Optimizing Memory Access Patterns through Automatic Data Layout Transformation

Jolly Chen[1,2], **Ana Lucia Varbanescu**[1], **Axel Naumann**[2]
jolly.chen@cern.ch
[1]University of Twente (NL) — [2]CERN (CH)

International Conference on Performance Engineering (ICPE'25)
Emerging Research Track
May 9, 2025

UNIVERSITY OF TWENTE.

Memory Access Patterns

- **Memory Access Patterns** (MAPs) = the way data is read from or written to memory
- Algorithm + **Data Layout** = MAP
  - In C++, the data *layout* is decided by the data structure *definition*
- Two common ways of organizing data:

Array of Structures (AoS)

```
1 struct S { double x1, x2, x3, x4; };
2 template <int N> using AoS = S[N];
```

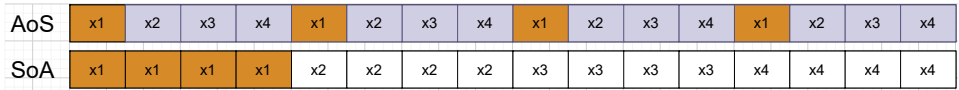
Structure of Arrays (SoA)

```
1 template <int N>
2 struct SoA { double x1[N], x2[N], x3[N], x4[N]; };
```

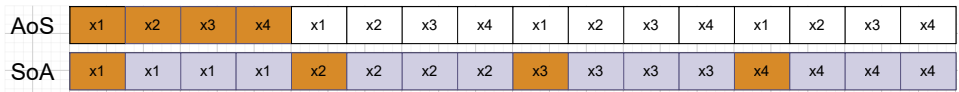
- MAPs strongly influence **cache efficiency** and **parallel data access latency**
  - It is important to choose a good data layout!

The Impact of the Data Layout

- Example where SoA performs better than AoS:
  - **Algorithm**: iterate over data, use *some* members, e.g., only `x1`



  - Better **cache efficiency** and more **vectorization** with SoA
  - Measured ∼**5x speedup** (AoS ∼300ms vs. SoA ∼5ms) on Intel Core i9-9900K CPU with 20 members
- ...but sometimes AoS is better
  - **Algorithm**: iterate over data, use *all* members



  - Measured ∼**2.5x speedup** (AoS ∼0.1s vs. SoA ∼0.25s) on Intel Core i9-9900K CPU with 8 data members

Layout Conversion

- Predicting the optimal layout for complex algorithms and data structures is non-trivial
  - → We should **experiment with different layouts**
- Changing the layout requires a lot of changes in data structure **definitions** and **access**

```
1 struct P { double x, y, z; };
2 using AoS = P[N];
3
4 void sum(AoS p) {
5   for (int i = 0; i < N; ++i) {
6     p[i].z = p[i].x + p[i].y + p[i].z;
7 }}
```

```
1 struct SoA { double x[N], y[N], z[N]; };
2
3
4 void sum(SoA p) {
5   for (int i = 0; i < N; ++i) {
6     p.z[i] = p.x[i] + p.y[i] + p.z[i];
7 }}
```

- We propose a solution that changes the data layout without changing the access syntax
  - — We use C++ **compile-time reflection** features, planned for C++26 (and beyond)

Our Solution

- Before:

```
struct P { double x, y, z; };
using AoS = P[N];

void sum(AoS p) {
  for (int i = 0; i < N; ++i) {
    p[i].z = p[i].x + p[i].y + p[i].z;
}}
```

```
struct SoA { double x[N], y[N], z[N]; };


void sum(SoA p) {
  for (int i = 0; i < N; ++i) {
    p.z[i] = p.x[i] + p.y[i] + p.z[i];
}}
```

- After:

```
struct P { double x, y, z; };
using AoS = P[N];

void sum(AoS p) {
  for (int i = 0; i < N; ++i) {
    p[i].z = p[i].x + p[i].y + p[i].z;
}}
```

```
struct PRef { double &x, &y, &z };
using SoA = rmpp::vector<P>;

void sum(SoA p) {
  for (int i = 0; i < N; ++i) {
    p[i].z = p[i].x + p[i].y + p[i].z;
}}
```

How Does It Work? – C++ Reflection

- Main proposal proposes two new operators (`wg21.link/p2996`):

`^^x`	"*Lift*" operand `x` to a **reflection** value
`[: refl :]`	"*Splice*" a reflection to **produce grammatical elements**

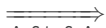
- Token Sequences (`wg21.link/p3294`):

`^^{ x }`	A token sequence
`queue_injection(...)`	Inject sequence at the end of `consteval` block

Implementation

User code

```
1
2 struct PRef {
3   double &x, &y, &z;
4 };
5
6 using SoA =
7     rmpp::vector<PRef>;
```



Implementation of `rmpp::vector`

```
1  namespace rmpp {
2  template <>
3  struct vector<PRef> {
4    std::vector<std::byte> storage;
5
6    consteval {
7    for (auto m : nonstatic_data_members_of(^^PRef)) {
8      auto val_type = type_remove_cvref(type_of(m));
9      queue_injection(^^{
10       std::span<typename[: \(val_type) :]>
11         \id(identifier_of(m));
12   });}}
13
14   PRef operator[](const size_t idx) {
15     consteval {
16     auto params = std::meta::list_builder{};
17     for (auto m :
18          nonstatic_data_members_of(^^PRef)) {
19       params += ^^{ \id(identifier_of(m))[idx] };
20     }
21     queue_injection(^^{return {\tokens(params)};});
22 }}}; }
```

Implementation

User code

```
1
2  struct PRef {
3    double &x, &y, &z;
4  };
5
6  using SoA =
7      rmpp::vector<PRef>;
```

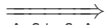
Implementation of `rmpp::vector`

```
1  namespace rmpp {
2  template <>
3  struct vector<PRef> {
4    std::vector<std::byte> storage;
5
6    consteval {
7    for (auto m : nonstatic_data_members_of(^^PRef)) {
8      auto val_type = type_remove_cvref(type_of(m));
9      queue_injection(^^{
10       std::span<typename[: \(val_type) :]>
11         \id(identifier_of(m));
12   });}}
13
14   PRef operator[](const size_t idx) {
15     consteval {
16     auto params = std::meta::list_builder{};
17     for (auto m :
18          nonstatic_data_members_of(^^PRef)) {
19       params += ^^{ \id(identifier_of(m))[idx] };
20     }
21     queue_injection(^^{return {\tokens(params)};});
22 }}}; }
```

Implementation

User code

```
1
2 struct PRef {
3   double &x, &y, &z;
4 };
5
6 using SoA =
7     rmpp::vector<PRef>;
```



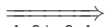
Implementation of `rmpp::vector`

```
1  namespace rmpp {
2  template <>
3  struct vector<PRef> {
4    std::vector<std::byte> storage;
5
6    // What the reflection translates to
7    std::span<double> x, y, z;
8
9    PRef operator[](const size_t idx) {
10     consteval {
11     auto params = std::meta::list_builder{};
12     for (auto m :
13          nonstatic_data_members_of(^^PRef)) {
14       params += ^^{ \id(identifier_of(m))[idx] };
15     }
16     queue_injection(^^{return {\tokens(params)};});
17 }}}; }
```

Implementation

User code

```
1
2  struct PRef {
3    double &x, &y, &z;
4  };
5
6  using SoA =
7      rmpp::vector<PRef>;
```



Implementation of `rmpp::vector`

```
1  namespace rmpp {
2  template <>
3  struct vector<PRef> {
4    std::vector<std::byte> storage;
5
6    // What the reflection translates to
7    std::span<double> x, y, z;
8
9    PRef operator[](const size_t idx) {
10     consteval {
11     auto params = std::meta::list_builder{};
12     for (auto m :
13          nonstatic_data_members_of(^^PRef)) {
14       params += ^^{ \id(identifier_of(m))[idx] };
15     }
16     queue_injection(^^{return {\tokens(params)};});
17 }}}; }
```

Implementation

User code

```
1
2  struct PRef {
3    double &x, &y, &z;
4  };
5
6  using SoA =
7      rmpp::vector<PRef>;
```

Implementation of `rmpp::vector`

```
1  namespace rmpp {
2  template <>
3  struct vector<PRef> {
4    std::vector<std::byte> storage;
5
6    // What the reflection translates to
7    std::span<double> x, y, z;
8
9    // What the reflection translates to
10   PRef operator[](const size_t idx) {
11     return { x[idx], y[idx], z[idx] };
12 }}; }
```

Implementation

User code

```
1
2  struct PRef {
3    double &x, &y, &z;
4  };
5
6  using SoA =
7      rmpp::vector<PRef>;
```

AoS to SoA

Implementation of `rmpp::vector`

```
1   namespace rmpp {
2   template <>
3   struct vector<PRef> {
4     std::vector<std::byte> storage;
5
6     // What the reflection translates to
7     std::span<double> x, y, z;
8
9     // What the reflection translates to
10    PRef operator[](const size_t idx) {
11      return { x[idx], y[idx], z[idx] };
12  }}; }
```

- Data can be accessed as `data.x[i]` and `data[i].x`

Summary

- Important to experiment with different data layouts
- Working on a **data layout converter** using C++ reflection
  - Prototype available on GitHub at: `https://github.com/cern-nextgen/reflmempp`
- **Goal**: make it easier to optimize memory access patterns
- Future work: more layouts and measure performance/usability

- Contact: `jolly.chen@cern.ch`

Prototype

Paper