on 32 bytes)
movaps	vmovaps (aligned on 16 bytes) + vinsertf128
shufps	vshufps + vinsertf128
xorps	vxorps
addps	vaddps
subps	vsubps
mulps	vmulps

long sequences of instructions, spanning the whole loop nests we are interested in. Liveness analysis requires a global data flow analysis.

Our optimizer only impacts the liveness of `ymm` registers. For all other registers, we simply update an operand of instructions, such as the stride of induction variables, which has no impact on live ranges. Registers `ymm` must be handled in a different way. They are not used in the loops we consider, and SSE code is unlikely to make any use of them. Still, there might be situations where SSE and AVX code coexist (such as hand written assembly, use of third party libraries, or ongoing code re-factoring – see §11.3 in [1]). Our optimizer cannot run any interprocedural liveness analysis and has only a local view of the program. We cannot make any assumption on the liveness of these registers. Our straightforward solution is to spill the registers we use in the preheader of the loop (created as needed), and restore them after the loop.

The situation is actually slightly more complicated because `ymm` is an extension of `xmm`, where `xmm` denotes the 128 least significant bits of `ymm`. We only restore the 128 most significant bits to guarantee a consistent state.

3.4 Induction variables

In the scenario where arrays are manipulated, a register is typically used as an index to keep track of the memory access of the iteration. For instance, the vector addition example in Figure 2 (a) makes use of register `rax` for two purposes: first, as a counter of iterations and second as an array index. In lines 2, 3, and 4 it is used as an index with respect to the base addresses of arrays A, B, and C from and to which the reads, and writes should be performed. Since the SSE data movement instructions operate on 16 bytes operands, the register is incremented by 16 bytes in line 5.

When the code is translated into AVX, the operand's size changes. Hence, the instruction responsible for incrementing the index should be adapted to the AVX version. In other words, the pace of change becomes 32 bytes instead of 16 bytes. Line 5 in Figure 2 (b) illustrates the modification performed by the translator for adapting indices to the optimized version.

3.5 Loop bounds

Replacing vector instructions by their wider instructions requires an adjustment of the total number of loop iterations. In the case of translating SSE into AVX SIMD instructions, the size of the vector operands is doubled. Hence, a single AVX iteration is equivalent to two consecutive SSE iterations.

The translator handles loop bounds by classifying vectorized loops into two categories: loops with a number of iterations known at compile-time; and loops where the bounds are only known at run-time.

3.5.1 Loop bound known at compile-time

When the total number of iterations is known at compile-time, the translator has two alternatives: either replace the total number of iterations by its half value or double the increment of the loop counter. It opts for the second choice, since the loop counter may serve as an array index at the same time, whose pace of change is doubled as discussed in the previous subsection. Line 5 in Figure 2 (b) illustrates the transformation.

The number of iterations of the loop with SSE SIMD instructions can be either even or odd. When it is even, $n = 2 \times m$, the optimizer simply translates SSE into AVX instructions and doubles the pace of change of the counter so that the transformed loop iterates m times. However, when the number of iterations is odd, $n = 2 \times m + 1$, the transformed code is composed of two basic blocks: primary, the loop that runs AVX instructions m times. Secondary, the SSE instructions that run the last SSE iteration. The latter instructions are vex.128[3] encoded to avoid SSE/AVX transition penalty.

3.5.2 Loop bound known only at run-time

This happens, for instance, when n is a function parameter, or when an outer loop modifies the bound value of the inner one. Suppose that a sequential loop executes a set of instruction n times, and its vectorized version executes equivalent instructions in m iterations such that: $n = m \times vf + r$, where r is the number of remaining sequential iterations ($r < vf$) that obviously cannot be performed at once.

Under such circumstances, the compiler generates a vectorized loop that iterates m times, and a sequential one that iterates r times. The values of m and r are calculated earlier, before control is given to one of the loops or both of them depending on the values of n discovered at runtime. It is also possible that the static compiler unrolls the sequential loop. Currently, this latter situation is not handled.

For a correct translation into AVX, an adjustment of the values of m and r is required. Let us consider the vectorized loop L2 (see Figure 3). The loop has

[3] vex.128 instructions are enhanced versions of legacy SSE instructions that operate on lower half of `ymm` registers and which zero the upper half.

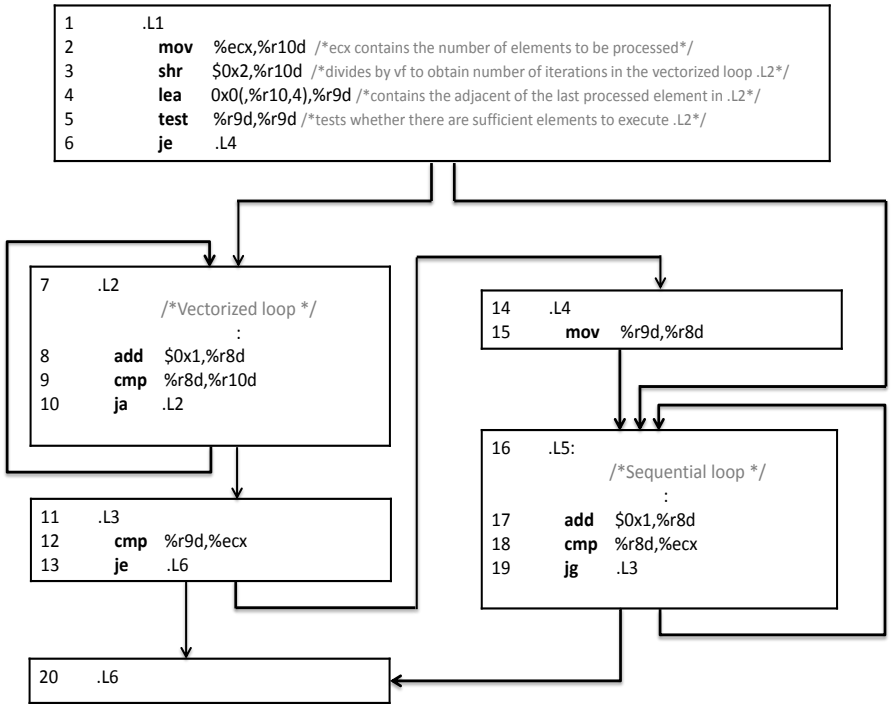


Fig. 3 Pattern of code when the number of iterations is known only at runtime

```
for(i=0; i < n-4; i++)
  A[i+4] = A[i] + 1;
```

Fig. 4 Example of loop-carried dependence

a bound register that is compared to the counter in line 9. The bound register holds the value m, that is computed earlier in line 3. This instruction initializes the bound register to a value equal to $\lfloor n/vf \rfloor$. The translator modifies this instruction to initialize it to $\lfloor n/(2 \times vf) \rfloor$.

Regarding the sequential loop L5, it iterates r times. The compiler initializes the counter by a value equal to $n-r$ which is also equal to $vf \times m$ (in line 15). Therefore, the loop iterates from $n-r$ to n for a total of r iterations. The translator traces the counter register `r8d` (in line 18). It finds that it received the contents of `r9d` (in line 15). It continues tracing `r9d` until line 4 where this latter is initialized by the value $m \times vf$. The translator then changes the initialization into $2 \times m \times vf$.

3.6 Aliasing and Data Dependencies

3.6.1 Overview of aliasing

Aliasing is the situation where multiple instructions access the same memory location. Dependence happens when at least one of the instructions is a write, as in the following code:

```
A[0] = B[0];   // writes into location A[0]
B[1] = A[0];   // reads from A[0]
```

There are three kinds of dependence: read-after-write, write-after-read, and write-after-write. The first one, also known as true dependence, forces the code to execute sequentially and the others allow code to be parallelized, subject to some slight code transformations. Sticking to the same example, it is not possible to execute both instructions at once; since, the second instruction that reads A[0] should wait until the first instruction writes into A[0].

A loop-carried dependence describes the situation where the dependence occurs between iterations of a loop. For instance, in Figure 4, a write into A[4] occurs at the first iteration, and a read from the same location occurs four iterations later. Considering the true dependence, the distance vector is the number of iterations between successive accesses to the same location. In this example, the distance is $d = (i + 4) - i = 4$. Therefore, executing up to four iterations in parallel is allowed; as well as, vectorizing up to four elements. However, vectorizing more than four elements violates the data dependence.

3.6.2 Issue of translating a loop with data dependencies

A blind widening of vectors from SSE's 128 bits to AVX's 256 bits might violate data dependencies. It may happen because this doubles the vectorizing factor, hence runs the risk to exceed the dependence distance.

3.6.3 Static interval-overlapping test

Static interval-overlapping test refers to a lightweight verification done by the translator at compile time (no code is generated) to ensure that the translation will not cause a dependence violation. The test consists of comparing the set of addresses touched by the new AVX SIMD instructions against all addresses of the remaining new AVX SIMD data movement instructions of the loop. A non-empty intersection signals an attempt to translate more elements than the dependence distance. Consequently, the translation process is aborted. The static test occurs in the following scenarios: when the addresses of the arrays are known at compile time, and the distance vector is constant during the execution of the loop.

For illustration, let us consider the original SSE code in Figure 2. The translator gathers SIMD instructions that involve an access to memory in lines 2, 3, and 4. The one in line 4 is a write; therefore, it is compared to the

```
for(i=0; i < n ; i++) {
  for(j=4; i+j < n; j++) {
    A[i+j] = A[i] + 1;
  }
}
```

Fig. 5 Example of aliasing with changing dependence distance

others. The set of addresses referenced by instruction in line 4, in the case it is translated into AVX, ranges between C and C+32. Similarly, for instruction in line 2 the range is between A and A+32. A non-empty intersection of intervals would stop the translation process. In this example, the optimizer proceeds. Likewise, the intersection test is done for instructions in line 3 and 4.

3.6.4 Dynamic interval-overlapping test

The dynamic interval-overlapping test refers to verification performed at runtime. This happens when the addresses of arrays manipulated are not known statically, or when the dependence distance in the inner loop is modified by the outer loop, as depicted in the example of Figure 5.

In these scenarios, the compiler generates the test as shown in basic block L1 in Figure 6. In case of empty intersection, the control flow is directed to the vectorized loop L2; otherwise, the sequential loop L3 is invoked. The basic block L1 contains the intersection test between the read and write of lines 13 and 16.

The test works as follow: an offset of 16 bytes is added to both `rdi` and `rdx` in lines 2 and 3, to verify whether the read interval [`rdi`, `rdi`+16] intersects with the write interval [`rdx`, `rdx`+16]. In order to adjust this code to work for the AVX version, our translator performs a pattern matching to identify the instructions in line 2 and 3, then it changes the offset from 16 (0x10) to 32 bytes (0x20).

3.7 Alignment constraints

Alignment is another issue faced during the translation of SIMD instructions. In a nutshell, the majority of x86 instructions are flexible with alignment; however, some of them demand data on which they operate to be aligned. From now on, we refer to these two categories respectively as unaligned and aligned instructions. Actually, the aligned SSE instructions require data to be aligned on 16 bytes. And, the aligned AVX instructions require data to be aligned on either 16 or 32 bytes [2].

When the optimizer encounters an unaligned SSE instruction, it translates it to its equivalent unaligned AVX instruction. However, when it encounters an aligned SSE instruction, it must apply one of these three options:

– translate it into its equivalent 32-byte aligned AVX instruction;

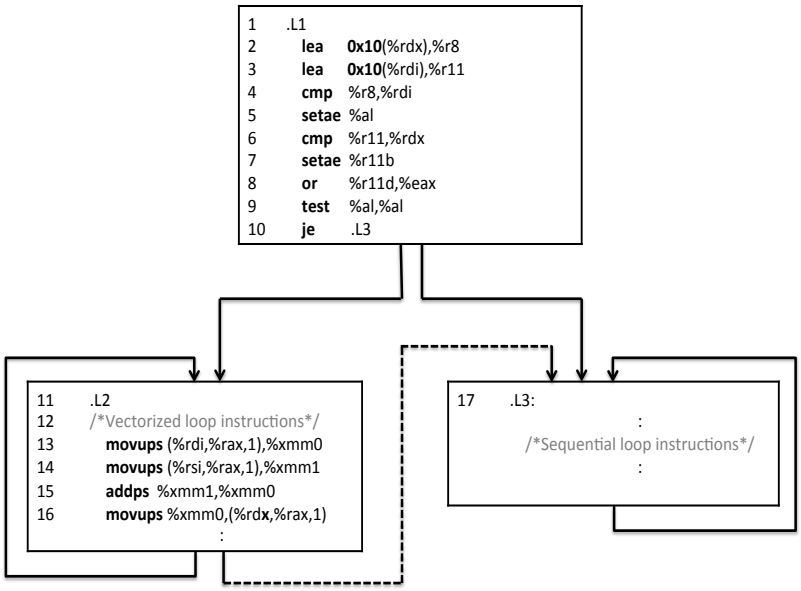


Fig. 6 Pattern of code when array addresses are known only at runtime

- translate it into its multiple equivalent 16-byte aligned AVX instructions that operates on half of the vector operand;
- translate it into its equivalent unaligned AVX instruction.

When the address of data on which the aligned SSE instruction operates is explicitly or implicitly provided in the code section as a static value, the optimizer tests whether it is also aligned on 32 bytes. When the test succeeds, the equivalent 32 bytes aligned AVX instruction is eligible for use. Otherwise, since the SSE aligned instruction is by definition aligned on 16 bytes; the optimizer blindly translates it into the equivalent 16 bytes aligned AVX instructions.

The reason why we do not use the unaligned AVX instructions in the mapping of aligned SSE instructions, although they can be used instead of multiple aligned AVX instructions on 16 bytes, is: multiple aligned instructions, when they are independent, execute in parallel in an out-order-processor. When they are aligned on 16 bytes, they run faster than a single unaligned instruction that performs extra work of testing the cross-cache line access. The drawback of this solution is that multiple instructions occupy more issue slots, in comparison with the single unaligned AVX instruction.

There is another approach that we intend to use in the future to cope with alignment problem. In fact, some compilers extensively use it and it consists of looping over data elements sequentially until most of the manipulated data

```
float m = MIN;
for(i = 0; i < n; i++) {
  if (m < array[i]) /* max(m, array[i]) */
    m = array[i];
}
```

Fig. 7 Search for the maximum element in an array of floats

```
1  .L2:
2          maxps   A(rax),xmm0
3          add     0x10,rax
4          cmp     rax,ARRAY_LIMIT
5          jle     .L2
6          movaps  xmm0,xmm1
7          psrld   8, xmm1 ; shift right logical
8          maxps   xmm1,xmm0
9          movaps  xmm0,xmm1
10         psrld   4,xmm1
11         maxps   xmm1,xmm0
12         movaps  xmm0,xmm1
```

Fig. 8 Search for the largest element in an array, vectorized for SSE

are aligned. At that moment the vectorized loop, which contains aligned instructions, is qualified for its use. While this implies more work at runtime, it also delivers additional performance.

To sum up, dynamic optimization allows us in some cases to check for data elements addresses to cope with alignment problem.

3.8 Reductions

Reduction is the process of reducing a set of values into a single one, for instance, summing the elements of a set.

Figure 7 depicts a reduction algorithm that searches for the largest floating point number in an array. The algorithm suffers from a loop-carried dependence whose distance is equal to 1 (read-after-write on m). In spite of the data dependence, the problem is prone to vectorization thanks to the associativity and commutativity of the `max` operation. Suppose we want to find the maximum value in the following set 5, 9, 3, 7. It is possible to execute simultaneously both operations $max(5, 9) = 9$ and $max(3, 7) = 7$, then execute $max(9, 7) = 9$.

3.8.1 Issue of translating a reduction

The code in Figure 8 depicts the assembly of Figure 7. The code has two sections. First, the loop in lines 2—5 searches simultaneously for four largest numbers in four different sections of the array. Precisely, the sections have indices that fall in i mod 4, i mod 4+1, i mod 4+2, and i mod 4+3. This yields

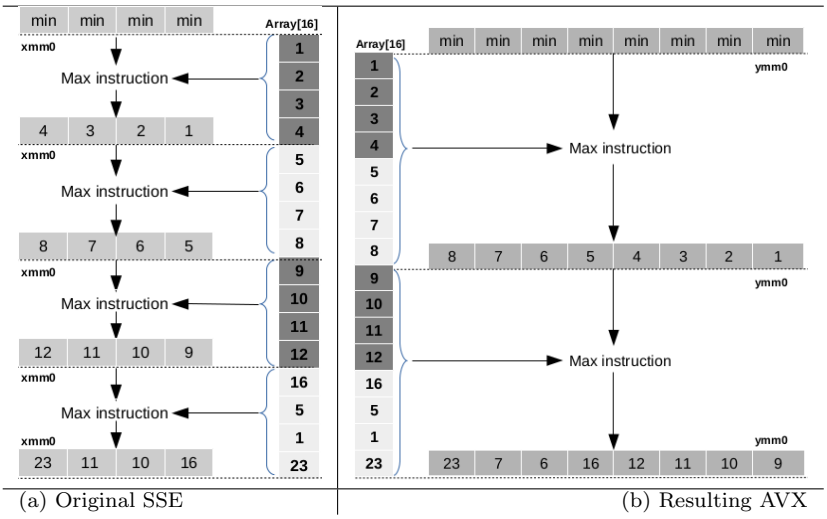


Fig. 9 Body of vectorized loop for vector max

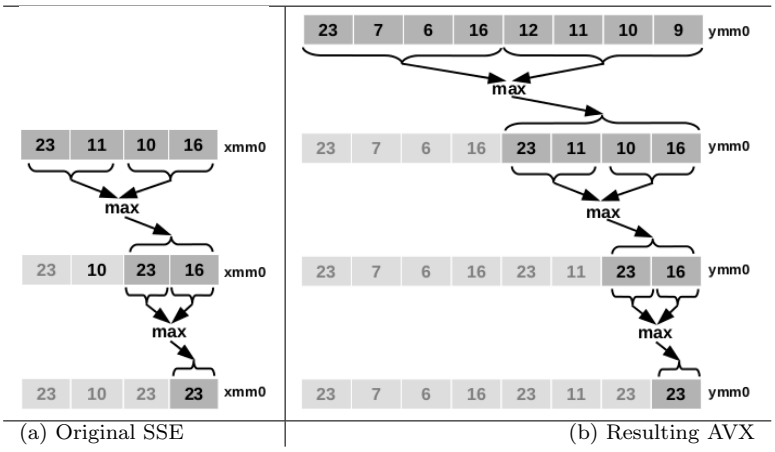
the `xmm0` register with four greatest elements seized during the simultaneous traversal of these regions.

Figure 9 (a) delineates the execution of these lines on an array of sixteen elements. At each iteration, a pack of four elements is read from memory and compared to values stored in `xmm0`, which is initially loaded with minimal values. The register is updated based on the result of the comparisons. Second, since `xmm0` contains a tuple of elements, lines 6—12 resolve the largest one from the others by shifting it into the lowest part of the register. Figure 10 portrays the execution of this latter.

Translation of reductions requires special care. Suppose we translate the loop body of Figure 8 into AVX. Its execution yields register `ymm0` with eight sub-results as depicted in Figure 9 (b), as opposed to four in the case of SSE. The loop epilogue (lines 6—12 of Figure 8) "reduces" them to the final result. It must be updated accordingly to take into account the wider set of sub-results, i.e. the new elements in the upper part of `xmm0`. This is achieved by introducing a single step, between lines 5 and 6, to reduce the 256-bit `ymm` register into its 128-bit `xmm` counterpart, as produced by the original code. In our running example, we need an additional shift-and-mask pair, as shown in Figure 10 (b). Whenever the translator is not able to find a combination of instructions to restore the state of the register `xmm`, it simply aborts optimizing this loop.

3.8.2 Subclass of reduction supported by the translator

This subsection shows a subclass of reduction for which it is possible to recover the same state of `xmm` as if the non-translated loop is executed. This subclass has a particular characteristic described in the following recursive sequence



(a) Original SSE (b) Resulting AVX

Fig. 10 Extracting the largest element from the SIMD register

form:

$$xmm_n = xmm_{n-1} \odot input_n \tag{1}$$

Where the operator $\odot$ is either an arithmetic or logical instruction that is both associative and commutative; and, the subscript n indicates the iteration number. Besides, the input is nothing but the memory read access of the same size as the `xmm` register.

Demonstration: Let us solve the recursive sequence in (1) that describes an SSE loop. In the last line of (2), the odd and even terms are separated since the operator is both associative and commutative (assuming n is even for the sake of simplicity):

$$\begin{aligned} xmm_n &= xmm_{n-1} \odot input_n \\ &= xmm_{n-2} \odot input_{n-1} \odot input_n \\ &= xmm_{n-3} \odot input_{n-2} \odot input_{n-1} \odot input_n \\ &= input_0 \odot input_1 \odot ... \odot input_{n-1} \odot input_n \\ &= [input_0 \odot input_2 \odot ... \odot input_n] \odot [input_1 \odot input_3 \odot ... \odot input_{n-1}] \end{aligned} \tag{2}$$

Since an AVX iteration is equivalent to two successive SSE iterations, there are two consequences. First, the number of iterations decreases by half. Second, a packed memory read in an AVX iteration is equivalent to the concatenation of memory reads of two consecutive iterations in SSE loop. In our notation, *input* represents a memory read for an SSE iteration and the same word in capital letters for an AVX iteration. Plus, $INPUT_L$ and $INPUT_H$ indicate the lower and higher halves of $INPUT$:

$$\begin{aligned} input_{2n} &= INPUT_{L_n} \\ input_{2n+1} &= INPUT_{H_n} \end{aligned} \tag{3}$$

Applying (3) in (2):

$$\begin{aligned} xmm_n &= [INPUT_{L_0} \odot INPUT_{L_1} \odot ... \odot INPUT_{L_{n/2}}] \odot \\ &\quad [INPUT_{H_0} \odot INPUT_{H_1} \odot ... \odot INPUT_{H_{n/2}}] \\ xmm_n &= [ymm\_lowerhalf_{n/2-1} \odot INPUT_{L_{n/2}}] \odot \\ &\quad [ymm\_higherhalf_{n/2-1} \odot INPUT_{H_{n/2}}] \\ &= ymm\_lowerhalf_{n/2} \odot ymm\_higherhalf_{n/2} \end{aligned} \tag{4}$$

Therefore, we conclude that applying the operator between the higher and lower halves yields the state of `xmm` as if it had executed the non-translated code.

4 Vectorization of Binary Code

4.1 Principle of scalar into vector optimization

It may happen that some applications contain scalar codes that were not vectorized by the compiler, even when they contain loops that have the properties required for correct and beneficial SIMD processing. One reason may be that the source code has been compiled for an architecture that does not support SIMD instructions. Another reason is that some compilers are not able to vectorize some class of codes [15], because they do not embed advanced data dependence analysis and loop transformation capabilities, as provided by the polyhedral model [12]. In this section, we widen the optimization scope by auto-vectorizing long running inner loops which contain scalar instructions. We rely on an open source framework, McSema [16], that lifts binary code into the LLVM-IR so that higher level loop optimizations can be performed. From the IR, it is then possible to delegate the vectorization burden to another framework, Polly [13], implementing techniques of the polyhedral model to vectorize candidate loops. Finally, we compile back the vectorized IR into binary using the LLVM JIT compiler. Figure 11 shows the auto-vectorization process.

This approach constitutes a proof-of-concept for dynamic full vectorization using an ambitious mechanism. Note that it is not meant to be integrated with the re-vectorization presented in the previous section, but rather to assess feasibility and performance of such a technique. Indeed, additionally to vectorization, the presented mechanism may support general high-level loop optimizing transformations such as loop interchange, loop tiling, loop skewing and loop parallelization. The feasibility of such transformations at runtime on binary code, thanks to this technique, will be showed in future works.

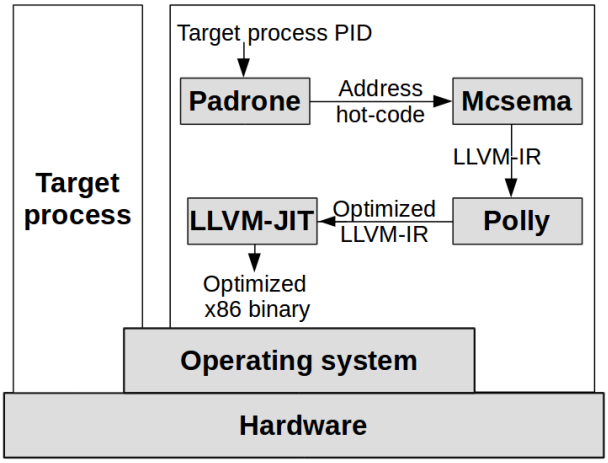


Fig. 11 Auto-vectorization process

4.2 Binary into intermediate representation using McSema

McSema is a decompilation framework whose functionality is to lift code from binary into LLVM-IR. First, the tool disassembles the x86 executable and creates an according control flow graph (CFG). Second, it translates the assembly instructions into LLVM-IR.

4.2.1 Integration of Padrone with McSema

Basically, McSema operates on x86-64 executable files. The `bin_descend` tool takes as arguments the binary file and the name of the function to disassemble, creates a CFG of assembly and marshals this data structure. The `cfg_to_bc` tool demarshals the file, and translates assembly instructions into equivalent LLVM-IR form that is written to a bit-code file.

We tuned McSema to take as input the address of the frequently executed function provided by Padrone and the processed image. Moreover, we avoid the overhead of writing into data storage by skipping the marshaling/demarshaling process so as the data structure produced by `bin_descend` is passed directly to `cfg_to_bc`. Figure 12 shows the integration of Padrone with McSema.

4.2.2 Adjusting McSema to produce a suitable LLVM-IR

The code lifting from binary to IR obviously requires to preserve the semantics of the binary instructions. FPU and arithmetic instructions usually alter the FPU and arithmetic status registers. Hence, McSema lifts the status registers as well to maintain the state of the processor. It is important to keep track of these bits since subsequent instructions like conditional jumps, bitwise or shift-rotate operations, etc. not only depend on their operands, but also on these flags.

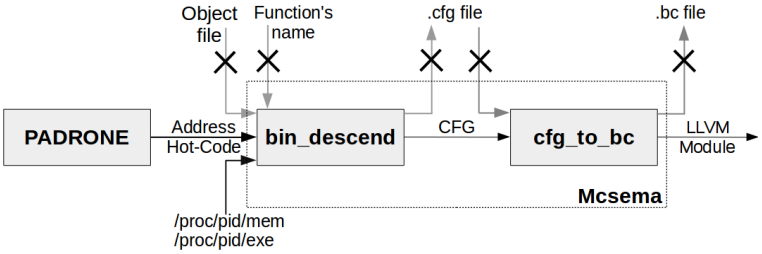


Fig. 12 Integration of Padrone with Mcsema

The consequent issue is that the translation of these instructions ends up in generating a large amount of corresponding LLVM-IR instructions. These instructions confuse the loop analyzer of Polly and thus prevent it from optimizing the bit-code.

For instance, Table 2 depicts the translation of an assembly loop into LLVM-IR. The comparison instruction is a signed subtraction of its operands which alters the AF, SF, OF, PF, CF, ZF flags as well. As a consequence, McSema translates the comparison in line 3 of the first column into lines 6—17 in the second column, where the subtraction is performed (line 9) and the states of the flags are set accordingly. Furthermore, the conditional jump do depend on the flags to direct the flow of execution. The Jump if Not Equal (JNE) tests whether the ZF flag is equal to zero. Correspondingly, line 4 of the first column is translated into lines 19—21. To determine the number of the loop iteration, Polly requires a single comparison instruction whose arguments are the induction variable and the loop trip count. However, McSema produces several comparison instructions free from the loop trip count operand. Thus, we prevent McSema from generating the instructions that alter the state of the flags, and we keep track of the subtraction's arguments. Depending on the conditional jump, an appropriate predicate is fed into the comparison instruction. In this example, the comparison takes three arguments, a predicate Signed Less or Equal (SLE), the induction variable and the loop bound. Lines 6—8 in the third column is the produced semantically equivalent code.

4.3 Vectorization of loops in LLVM-IR using Polly

Polly [13] is a static loop optimization infrastructure for the LLVM compiler, capable of optimizing data locality and exposing parallelism opportunities for loop nests that are compliant with the polyhedral model. When the loop bounds and memory accesses are detected as affine, it creates their representation in the polyhedral model, and reschedules the accesses while respecting the data dependencies.

The main objective in the current step is to vectorize sequential code. We make use of the compiler passes provided by Polly. First, the canonicalization passes are run, which transform the IR into a suitable form for Polly. Second,

Table 2 Adjusting McSema to produce a suitable LLVM-IR

x86 assembly	LLVM-IR produced by McSema	Adjusted LLVM-IR
1. L1: 2. . . . 3. cmp $1024, %rax 4. jne L1	1. %RAX_val = alloca i64 2. %ZF_val = alloca i1 3. 4. %block_L1 5. . . . 6. /* instructions modify the 7. state of ZF */ 8. %117 = load i64* %RAX_val 9. %118 = sub i64 %117, 1024 10. %127 = icmp eq i64 %118, 0 11. store i1 %127, i1* %ZF_val 12. 13. /* instructions modify the 14. states of the remaning flags 15. AF, SF, OF, PF, CF depending 16. on the subtract instruction */ 17. . . . 18. 19. %136 = load i1* %ZF_val 20. %137 = icmp eq i1 %136, false 21. br i1 %137, label %block_L1, label %block_L2 22. 23. %block_L2	1. %RAX_val = alloca i64 2. %ZF_val = alloca i1 3. 4. %block_L1 5. . . . 6. %156 = load i64* %RAX_val 7. %157 = icmp sle i64 %156, 4096 8. br i1 %157, label %block_L1, label %block_L2 9. %block_L2

the Static Control Parts (SCoPs), which are subgraphs of the CFG defining loops with statically known control flow, are detected. Basically, Polly is able to optimize regions of the CFG, namely loops, with fixed number of iterations and conditionals defined by linear functions of the loop iterators (i.e., induction variables). Third, the SCoPs are abstracted into the polyhedral model. Finally, the data dependencies are computed in order to expose the parallelism in the SCoPs. At the time being, our work is confined only to inner loops. We use a method provided by the dependence analysis pass to check whether the loop is parallel. An algorithm that checks whether the data accesses are consecutive along iterations has been developed. We cast the LLVM-IR loads and stores into vector loads and stores, so that the JIT generates SIMD instructions. The induction variable is modified depending on the vector factor as well.

4.4 JITting the IR into binary

4.4.1 Handling global variables

The lifting of binary into LLVM-IR consists of both lifting the instructions and the memory addresses on which they operate. Hence, McSema declares arrays on which the IR instructions perform their operations. However, compiling the IR yields instructions and array allocations in the optimizer's process space. The generated instructions perform their operations on the initiated

arrays. Therefore, injecting this code into the target process would result in bad address references.

4.4.2 Marking the IR operands

As a solution, we mark the operands lifted by McSema with the physical addresses in the original binary; so as, while JITting, the recovered addresses are encoded in the generated instructions. The compilation of LLVM-IR into binary goes through multiple stages. At the beginning, the IR instructions form an Abstract Syntax Tree (AST) that is partially target independent. At each stage, the instruction in the AST is reincarnated into a different data type which is decorated with more target dependent information. At each of these phases, the addresses recovered, while McSema lifts the original binary, are transferred until the generated instructions are encoded.

5 Experimental Results

In this section, we present our experimental apparatuses and the results obtained for both dynamic vectorization transformations adopted in our work. We show for each of the approaches the hardware and software environments, the benchmarks, as well as the metric and experimental results.

5.1 Re-Vectorization experimental results

5.1.1 Hardware/Software

Our experiments were conducted with a 64-bit Linux Fedora 19 workstation featuring an Intel i7-4770 *Haswell* processor clocked at 3.4 GHz. Turbo Boost and SpeedStep were disabled in order to avoid performance measure artifacts associated with these features.

We observed that different versions of GCC produce different results. We made our experiments with two relatively recent version of GCC: GCC-4.7.2 (Sep. 2012) and GCC-4.8.2 (Oct. 2013) available on our workstation. ICC-14.0.0 was used whenever GCC was not able to vectorize our benchmarks.

5.1.2 Benchmarks

We handled two kinds of benchmarks. The first kind consists in a few handcrafted loops that illustrate basic vectorizable idioms. The second kind is a subset of the TSVC suite [15]. Table 3 summarizes the main features for each benchmark. All TSVC kernels manipulate arrays of type `float`. We also manually converted them to `double` to enlarge the spectrum of possible targets and assess the impact of data types.

Table 3 Experimental Kernels

Name	Short description
vecadd	addition of two arrays
saxpy	multiply an array by a constant and add a second
dscal	multiply an array by a constant
s000	addition and multiplication of arrays
s115	triangular saxpy
s125	product and addition of two-dimensional arrays
s174	addition of an array with a part of the second array storing in an other part of the latter
s176	convolution
s251	scalar and array expansion
s311	sum of elements of a single array (reduction)
s314	search for maximum element in an array (reduction)
s319	sum of elements of multiple arrays
s1351	addition of two arrays using `restrict` pointers

We compiled most of our benchmarks with GCC using flags `-O3 -msse4.2` (`-O3` activates the GCC vectorizer). Only `s311` and `s314` were compiled with ICC because both versions of GCC were unable to vectorize them.

We decouple the analysis of the optimizer overhead, and the performance of the re-vectorized loops.

Each benchmark (both hand-crafted and TSVC) essentially consists in a single intensive loop that accounts for nearly 100 % of the run time. This is classical for vectorization studies as it shows the potential of the technique (its asymptotic performance). As per Amdahl's law [3], the speedup on a given application can easily be derived from the weight of the loop in the entire application: let α be the weight of the loop, and s the speedup of this loop, the overall speedup is given by:

$$s' = \frac{1}{1 - \alpha + \frac{\alpha}{s}}$$

As expected, $\lim_{\alpha \to 1} s' = s$. This is the value we observe experimentally.

5.1.3 Performance Results

We assess the performance of our technique by means of two comparisons. First we measure the raw speedup, i.e. we compare the transformed AVX-based loop against the original SSE-based loop. Then, we also compare it against the native AVX code generated by GCC with flags `gcc -mavx -O3` (except `s311` and `s314` whose native AVX codes are generated by ICC with flags `icc -mavx -O3`). Table 4 reports the speedups of both native compiler for AVX and our re-vectorizer compared to SSE code. In the case of our re-vectorizer, we also report how it compares to the native compiler targeting AVX. These numbers are shown graphically in Figures 13 and 14 for data type `float` and `double` respectively. As an example, the first row (`dscal`) shows that the AVX code produced by GCC runs 1.4× faster than the SSE

Table 4 Performance Improvement over SSE

	Kernel	GCC 4.8.2 native AVX	GCC 4.8.2 our revect.	GCC 4.8.2 % vs. native	GCC 4.7.2 native AVX	GCC 4.7.2 our revect.	GCC 4.7.2 % vs. native
float	dscal	1.40×	1.66×	+19%	1.60×	1.37×	-14%
	saxpy	1.10×	1.67×	+52%	1.88×	1.35×	-28%
	vecadd	1.00×	1.66×	+66%	1.74×	1.26×	-28%
	s000	0.99×	1.19×	+20%	1.07×	0.69×	-36%
	s125	1.02×	1.25×	+23%	1.49×	1.25×	-16%
	s174	0.86×	1.34×	+56%	1.50×	1.09×	-27%
	s176	1.48×	1.52×	+3%	1.54×	1.22×	-21%
	s251	1.17×	1.35×	+15%	1.57×	0.92×	-41%
	s319	1.42×	1.61×	+13%	1.80×	1.05×	-42%
	s1351	0.90×	0.91×	+1%	0.92×	0.87×	-5%
	s115	translator aborts					
double	dscal	1.17×	1.11×	-5%	1.60×	1.31×	-18%
	saxpy	1.01×	1.18×	+18%	1.57×	1.10×	-30%
	vecadd	0.82×	1.34×	+63%	1.47×	0.97×	-34%
	s000	0.95×	0.99×	+5%	0.98×	0.67×	-32%
	s125	0.90×	0.91×	+0%	0.91×	0.91×	0%
	s174	0.89×	1.33×	+51%	1.49×	0.98×	-34%
	s251	0.94×	0.96×	+2%	0.97×	0.96×	-1%
	s319	1.25×	1.33×	+6%	1.33×	0.90×	-33%
	s1351	0.88×	0.91×	+3%	0.91×	0.90×	-1%
	s115	translator aborts					
	s176	translator aborts					

	Kernel	Reduction with icc native	our revect.	% vs. native
float	s311	1.80×	1.79×	-0.55%
	s314	1.79×	1.79×	0%
double	s311	1.79×	1.79×	0%
	s314	translator aborts		

+ native AVX: the execution time of the native SSE divided by native AVX.
+ our revect.: the execution time of native SSE divided by AVX generated by our revectorizer.
+ % vs. native: the percentage of improvement of "our revect" compared to "native AVX".

version. The code produced by our re-vectorizer runs 1.66× faster than the SSE version, that is a 19 % improvement over the AVX version.

We confirmed that the difference in the code quality between the SSE references produced by both compilers is small compared to the variations observed between SSE and AVX.

GCC-4.8.2 As a general trend, our re-vectorizer is able to improve the performance of eligible loops, up to 67 %. Surprisingly, we also constantly outperform GCC for AVX, up to 66 % in the case of `vecadd`. The reasons are:

1. When targeting AVX, GCC-4.8.2 generates a prologue to the parallel loop to guarantee alignment of one of the accessed arrays. Unfortunately, the loop does not take advantage of the alignment and relies on unaligned memory accesses (`vmovups` followed by `vinsertf128` when a single `vmovaps` sufficed). When targeting SSE, there is no prologue, and the loop relies on

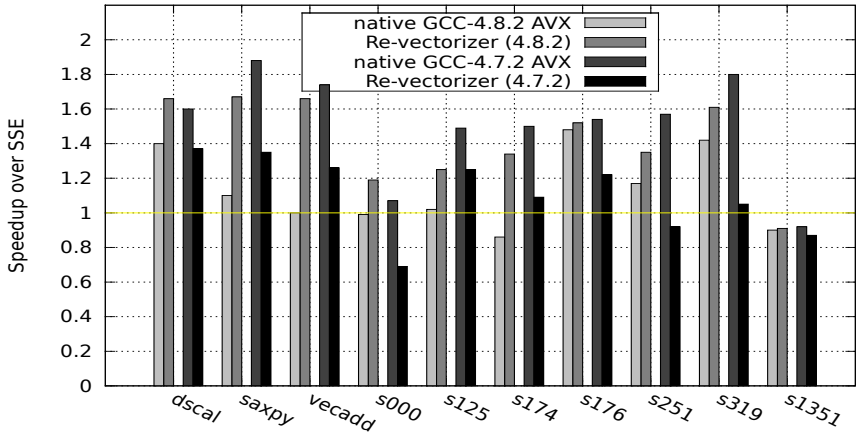


Fig. 13 Speedups for type `float`

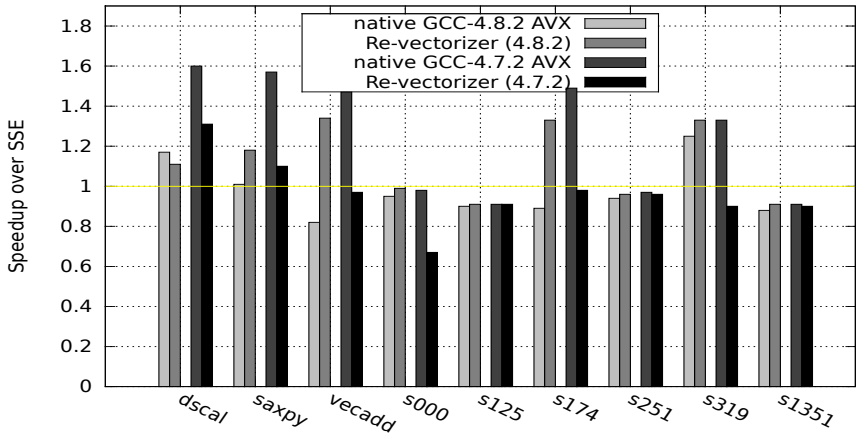


Fig. 14 Speedups for type `double`

16-byte aligned memory accesses. In fact, AVX code generated by GCC-4.7.2 is more straightforward, without prologue, and similar to our own code generation, and corresponding performance also correlates.

2. GCC-4.8.2 tries to align only one of the arrays. Testing for alignment conditions of all arrays and generating specialized code for each case would result in excessive code bloat. The static compiler hence relies on unaligned memory accesses for all other arrays. Because we operate at runtime, we have the capability to check actual values and generate faster aligned accesses when possible.

In the case of `s115`, we correctly identified the vectorized hot loop, but we were not able to locate its sequential counterpart (the loop at label L5 in Figure 3) in the function body, needed to execute a few remaining iterations when

the trip count is not a multiple of the new vectorization factor (as described in Section 3.5.2). The reason is that the native compiler chooses to fully unroll this epilogue. Our optimizer simply aborted the transformation.

The only negative effect occurs with `s1351`, with a 9% slowdown. Note that the native AVX compiler also yields to a 10 % slowdown. In this example, the compiler generates unaligned packed instructions. Precisely, a combination of two instructions that move separately the low and high 128 bits of operands between memory and registers. This degrades the performance of AVX. To verify that it is an alignment issue, we made a manual comparison between SSE and AVX versions of `s1351` and forced data to be 32-byte aligned. As consequence, the compiler generates aligned instructions. Under this circumstance, the AVX version outperforms SSE.

Performance of our re-vectorizer as well as native GCC AVX is generally lower when applied to kernels operating on type `double`. The reason is that arrays with the same number of elements are twice larger, hence increasing the bandwidth-to-computation ratio, sometimes hitting the physical limits of our machine, as well as increasing cache pressure. We confirmed that halving the size of arrays produces results in line with the `float` benchmarks.

Three benchmarks failed with type `double`: `s115`, `s176`, and `s314`. This is due to a current limitation in our analyzer: the instruction `movsd` may be translated in two different ways, depending on the presence of another instruction `movhpd` operating on the same registers. Our analyzer currently considers instructions once at a time, and must abort. Future work will extend the analysis to cover such cases.

GCC-4.7.2 With GCC-4.7.2, our re-vectorizer sometimes degrades the overall performance compared to SSE code. We observe that this happens when the same register (`ymm0`) is used repeatedly in the loop body to manipulate different arrays. This increases significantly the number of partial writes to this register, a pattern known to cause performance penalties [1]. This is particularly true in the case of `s125`. Despite these results, since our optimizer operates at runtime, we always have the option to revert to the original code, limiting the penalty to a short (and tunable) amount of time.

As opposed to GCC-4.8.2, we systematically perform worse than native AVX. This is expected because the native AVX compiler often has the capability to force alignment of arrays to 32 bytes when needed. Since we start from SSE, we only have the guarantee of 16-byte alignment, and we must generate unaligned memory accesses. The net result is the same number of memory instructions as SSE code, while we save only on arithmetic instructions. Note, though, that we do improve over SSE code in many cases, and we have the capability to revert when we do not.

ICC-14.0.0 For both of `s311` and `s314`, our re-vectorizer produces codes that run almost 1.80× faster than native SSE, and they have almost the same performance as the native AVX.

name	s000	s125	s174	s176	s319	s1351	vecadd	saxpy	dscal
average	1.4	1.0	1.0	1.4	1.5	1.1	0.8	1.3	1.2
stddev	0.4	0.3	0.2	0.2	0.05	0.03	0.1	0.03	0.05

Table 5 Re-vectorization overhead (ms). Average and standard deviation over 10 runs.

5.1.4 Overhead

The overhead includes profiling the application to identify hot spots, reading the target process' memory and disassembling its code section, building a control flow graph and constructing natural loops, converting eligible loops from SSE to AVX, and injecting the optimized code into the code cache. Profiling has been previously reported [22] to have a negligible impact on the target application. With the exception of code injection, all other steps are performed in parallel with the execution of the application.

On a multicore processor, running the re-vectorizer on the same core as the target improves communication between both processes (profiling, reading original code, and storing to the code cache) at the expense of sharing hardware resources when both processes execute simultaneously. The opposite holds when running on different cores. Since our experimental machine features simultaneous multi-threading (Intel Hyperthreading), we also considered running on the same physical core, but two different logical cores.

In our results for all configurations, the time overhead remains close to the measurement error. Table 5 reports the overhead in milliseconds of re-vectorizing loops, for each benchmark. We ran the experiment ten times and we report the average and the standard deviation.

On the application side, storing and restoring the `ymm` registers represent a negligible overhead, consisting in writing/reading a few dozen bytes to/from memory.

5.2 Vectorization Experimental Results

5.2.1 Hardware/Software

The experiments were carried using a machine equipped with an Intel Core i5-2410M processor based on the Sandy Bridge architecture. The clock rate is 2.3 GHz. As before, the Turbo Boost and SpeedStep were disabled. The hardware resources are managed by an Ubuntu 14.04.1 LTS operating system. Furthermore, the benchmarks were compiled with GCC 4.8.4.

5.2.2 Benchmarks

The integration of the complex pieces of software which are Padrone, McSema, and Polly, is not yet finalized at the time of writing, thus forcing us to investigate only a subset of the benchmarks addressed previously with re-vectorization. The kernels operate on arrays of double-precision floating point

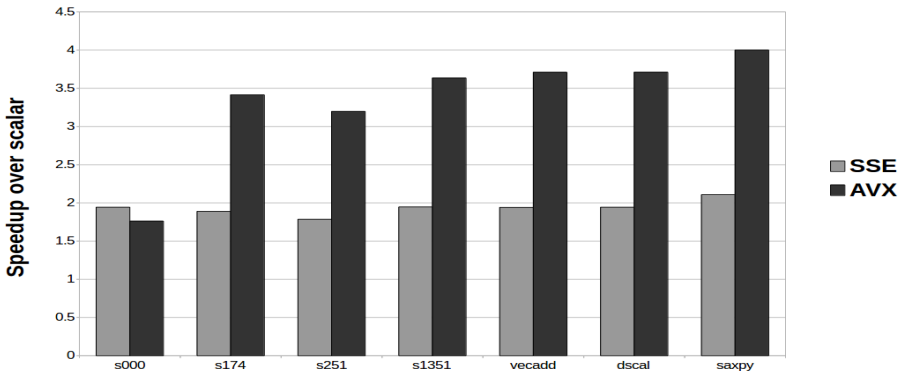


Fig. 15 SSE and AVX Autovectorization Speedups

elements which are 32-bit aligned. The benchmarks were compiled with flags `-O3 -fno-tree-vectorize` which disable the vectorization and maintain the rest of the optimizations. Besides, the methods are declared with attribute `noinline` to disable function inlining.

5.2.3 Performance Results

We assess the performance of vectorization by measuring the speedup of the optimized loops (vectorized for SSE or AVX) against the unvectorized loops. Results are shown in Figure 15.

First, we observe that auto-vectorization improves the performance by factors on par with the vector width. The SSE version outperforms the scalar version up to 1.95× in the case of `s1351`. As for `dscal`, the AVX version runs 3.70× faster.

Second for `saxpy`, we even obtain a superlinear speedup: it runs 2.10× faster. We compare this value to the speedup obtained with GCC 4.8.4 which is 2.04×. This small difference is due to the instructions generated by GCC 4.8.4 and the LLVM's JIT which are not similar.

Finally, for `s000`, we notice that the AVX version's performance is slightly less than the one of SSE. The reason is that `s000` is prone to a bottleneck between the register-file and cache. In other words, the microbenchmark intensively accesses memory, and the limitation of bandwidth results in introducing stalls into the CPU pipeline. Hence, AVX data movement instructions commit with extra cycles with regards to SSE.

5.2.4 Overhead

The overhead of fully vectorizing loops is reported in Table 6. As expected, the values are much higher than in the case of re-vectorization, typically two orders of magnitude. This is due to the complexity of vectorization, to the approach based on lifting the code to LLVM-IR and applying the polyhedral

name	s000	s174	s251	s1351	vecadd	dscal	saxpy
average	205.4	112.0	162.5	113.8	106.9	104.6	123.3
stddev	7.5	5.7	4.5	4.5	3.0	3.4	3.3

Table 6 Vectorization overhead (ms). Average and standard deviation over 10 runs.

model, and partly the fact that we connect building blocks whose interfaces have not been designed for efficiency at runtime.

6 Related Work

Vectorization has been initially proposed as a purely static code optimization which equips all industrial-grade compilers. Retargetable compilers have introduced the need to handle several targets, including various levels of SIMD support within a family [19].

Liquid SIMD [9], from Clark et al., is conceptually similar to our approach. 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. The difference is that we require no additional hardware.

Recent work integrates initial automatic vectorization capabilities in JIT compilers for Java [17] [11]. Our focus is on increasing the performance of native executables.

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 [14]. Instead, our proposal considers a single ISA, and migrates SIMD instructions to a higher level of features, targeting wider registers, but retaining the rest of the surrounding code.

Vapor SIMD [23] 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. It was further extended [18] into a scheme that 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. In contrast, the present work focuses on *plain* native executables, without any bytecode, JIT compiler, or annotated (fat) binary. Vapor SIMD also automatically exploits the highest level of SSE available on the target processor. This paper only considers SSE as a whole vs. AVX, however it could easily be extended to support various versions of SSE.

Riou et al. [22] dynamically modify a running executable by substituting a SSE hot loop by a statically compiled version targeting AVX. In this paper,

we dynamically generate code for the new loop, and do not rely on a statically prepared patch.

Regarding automatic loop optimization and parallelization of binary code, a close work is the one of Pradelle et al. [21], however the described approach is purely static. Their automatic parallelizer first parses the binary code and extracts high-level information. From this information, a C program is generated. This program captures only a subset of the program semantics, namely, loops and memory accesses. This C program is then parallelized using the polyhedral parallelizer Pluto [20]. The original program semantics is then reinjected, and the transformed parallel loop nests are recompiled by a standard C compiler.

As a binary optimizer, ADORE [7] uses hardware counters to identify hotspots and phases and to apply memory prefetching optimizations. Similar goals are addressed in [5] where a dynamic system inserts prefetch instructions on-the-fly where it has been evaluated as effective by measuring the load latency. Both approaches focus on reducing the memory latency of memory instructions.

Various tools have been developed with DynamoRIO [6]. The framework we used keeps all the client and toolbox in a separate address space and modifies as little as necessary of the original code, while DynamoRIO links with the target application and execute code only from the code cache.

Dynamic binary translation also operates at run-time on the executable, but translates it to a different ISA. It has been implemented several ways, such as Qemu [4], FX!32 [8], or Transmeta's Code Morphing System [10]. We generate code for the same ISA (only targeting a different extension). This gives us the ability to avoid much of the complexity of such translators, and to execute mostly unmodified application code, focusing only on hotspots.

7 Conclusion and Perspective

In this paper, we have focused on maximizing the utilization of SIMD extensions on-the-fly. For this purpose, we use a lightweight profiling to detect frequently executed sections of code. We adopt two techniques based on dynamic binary rewriting and which optimize loops belonging to the hot-code. On the one hand, eligible loops that were compiled for the SSE SIMD extension are converted to AVX at runtime. We leverage the effort of the static vectorizer, and only extend the vectorization factor, while guaranteeing that the transformation is legal. We show that the overhead is minimal, and the speedup very significant. Moreover, runtime information such as actual alignment of memory accesses, can also result in substantial performance improvements. On the other hand, the scalar loops are lifted to the LLVM intermediate representation. At this level, which hides the target machine details, we have showed that it is possible to handle the loops using the polyhedral model. They are then vectorized when possible. Finally, they are compiled into executable form using the LLVM JIT compiler. The results show the effectiveness of the approach.

In the near future, we plan to extend the range of applied loop optimizations by taking advantage of the whole polyhedral model framework at runtime, thus handling on-the-fly automatic loop tiling and automatic loop parallelization, for instance.

References

1. Intel 64 and IA-32 Architectures Optimization Reference Manual (2014)
2. Intel 64 and IA-32 Architectures Software Developer's Manual (2014)
3. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Spring Joint Computer Conference, AFIPS '67 (Spring), pp. 483–485. ACM (1967). DOI 10.1145/1465482.1465560
4. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Usenix ATC, Freenix Track, pp. 41–46 (2005)
5. Beyler, J.C., Clauss, P.: Performance driven data cache prefetching in a dynamic software optimization system. In: ICS'07, pp. 202–209. ACM (2007)
6. Bruening, D.: Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, MIT (2004)
7. Chen, H., Lu, J., Hsu, W.C., Yew, P.C.: Continuous adaptive object-code reoptimization framework. In: Advances in Computer Systems Architecture, *LNCS*, vol. 3189 (2004)
8. Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S.B., Yates, J.: FX!32: A profile-directed binary translator. IEEE Micro **18**(2), 56–64 (1998)
9. Clark, N., Hormati, A., Yehia, S., Mahlke, S., Flautner, K.: Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In: HPCA (2007)
10. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: CGO (2003)
11. El-Shobaky, S., El-Mahdy, A., El-Nahas, A.: Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In: ICOOOLPS (2009)
12. Feautrier, P., Lengauer, C.: Polyhedron model. In: Encyclopedia of Parallel Computing, pp. 1581–1592. Springer (2011)
13. Grosser, T., Groesslinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(04) (2012)
14. Li, J., Zhang, Q., Xu, S., Huang, B.: Optimizing dynamic binary translation for SIMD instructions. In: CGO (2006)
15. Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: PACT'11, pp. 372–382 (2011)
16. McSema: x86 to machine code translation framework. `https://github.com/trailofbits/mcsema`
17. Nie, J., Cheng, B., Li, S., Wang, L., Li, X.F.: Vectorization for Java. In: NPC (2010)
18. Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A., Zaks, A.: Vapor SIMD: Auto-vectorize once, run everywhere. In: CGO (2011)
19. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: CGO (2006)
20. Pluto: An automatic parallelizer and locality optimizer for multicores
21. Pradelle, B., Ketterlin, A., Clauss, P.: Polyhedral parallelization of binary code. ACM TACO **8**(4), 39:1–39:21 (2012)
22. Riou, E., Rohou, E., Clauss, P., Hallou, N., Ketterlin, A.: Padrone: a platform for online profiling, analysis, and optimization. In: Dynamic Compilation Everywhere (2014)
23. Rohou, E., Dyshel, S., Nuzman, D., Rosen, I., Williams, K., Cohen, A., Zaks, A.: Speculatively vectorized bytecode. In: HiPEAC (2011)
24. Valensi, C.: A generic approach to the definition of low-level components for multi-architecture binary analysis. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines (2014)