Portable and Efficient Auto-vectorized Bytecode: a Look at the Interaction between Static and JIT Compilers

Erven Rohou

Portable and Efficient Auto-vectorized Bytecode: a Look at the Interaction between Static and JIT Compilers

Erven Rohou – HiPEAC member*

INRIA, Centre Inria Rennes - Bretagne Atlantique
Campus de Beaulieu
Rennes, France

Abstract. Heterogeneity is a confirmed trend of computing systems. Bytecode formats and just-in-time compilers have been proposed to deal with the diversity of the platforms. By hiding architectural details and giving software developers a unified view of the machine, they help improve portability and manage the complexity of large software.
Independently, careful exploitation of SIMD instructions has become crucial for the performance of many applications. However, auto-vectorizing compilers need detailed information about the architectural features of the processor to generate efficient code.
We propose to reconcile the use of architecture neutral bytecode formats with the need to generate highly efficient vectorized native code. We make three contributions. 1) We show that vectorized bytecode is a viable approach that can deliver portable performance in the presence of SIMD extensions, while incurring only minor penalty when SIMD is not supported. In other words, the information that a loop can be vectorized is the vectorized loop itself. 2) We analyze the interaction between the static and just-in-time compilers and we derive conditions to deliver performance. 3) We add vectorization capabilities to the CLI port of the GCC compiler.

1 Motivations

In this study, we attempt to reconcile two apparently contradictory trends of computing systems. On the one hand, hardware heterogeneity favors the adoption of bytecode format and late, just-in-time (JIT) code generation. On the other hand, exploitation of hardware features, in particular SIMD extensions, is key to extract performance from the hardware.

1.1 Mobile Long-lived Applications and Processor Heterogeneity

Heterogeneity of computing systems is a global trend. On embedded systems, this trend has been driven by the drastic constraints on cost, power and performance. General purpose computers also feature some degree of variability: availability of a

* This work was partially supported by the European project ACOTES and the HiPEAC Network of Excellence. The author would like to thank colleagues at IBM, INRIA, STMicroelectronics and Thales for fruitful discussions.

floating point unit, width of the vectors of the SIMD unit, number of cores, kind and features of the GPU, and so on. Some predict that technology variability will make it hard to produce homogeneous manycores, and that the large number of available cores will push to specialize cores for dedicated tasks [7].

The lifetime of applications is much longer than that of the hardware. This problem is known as *legacy code* in the industry. Embedded systems rarely offer binary compatibility because of the associated cost. Servers and personal computers usually do, but at a significant design cost (with occasional disruptions like DEC, or Apple). However, compatibility is limited to functionality; old code can only take advantage of increased clock frequency (which, incidentally, has recently stopped) and improved microarchitecture, but not of additional features or increased parallelism. Applications also become mobile. Because of the ubiquity of computing devices, application developers must make sure that their applications run on dozens of platforms, some being unknown or not completely specified.

Bytecode formats and just-in-time compilers have been proposed to deal with heterogeneity. Bytecode can be deployed to any system as long as a JIT compiler is available for each core on which the code is going to run. Application developers do not even have to know the hardware on which their code will eventually run. Processor virtualization, i.e. virtual machine and JIT compilers, is a mature and widely spread technology: Java applications can be found from games in cell phones to web servers and banking applications, and CLI [8] (the core of the .NET initiative [16]) is growing fast. Virtualization can address the above mentioned problems: it reduces the burden put on software developers who no longer need to deal with varying hardware, it guarantees that application lifetimes can span several generations of hardware, and, to some degree, it makes it possible for old code to exploit new hardware features.

1.2 Exploitation of Word-Level Parallelism

The careful exploitation of SIMD instructions is crucial for the performance of many applications. All major instruction sets provide SIMD extensions (SSE on x86 processors, Altivec on PowerPC, VIS on Sparc, etc.), and keep adding new vector instructions (SSE4.1, SSE4.2, SSE4a). Even though significant progress has been made in the recent years, good auto-vectorization is still an open difficult problem. This is illustrated by the abundance of literature [1, 3, 14, 17], the ongoing work in open source compilers like GCC, or simply the existence of source-level *builtins* that let programmers insert instructions by hand when the compiler fails to detect a pattern.

Auto-vectorization is a complex optimization for several reasons:

- strong conditions must be met by the code, in particular in terms of data dependencies;
- for better applicability, one wants the optimization to also apply to outer loops and to handle strided accesses and other complex patterns;
- each particular instruction set has a very specific set of vector instructions, and associated constraints [17]: required alignment, available registers, etc.

1.3 Putting Things Together

If static compilers have a hard time vectorizing loops, the situation is much worse for JIT compilers. A simple look at the vectorizer in GCC gives an idea — more than 20,000 lines, not counting data dependence analysis and the construction of the SSA form. The complexity of the analysis and transformation makes the vectorizer unfit for JIT compilers which are often running on memory and CPU constrained environments.

Conversely, *statically* auto-vectorizing loops for a bytecode representation is challenging because the actual features and constraints of the execution platform are unknown at compile-time. At run-time, SIMD extensions might not be available.

In this paper, we investigate how processor virtualization and auto-vectorization can be reconciled. We make three contributions:

1. we show that vectorized bytecode is a viable approach, that can yield the expected speedups in the presence of SIMD instructions, and a minor penalty in its absence;
2. we measure the performance of loop kernels on several architectures, we analyze the interaction between the static and the JIT compilers and we provide suggestions for a good performance of vectorized bytecode;
3. we describe our modifications to the CLI port of the GCC compiler to emit vectorized bytecode, and we make them publicly available in the GCC repository.

This paper is organized as follows. Section 2 presents the high level view and rationale of our approach, while Section 3 goes over the details of the implementation. We develop our experiments and our analyses in Section 4. Related work is reviewed in Section 5 and we conclude in Section 6.

2 Reconciling Processor Virtualization and Auto-vectorization

This paper uses or refers to compilation and optimization techniques, such as function inlining, loop unrolling, data dependence analysis, SSA form. We did not introduce any new technique *per se*, but rather we take them for granted and we use them. The interested reader can refer, for example, to [15].

2.1 Split-Compilation

Our proposal builds on top of the idea of split-compilation. Split-compilation refers to the fact that a given source code undergoes two compilation steps before it becomes machine code.

1. The first step translates source code to bytecode. This happens on the programmer's workstation. This means that the resources available to the compiler are virtually unlimited: gigahertz, gigabytes and minutes of compile-time are common. However, no assumption can be made on the actual platform on which the application will eventually run.

2. The second step converts the bytecode to machine code. It happens *just-in-time*, i.e. on the final device and at run-time. Resources are likely to be limited, especially on an embedded system like a cell phone or a DVD player. Compile-time is also visible to the end-user, and thus it must be kept as small as possible.

The key of split compilation is to move as much complexity as possible from the second step to the first one [19]. The first pass is in charge of all target independent optimizations. Target specific optimizations obviously cannot be applied. Expensive analyses, however, can be run, and their results encoded in the bytecode, so that the JIT compiler can directly benefit from their outcome.

2.2 Vectorized Bytecode

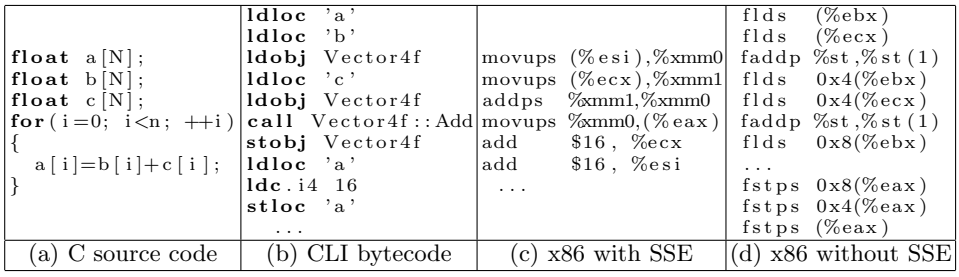
Previous work by Leśnicki et al. annotated the bytecode to mark the variables and types of interest to the JIT compiler (see Section 5 for more details). We believe that this kind of annotation was rather difficult to generate and left too much work to the online vectorizer. Instead, we choose a more drastic approach: the information that a loop can be vectorized is the vectorized loop itself. All the expensive loop transformation is done in the first pass, and we make sure that it can be undone at a low cost if necessary. It is cheaper (at run-time) to undo a speculative vectorization than to do it when necessary.

As further explained in Section 3, we base our work on the CLI format. However, it is very important that we do not extend the format itself. The vectorized bytecode we produce must run unmodified on any CLI compliant virtual machine. Vector information is expressed by means of new types and methods. Vector operations in the user code appear as method invocations.

We need to achieve three objectives:

1. in the presence of SIMD extensions in the instruction set, a JIT compiler aware of our optimization must produce fast machine code;
2. in the absence of SIMD extensions, or when using any other JIT compiler, the code must run correctly; and
3. the penalty when running vectorized bytecode without SIMD support (whether in the JIT compiler or in the instruction set) must be minimum.

All three objectives are made possible by the dynamic code generation mechanism. Point 1 is "simply" a different code generation. The static and JIT compilers must agree upon a naming convention for special types and methods. When a call to such a method is encountered, a specialized instruction pattern is emitted, instead of a call. Figure 1 illustrates this for a simple vector addition. Column (a) shows the C code for this simple loop. Column (b) shows a part of the loop once translated to CLI bytecode. Bear in mind that the execution model relies on an execution stack. `Vector4f` is a type defined in a library that represents a vector of four single precision floating point numbers. `ldloc 'b'` places the address of a vector element on the stack. `ldobj` consumes the address and loads the element on the stack. The same is true for element `c`. The method `Add` is then called to perform the addition. The result is stored at the address initially pushed on the stack by the `ldloc 'a'` instruction. The remaining instructions increment the induction variable



```
float a[N];
float b[N];
float c[N];
for(i=0; i<n; ++i)
{
  a[i]=b[i]+c[i];
}
```

```
ldloc 'a'
ldloc 'b'
ldobj Vector4f
ldloc 'c'
ldobj Vector4f
call Vector4f::Add
stobj Vector4f
ldloc 'a'
ldc.i4 16
stloc 'a'
  ...
```

```
movups (%esi),%xmm0
movups (%ecx),%xmm1
addps  %xmm1,%xmm0
movups %xmm0,(%eax)
add    $16, %ecx
add    $16, %esi
  ...
```

```
flds  (%ebx)
flds  (%ecx)
faddp %st,%st(1)
flds  0x4(%ebx)
flds  0x4(%ecx)
faddp %st,%st(1)
flds  0x8(%ebx)
...
fstps 0x8(%eax)
fstps 0x4(%eax)
fstps (%eax)
```

(a) C source code | (b) CLI bytecode | (c) x86 with SSE | (d) x86 without SSE

Fig. 1. Code generation schemes

`a`. When translating this bytecode (column (c), showing x86 assembly code [10]), the JIT compiler recognizes the type `Vector4f` and it emits a `movups` instruction that targets an SSE register. Similarly, `Add` is recognized, and a single `addps` instruction is emitted.

Point 2 consists in providing a library which implements all the functions defined by the naming convention. A compiler unaware of the special semantics will emit regular code, the same way as any other code.

Point 3 combines the just-in-time code generation with inlining. We make the assumption that JIT compilers always have the capability to inline functions. This is not a strong assumption because bytecode and JIT compilers were initially designed for object oriented languages, which tend to encourage small functions, including accessors (setters/getters) and constructors. Inlining has been key for performance since the beginning of this technology. The vector operations provided in the library are very basic, they include arithmetic, constructors, and load or store operations. They are good candidates for inlining. The end result after minimal cleanup is code similar to the column (d) of Figure 1. The vector operations are effectively unrolled by an amount equal to the width of the vector. Unrolling is known to help performance at the expense of code size. However, we do not expect any significant code bloat because only the small loops corresponding to vector operations are unrolled.

Vectorizing the bytecode gives another advantage: in cases when the static compiler is not able to generate the SIMD instructions, programmers can still manually insert builtins in the source code, as they usually do for performance critical loop nests. It makes no difference to the JIT compiler whether those builtins were automatically generated or hand written.

2.3 Other Design Decisions

SIMD instruction sets vary a lot in number of supported idioms, expressiveness, and constraints. Many choices can be made to best match the abstract vector representation of the bytecode to all possible instances of vector instruction sets. Because we rely on an existing compiler (see Section 3), our choices are limited and we mostly follow the decisions made in the GCC GIMPLE representation. For further details about those design decisions, we refer to the discussion "Generality vs. applicability" of [17].

Alignment Alignment constraints and realignment idioms are a typical burden of vectorizing compilers. We face the additional problem that the static compiler does not know whether the target supports unaligned accesses. We have two options.

- We support unaligned accesses in the bytecode. The static compiler generates simpler code. It is up to the JIT compiler to realign memory accesses if needed.
- Or we require aligned memory accesses in the bytecode. In this case, the static compiler generates the realignment code in the bytecode. The JIT compiler is guaranteed to see only aligned memory accesses.

We decided for the former approach, because it generates simpler code. The latter, while always correct, requires extra work from the JIT compiler when misaligned accesses are available: it needs to eliminate redundant checks or even entire loops that were generated to realign accesses by peeling some iterations off the main computation loop. In the former approach, the static compiler can pass alignment information to the JIT compiler, so that no unnecessary realignment is generated when arrays are known to be properly aligned.

Vector Width Vector width is another parameter dictated by the architecture, hence unknown in the static compiler. We take the following approach: since most architectures have 128-bit wide vector operations, this is the width we vectorize for. An architecture with a different vector width, like the upcoming AVX or Larrabee) will fall back on the scalar implementation as described in this section. A smarter JIT compiler could try adjust to the actual width, but this needs additional data dependence analysis at run-time, or extra annotations that specify the maximum vectorization factor for each loop.

Multiversioning We could have made the choice to generate two versions of each loop: a vectorized one, and a scalar one. The consequence, however, is that the static compiler should use a reduced set of vector instructions, unless it runs the risk that the vectorized code is too specialized and never runs on many architectures. Another option is to generate more than two versions of the same loop (a technique generally called multiversioning), to adjust to most targets. Obviously the cost is code size increase. Generating several versions of each vectorizable loop in not even an option for embedded systems, for example.

Our approach has the advantage that it exposes all the opportunities to the JIT compiler in a single version of the loop, while letting it gracefully handle the patterns that do not have hardware support.

3 Implementation

Proper evaluation of our proposal requires aggressive static and JIT compilers to make sure that results are not biased because of poor optimizations unrelated to our focus. Compilers — static and JIT — are huge pieces of software. We leveraged two open-source software projects: GCC for the static compiler, and Mono for the virtual machine and JIT compiler.

We chose to implement our experiments in the GCC compiler for several reasons beyond the availability of the source code. The good quality of the generated code

makes the results trustworthy, and the well documented auto-vectorizer [18], despite its internal complexity, is easy to retarget thanks to the GCC machine model.

We have shown that the CLI format is appropriate for deployment onto embedded systems [4, 6]. CLI is a standard format [8], which means that it is more likely to be portable to various architectures. In fact, several commercial and open-source projects already provide execution environments for the CLI format (see Section 5).

We previously developed a GCC back-end for the CLI format [5, 20], however with very limited support for vector types. One of the contributions of this work is to add vectorization capabilities to the CLI back-end. It is publicly available in the branch `st/cli-be` of the GCC repository.

3.1 Machine Model Technicalities

Activating the GCC vectorizer consists in modifying a few places in the machine description files. First we need to globally instruct the compiler that vector modes are supported by defining the function `cil32_vector_mode_supported_p`.

Then, we need to provide the width of the available vector types. This is accomplished through the macro `UNITS_PER_SIMD_WORD`.

Ideally, the CLI machine description does not need any register at all, since operations are carried on the evaluation stack, and the set of local variables (CLI *locals*) used to store values is infinite, making register allocation a non-issue. Still, GCC needs a minimal set of registers for its own mechanics. In particular, the largest *mode* that can be produced by the vectorizer is computed from the largest set of contiguous registers in the same class. The existing machine description defined only one 32-bit register, thus making vectorization impossible. We increased the number of available registers by modifying the macros `FIXED_REGISTERS` and `CALL_USED_REGISTERS`. We also modified `FIRST_PSEUDO_REGISTER` accordingly.

Finally, we simply add the definition of the supported vector modes and all the supported arithmetic instructions to the machine description file `cil32.md`, as well as the special `movmisalign` instruction used by GCC to generate misaligned accesses.

3.2 Intermediate Representation

We keep the GIMPLE representation, and the vectorizer as a whole, unmodified. Vector types are produced by the vectorizer as usual, based on the information derived from the machine model. The differences appear in the stack based intermediate representation (IR), introduced in [20] to replace RTL in the CLI back-end. This representation was shown to be better for emitting CLI for two main reasons: it has a concept of evaluation stack, and it is strongly typed, a necessary condition to emit correct CLI.

Most vector operations are eventually translated in the bytecode as calls to well defined library functions (builtins). However, function calls tend to make the code more difficult to analyze and optimize. For this reason, we try to postpone the emission of the builtins as much as possible. In particular, we rely on the existing policy in GIMPLE and the stack-based IR assuming that arithmetic operators are polymorphic: we do not add any new arithmetic nodes operating on vectors. Rather, we extend the semantics of existing operators to accept the new vector types.

We handle the vector constructors (the `CONSTRUCTOR` GIMPLE node), with a new IR statement named `VEC_CTOR`. To distinguish vector loads and stores, we also introduce a `LDVEC` and a `STVEC` instruction. Similarly to GIMPLE, we introduce new statements for operators which do not have any scalar equivalent, for example the dot product, or the saturating arithmetic.

We add a pass just before the CLI emission to recognize the vector statements and to transform them to calls to builtins. This pass walks over all the statements of a function, and computes the status of the stack before each of them (this is always possible without dataflow analysis thanks a special constraint of the CLI format, see § III, 1.7.5 of [8]). Arithmetic statements that operate on vector types are replaced by the corresponding builtin. Constructors and `LDVEC` `STVEC` are rewritten.

3.3 Execution Environment

We used Mono [13] as our execution platform. Mono is an open development initiative to develop a UNIX version of the Microsoft .NET environment. It contains a CLI virtual machine — complete with a class loader, a JIT compiler and a garbage collector — as well as a class library and a compiler for the C# language. In this project, we rely on the JIT compiler of the VM.

The Mono environment contains the library Mono.Simd.dll. It defines all the 128-bit vectors types (four single precision floats, four 32-bit integers, eight 16-bit integers, etc.), and the basic arithmetic operations on them. A source implementation of the types and methods is provided in C#, and compiled to CLI. This code is used as a backup for JIT compilers unaware of the special naming convention. We rely on the naming convention defined by Mono in this library. On the x86 platform, Mono's JIT compiler recognizes the special semantics. Many other platforms are supported by Mono, but the SIMD extensions are not implemented yet.

4 Experiments and Analysis

This section presents our experiments with some loop kernels. It shows how initial results were far from acceptable, and it analyzes what are the minimum conditions to produce efficient code.

4.1 Setup

Our goal is to illustrate the advantage of vectorizing *bytecode*. The focus in on the specificities of target independent bytecode. Effectiveness of vectorization as an optimization technique is *not* the point of our work, it has been proved already elsewhere, and we take it for granted, as any other optimization mentioned in this paper. For this reason, we decided to show some results only on small kernels that illustrate the *key features* of vectorization. Using real application would only blur the specific behaviors we are interested in. We use the same benchmarks as [17]. See Table 1 for a short description. They cover several data types and type sizes (single precision floating point, 8-bit and 16-bit integers). They also illustrate various features of the vectorizer: simple arithmetic, reductions, use of a constant. All kernels operate on arrays of 1000 elements, except *sum_u8* and *sum_u16* which operate on

Name	Description	Data type	Features
vecadd_fp	addition of two vectors	floating point	arithmetic
sdot_fp	dot product of two vectors	floating point	reduction
saxpy_fp	constant times a vector plus a vector	floating point	constant
dscal_fp	scale a vector by a constant	floating point	constant
max_u8	find maximum over elements of a vector	8-bit char	reduction
sum_u8	sum the elements of a vector	8-bit char	reduction
sum_u16	sum the elements of a vector	16-bit short	reduction

Table 1. Description of the benchmarks

10,000 elements to keep the running time in the order of a few seconds. Each kernel is also wrapped by a main loop that executes many times.

Since those benchmarks, from the BLAS suite, are written in Fortran, we wrote a straightforward implementation in C. We used the latest release of Mono at the time of writing: version 2.4.2.3. The CLI backend is based on GCC version 4.4. In order to demonstrate both the performance advantage and the portability, we run our experiments on several hardware platforms:

- a desktop PC featuring an Intel Core2 Duo clocked at 3 GHz — supporting the SIMD extensions MMX, SSE, SSE2, SSSE3 — running Linux 2.6.27;
- a Sun Blade 100 featuring a TI UltraSparc IIe, clocked at 500 MHz, running Linux 2.6.26.

Note that, even though the UltraSparc has a SIMD instruction set extension VIS, the Mono JIT compiler does not exploit these extensions yet. It does support the SSE extension on the x86 architecture. We also simulate an x86 platform without SIMD support by running the experiments on the same desktop PC and by disabling the SIMD intrinsics recognition.

4.2 Initial Results

In this first experiment, we run the benchmarks with three configurations on the x86 platform. In all cases of Table 2, the C programs are compiled to bytecode, and the bytecode is run by the Mono JIT compiler.

1. Firstly, we generate the bytecode without the vectorizer. That is, we use GCC with the command line flags `-O2 -fno-tree-vectorize` (note that, as of today, `-O2` would be sufficient since the vectorizer is only enabled at `-O3`). This gives us our baseline, reported in the column *scalar* of Table 2.
2. Secondly, we compile the benchmarks again, with the vectorizer enabled, using `-O2 -ftree-vectorize`. Running times are reported in the columns *vectorized* as absolute values and as performance relative to the scalar version (defined as the scalar base time divided by the new time). The next column, labeled *max* indicates the rough maximum speedup one would naively expect, based on the data type.
3. Finally, the vectorized bytecode produced in the previous step in run again, but without SIMD support in the JIT compiler. On x86, this is achieved by adding the flag `--optimize=-simd` to Mono.

benchmark	scalar	vectorized			no SIMD	
	time	time	rel. perf.	max	time	rel. perf.
vecadd_fp	1893	815	2.3	4	4475	0.42
sdot_fp	3039	3039	1.0	4	3039	1.0
saxpy_fp	2429	1224	2.0	4	8531	0.28
dscal_fp	1921	733	2.6	4	4122	0.47
max_u8	3156	346	9.1	16	6101	0.52
sum_u8	9030	2613	3.5	16	32528	0.28
sum_u16	9334	5223	1.8	8	42339	0.22

Table 2. Initial performance results on x86 (10^6 iterations, time in ms)

benchmark	scalar	previous	no SIMD	
	time	rel. perf.	time	rel. perf.
max_u8	3156	0.52	5185	0.61
sum_u8	9030	0.28	20569	0.44
sum_u16	9334	0.22	27565	0.34

Table 3. Performance results on x86, with inlining

At first glance, we can make the following high-level comments:

- *sdot* is not vectorized. This is confirmed by the activating the vectorizer's debugging messages. Figuring out the cause of this missed optimization was beyond the scope of this paper, and we omitted *sdot* in the rest of this paper.
- The vectorized bytecode shows significant speedups, ranging from 1.8 to 9.1.
- Even though the speedups are high, one might expect even more: 32-bit floating point values packed in 128-bit vectors gives an upper bound value for the speedup of 4 for the benchmarks *vecadd*, *saxpy*, *dscal*. The kernels *max_u8* and *sum_u8* operate on 8-bit integers, let us expect a 16x speedup. *max_u8* achieves a reasonable score of 9.1, but *sum_u8* obviously has a problem.
- The performance of the vectorized code run without SIMD support is unacceptable, with relative performance in the range 0.22 to 0.52.

4.3 Missed Inlining

We first look at the disastrous performance of the code in the absence of SIMD support, especially the bottom two kernels of Table 1. It turns out that the function which implements the vector operation (sum or max) is not inlined as we expected. It is inlined in the case of the floating point kernels. We found out that the coding style in Mono.Simd differs according to the size of the vector: 4-element vectors arithmetic (e.g. floating point) is implemented as four sequential statements, while 8-element and more vectors are implemented with a loop. This loop changes the outcome of the inlining heuristic of the JIT compiler. We rewrote the code of the library and reran those three tests, obtaining the numbers of Figure 3.

Unfortunately, the call to max is still not inlined, but we still obtain slightly better code for *max_u8*. Even though the slowdown of the *sum* kernels is still unacceptable, the improvement is also noticeable.

benchmark	scalar	vectorized			no SIMD	
	time	time	rel. perf.	max	time	rel. perf.
vecadd_fp	1197	537	2.2	4	1228	1.0
saxpy_fp	1544	724	2.1	4	1890	1.2
dscal_fp	1045	657	1.6	4	1095	1.0
max_u8	3541	227	15.6	16	3735	1.1
sum_u8	6707	1277	5.3	16	8925	1.3
sum_u16	6710	2547	2.6	8	8198	1.2

Table 4. Final results on x86

4.4 Manual Improvements

We then take advantage of the tracing capabilities of the JIT compiler and we dump the code emitted for *vecadd* for manual inspection. Functions calls are effectively inlined, however the resulting x86 code looks extremely poor. The loop contains 91 instructions. In comparison, the emitted code for the scalar version is 12 instructions long. Since combining vectorization and inlining amounts to unrolling the loop four times, one would expect in the order of 48 instructions, not taking into account induction variable simplification. Careful analysis shows that the input values (the vector elements) are copied twice in the function stack frame before being loaded in the floating point unit, and the output values are copied three times on their way from the floating point unit to their final destination. Our hypothesis is that the JIT compiler is missing a simple pass of copy propagation and dead code elimination after inlining.

We manually optimize the code produced by Mono, and link it again with the loop driver in the main program[1]. Since the loop is a single basic block, copy propagation and dead code elimination are straightforward, and very much within the capabilities of a JIT compiler. Running the experiment again, we measure 1228 ms, which is 3.6 times faster that the original. We apply the same simple cleanup to all vectorized benchmarks, we obtained improvements ranging from 3.6 to 5.2, with the exception of *max_u8* which scores "only" 1.6 because of the presence of the function calls.

In the interest of fairness, we apply the same simple manual code optimizations on all the generated versions of the loops of Table 2: scalar, vectorized, and vectorized without SIMD support. The consolidated results are presented in Table 4.

4.5 UltraSparc

We run the same set of experiments on the UltraSparc workstation, with the exception that Mono does not implement the SIMD code generation for this processor yet. All the finding on the x86 architectures have their counterpart on the UltraSparc, which is not surprising since the JIT compiler engine is the same. Because of the RISC nature of the UltraSparc, loading a long immediate in a register requires two instructions, and the address computation of an array element requires a third

[1] In doing this, we potentially bias our results by omitting the compile-time of the JIT compiler. We verified that this is not the case: Mono reports compiling time below 0.25 ms for each of our loop kernels.

benchmark	automatic			after manual cleanup		
	scalar	vectorized	rel. perf.	scalar	vectorized	rel. perf.
vecadd_fp	4215	11193	0.38	2810	1947	1.4
saxpy_fp	5216	21011	0.25	3812	3239	1.2
dscal_fp	3642	10687	0.34	2608	1787	1.5
max_u8	3151	8613	0.37	3032	3188	0.95
sum_u8	10022	65864	0.15	8019	8559	0.94
sum_u16	10756	109866	0.10	8788	11256	0.78

Table 5. Final performance on Sparc, 10^5 iterations, time in ms

instruction for the addition. Since the first two instructions load a constant address, they are loop invariant and it was an additional obvious manual optimization to hoist them outside the loop. Table 5 shows the final results.

Speedups come from the fact that inlining the vector operations is similar to unrolling the small loops. In the case of the reduction kernels, this speedup is offset by the reduction epilogue and the less-than-optimal code generation.

4.6 Final Comments

The optimizations we run manually are basic compiler optimizations, which run in linear time. They are by no means out of reach of a JIT compiler. It should simply be a matter of running them again before code emission.

The generated vectorized code can be further improved, beyond our manual cleanup. In particular, intermediate results are continuously loaded and stored on the stack at each iteration. The scalar version manages to keep intermediate values in registers. Inductions variables are not optimized in the vectorized code as aggressively as in the scalar code. Those reasons explain why the floating point benchmarks do not reach a better speedup.

Reduction kernels are impacted by another problem. An epilogue performs the final reduction over the 8 or 16 elements of the vector result. This code is less optimized than the loop body, and it impacts to total performance. Adding new builtins for *horizontal* operations (like the sum of the elements of a vector) could help the JIT compiler generate better code even for the epilogue.

5 Related Work

Bytecode formats and JIT compilers have existed for decades. CLI has recently drawn a lot of attention. Many projects exist, both commercial and open source. Microsoft proposed the .NET [16] framework. Mono [13] has been presented in Section 3. ILDJIT [2] is a distributed JIT compiler for CLI. LLVM [11] is a compiler infrastructure that defines a virtual instruction set suitable for sophisticated transformation on object code.

The original work in the GCC auto-vectorizer has been presented by Nuzman and Henderson in [17, 18]. They showed how the vectorizer algorithms can be implemented in a target-independent way, and driven by the compiler machine model. In

their approach, the generated code *is* target-dependent, while in ours even the generated code is target-independent. We postpone the specialization to the run-time.

Leśnicki et al. [12] proposed to annotate the CLI bytecode with information to help the run-time vectorizer. In that approach, the vectorization happens at run-time, but the static compiler does the analysis and hints at the interesting loops. However, marking all the relevant loops and variables with the appropriate and usable information, while keeping legal CLI code, proved difficult. In this paper, we propose to entirely apply the vectorization in the static compiler, and to possibly revert to scalar code when necessary.

El-Shobaky et al. [9] also propose to apply the vectorization at run-time. They unroll loops to duplicate loop bodies, and they modify the code selector of the JIT compiler to group corresponding instructions using tree pattern matching techniques. However only a small number of operators are supported, and the number of additional rules in the code selector will grow rapidly when more complex patterns are needed. The approach also suffers a number of limitations, as described in the paper.

Vector LLVA [1] is similar to our approach, but for the LLVM instruction set. However, in contrast to our proposal, the authors do extend the bytecode itself, breaking the compatibility, and they do not investigate the behavior of vector code on non-SIMD machines.

Clark et al. propose to rely on a simple dynamic translation mechanism to recognize the instruction patterns that can be vectorized (those patterns have been produced by scalarizing the output of a vectorizing compiler). They are able to handle data widths increases. The solution, however, is entirely in hardware and thus limited in the size of the instruction window in which it can recognize patterns.

6 Conclusion

In this paper, we propose a scheme to reconcile platform neutral binary formats like bytecodes, and the careful exploitation of SIMD extensions of instruction sets.

Our contribution is three fold: first, we added vectorization capability to the GCC CLI backend and we made our developments publicly available. Second, we showed that vectorized bytecode is a viable approach to deliver performance in the presence of SIMD instructions while not incurring any penalty on non-SIMD instruction sets. And third, we analyzed under what conditions the bytecode produced by the static compiler can be efficiently executed on various processors. In particular, the JIT compiler must guarantee that some basic optimizations will be run in order to not to degrade the performance.

References

1. Robert L. Bocchino, Jr. and Vikram S. Adve. Vector LLVA: a Virtual Vector Instruction Set for Media Processing. In *VEE'06*, pages 46–56, June 2006.
2. Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.
3. Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA'07*, pages 216–227, Washington, DC, USA, 2007.

4. Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *Conference on HiPEAC*, pages 130–144, Göteborg, Sweden, January 2008. Springer.
5. Roberto Costa, Andrea C. Ornstein, and Erven Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.
6. Roberto Costa and Erven Rohou. Comparing the size of .NET applications with native code. In *3rd Intl Conference on Hardware/software codesign and system synthesis*, pages 99–104, Jersey City, NJ, USA, September 2005. ACM.
7. Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050 of *LNCS*, pages 5–29. 2007.
8. ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.
9. Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 63–69, New York, NY, USA, 2009. ACM.
10. Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, February 2008.
11. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, Palo Alto, California, Mar 2004.
12. Piotr Leśnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andrea Ornstein, and Erven Rohou. Split compilation: an application to just-in-time vectorization. In *GREPS'07, in conjunction with PACT*, Braşov, Romania, September 2007.
13. The Mono Project. `http://www.mono-project.com`.
14. José M. Moya, Javier Rodríguez, Julio Martín, Juan Carlos Vallejo, Pedro Malagón, Álvaro Araujo, Juan-Mariano Goyeneche, Agustín Rubio, Elena Romero, Daniel Villanueva, Octavio Nieto-Taladriz, and Carlos A. López Barrio. SORU: A reconfigurable vector unit for adaptable embedded systems. In *ARC '09: Proceedings of the 5th Intl. Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pages 255–260, 2009.
15. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
16. Microsoft .NET. `http://www.microsoft.com/NET`.
17. Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO'06*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.
18. Dorit Nuzman and Ayal Zaks. Autovectorization in GCC – two years later. In *GCC Developers' Summit*, pages 145–158, June 2006.
19. Erven Rohou. Combining processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Emerging Uses and Paradigms for Dynamic Binary Translation*, number 08441 in Dagstuhl Seminar Proceedings.
20. Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A stack-based internal representation for GCC. In *GROW'09*, pages 37–48, Paphos, Cyprus, 2009.