Speculatively Vectorized Bytecode

Erven Rohou
INRIA Rennes – Bretagne Atlantique
Rennes, France
erven.rohou@inria.fr

Sergei Dyshel
IBM Haifa Labs
Haifa, Israel
sergeid@il.ibm.com

Dorit Nuzman
IBM Haifa Labs
Haifa, Israel
dorit@il.ibm.com

Ira Rosen
IBM Haifa Labs
Haifa, Israel
irar@il.ibm.com

Kevin Williams
INRIA Rennes – Bretagne Atlantique
Rennes, France
kevin.williams@inria.fr

Albert Cohen
INRIA Saclay – Île-de-France
Orsay, France
albert.cohen@inria.fr

Ayal Zaks
IBM Haifa Labs
Haifa, Israel
zaks@il.ibm.com

ABSTRACT

Diversity is a confirmed trend of computing systems, which present a complex and moving target to software developers. Virtual machines and just-in-time compilers have been proposed to mitigate the complexity of these systems. They do so by offering a single and stable abstract machine model thereby hiding architectural details from programmers.

SIMD capabilities are common among current and expected computing systems. Efficient exploitation of SIMD instructions has become crucial for the performance of many applications. Existing auto-vectorizers operate within traditional static optimizing compilers, and use details about the target architecture when generating SIMD instructions. Unfortunately, auto-vectorizers are currently too complex to be included in a constrained Just-In-Time (JIT) environment.

In this paper we propose *Vapor SIMD*: a speculative approach for effective just-in-time vectorization. Vapor SIMD first applies complex ahead-of-time techniques to vectorize source code and produce bytecode of a standard portable format. Advanced JIT compilers can then quickly tailor this bytecode to exploit SIMD capabilities of appropriate platforms, yielding up to 14.7× and 11.8× speedups on x86 and PowerPC platforms (including JIT-compilation time). JIT compilers can also seamlessly revert to non-vector code, in the absence of SIMD capabilities or in the case of a third-party non-vectorizing JIT compiler, yielding 93% or more of the original performance.

HiPEAC 2011 Heraklion, Crete, Greece

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*; D.2.13 [**Software Engineering**]: Reusable Software; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Performance, Languages, Algorithms

1. MOTIVATION

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, requires resource-hungry compiler analyses and optimizations.

1.1 Applications Lifetime and Processor Heterogeneity

Software applications have longer lifetimes than the hardware they run on. These long-living applications are referred to as *legacy code*. To support legacy code, hardware vendors offer binary compatibility between generations of processors. However, this compatibility is limited. Legacy code can only take advantage of increased clock frequency and improved micro-architecture, but not of additional architectural features. Embedded systems rarely offer binary compatibility because of the induced costs and inefficiencies.

Diversity of computing systems is a global trend. On embedded systems, this trend has been driven by the drastic constraints on cost, power and performance. On general purpose computers, variability exists in the availability of a floating point unit, width of SIMD units, number of cores, type and features of the GPU, etc. It is predicted that technology variability will limit the proliferation of homogeneous

many-cores, and that the large number of available cores will push to specialize cores for dedicated tasks [7].

The ubiquity of computing devices drives systems designers to efficiently support the same application on dozens of platforms, some of which may not be known or well defined at compile time. Following Java's "write once run anywhere" approach, processor virtualization (bytecode formats and just-in-time compilers) has been proposed to deal with heterogeneity [7]. Bytecode can be deployed to any system provided an interpreter or JIT compiler is available for it. Application developers are not required to know the hardware on which their code will eventually run.

Processor virtualization addresses the problem of both legacy code and heterogeneous processors by: (1) reducing the burden laid upon software developers who no longer need to deal with varying hardware, (2) guaranteeing that application lifetimes can span several generations of hardware, and, (3) making it possible for legacy code to exploit new hardware features. For example, applications written before floating point hardware was available can still take advantage of a FPU, since the JIT compiler for the new platform will take it into account.

1.2 Exploitation of Word-Level Parallelism

The successful exploitation of SIMD instructions is crucial for the performance of many applications. All major hardware vendors provide SIMD extensions (SSE on x86 processors, AltiVec on PowerPC, VIS on Sparc, NEON on ARM, etc.) They also continuously add new vector instructions (SSE4.1, SSE4.2, SSE4a, VSX, etc.) Programmers commonly use source-level *builtins* (a.k.a. *intrinsics*) to take advantage of SIMD units. Alternatively, compilers have included auto-vectorization as an advanced optimization. Despite significant progress, good automatic vectorization is still an open and difficult problem [4, 14, 16, 2], for several reasons:

1. strong conditions must be met by the code and proven by the compiler, in particular in terms of data dependencies;
2. complex patterns such as nested loops and strided access need to be detected and handled, to extend the applicability and scope of vectorization;
3. each architecture has its own set of SIMD instructions and associated constraints such as alignment requirements, available registers, etc.

1.3 Just-in-time Vectorization

Applying automatic vectorization ahead-of-time in a classical compiler is already a difficult task, occupying over 20,000 lines of code in GCC; designing an efficient just-in-time vectorizer running in memory- and CPU-constrained environments is far more challenging. Conversely, applying auto-vectorization *ahead of time* to a bytecode representation is not feasible, as the features and constraints of the execution platform may not be known at static compilation-time (or would defeat the portability purpose). In this paper we show how a new *Vapor SIMD* approach can reconcile auto-vectorization with processor virtualization. We prove that portable vectorized bytecode is a viable approach: it can yield significant speedups using lightweight JIT transformations to generate SIMD instructions, while incurring minor or no penalty in their absence.

2. RECONCILING VECTORIZATION AND PROCESSOR VIRTUALIZATION

Any just-in-time vectorizer must meet the following three main objectives:

- **Robust.** The approach must be general enough to allow execution, both when using SIMD capabilities and also in the absence of SIMD extensions, or when using an unmodified, non-vectorizing JIT compiler
- **Risk-free.** The penalty of running vectorized bytecode without SIMD support is kept at a minimum.
- **Efficient.** The improvement of running vectorized bytecode with SIMD support is maximized.

Our proposed solution for achieving these goals is based on the notion of *split-compilation* [5]. The source code undergoes two (or more) separate compilation stages before being translated into machine code. Optimizations are carefully coordinated and distributed over these compilation stages. Vapor SIMD splits compilation into two stages:

1. The first stage translates source code into bytecode. This happens on the programmer's workstation, *ahead-of-time*. Resources available to the compiler are virtually unlimited: gigahertz, gigabytes and minutes of compile-time are common. However, the target is a neutral bytecode: it is independent of the platform on which the application will eventually run. We refer to this stage as the *offline* compiler.
2. The second stage converts the bytecode into machine code. This takes place *just-in-time*, i.e. on the final device, and at run time. Resources are likely to be limited, especially on an embedded system. Furthermore, compile time is visible to the user, hence it must be as efficient as possible. We refer to this stage as the *online* compiler.

The crux of split-compilation is to move as much complexity as possible from the online stages to the offline ones. Offline stages are responsible for all target independent optimizations; expensive analyses can be run, and the results encoded in the bytecode. Online stages later use these encodings to both reduce compilation time and increase code quality.

Aggressive offline stages address the first two difficulties of automatic vectorization presented in Section 1.2. Online stages allow for fine adjustments to the actual instruction set. Online code generation also makes it possible to exploit properties that could not be proven statically, but are discovered at run time. We did not explore the latter option in this paper, but it is a well known benefit of deferred compilation scenarios, including online adaptive optimization [1] and partial evaluation [6].

3. IMPLEMENTATION

We implemented Vapor SIMD using the CLI ECMA 335 bytecode format [8]. We convey vector information via a naming convention that applies to vector types and methods. Vector operations in bytecode are implemented as standard method calls. Note that we do not extend the CLI format — the vectorized bytecode we produce runs unmodified on any CLI-compliant virtual machine. The semantics of the special types and methods are provided in an auxiliary library (.dll).

3.1 Naming Convention

In a split-compilation scheme, the offline compiler must carry the vector semantics through the bytecode to facilitate the work of the online compiler. We chose to rely on a naming convention for this purpose. More specifically, the producer and the consumer of the bytecode need to agree upon:

- Type names: the online compiler needs to be aware of vector types and map them to the correct hardware registers (e.g. `%xmm` on SSE).
- Arithmetic operators: the online compiler must inline and optimize vector operations implemented as method calls in bytecode.
- Aligned and unaligned memory accesses: *unaligned* accesses are generated. A special method name can be generated for aligned accesses, for the JIT to generate optimized code (e.g., when the offline compiler peels a loop to align a memory access).

3.2 Code Generation

We define a *conscious-JIT* as a JIT that is aware of the naming convention used by the offline compiler to convey vectorization opportunities, and can generate efficient SIMD instructions accordingly. Conversely, an *agnostic-JIT* is any JIT that can process the same standard bytecode but is unaware of this naming convention or cannot make use of it to generate SIMD instructions (e.g. due to lack of hardware support).

The above naming convention can be further developed into several split-compilation configurations. These are presented in Figure 1 where: (A) is a fully scalar configuration with no vectorization, (B) has offline vectorization with online execution by a *conscious-JIT* and (C) has offline vectorization with online execution by an *agnostic-JIT*.

Figure 2 illustrates the code generation for a simple vector addition. Column (a) shows the C source code for this simple loop. Column (b) shows a segment of the vectorized CLI bytecode produced by the offline compiler. `Vector4f`, a type defined by our naming convention, is a vector of four single precision floating point numbers. The three `ldloc` instructions load the addresses of the vectors `'a'`, `'b'` and `'c'` onto the stack. The method `Add` performs the addition on vectors `'b'` and `'c'` and stores the result in `'a'`. The remaining instructions increment the induction variable `'a'`.

On an x86 target, a conscious-JIT translates this bytecode into that of column (c). It recognizes the variables of type `Vector4f` and emits `movups` instructions (move unaligned packed single precision) that load vectors into SSE registers. Similarly, `Vector4f::Add` is recognized and a single `addps` instruction (add packed single precision) is emitted.

Column (d) presents a similar online compilation stage performed by an agnostic-JIT running on an x86 target. Here, the methods of our supporting library are inlined into the calling method. After applying standard optimizations (copy and constant propagation, constant folding) the scalar code of column (d) is emitted. The net effect of this agnostic-optimization is equivalent to loop tiling with a tile size equal to the vector width.

This example illustrates how we achieve the three objectives: in column (b) robustness is achieved with our naming convention (and supporting library), efficiency is achieved by our conscious-JIT in column (c) and column (d) demonstrates that using an agnostic-JIT is risk-free as it inlines and optimizes our speculative vector code.

3.3 Design Considerations

SIMD instruction sets vary considerably in number of supported idioms, expressiveness, and constraints. Many choices can be made to best match the abstract vector representation of the bytecode to the actual instances of vector instruction sets. Our experimental setup uses the GCC toolchain and we largely follow the decisions made in the GCC GIMPLE representation [16].

Alignment constraints and realignment idioms are a typical burden on vectorizing compilers. We face the additional problem that the offline compiler does not know whether the target supports unaligned accesses or not. This leaves us with two options:

1. Support unaligned accesses in the bytecode. The offline compiler generates vector loads/stores that directly access the desired addresses without realignment.
2. Require aligned memory accesses in the bytecode. In this case, the offline compiler generates realignment code explicitly in the bytecode. The online compiler is guaranteed to see only aligned memory accesses.

The second approach allows the online compiler to easily generate efficient code even for platforms that do not support misaligned accesses (like the Cell SPU ISA and PowerPC's AltiVec). Such platforms require loop-level analyses and transformations in order to generate high-quality realignment, which are best handled by the static compiler. This approach creates relatively complicated bytecode, involving explicit realignment idioms, and special setup code. For platforms that do not support these idioms, but instead support misaligned accesses (like the x86's SSE), this approach requires that the online compiler would be able to reverse the optimization, which is a non-trivial task in general. For this reason it is also less suitable for the agnostic-JIT scenario.

The first approach, on the other hand, generates much simpler bytecode, with misaligned accesses that can be directly mapped to the instructions of the target platform, even by an agnostic-JIT (i.e., a sequence of scalar accesses). The drawback here is that it may not result in the best quality vector code for platforms that have better means to deal with misaligned accesses. However, the offline compiler can pass alignment information to the online compiler, to prevent generating unnecessary realignment operations when arrays are known to be properly aligned. In particular, when the offline compiler can prove that accesses are already aligned, it can generate special "aligned load/store" method calls.

For these reasons the first approach is more suitable for the purposes of this study. More specifically, all vector loads will thus be expanded to generic, unaligned method calls by the offline compiler. But due to the higher cost of misaligned stores, the offline compiler attempts to peel vectorized loops to generate special aligned store method calls [16] (generation of a scalar prolog to align the stores in the vectorized kernel).

Vector width is another parameter dictated by the target architecture, hence unknown to the offline compiler. We take

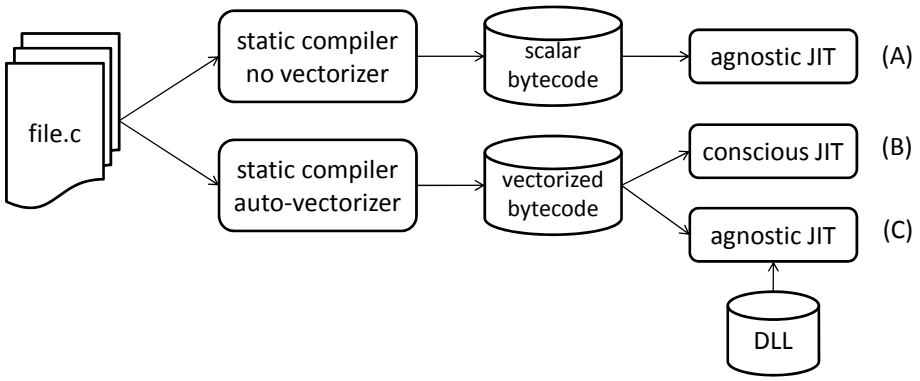

Figure 1: Scenarios: (A) regular flow; (B & C) Vapor SIMD flows

`float a[N];`
`float b[N];`
`float c[N];`
`for (i=0; i<n; ++i)`
`{`
`  a[i]=b[i]+c[i];`
`}`
	`.locals (Vector4f* 'a')`
	`ldloc 'a'`
	`ldloc 'b'`
	`ldloc 'c'`
	`call Vector4f::Add`
	`  ...`
	`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

Figure 2: Code generation schemes

the following approach — based on the fact that most current architectures support 128-bit wide vector operations, this is the width we vectorize for. An architecture with a different vector width will fall back on the scalar implementation as described earlier. This is again a lost opportunity, but not a performance degradation. A more advanced online compiler could try to adjust the width to fit its architecture, but this needs additional data dependence analysis at run-time, or extra annotations to specify the maximum vectorization factor for each loop. This is beyond the scope of this paper and is a subject of future work.

4. EXPERIMENTAL SETUP

To evaluate our Vapor SIMD approach we make use of the following tools, depicted in Figure 3. We use GCC version 4.4 to perform offline auto-vectorization of C benchmarks. The gcc4cli back-end emits vectorized CLI bytecode, it is also based on GCC version 4.4 [20]. We use two online CLI environments: Mono virtual machine [13] development version 2.8 and Microsoft .NET framework version 2.0. This .NET framework is a standard installation and is agnostic, being unaware of the offline vectorization and naming convention. The Mono environment contains the Mono.Simd.dll library. It defines 128-bit vectors types (four single precision floats, four 32-bit integers, eight 16-bit integers, etc.), and the basic arithmetic operations to manipulate them. We adopt the naming scheme of this Mono library to create our split-compilation naming convention. We also augment this library to support more combinations of types and methods.

On the x86 platform, Mono's JIT compiler already supports the SSE instruction set. In order to validate our framework on multiple SIMD platforms we extended Mono's PowerPC port to support the AltiVec instruction set. It provides a mapping from the Mono.Simd library to the equivalent AltiVec instructions.

Conscious-JITs directly map vectors to machine registers, therefore an ABI that passes vectors by value is preferable. An agnostic-JIT, which must support 16-byte data structures, favors passing parameters by address. To accommodate this difference, our conscious-JIT includes a CLI-to-CLI pre-pass. In principal this pre-pass is part of the JIT. We could also modify Mono to handle the precise output of GCC. However, in software engineering terms, it proved simpler to write a small pass to adjust the CLI to what Mono already recognizes natively.

We use benchmarks similar to those used by the authors of the GCC vectorizer [16]; see Table 1 for a short description. They cover several data types and type sizes (single precision floating point (fp), 8-bit and 16-bit integers). They also illustrate various features of vectorization: simple arithmetic (first group), integer reduction (second group), and floating point reduction (third group), potentially also using constants. Each kernel is wrapped in a main loop that executes many times. We used the latest version of Mono at the time of writing (development version 2.8) as our conscious JIT, with all its optimizations enabled (`-O=all`), and also used it as an agnostic-JIT by disabling its SIMD capabilities (`-O=all,-simd`).

5. EXPERIMENTS AND ANALYSIS

The experimental analysis in this section shows how our three objectives are met. **Robustness** is implicitly proven by the fact that our bytecode runs on different platforms and JIT compilers. We present the following experiments:

A. the performance of vectorized bytecode with several agnostic JITs (**Risk free** objective);

Name	Description	Data type	Features
vecadd_fp	add two vectors	floating point	arithmetic
saxpy_fp	constant times a vector plus a vector	floating point	constant
dscal_fp	scale a vector by a constant	floating point	constant
max_s16	find maximum over elements of a vector	16-bit signed short	reduction
max_u8	find maximum over elements of a vector	8-bit unsigned char	reduction
max_s8	find maximum over elements of a vector	8-bit signed char	reduction
sum_u8	sum the elements of a vector	8-bit unsigned char	reduction
sum_u16	sum the elements of a vector	16-bit unsigned short	reduction
sdot_fp	dot product of two vectors	floating point	reduction
sum_fp	sum the elements of a vector	floating point	reduction

Table 1: Benchmark description

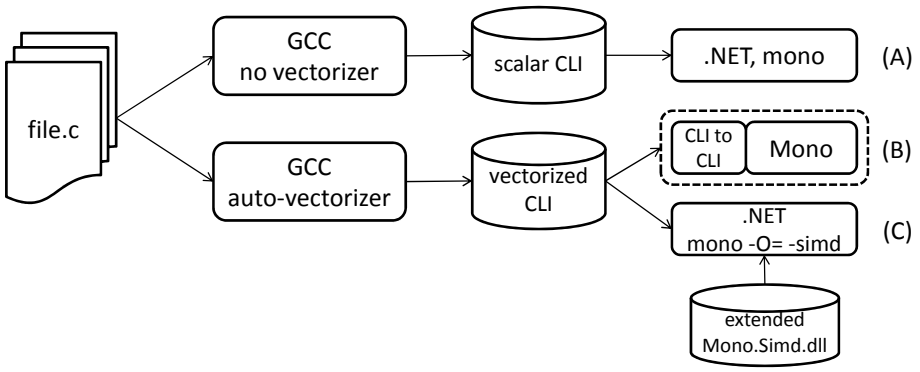

Figure 3: Implementation of scenarios

B. the performance of vectorized bytecode with a conscious-JIT (**Efficiency** objective);

C. a closer look at the behavior of vectorized bytecode on an x86 platform across SSE versions;

D. the compile time and code size implications of bytecode vectorization.

In this section, we evaluate the performance of our vectorized bytecode on different platforms. Our goal is to illustrate the viability of speculatively vectorizing bytecode for an unknown target architecture. Therefore, our main focus is efficiency across a range of different platforms, both with and without SIMD support, rather than the effectiveness of vectorization itself.

5.1 Performance of vectorized bytecode, agnostic JIT

The purpose of this first experiment is to validate our risk-free objective — incur only minor penalties in an agnostic scenario. It also implicitly validates the robustness objective — vectorized bytecode correctly runs on arbitrary CLI platforms.

The goal is to obtain reasonable performance of vectorized-then-scalarized code, despite the overheads. Some are inherent to the vectorization process, others are related to the bytecode representation of the vector idioms: calls to Mono.Simd library functions and vector variables are structures.

We experimented with three different architectures and two JIT compiler technologies, all combinations being agnostic:

- an Intel Core2 Duo at 1.33 GHz, Windows XP Pro and .NET version 2.0.50727;
- an Intel Core2 Duo at 3 GHz, Fedora 12 Linux 2.6.31 and Mono 2.8 with SIMD support disabled;
- a Sun Blade 100 TI UltraSparc IIe, at 500 MHz, Linux 2.6.26 and Mono 2.6.3;
- a PowerMac 7450 at 800 MHz, Linux 2.6.29 and Mono 2.8 with SIMD support disabled.

The VIS SIMD extensions of UltraSparc are not supported by the Mono JIT compiler version 2.6.3, and the JIT compiler of the .NET Framework is not aware of the vector naming convention, thus turning these two platforms effectively into agnostic platforms.

Table 2 presents the run times for these agnostic platforms. For each platform, the first column shows the run time t_A of the reference, non vectorized, bytecode (scenario A in Figure 3), the second column shows the run time t_C of the vectorized bytecode (scenario C), and the third column computes the relative performance as a speedup t_A/t_C. The first average row uses the automatically generated scalar versions of *sdot_fp* and *sum_fp*, whereas the second (italic) *average_opt* row uses manually optimized versions of these kernels (*sdot_fp_opt* and *sum_fp_opt*, as detailed below).

These numbers confirm that the penalty of running vectorized bytecode through an agnostic-JIT is limited. Moreover, in many cases performance even improves.

Looking at overall averages, only the .NET platform exhibits performance degradation, of 7% and 14%. The internals of the .NET platform are not documented and its performance is difficult to analyze. However, the JIT has

benchmark	.NET x86			Mono x86 (no SIMD)			UltraSparc			Mono PowerPC (no SIMD)		
	scalar	scalarized	rel.	scalar	scalarized	rel.	scalar	scalarized	rel.	scalar	scalarized	rel.
vecadd	6281	5062	1.2	1891	1972	1.0	4214	3918	1.1	2497	1915	1.3
saxpy	6984	10015	0.7	2433	3646	0.7	5226	7451	0.7	2528	3017	0.8
dscal	4750	5437	0.9	1836	1884	1.0	3652	3774	1.0	1646	1629	1.0
max_s16	2593	2609	1.0	1787	792	2.3	2810	2502	1.1	889	949	0.9
max_u8	2218	2734	0.8	1845	783	2.4	2968	2493	1.2	798	1019	0.8
max_s8	2376	2876	0.8	3060	955	3.2	3210	2617	1.2	908	1181	0.8
sum_u8	17265	19922	0.9	10632	9878	1.1	10034	8449	1.2	5054	6345	0.8
sum_u16	22344	24250	0.9	10678	7715	1.4	10771	11542	0.9	5054	6340	0.8
sdot_fp	4609	4328	1.1	5060	1150	4.4	7261	3533	2.1	3921	1625	2.4
sum_fp	2328	2266	1.0	5035	798	6.3	5832	2334	2.5	3928	869	4.5
sdot_fp_opt	*3234*		*0.8*	*1862*		*1.6*	*6904*		*1.7*	*2447*		*1.5*
sum_fp_opt	*1375*		*0.6*	*1363*		*1.7*	*4348*		*1.9*	*1752*		*2.0*
average			0.93			2.38			1.30			1.41
average_opt			*0.86*			*1.64*			*1.20*			*1.07*

Table 2: Performance of (scalarized) vectorized bytecode on agnostic-JIT (time in milliseconds)

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

```
for (i=0; i<n; i+=4) {
  s0 += a[i];
  s1 += a[i+1];
  s2 += a[i+2];
  s3 += a[i+3];
}
s = s0 + s1 + s2 + s3;
```

Figure 4: Effect of unrolling and modulo variable expansion

a low code size threshold for inlining [15]. Even though the arithmetic operations in Mono.Simd.dll are small, they might not be inlined. This limits the benefit of tiling and unrolling (inherent in vectorized code) and may explain why the relative performance is close to 1.

On other platforms, degradations are observed only on the *saxpy* kernel, and on the integer-reduction kernels on PowerPC, as we now explain. In the first group of kernels, the unrolling effect of scalarization compensate the offline vectorization overheads, which in this case include loop-peeling to align the store instruction. In *saxpy* however, vector temporaries are needed for intermediate computations, and because vector types are represented as structures, these temporaries are allocated on the stack instead of keeping them in registers. We are exploring alternative scalarization schemes to avoid this overhead.

The next two groups of kernels feature reduction computations. The effect of scalarizing vectorized reduction code (with the arithmetic operation inlined) is loop unrolling followed by modulo variable expansion (MVE). The code sample of Figure 4 illustrates this effect. Here we can see the critical path in the scalar loop is the circular data dependence on the accumulator. In the scalarized loop, it has been split into four independent components, thus improving performance. However, this transformation may result in aggressive unrolling (the char and short kernels are unrolled 16 and 8 times respectively) spilling of intermediate values, and in turn additional memory traffic.

On PowerPC this additional memory traffic is the key factor behind the performance degradation of the integer kernels (second group). The x86 platform is less sensitive to the additional memory traffic, but benefits from the relaxed dependences due to the MVE effect.

Finally, the scalarized loops in the last group of floating point reduction kernels benefit from MVE and do not suffer from spilling (as the scalarized loop is unrolled only 4 times). In addition, the lack of global-register allocation for floating-point registers in Mono results in very poor code generated for the scalar versions, which further magnifies the positive effect of MVE in the scalarized versions.

To confirm this analysis, we examine the effect of scalarization in the hand optimized (unrolling and MVE) kernels *sdot_fp_opt* and *sum_fp_opt*. The experimental results for these kernels show a considerable reduction of the performance difference. However, the scalarized version is still faster than the original code. The main reason for this seems to be improved address computations: the scalarized version operates on vector elements using structure-field offsets, resulting in a single instruction per access (of the form `load(base+offset)`). The scalar version uses array indices which are expanded to a sequence of two instructions (`addr=base+4*(i + offset); load(addr)`).

Potentially, the *offline* compiler could apply loop unrolling and MVE. The increased code size is not necessarily amortized with increased performance but a "better" JIT compiler could implement these transformations. However, we have shown that already implemented transformations achieve the similar performance as a result of collaboration with the offline compiler.

In summary, these experiments confirm that our Vapor SIMD approach can be risk-free in terms of performance. In the absence of SIMD support, speculative vectorization incurs only a minor penalty in the worst case. In favorable cases, it can even yield substantial performance improvements.

5.2 Performance of vectorized bytecode, conscious JIT

The efficiency objective refers to performance improvements delivered by a conscious-JIT when SIMD support is available. In this section, we experiment with:

- an Intel Core2 Duo at 3 GHz, Fedora 12 Linux 2.6.31 and Mono 2.8. This platform supports the SIMD extensions SSE, SSE2 and SSSE3;

benchmark	x86 + SSSE3			PowerPC + AltiVec			VF
	scalar	vectorized	speedup	scalar	vectorized	speedup	
vecadd	1891	928	2.0	2497	590	4.2	4
saxpy	2433	995	2.4	2528	623	4.1	4
dscal	1836	888	2.1	1646	403	4.1	4
max_s16	1787	413	4.3	889	143	6.2	8
max_u8	1845	308	6.0	798	106	7.5	16
max_s8	3060	2611	1.2	908	108	8.4	16
sum_u8	10632	2674	4.0	5054	651	7.8	16
sum_u16	10678	5235	2.0	5054	1278	4.0	8
sdot_fp	5060	1740	2.9	3921	650	6.0	4
sum_fp	5035	1235	4.1	3919	333	11.8	4
sdot_fp_opt	*1862*		*1.1*	*2447*		*3.8*	4
sum_fp_opt	*1363*		*1.1*	*1752*		*5.2*	4

Table 3: Performance of vectorized bytecode with conscious-JITs (time in milliseconds)

- a PowerMac 7450 at 800 MHz, running Linux 2.6.29 and a development version of Mono (based on 2.8) with our support for AltiVec.

Table 3 reports the performance of the vectorized bytecode when run with the conscious-JITs. For each platform, the first column contains the run time t_A of the scalar bytecode (scenario A) and the second column contains the run time t_B of the vectorized bytecode (scenario B). The third column computes the relative performance as t_A/t_B, i.e. the speedup achieved by the vectorized bytecode. The last column indicates the vectorizing factor (VF) which is the number of elements processed together by a SIMD instruction.

The observed speedups on PowerPC are mostly in line with the expected speedups from vectorization: they are comparable to the Vectorization Factor (VF), minus the usual overheads of vectorization (handling misaligned accesses, initializing vectors of constants, reduction prologue and epilogue, etc). However, code generation issues related to the actual JIT used in the experiment offset the results in some cases.

The vectorization impact on PowerPC is close to VF for the first group of kernels, the impact is about VF/2 for the second group of kernels, and for the third group we observe super-linear speedups. The vectorized reduction kernels of the second group suffer from the lack of global vector register allocation in Mono, which results in loading and storing of the reduction variable in each iteration of the vectorized loop. This explains the reduced speedups on the integer reduction kernels. Mono also does not perform global register allocation for (scalar) floating point registers, resulting in even more redundant loads and stores in the scalar code than in the vectorized code, which explains the super-linear speedups on the floating point reduction kernels of the third group. In order to overcome these code generation deficiencies of Mono, we manually optimized the scalar version of the floating point reduction kernels (see the *"*_opt"* kernels in the Table). For these, the vectorization speedup is indeed more reasonable (again, close to VF).

Similar trends are observed on the x86 platform, however scaled down by half. Again, this is due to Mono's poor register allocation capabilities: more variables are needed in the vectorized version (e.g. additional induction variable to advance addresses by the vector width), which on x86 with its few available registers results in register spilling. These code generation issues are not inherent to JITs in general, and work is undergoing on Mono's development branch to improve its register allocator. This would allow the benefit of vectorization to manifest itself also on targets that do not have many registers as does PowerPC.

5.3 Supporting Architecture Evolutions

One of the motivations of our work is to support evolutions of architectures. As an illustration, we now take a closer look at the behavior of vectorized bytecode on the SSE platform. SSE is a SIMD extension of Intel's x86 instruction set [10]. It has several versions: SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2. Table 4 describes the SSE features used in our benchmark set, as well as their SSE versions. This section studies how vectorized bytecode behaves on varying levels of SIMD support.

Running the benchmarks on different machines would not yield comparable run times. Instead, we run all the benchmarks on a single platform that support the highest level of SSE, and we selectively disable some features in the JIT compiler. Our target platform for this experiment is:

- an Intel Core i7 server, at 2.93 GHz, Fedora 11 Linux 2.6.29.5 and Mono 2.8. It supports the SIMD extensions SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2.

Note that features unique to SSSE3 are not exploited by our kernels and are therefore omitted in the rest of our discussion.

Table 5 summarizes the run times of all kernels for each supported SSE level. Relative performance is computed as the speedup over the scalar implementation, as in previous tables. In the interest of readability, we do not replicate numbers that are equal to the column on their left, but use an '=' symbol instead. This occurs in cases where a kernel has exploited the highest level of SSE; higher levels will not bring any additional speedup (*vecadd* for example does not take advantage of any level above 1). It also occurs when intermediate SSE levels are not relevant to the kernel. For example, *max_s8* benefits from SSE and SSE4.1, but SSE2 and SSE3 do not provide any feature of interest.

The main observation is that speedups increase monotonically with the SSE level. The JIT compiler automatically takes advantage of the available hardware support to provide acceleration, and falls back to scalarized code for unsupported features. This is especially visible in the case of *max_s8* and *max_u8*, which are identical except for the

feature	level	vecadd	saxpy	dscal	max_s16	max_u8	max_s8	sum_u8	sum_u16	sdot	sum_fp
load/store	1	√	√	√	√	√	√	√	√	√	√
add float	1	√	√							√	
mul float	1		√	√						√	
max short	2				√						
max uchar	2					√					
add short	2							√			
add char	2								√		
horizontal add fp	3									√	√
max char	4.1						√				
highest SSE level		1	1	1	2	2	4.1	2	2	3	3

Table 4: SSE level of SIMD features used in benchmarks

signedness of the max operator. The signed max on packed bytes is introduced only in SSE4.1, whereas unsigned max first appeared in SSE2. The kernels exhibit different behaviors with respect to SSE support. In any case, extended SIMD support implies improved performance.

The *horizontal add* instruction was added in SSE3. It is used by *sdot* and *sum_fp*, but only for the final reduction in the loop epilogue, hence the limited impact on performance. The code size, however, is reduced.

Note that the *scalarized* column exhibits trends similar to column *"x86, no SIMD"* of Table 2, but they differ slightly as the machines are different. The notable exception is *sum_fp*: scalar and vectorized code are identical to the code produced on the Core2 Duo, but the Core i7 is able to achieve a 6.5x speedup thanks to its micro-architecture.

5.4 Code Size and JIT Compilation Time

Vectorization typically performs code duplication and as a consequence an increase in code size is common [2, 17]. Several reasons for code duplication are:

- versioning: multiple versions of loops are emitted to handle situations that cannot be determined at compile time;
- peeling: guarantees alignment when memory accesses cannot be proven to be aligned;
- epilogues: which compute the final part of reduction kernels.

Vapor SIMD assumes vector accesses to misaligned memory addresses are permitted in bytecode. Consequently, neither loop versioning nor peeling are required. Further, its use of builtins minimizes the impact of epilogues. Despite avoiding these classical sources of code bloat, Vapor SIMD increases bytecode size in our loop kernel benchmarks by a factor of 1.6 to 2.5. Table 6 presents these bytecode sizes, where s_A is produced by our non-vectorizing offline compiler and s_B is vectorized according to the Vapor SIMD naming convention. Factors which increase bytecode size in Vapor SIMD are: (1) its use of (static) method calls (5 bytes) over stack operations (1 byte) for loading, storing and arithmetic, and (2) the addition of builtins returning target specific constants (vector width, etc.)

The algorithmic complexity of the Vapor SIMD optimization implemented in a conscious-JIT is at most linear with respect to the size of the intermediate representation, itself being essentially proportional to the size of the bytecode (assuming a bounded inlining heuristic). Paradoxically, the compilation time of an agnostic-JIT can be slightly higher; this is due to explicit loop unrolling in the implementation of builtins. Table 6 again presents JIT compilation time for our loop kernel benchmarks, where t_A represents an agnostic-JIT compiling non-vectorized bytecode, t_B shows a conscious-JIT compiling vectorized bytecode and t_C presents an agnostic-JIT compiling vectorized bytecode.

Note that these code size and compile time increases affect only vectorized portions of code, which usually constitute a small part of an application. Though increased by scalarization, compile time is still negligible. Indeed the speedups reported in Section 5 include JIT compilation time, yet there's no visible affect on performance.

6. RELATED WORK

Leśnicki et al. [12] designed a split-compilation approach using GCC and CLI to support vectorization by an online JIT compiler. The proposal annotates the CLI bytecode with information to help the run-time vectorizer. However, marking all relevant loops and variables with appropriate and usable information, while maintaining correct CLI code, proved very difficult. Our approach shifts more processing to the offline compiler, conveying information within the CLI code itself, while providing a mechanism for efficient scalarization where needed. We also provide experimental results demonstrating the feasibility of implementing our approach.

The original work in the GCC auto-vectorizer [16, 17] presented the design and implementation of target-independent vectorization algorithms. These algorithms are driven by the machine model of the compiler and generate target-dependent (assembly) code. On the contrary, we generate target-independent CLI bytecode, delaying the specialization to a target-dependent JIT compiler. Nevertheless, our approach does leverage the target-independence of GCC's auto-vectorization algorithms, as did Leśnicki et al [12]. Recent work [19] studies the interaction between static and dynamic compilers in the context of vectorization, and applies manual optimizations. Our approach is automatic.

Another approach developed by Lattner and Adve [3] uses LLVM [11] for representing SIMD operations in intermediate, target-independent bytecode format. In contrast to our proposal which uses standard CLI, their approach does extend the bytecode format itself, thereby breaking its compatibility. In addition, we investigate the behavior of running vector code on non-SIMD machines, and we evaluate compilation time in a more resource-constrained JIT environment. Finally, we propose an integrated (split-compilation) approach with an offline, target-independent automatic vectorizer.

benchmark	scalar	scalarized		SSE		SSE2		SSE3		SSE4.1	
		time	rel.	time	rel.	time	rel.	time	rel.	time	rel.
vecadd	2545	1525	1.7	721	3.5	=	=	=	=	=	=
saxpy	2458	2965	0.8	754	3.3	=	=	=	=	=	=
dscal	1531	1643	0.9	608	2.5	=	=	=	=	=	=
max_s16	1589	748	2.1	702	2.3	298	5.3	=	=	=	=
max_u8	1588	723	2.2	712	2.2	184	8.6	=	=	=	=
max_s8	2860	741	3.9	725	3.9	=	=	=	=	195	14.7
sum_u8	12486	6696	1.9	6578	1.9	1461	8.5	=	=	=	=
sum_u16	15210	6879	2.2	6791	2.2	2895	5.3	=	=	=	=
sdot	5282	1183	4.5	880	6.0	=	=	850	6.2	=	=
sum_fp	5357	819	6.5	820	6.5	=	=	812	6.6	=	=

Table 5: Performance with conscious-JIT depending on SSE level (time in milliseconds)

benchmark	VF	JIT compilation time (milliseconds)				(rel.)		(rel.)	Bytecode size (bytes)			(rel.)
		t_A	t_B	t_C	$t_B - t_A$	t_B/t_A	$t_C - t_A$	t_C/t_A	s_A	s_B	$s_B - s_A$	s_B/s_A
vecadd	4	0.054	0.077	0.164	0.023	1.4	0.11	3.0	50	96	46	1.9
saxpy	4	0.057	0.165	0.23	0.108	2.9	0.173	4.0	52	111	59	2.1
dscal	4	0.09	0.109	0.167	0.019	1.2	0.077	1.9	42	91	49	2.2
max_s16	8	0.077	0.203	0.262	0.126	2.6	0.185	3.4	63	103	40	1.6
max_u8	16	0.074	0.282	0.321	0.208	3.8	0.247	4.3	55	136	81	2.5
max_s8	16	0.071	0.269	0.306	0.198	3.8	0.235	4.3	57	138	81	2.4
sum_u8	16	0.073	0.23	0.397	0.157	3.2	0.324	5.4	53	95	42	1.8
sum_u16	8	0.071	0.204	0.259	0.133	2.9	0.188	3.6	53	95	42	1.8
sdot_fp	4	0.074	0.182	0.324	0.108	2.5	0.25	4.4	59	125	66	2.1
sum_fp	4	0.079	0.174	0.246	0.095	2.2	0.167	3.1	59	103	44	1.7

Table 6: JIT compilation time and bytecode size

Clark et al. [4] propose to convey SIMD opportunities within standard scalar assembly code, relying on a dynamic translation mechanism to recognize instruction patterns that can be vectorized (in hardware). A static vectorizing compiler is responsible for producing these patterns, by scalarizing (and outlining) its output. They are able to handle certain increases in SIMD widths as well. Their hardware-based solution, however, is limited in the size and complexity of the patterns it can recognize, and cannot be applied to arbitrary SIMD targets. Our approach also provides the ability to scalarize code that has been speculatively vectorized ahead-of-time, but does so only where needed, just-in-time rather than ahead-of-time.

Pajuelo et al. [18] propose a purely hardware solution that detects memory accesses with constant strides and generates vector instructions speculatively. We focus on portability and demonstrate our approach on several targets.

An auto-vectorizing JIT compiler for Java is considered by El-Shobaky et al. [9]. Their solution is based on tree pattern matching and is limited in terms of supported idioms and targets: packing short data types into 32-bit containers on x86, rather than generating instructions of different SIMD extensions, which is our focus.

7. CONCLUSION AND FUTURE WORK

In this paper we demonstrate a scheme to exploit SIMD capabilities available in many existing platforms, while incurring a negligible penalty on other non-SIMD platforms. Our Vapor SIMD approach makes use of split-compilation, leveraging powerful automatic vectorization optimizations available in offline compilers, in conjunction with a standard intermediate language that is supported by several JIT compilers for different platforms. The crux of our approach is to run advanced, time-consuming analyses and transformations speculatively, ahead-of-time, conveying their results in a standard format, in a way that can be seamlessly reverted where needed. Experimental results show that this approach can deliver high-level performance improvements exploiting different levels of SIMD capabilities efficiently, while being robust and applicable to arbitrary CLI environments.

We plan to extend our approach to support just-in-time adjustment of the vector size to that of the target platform, so that the same vectorized bytecode could be run on 128-bit vector platforms (like SSE and AltiVec) as well as 64-bit and 256-bit vector platforms (like MMX and AVX, respectively). Misalignment is another area of improvement, developing a scheme to seamlessly adjust the vectorized code to use the best realignment mechanisms available to the target platform. This direction would be particularly suitable to a split-compilation scheme: it may involve offline loop transformations and tight collaboration between the offline and online compilers.

Acknowledgements

This work was partly supported by the European Commission through the HiPEAC Network of Excellence, and by French Ministry of Industry and STMicroelectronics through the Mediacom contract. The design of Vapor SIMD benefitted from numerous discussions with Sami Yehia.

8. REFERENCES

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

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

[3] Robert L. Bocchino, Jr. and Vikram S. Adve. Vector LLVA: a Virtual Vector Instruction Set for Media Processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 46–56, New York, NY, USA, June 2006.

[4] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztián Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 216–227, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Albert Cohen and Erven Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Design Automation Conference (DAC '10)*, pages 102–107. IEEE Computer Society, June 2010. Special session on Embedded Virtualization.

[6] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 1998.

[7] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramírez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050 of *LNCS*, pages 5–29. Springer-Verlag, 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 *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, pages 63–69. ACM, 2009.

[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 *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, CA, USA, March 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 *International Workshop on GCC for Research in Embedded and Parallel Systems*, 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 *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '09)*, pages 255–260. Springer-Verlag, 2009.

[15] Gregor Noriskin. Writing High-Performance Managed Applications: A Primer. In *MSDN Library*. Microsoft, 2003.

[16] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Dorit Nuzman and Ayal Zaks. Autovectorization in GCC – two years later. In *GCC Developers' Summit*, pages 145–158, June 2006.

[18] Alex Pajuelo, Antonio González, and Mateo Valero. Speculative dynamic vectorization. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*, pages 271–280, 2002.

[19] Erven Rohou. Portable and efficient auto-vectorized bytecode: a look at the interaction between static and JIT compilers. In *Second International Workshop on GCC Research Opportunities (GROW '10)*, Pisa, Italy, January 2010.

[20] Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A stack-based internal representation for GCC. In *First International Workshop on GCC Research Opportunities (GROW '09)*, pages 37–48, Paphos, Cyprus, January 2009.