Document Number: N4395
Date: 2015-04-10
Project: Programming Language C++, Library Working Group
Reply-to: Matthias Kretz <kretz@compeng.uni-frankfurt.de>

SIMD Types: ABI Considerations

ABSTRACT

This document discusses the ABI implications from the SIMD types described in [N4184] and [N4185]. I investigate strategies to automatically adapt between different translation units compiled for different microarchitectures of the same architecture. None of the strategies lead to a solution without surprises. The solutions section therefore looks at how the default vector type may need to be declared to make ABI incompatibilities a conscious choice of the user.

CONTENTS

1 About This Document

This document is derived from a larger document about the Vc library. For the sake of simplicity I refrained from changing the naming conventions of types/functions in this paper:

- I want to focus on functionality first. We can "correct" the names later.
- It is easier to find the reference to an existing implementation.

Disclaimer: I did not get everything "right" in the Vc implementation yet. Some details of the interface definitions I present here do not fully reflect the current state of the Vc SIMD types.

1.1 Shorthands in the document

- $\mathcal{W}_{\mathtt{T}}$: number of scalar values (width) in a SIMD vector of type `T` (sometimes also called the number of SIMD lanes)

1.2 Scope

This document talks about a quality-of-implementation issue. None of the extra compiler features discussed here need to / should be part of the SIMD types specification. This is an investigation of the implications of a specification along the lines of [N4184] and may influence some decisions (mainly about the *default* SIMD types) going forward.

2 ABI Introduction

An ABI describes machine-, operating system-, and compiler-specific choices that are not covered by a programming language standard. Some of the issues are function calling conventions, memory/stack organization, and symbol name mangling. For example, Matz et al. [4] and *Itanium C++ ABI* [3] standardize and document the ABI for Linux. For Windows the ABI is implicitly defined by its development tools. For all targets, the goal is to have an ABI that allows interoperability. Developers expect that their choice of compiler (and compiler version) does not have an influence on the TUs (*translation units*) that can be linked together correctly. Compiler vendors and operating system vendors have a great interest in providing this guarantee.

3 ABI RELEVANT DECISIONS IN VC

The interface choices for Vc have a direct influence on the ABI of the Vc library.

3.1 FUNCTION PARAMETERS

If a `Vector<T>` is used as a function parameter, there are two principal choices for implementing the parameter passing in the function call convention:

1. The vector can be passed as a single SIMD register.
2. The vector is pushed onto the stack and thus passed via memory.

Choice 1 is the most efficient choice for the majority of situations. This requires a trivial copy constructor and destructor in `Vector<T>` (which recursively requires the copy constructor and destructor to be trivial for all non-static data members) with the Linux ABI [4].

If a `union` of intrinsic type and array of scalar type is used to implement the data member of `Vector<T>`, the Linux x86_64 ABI requires different parameter passing, which is derived from all members of the `union`. With a trivial copy constructor and destructor a `union` with `sizeof()` = 16 must be passed as two SSE registers or two general purpose registers, while with `sizeof()` = 32 the parameter must be passed via memory [4, §3.2.3]. (According to the rules in Matz et al. [4, §3.2.3] there is a workaround to achieve parameter passing via SIMD registers: The array inside the `union` must be declared with zero entries.)

This shows that, in addition to the interface definition, the concrete implementation strategy also has an influence on the resulting ABI of the vector types. This needs to be considered carefully when implementing the library. Consequently, an implementation should avoid a `union` based implementation (unless the ABI for the target system works differently) and rather use a different compiler extension for explicit aliasing, such as GCC's `may_alias` attribute.

The discussion above equally applies to `Mask<T>` and all derived types, of course.

3.2 LINKING DIFFERENT TRANSLATION UNITS

A user can compile two TUs with different compiler flags for the target microarchitecture (for example, so that one is compiled for SSE and the other for AVX). This most likely happens with one TU in a library and the other in an application. Then Vc vector or mask types in the interfaces between the TUs are incompatible types. The most complicated architecture with regard to SIMD differences probably is x86: Very old systems have no usable SSE support, old systems support SSE, current systems

AVX or AVX2, and future systems AVX-512. $\mathcal{W}_{\texttt{T}}$ is different: `xmm` vs. `ymm` vs. `zmm` registers. Consequently, it appears like a good idea for packagers to not treat x86 as a single architecture anymore, but several ones. There is great interest in not having to take that path, though. The following sections will explore the issue in detail.

4 Problem

The `Vector<T>` type is defined as a target-dependent type, which, similarly to `int`[1], uses the most efficient register size on the target system. For SIMD registers this implies that the number of values $\mathcal{W}_{\texttt{T}}$ stored in a `Vector<T>` object can be different between different microarchitectures of the same architecture. The SIMD Types interface [N4184] at least ensures that the types `Vector<T>` are different if the register sizes differ. Therefore, the use of `Vector<T>` is safeguarded against incompatible linking, which would result in spurious runtime errors.

For the following discussion, consider an Intel Haswell system, which implements the x86_64 architecture and AVX2 SIMD registers & operations as a part of its microarchitecture (for simplicity, ignore the MMX instruction set). Then,

- with AVX2 $\mathcal{W}_{\texttt{float}} = 8$ and $\mathcal{W}_{\texttt{int}} = 8$,
- with AVX $\mathcal{W}_{\texttt{float}} = 8$ and $\mathcal{W}_{\texttt{int}} = 4$,
- with SSE $\mathcal{W}_{\texttt{float}} = 4$ and $\mathcal{W}_{\texttt{int}} = 4$, and
- without using SIMD functionality $\mathcal{W}_{\texttt{float}} = 1$ and $\mathcal{W}_{\texttt{int}} = 1$ (the `Scalar` implementation mentioned in [N4184]).

The `Vector<T>` incompatibility between different SIMD instruction sets implies that a TU built for Intel SandyBridge differs in ABI to a TU built for Haswell. This breaks with the guarantee compiler vendors would like to retain: the ABI for a given architecture should stay stable. With the current `Vector<T>` proposal, implemented on top of SIMD intrinsics, the ABI would only be stable within microarchitectures.

One could argue that it is technically correct that some microarchitectures (those with differing SIMD widths) of the same architecture are partially incompatible, and thus the ABI could/should reflect this. On the other hand, it is very desirable that such incompatibilities are either hidden from (or consciously enabled by) the user. Thus, if it is at all possible to have the compiler automatically adapt between the microarchitectural differences, then implementors should invest in getting the `Vector<T>` ABI right from the outset.

1 “Plain ints have the natural size suggested by the architecture of the execution environment” [2, §3.9.1 p2]

4.1 fixed w_t in interfaces is not the solution

A common idea for solve the above issue, is to request that the SIMD type uses a user-defined width (cf. Fog [1] and Wang et al. [5]). Then the type would use the same $\mathcal{W}_T$ on any target and the types would be equal in different TUs.

There are two issues with this:

1. There is no guarantee that the specific $\mathcal{W}_T$ can be implemented efficiently on all target systems. Consider, for example, the common choice of $\mathcal{W}_{\texttt{float}} = 4$ compiled for an Intel Xeon Phi. The type would have to be implemented with a 512-bit SIMD register where 75% of the values are masked off. On a target without SIMD support, four scalar registers would have to be used, which increases register pressure.[2]

2. Even though the types are equal, the specific parameter passing implementation might be different. Consider a `vec<float, 8>` type translated for either AVX or SSE. Then the function

   ```
   void f(vec<float, 8>)
   ```

   would use `ymm0` with AVX and `xmm0` and `xmm1` with SSE to pass the function parameter from the caller to the function. Thus, if this were the preferred solution for implementors, vector types would have to be passed via the stack for function parameter passing (cf. Section 3.1). In addition, the in-memory representation and alignment requirements for the different microarchitectures must be defined in such a way that they work correctly on all systems.

From my experience, and in order to enable full scaling to different SIMD targets, I prefer a solution where a fixed $\mathcal{W}_T$ is only chosen because it is dictated by the algorithm, not because of technical complications with ABI compatibility.

4.2 derived types

A class that is derived from `Vector<T>` or a class that has a non-static `Vector<T>` member will not have a different type in different TUs which are compiled for different SIMD widths. Thus, the linkage safety built into `Vector<T>` does not work for any derived types. Furthermore, this suggests that a solution that transparently adapts the ABI differences must be rather invasive.

The compiler would have to compile Scalar, SSE, AVX, and AVX2 (to stay with the x86_64 example) variants of all derived types and functions that use these types.

2 With luck this might just be the right loop-unrolling to achieve good performance, but it is the wrong mechanism to achieve this effect.

The symbols would need additional information about the SIMD target as part of the name mangling.

Automatic adaption (such as a call from an AVX TU to an SSE TU) between derived types will be a problem, though. Consider that TU1 creates an object of a derived type `D`. A call to a member function, which is not declared `inline` and instead was compiled for a different SIMD width in TU2 now would require a transparent conversion of the object from one SIMD width to a different SIMD width. There cannot be a generic strategy to perform such a conversion without breaking the semantics guaranteed to the implementation of `D`.

4.3 Serial semantics

Consider an ABI adaption strategy that splits a function call from TU1 with a `Vector<T>` argument with $\mathcal{W}_{\mathtt{T}}^{(1)}$ to multiple function calls to the function compiled with $\mathcal{W}_{\mathtt{T}}^{(2)} = \frac{\mathcal{W}_{\mathtt{T}}^{(1)}}{N}$ in TU2. This approach exposes non-serial semantics. This manifests, for instance, if two functions are intended to be called in serial succession, communicating via a global (or thread-local) variable.[3] If the adaption from an AVX2 TU to an SSE TU is done via calling the SSE function twice with the low and high parts of the vector argument, then the first function will be called twice, before the second function is called twice.

Consider the example in Listing 1. The developer expected serial semantics in function `h`. Instead, `f` is called twice, before `g` is called twice. Therefore, the conclusion is that adapting between different SIMD widths cannot be done via splitting a function call into multiple function calls.

4.4 Largest common SIMD width

Consider a compiler implementation that identifies types that depend on $\mathcal{W}_{\mathtt{T}}$ and automatically compiles these symbols for all possible $\mathcal{W}_{\mathtt{T}}$ the target supports (extending the mangling rules accordingly). Then, when the TUs are linked to a single executable, the linker can detect whether for some symbols some $\mathcal{W}_{\mathtt{T}}$ translations are missing. In this case it can drop these $\mathcal{W}_{\mathtt{T}}$ symbols. The same could be done by the loader when the program is dynamically linked, right before executing the program. The largest remaining $\mathcal{W}_{\mathtt{T}}$ symbols can then be used to execute the program.

This solution should work as long as no dynamically loaded libraries are used (e.g. Plug-ins). Because, if an incompatible library (i.e. one that does not have the symbols for the currently executing $\mathcal{W}_{\mathtt{T}}$) is loaded, the program cannot switch back down to

3 This is probably a bad design, but that does not invalidate the problem.

```
// a.cc (SSE2 : float_v::size() == 4):
static float_v globalData;
void f(float_v x) { globalData = x; }
float_v g() { return globalData; }

// b.cc (AVX2 : float_v::size() == 8):
float_v h(float_v x) {
  f(x);  // calls f(x[0...3]) and f(x[4...7])
  // now globalData is either x[0...3] or x[4...7], depending on the order of
  // calls to f above
  return g();  // calls concatenate(g(), g())
}

int main() {
  cout << h(float_v::IndexesFromZero());  // {0 1 2 3 4 5 6 7}
  return 0;
}

// prints:
// 0 1 2 3 0 1 2 3
// or:
// 4 5 6 7 4 5 6 7
```

Listing 1: Impure functions break the adaption strategy of using multiple calls to TUs with shorter SIMD width.

a smaller $\mathcal{W}_{\mathrm{T}}$. Thus, at least the ABI compatibility with dynamically loaded symbols cannot be guaranteed by this approach.

4.5 SIMD-enabled functions

The SIMD-enabled functions described in [N3831] provide the semantic restriction which works around the issue described in Section 4.3. The code in Listing 1 would still produce the same result, but because of the semantic restriction for the functions `f` and `g` the undefined behavior would be expected.

On the other hand, a member function of a class with members of vector type that accesses such members will still not be automatically adaptable between different TUs. Consider Listing 2. The call to `D::f` on line 21 will pass a `this` pointer to an object storing two `float_v` objects with $\mathcal{W}_{\texttt{float}} = 8$ placed next to each other in memory. The function `D::f`, on the other hand, (line 10) expects two `float_v` objects with $\mathcal{W}_{\texttt{float}} = 4$ consecutively in memory (Figure 1). In order to adapt such differences between TUs automatically, the adaptor code would have to create two temporary objects of type `D` (with the ABI in `a.cc`), copy the data, call the function `D::f` twice, copy the resulting temporary objects back into the original object and return. But such a strategy breaks with the call to `next->f()`. Non-vector members

```
typedef Vector<float, Target::Widest> float_v;
struct D {
  float_v x, y;
  unique_ptr<D> next;
  D() : x(float_v::IndexesFromZero()), y(0) {}
  void f() [[simd]];
};

// a.cc (widest float_v::size() == 4):
void D::f() [[simd]] {
  y = (y + 1) * x;
  if (next) {
    next->f();
  }
}

// b.cc (widest float_v::size() == 8):
int main() {
  D d;
  d.next.reset(new D);
  d.f();
}
```

Listing 2: Member functions as SIMD-enabled functions?

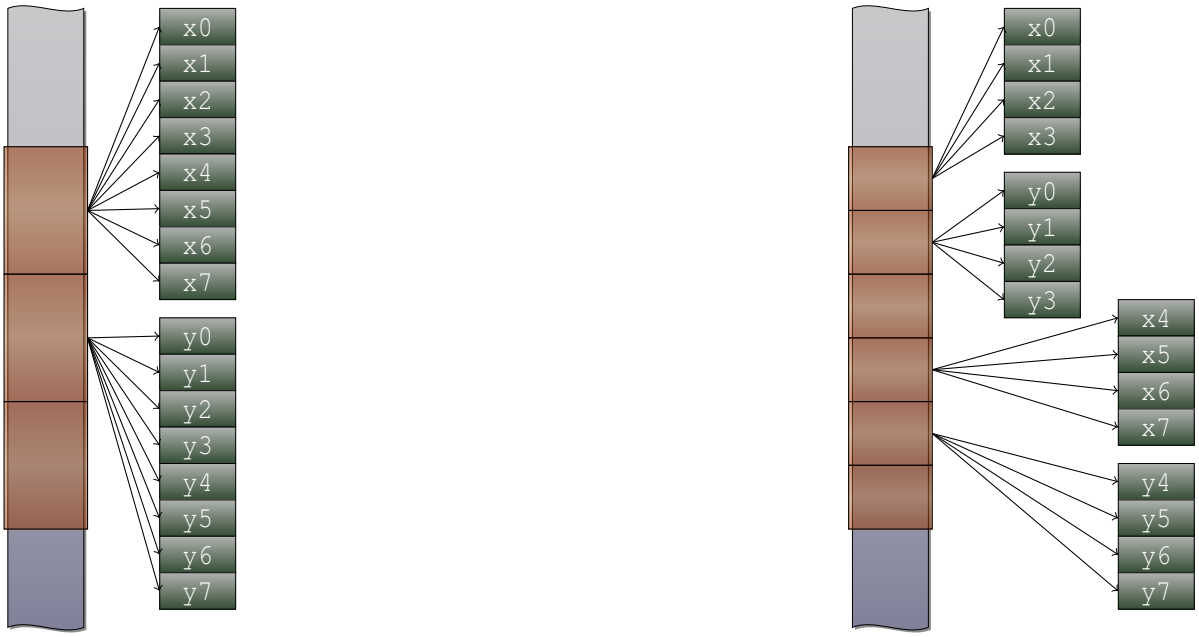

Figure 1: Memory layout differences depending on ABI. The memory layout of `d` in the caller is shown on the left. However, the function `D::f` expects the memory layout as shown on the right.

cannot be transformed generically and the `next` pointer would therefore point to an untransformed object.

Effectively, the strength of vector types (namely target-optimized data structures) inhibits the creation of automatic ABI adaption between TUs with different $\mathcal{W}_T$.

5 SOLUTION SPACE

In order to enable compilers to keep ABI compatibility for the complete x86_64 architecture, the solution needs to …

1. …make ABI breakage of derived types impossible (or obvious to the user). (cf. Section 4.2)
2. …keep one function call as one function call. (cf. Section 4.3)
3. …not require a specific $\mathcal{W}_T$ from dynamically loadable libraries. (cf. Section 4.4)

5.1 DROP THE DEFAULT VECTOR TYPE

After extensive consideration and some prototyping I have not found an idea to transparently solve the issue while keeping a default vector type with varying $\mathcal{W}_T$ for different microarchitectures. At this point the only solution I can conceive is a vector type that does not have a default $\mathcal{W}_T$, or at least not one that follows the microarchitecture.

Feedback on [N4184] suggested to use a policy type to select the Vector implementation instead of namespaces. Therefore, the following discussion refers to the following class template (with possible tag types and a portable default for x86_64):

```
namespace Vc {
  namespace Target {
    struct Scalar {};  // always present
    struct Sse2 {};    // x86(_64) specific
    struct Avx {};     // x86(_64) specific
    struct Avx2 {};    // x86(_64) specific
    typedef target_dependent Widest;  // always present
  }
  template <typename T, typename Impl = Target::Sse2> class Vector;
}
```

The user who wants to have a different default behavior can do something along the lines of:

```
template <typename T> using Vector = Vc::Vector<T, Vc::Target::Avx>;
```

Or, to get the behavior described in [N4184]:

```
template <typename T> using Vector = Vc::Vector<T, Vc::Target::Widest>;
```

With this declaration of the default, the ABI can be stable for the complete x86_-64 architecture until the user selects a specific microarchitectural subset (such as `Target::Avx`) explicitly. Thus, ABI incompatibilities will only occur after a (at least to a certain degree) conscious choice of the user. And at the same time it requires a minimal amount of code to get the behavior described in [N4184], which can be very useful for controlled environments, such as homogeneous cluster systems and some in-house software.

5.2 Improving multi-w_t support

I believe it should be possible for compilers to improve deployment to targets with different $\mathcal{W}_T$ for different microarchitectures. This would follow along the lines described in Section 4.4. While this does not fully solve the ABI compatibility issue of $\mathcal{W}_T$ depending on the microarchitecture, it still enables a considerably simpler vehicle for binary distribution. Dynamically loaded libraries (Plug-ins) would require extra care by developers, but otherwise applications could transparently use the best available SIMD instruction set without additional work by application developers.

6 Conclusion & Intended Feedback

This paper has shown that there is still unclear direction how to define a portable vector type which enables compiler vendors to make a target architecture ABI-compatible throughout all SIMD variants. The requirements of C++ users certainly differ on this point and there will probably be no clear one-fits-all answer. This issue needs more experience and work in real-world use to come to a final conclusion.

Regarding the default vector width I am looking for feedback how to proceed. It appears like Section 5.1 is the only solution, at the expense of dropping the best-width vector type in favor of a more conservative choice. Did I miss an important idea or will this be the way forward?

Regarding the multi-$\mathcal{W}_T$ support, I am convinced that the compiler can alleviate some of the current pain we have with differences between microarchitectures. I know that multi-targeting is a goal for others as well. Therefore, I would be very interested to learn of anything that could be done from the language side or in the definition of the SIMD types to enable a better solution.

A ACKNOWLEDGEMENTS

This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

Thanks to JF Bastien for his review and questions.

B BIBLIOGRAPHY

[1] Agner Fog. *C++ vector class library*. 2014. URL: http://www.agner.org/optimize/vectorclass.pdf.

[N3831] Robert Geva and Clark Nelson. *N3831: Language Extensions for Vector level parallelism*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3831.pdf.

[2] *ISO/IEC 14882:2014. Information technology — Programming languages — C++*. Standard. ISO/IEC JTC 1/SC 22, 2014. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029.

[3] *Itanium C++ ABI*. CodeSourcery et al. URL: http://www.codesourcery.com/cxx-abi/ (visited on 12/12/2014).

[N4184] Matthias Kretz. *N4184: SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf.

[N4185] Matthias Kretz. *N4185: SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4185.pdf.

[4] Michael Matz et al. *System V Application Binary Interface*. Nov. 2014. URL: http://www.x86-64.org/documentation_folder/abi-0.99.pdf.

[5] Haichuan Wang et al. "Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library." In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. New York, NY, USA: ACM, 2014, pp. 9–16. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568059.