C++ Reflection for Data Layout Abstraction

Jolly Chen
jolly.chen@cern.ch
CERN & University of Twente

Kokkos Tea-Time — February 18, 2026

NexTGen
Next Generation Triggers

UNIVERSITY
OF TWENTE.

Memory Layout Problem

1 Introduction

- Good performance requires good memory access patterns (MAPs)
  - Depends on the data layout
  - Depends on the architecture and computation
- Common dilemma: array-of-structures (AoS) or structure-of-arrays (SoA)?

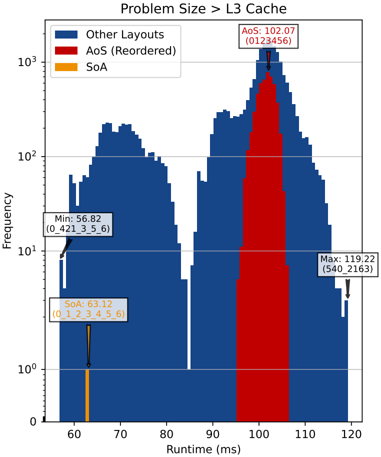


Figure: Runtime distribution for different data layouts of a simplified particle object with the "invariant mass" computation on AMD EPYC 9534.

AoS to SoA Manually

1 Introduction

AoS

id	x	y	id	x	y	id	x	y

```cpp
struct Particle {
    int id;
    float x;
    double y;
};



int main() {
    std::vector<Particle> particles(10);
    particles[2].id = 0;
    particles[2].x = 1.0;
    ...
}
```

SoA

id	id	id	x	x	x	y	y	y

```cpp
struct Particle {
    std::vector<int> id;
    std::vector<float> x;
    std::vector<double> y;

    Point(size_t n) : x(n), y(n), z(n) {}
};

int main() {
    Particles particles(10);
    particles.id[2] = 0;
    particle.x[2] = 0;
    ...
}
```

- Decouple memory organisation from logical data accesses
- We want to easily switch between AoS and SoA
- We want to keep `particles[0].id`

Table of Contents

1 Introduction

- Introduction
- C++26 Reflection (P2996)
- SoA Wrapper
- SoA Wrapper with Token Injection
- Structure Splitter
- Performance Comparison
- My experience with Reflection
- Conclusion

C++26 Reflection (P2996)

- Upcoming new operators in C++26:

`^^x`	*"Lift*" operand `x` to a **reflection** value of type `std::meta::info`
`[: refl :]`	*"Splice"* a reflection to **produce grammatical elements**

- We can query the reflection of data members of a struct `S` using:

```
std::vector<std::meta::info> members =
    nonstatic_data_members_of(^^S, std::meta::access_context::current());
```

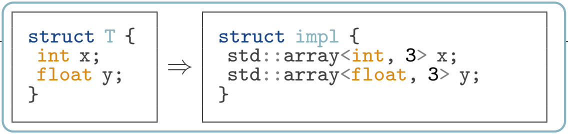
In the following slides, we shorten this to `nsdms(^^S)`

- We can also define structures:

```
struct S;
constexpr auto s_int_refl = define_aggregate(^^S, {
  data_member_spec(^^int,    {.name="i", .alignment=64}),
  data_member_spec(^^double, {.name="k"})
});
```

This defines `S` as `struct S { int i; double k; };`

Struct to Struct of Arrays Example in P2996



```
struct T {
 int x;
 float y;
}
```

⇒

```
struct impl {
 std::array<int, 3> x;
 std::array<float, 3> y;
}
```

```
template <typename T, size_t N>
struct struct_of_arrays_impl {
  struct impl;

  consteval {
    std::vector<std::meta::info> old_members = nsdms(^^T);
    std::vector<std::meta::info> new_members = {};
    for (std::meta::info member : old_members) {
        auto array_type = substitute(^^std::array, {
            type_of(member),
            std::meta::reflect_constant(N),
        });
        auto mem_descr = data_member_spec(array_type,
                            {.name = identifier_of(member)});

        new_members.push_back(mem_descr);
    }
    define_aggregate(^^impl, new_members);
  }
};
template <typename T, size_t N>
using struct_of_arrays = struct_of_arrays_impl<T, N>::impl;
```

For each data member in `T`

Construct array type with type of `member`

Create data member spec with the array type and same name as `member`

Define `impl` with list of arrayified members

Struct to Struct of Arrays Example in P2996

2 C++26 Reflection (P2996)

```
template <typename T, size_t N>
struct struct_of_arrays_impl {
  struct impl

  consteval {
    std::vect
    std::vect
    for (std:
        auto a
            ty
            st
        });
        auto n


        new_me
    }
    define_aggregate(  impl, new_members);
  }
};
template <typename T, size_t N>
using struct_of_arrays = struct_of_arrays_impl<T, N>::impl;
```

We can automate converting AoS members to SoA now, but how do we keep AoS-style access semantics?

↓ ↓ ↓

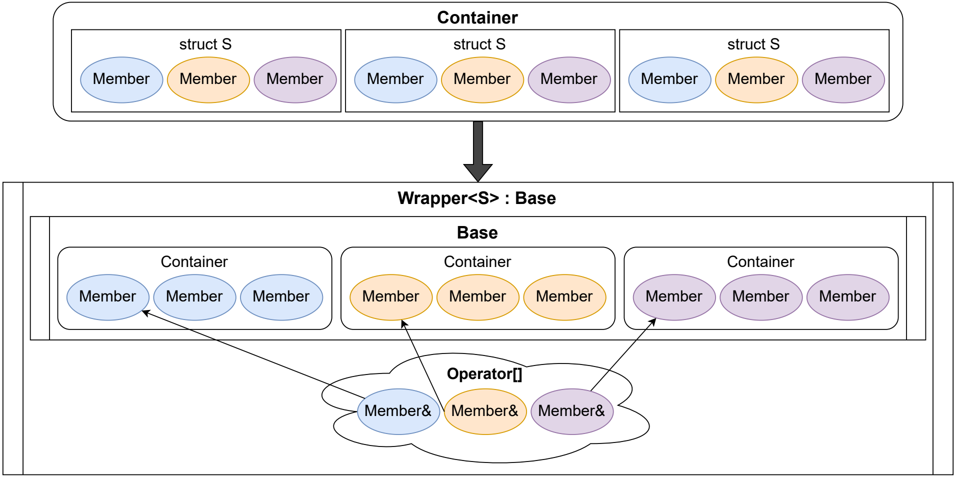
Can we define `operator[]` with `define_aggregate`?

define_aggregate **Limitations**

We can only define *data* members and only for incomplete types:

```cpp
struct Incomplete;

struct Defined {
    Defined operator[](int i) { ... } };

template <typename Base>
struct Templated {
    Templated operator[](int i) { ... } };

consteval {
    define_aggregate(^^Incomplete, {}); // OK
    define_aggregate(^^Defined, {});    // Redefinition error
    // OK, but no access to base template members
    define_aggregate(^^Templated<S>, {});
}
```

SoA Wrapper


Container
struct S
Member
Member
Member
struct S
Member
Member
Member
struct S
Member
Member
Member
Wrapper<S> : Base
Base
Container
Member
Member
Member
Container
Member
Member
Member
Container
Member
Member
Member
Operator[]
Member&
Member&
Member&

User Interface

3 SoA Wrapper

```
// Original Structure
struct Point {
  float &x;
  int &y;
  double &z;

  double sum() const { return x + y + z; };
};

using PointSoA = Wrapper<Point, std::vector>;

int main() {
  PointSoA q = {{0.0f, 0.0f, 0.0f}, {1, 1, 1}, {2.0, 2.0, 2.0}};
  auto s = q[0].sum();
  q[0].z = 42;
  ...
}
```

References to point to underlying SoA members and to keep member methods

Stores each data member of Point as std::vector member

Returns `Point`

Wrapper Generator

3 SoA Wrapper

```
template <class Func>
consteval auto transform_members(std::meta::info type, Func f) {
  return nsdms(type) | std::views::transform([=](std::meta::info member) {
    return data_member_spec(f(type_of(member)), {.name = identifier_of(member)});
  });
}

template <typename S, template <class> class F>
struct WrapperGenerator {
  struct Base;

  consteval {
    define_aggregate(
        ^^Base, transform_members(^^S,
            [](std::meta::info type) {
                return substitute(^^F, {remove_cvref(type)}); }));
  }

  ...
};
```

Get a list of member specs by applying function `Func` on all members of `type`

Define `Base` with each member of `S` transformed to a container of type `F` (e.g., `std::vector`)

Wrapper Generator

3 SoA Wrapper

For the example:

```cpp
// Original Structure
struct Point {
  float &x;
  int &y;
  double &z;

  double sum() const { return x + y + z; };
};
```

We get:

```cpp
template<>
struct WrapperGenerator<Point, std::vector> {
  struct Base {
    std::vector<float> x;
    std::vector<int> y;
    std::vector<double> z;
  };

  ...
};
```

Random Access Operator

3 SoA Wrapper

```
template <typename S, template <class> class F>
struct WrapperGenerator {
  ...

  class Wrapper : public Base {
    S operator[](int i) {
      auto construct_output = [&]<size_t... Is>(std::index_sequence<Is...>) -> S {
        return { static_cast<Wrapper &>(*this).[: nsdms(^^Base)[Is] :][i]... };
      };
      constexpr auto indices = std::make_index_sequence<nsdms(^^Base).size()>{};
      return construct_output(indices);
    }
}};
template <typename S, template <class> class F>
using Wrapper = WrapperGenerator<S, F>::Wrapper;
```

`std::vector<info>` is not a pack so we need an index sequence

For `Point`, this returns `Point{ x[i], y[i], z[i] }`

Hopefully in the future with a range splice:

```
S operator[](int i) {
  constexpr auto members = nonstatic_data_members_of(^^Base);
  return { static_cast<Wrapper &>(*this).[: ...members :][i]... };
}
```

Table of Contents

4 SoA Wrapper with Token Injection

- ▶ Introduction
- ▶ C++26 Reflection (P2996)
- ▶ SoA Wrapper
- ▶ SoA Wrapper with Token Injection
- ▶ Structure Splitter
- ▶ Performance Comparison
- ▶ My experience with Reflection
- ▶ Conclusion

Token Injection (P3294)

4 SoA Wrapper with Token Injection

P3294 proposes code injection via token sequences of type `std::meta::info`, created with:

```
^^{ balanced-brace-tokens }
```

The injection can be queued up at the end of the current constant evaluation using:

```
consteval {
    std::meta::queue_injection(^^{ ... } )
}
```

Three *interpolators* are provided to access outside context within a token sequence:

`\(e)`	Replaced by the evaluation of the expression `e`
`\id(string, string-or-int`$_{opt}$`...)`	Replaced by the concatenation of string-like or integral values as an identifier
`\tokens(e)`	Replaced by contents of the token sequence `e`

SoA Wrapper with Token Injection

Support for more complex data types

```
template <typename T>
struct LorentzVector { T &fX, &fY, &fZ, &fT; };

struct Particle {
  int &id;
  LorentzVector<double> momentum;
  std::span<float> vector;

  void SetId(int new_id) { id = new_id; }
};

using SoA = rmpp::Wrapper<Particle>;

int main() {
  constexpr size_t n = 3;
  alignas(64) std::vector<std::byte> buf(SoA::ComputeSize(n));
  SoA maos(buf.data(), buf.size(), n);

  maos.push_back({0, {0,0,0,0}, {33,33,33}, {1, 2}});
  maos[0].SetId(9);
  maos[1].momentum.fX = 8888;
  ...
}
```

`std::span` to refer to `std::vector`

AoS access to nested struct!

Wrapper Definition

4 SoA Wrapper with Token Injection

```cpp
template <typename S>
struct Wrapper {
  std::span<std::byte> storage;
  size_t m_size;     // Current number of data elements
  size_t m_capacity; // Maximum number of data elements

  consteval { gen_soa_members(^^S, true); }
}
```

Wrapper Definition

4 SoA Wrapper with Token Injection

```cpp
template <typename S>
struct Wrapper {
  std::span<std::byte> storage;
  size_t m_size;     // Current number of data elements
  size_t m_capacity; // Maximum number of data elements

  // Desired output of gen_soa_members
  std::span<int> id;

  struct LorentzVectorSoA {
    std::span<double> fX;
    std::span<double> fY;
    std::span<double> fZ;
    std::span<double> fT;
  };
  LorentzVectorSoA momentum;

  std::span<float> vector;
  std::vector<size_t> vector_offsets;
  // end output of gen_soa_members
}
```

For the example:

```cpp
struct Particle {
  int &id;
  LorentzVector<double> momentum;
  std::span<float> vector;
};
```

Generating SoA Members (1/2)

4 SoA Wrapper with Token Injection

```
consteval std::meta::info gen_soa_members(std::meta::info S, bool inject) {
  std::vector<std::meta::info> decl_tokens;      // List of token sequences ^^{ ...}
  std::vector<std::meta::info> visited_structs; // List of reflected types

  for (auto member : nsdms(S)) {
    auto type = remove_cvref(type_of(member));
    auto name = identifier_of(member);

    if (type is container) {
      auto value_type = get_scalar_type(type);
      decl_tokens.push_back(^^{ std::span<typename[:\(value_type):]> \id(name); });
      decl_tokens.push_back(^^{ std::vector<size_t> \id(name, "_offsets"sv); });
    } else if (type is nested struct) {
      if (!contains_refl(visited_structs, type)) {
        visited_structs.push_back(type);

        auto subdecl_tokens = gen_soa_members(type, false);
        decl_tokens.push_back(^^{ struct \id(identifier_of(type), "SoA"sv){
                                  \tokens(subdecl_tokens)}; });
      }

      decl_tokens.push_back(^^{ \id(identifier_of(type), "SoA"sv) \id(name); });
    ...
```

For the example:

```
std::span<float> vector;
std::vector<size_t> vector_offsets;
```

Recursively SoAify nested structures

Generating SoA Members (2/2)

4 SoA Wrapper with Token Injection

```
    ... // Continuation of previous code
    } else { Scalar Type
      decl_tokens.push_back(^^{ std::span<typename[:\(type):]> \id(name); });
  }}

  std::meta::info decl_concat = ^^{};
  for (auto tks : decl_tokens) {
    decl_concat = ^^{ \tokens(decl_concat) \tokens(tks) };
  }

  if (inject) { queue_injection(decl_concat); }
  return decl_concat;
}
```

Concatenate token sequences with interpolator

Random Access Operator (1/3)

```
S operator[](const size_t idx) {
  consteval {
    std::meta::list_builder member_data_tokens{};
    for (auto member : nsdms(^^S)) {
      member_data_tokens +=
        generate_view_tokens(member, ^^idx, ^^{ \id(identifier_of(member))});
    }
    queue_injection(^^{ return S{\tokens(member_data_tokens)}; });
}}
```

Random Access Operator (2/3)

4 SoA Wrapper with Token Injection

For the example:

```
struct Particle {
  int &id;
  LorentzVector<double> momentum;
  std::span<float> vector;
};
```

```
S operator[](const size_t idx) {
  // Desired result after token sequence injection
  return S{id[idx],
           {momentum.fX[idx], momentum.fY[idx], momentum.fZ[idx], momentum.fT[idx]},
           vector.subspan(vector_offsets[idx],
                          vector_offsets[idx + 1] - vector_offsets[idx])};
}
```

Random Access Operator (3/3)

4 SoA Wrapper with Token Injection

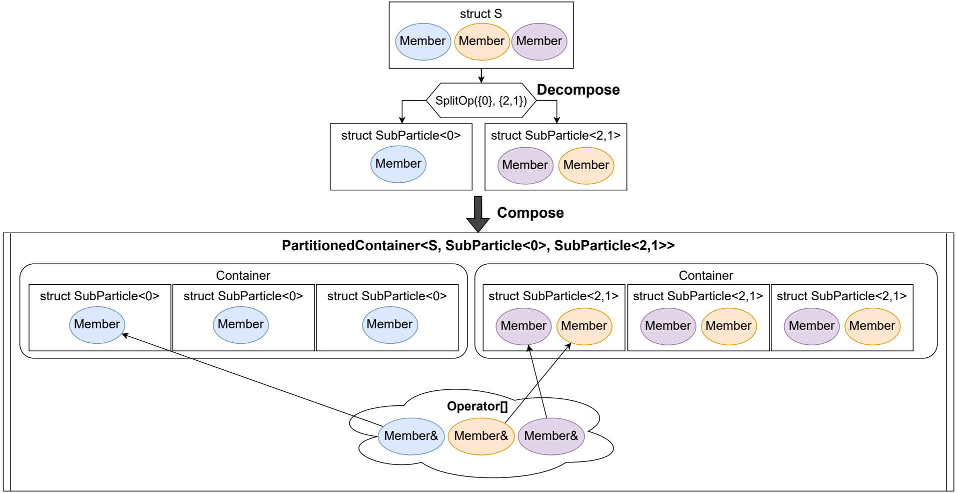
```
consteval std::meta::info generate_view_tokens(std::meta::info member,
                                               std::meta::info id_tokens) {
  auto name = identifier_of(member);
  auto type = type_of(member);

  if (type is container) {
    auto offset = ^^{ \id(name, "_offsets"sv)[idx] };
    auto size = ^^{ \id(name, "_offsets"sv)[idx + 1] - \tokens(offset) };
    return ^^{ \tokens(id_tokens).subspan(\tokens(offset), \tokens(size)) };
  } else if (type is nested struct) {
    std::meta::list_builder substruct_tokens{};
    for (auto submember : nsdms(type)) {
      auto submember_id = ^^{ \tokens(id_tokens).\id(identifier_of(submember)) };
      substruct_tokens += generate_view_tokens(submember, submember_id);
    }

    return ^^{ { \tokens(substruct_tokens) } };
  }

  Scalar Type
  return ^^{ \tokens(id_tokens)[idx] };
}
```

Table of Contents

5 Structure Splitter

- ▶ Introduction
- ▶ C++26 Reflection (P2996)
- ▶ SoA Wrapper
- ▶ SoA Wrapper with Token Injection
- ▶ Structure Splitter
- ▶ Performance Comparison
- ▶ My experience with Reflection
- ▶ Conclusion

Structure Splitter



User Interface

5 Structure Splitter

```
struct Point {
  float &x;
  int &y;
  double &z;
};

template <size_t... Is>
struct SubPoint;

// Decompose
consteval { SplitStruct<Point, SubPoint>(SplitOp({0}), SplitOp({1}), SplitOp({2})); }

// Compose
using PointSoA = PartitionedContainer<Point, SubPoint<0>, SubPoint<1>, SubPoint<2>>;
```

```
template <typename In, template <auto> typename Out,
        typename... SplitOps>
consteval void SplitStruct(SplitOps... ops);
```

We can promote compile-time values to static storage!
Allows `std::vector` as non-type template parameter.

```
consteval auto SplitOp(std::vector<int> indices) {
  return std::define_static_array(indices); }
```

Creating Substructures

5 Structure Splitter

```cpp
consteval { SplitStruct<Point, SubPoint>(SplitOp({0}), SplitOp({1}), SplitOp({2})); }
```

↓↓↓

```cpp
template <size_t... Is> struct SubPoint;

template <typename In, template <auto> typename Out, typename... SplitOps>
consteval void SplitStruct(SplitOps... ops) {
  auto unpack_op = [](std::span<const int> op) {
    std::vector<std::meta::info> unpacked;
    for (size_t i: op) {
      unpacked.push_back(std::meta::reflect_constant(i));
    }
    return unpacked;
  };

  (define_aggregate(substitute(^^Out, unpack_op(ops)), GetMemberSpecs<In>(ops)), ...);
};
```

↓↓↓

```cpp
template<> struct SubPoint<0> { float x; }
template<> struct SubPoint<1> { int y; }
template<> struct SubPoint<2> { double z; }
```

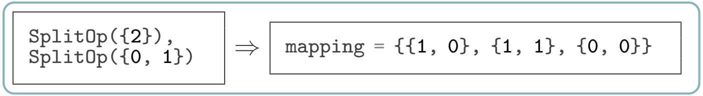
Container For Substructures

5 Structure Splitter

```
template <typename ProxyRef, typename... T>
struct PartitionedContainerContiguous {
private:
  struct Partitions;
  consteval {
    define_aggregate(^^Partitions,
                     {data_member_spec(substitute(^^std::span, {^^T}))...}); }

  Partitions p;
  std::byte *storage;
  size_t n;

  static constexpr auto mapping = find_in_partitions(^^ProxyRef, ^^Partitions);
  ...
}
```

Mapping to ProxyRef

5 Structure Splitter



```
SplitOp({2}),
SplitOp({0, 1})
```

⇒

```
mapping = {{1, 0}, {1, 1}, {0, 0}}
```

```
consteval auto find_in_partitions(std::meta::info original_type,
                                  std::meta::info partitioned_type) {
  auto original_members = nsdms(original_type);
  auto partitions = nsdms(partitioned_type);
  std::vector<std::pair<size_t, size_t>> mapping(original_members.size());

  for (size_t ip = 0; ip < partitions.size(); ++ip) {
    auto partition_type = template_arguments_of(type_of(partitions[ip]))[0];
    auto partition_members = nsdms(partition_type);
    for (size_t im = 0; im < partition_members.size(); ++im) {
      for (size_t io = 0; io < original_members.size(); ++io) {
        if (identifier_of(original_members[io]) == identifier_of(partition_members[im])) {
          mapping[io] = {ip, im};
          break;
  }}}}
  return std::define_static_array(mapping);
}
```

Random Access Operator

5 Structure Splitter

```cpp
inline ProxyRef operator[](const size_t index) const {
  return [&]<size_t... Is>(std::index_sequence<Is...>) constexpr -> ProxyRef {
    constexpr auto partitions = nsdms(^^Partitions);
    return ProxyRef{
      p.[: partitions[mapping[Is].first] :][index]
       .[: nsdms(template_arguments_of(
            type_of(partitions[mapping[Is].first]))[0])[mapping[Is].second] :]
      ...};
  }(std::make_index_sequence<nsdms(^^ProxyRef).size()>());
}
```

Table of Contents

6 Performance Comparison

- ▶ Introduction
- ▶ C++26 Reflection (P2996)
- ▶ SoA Wrapper
- ▶ SoA Wrapper with Token Injection
- ▶ Structure Splitter
- ▶ Performance Comparison
- ▶ My experience with Reflection
- ▶ Conclusion

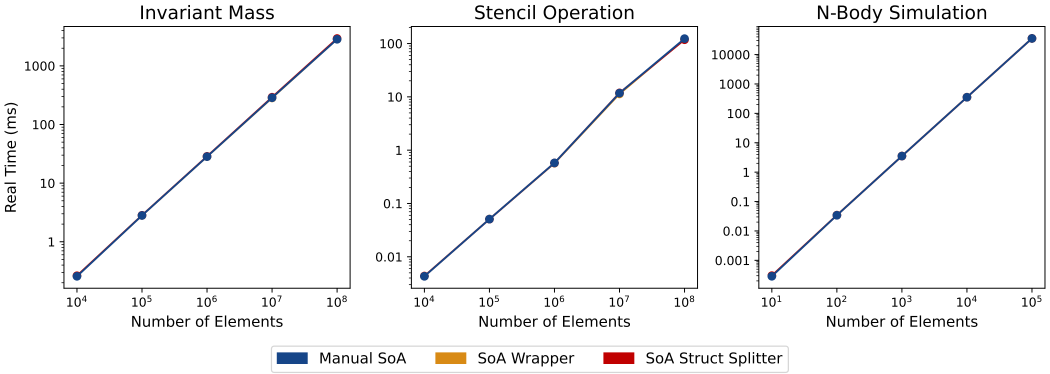
Performance Comparison

- AMD EPYC 9654 CPU, 1 core
- Available Compilers:
  - Clang P2996 compiler (commit: 2ea0a79): most up-to-date with P2996
  - EDG experimental reflection front-end: supports token sequences but is not up-to-date with P2996
  - GCC trunk: did not have time to test this yet
- `-O3 -ftree-vectorize -march=native -ffast-math`
- Three benchmarks:
  - Invariant mass computation with PxPyPzM vectors
  - Stencil operation (1D poisson equation solver)
  - N-body simulation

- Performance of all SoA versions should have the **same performance** as manual SoA

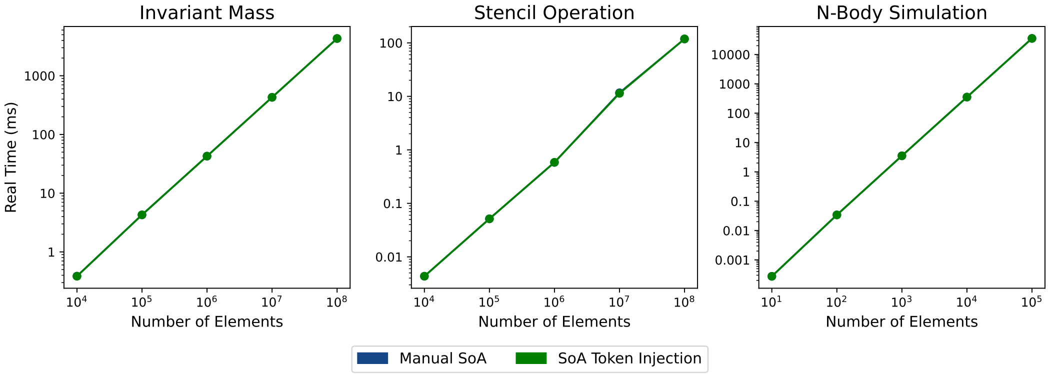
Runtime Performance Results with Clang-P2996

6 Performance Comparison



Runtime Performance Results with EDG

6 Performance Comparison



Compilation Time

6 Performance Comparison

Measured with `-ftime-report`

SoA Manual	SoA Wrapper	SoA Token Injection	SoA Struct Splitter
3.50 seconds	6.43 seconds	3.93 seconds	6.94 seconds

- Some more time spent on front end
  SoA Wrapper: +0.71 seconds **SoA Struct Splitter:** +1.30 seconds
- Some more time spent on optimizer
  SoA Wrapper: +1.21 seconds **SoA Struct Splitter:** +1.17 seconds

Disclaimers:

- Implementations are still experimental
- Clang-p2996 is with `libc++` and EDG is based on GCC with `libstdc++`

My Experience with Reflection

7 My experience with Reflection

- Metaprogramming with (a list of) types feels more straightforward now with `std::vector<std::meta::info>`, `nonstatic_data_members_of`, `template_arguments_of`, `template for`, etc.
- Token sequence injection feels incredibly powerful

- `define_aggregate` has a limited scope, e.g., can only add members to incomplete classes
  - Often encountered situations where I want to add to an already defined struct
- Would be nice to have a range splice to do something with a range of reflections without a helper lambda

My Experience with Reflection

7 My experience with Reflection

- Error messages can still improve

The error you get if you do something like:

```
consteval {
  auto refl = ^^int;
  auto m = nsdms(refl);
}
```

```
<source>:8:5: error: evaluating expression of a consteval block must be a constant expression

    8 |     consteval {
      |     ^
/cefs/a0/a065e145c2a13d121c68cfe8_clang-bb-p2996-trunk-20260204/bin/../include/c++/v1/meta:673:12:
      | note: subexpression not valid in a constant expression
  673 |     return __metafunction(detail::__metafn_get_begin_member_decl_of, reflectedEntity, ^^sentinel);
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/cefs/a0/a065e145c2a13d121c68cfe8_clang-bb-p2996-trunk-20260204/bin/../include/c++/v1/meta:596:21: note: in call to 'm_front.operator()(^^(type))'
  596 |     , m_currInfoItr{m_front(reflectedEntity)}
      |                     ^~~~~~~~~~~~~~~~~~~~~~~
/cefs/a0/a065e145c2a13d121c68cfe8_clang-bb-p2996-trunk-20260204/bin/../include/c++/v1/meta:635:7: note: in call to 'iterator(^^(type))'
  635 |     : m_first(reflection), m_last()
      |       ^~~~~~~~~~~~~~~~~~
/cefs/a0/a065e145c2a13d121c68cfe8_clang-bb-p2996-trunk-20260204/bin/../include/c++/v1/meta:1047:21: note: in call to 'range(^^(type))'
 1047 |   return __filtered(range(r).to_vec(), [=](info r) { return is_accessible(r, ctx); });
      |                     ^~~~~~~
/cefs/a0/a065e145c2a13d121c68cfe8_clang-bb-p2996-trunk-20260204/bin/../include/c++/v1/meta:1086:21: note: in call to
      | 'members_of(^^(type), {^^(declaration), ^^(null)})'
 1086 |   return __filtered(members_of(r, ctx), is_nonstatic_data_member);
      |                     ^~~~~~~~~~~~~~~~~
<source>:9:18: note: in call to 'nonstatic_data_members_of(^^(type), {^^(declaration), ^^(null)})'
    9 |       auto mem = nonstatic_data_members_of(^^int, std::meta::access_context::current());
      |                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:8:5: note: in call to '[]() -> void {
    auto mem = nonstatic_data_members_of(^^(...), std::meta::access_context::current()); }.operator()()'
    8 |     consteval {
      |     ^~~~~~~~~~
    9 |       auto mem = nonstatic_data_members_of(^^int, std::meta::access_context::current());
```

Conclusion

- Try out reflection yourself!
  - Clang: https://godbolt.org/z/71647q5Mo
  - EDG: https://godbolt.org/z/4hK564scs
  - GCC: https://godbolt.org/z/bx9zjMo8P
- SoA Abstractions: https://github.com/cern-nextgen/reflmempp
- Performance benchmarks: https://github.com/cern-nextgen/wp1.7-soa-benchmark
- Contact: jolly.chen@cern.ch

This work has been funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.