Delegating Behaviors in C++: A Practical Tour of the Available Mechanisms

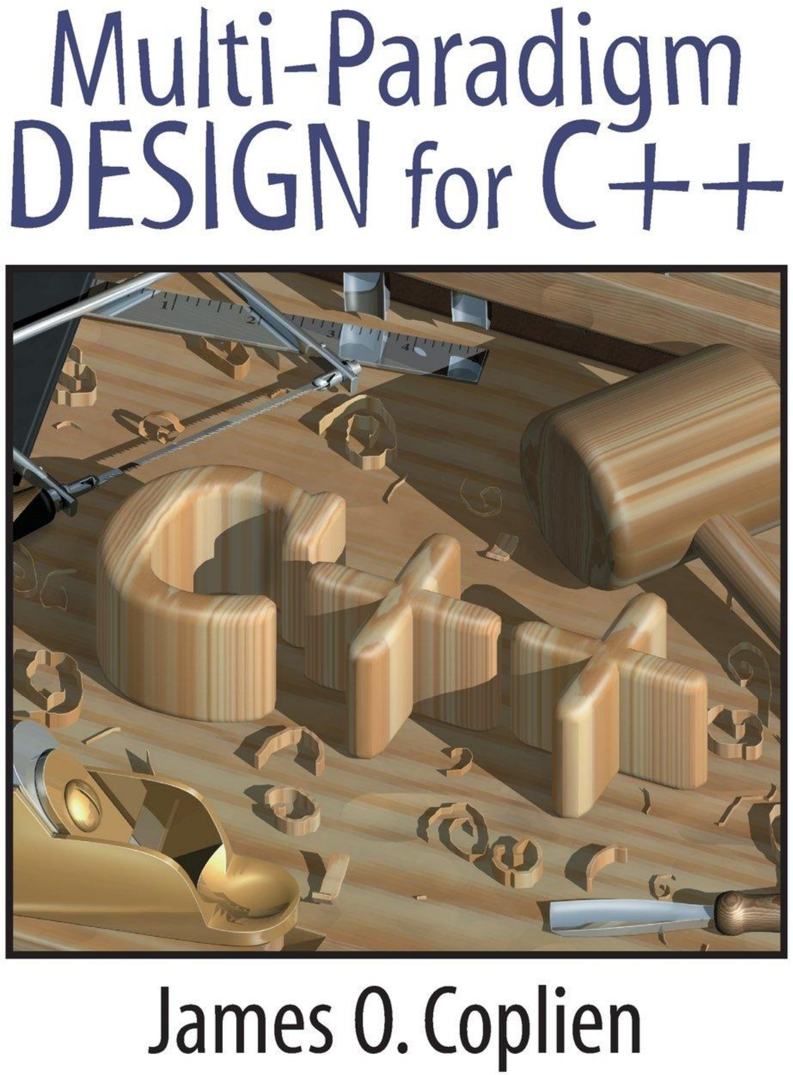
Daniele Pallastrelli, daniele77.github.io

25.10.2025, Italian C++

So… what is the truly unique characteristic of C++?

C++ is multiparadigm.

That’s what makes it truly powerful — and sometimes dangerous.



No building is made from one material – neither should your software

Use the right material for the right part.

Passing Behavior – The C++ way

Why passing behavior matters

Reusability — Extensibility — Testability

Non-functional properties that define quality.

Reusability

A component can be reused as-is across code and projects

Reusability

```
void sort_users(std::vector<User>& users, ??? cmp)
{
    // uses cmp as criterion to sort users
}
```

Reusability

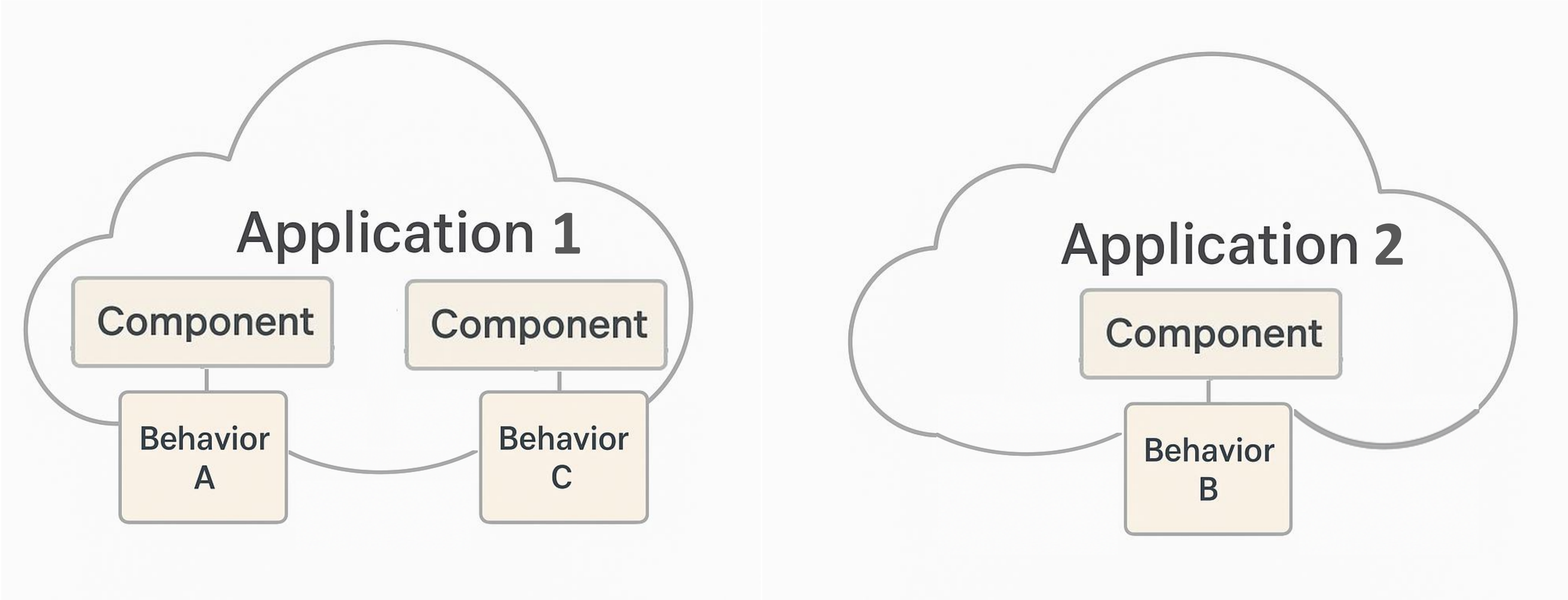
```
class button : public widget
{
public:
    void on_click(??? handler)
    {
      // set the handler for the onclick event
    }
};
```

Reusability

```cpp
class user
{
    // ...
};

class credential
{
    // ...
};

class role
{
    // ...
};
```

Inject an authentication mechanism (behavior)

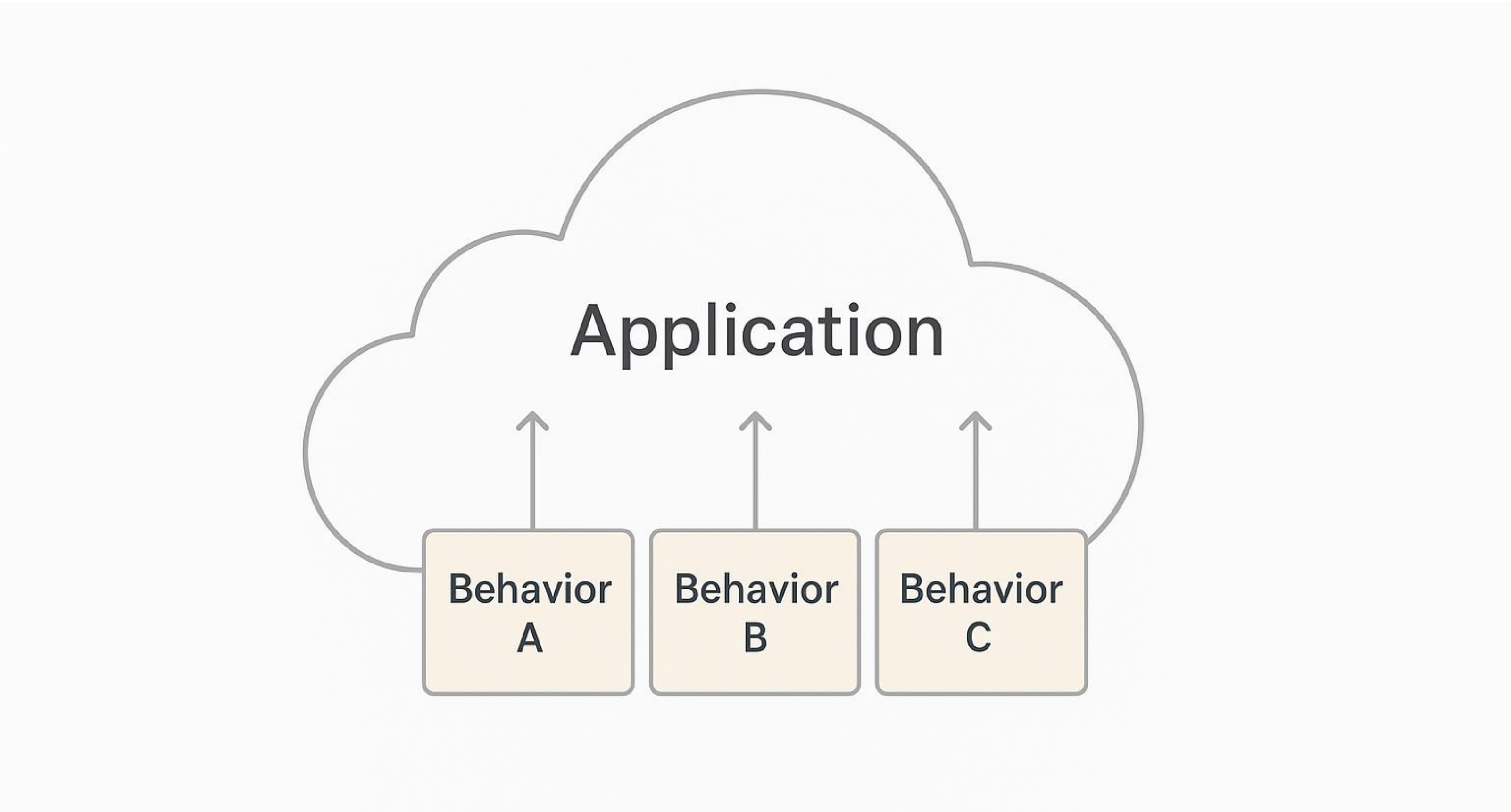
Reusability



Extensibility

An application is extendible if it allows developers to **add new features** or **modify existing behaviors without altering the application's core code** or **requiring significant changes to its existing structure**.

Extensibility



Testability

A software component is testable if its behavior can be reliably verified without the component's core source code having to be modified or 'opened up' for internal inspection.

Unit test => **Reusability**

Reuse the component in a test environment.

Integration test => **Extensibility**

Test the *whole system* and extend it with test doubles or stubs at its boundaries.

What Can We Pass as a Behavior?

```
void sort_users(std::vector<User>& users, ??? cmp)
{
    // uses cmp as criterion to sort users
}
```

What Can We Pass as a Behavior? A Function

```cpp
bool compare_by_age(const User& a, const User& b)
{
    return a.age < b.age;
}


...
sort_users(users, compare_by_age);
...
```

What Can We Pass as a Behavior? A Lambda

```
sort_users(users, [](const User& a, const User& b) { return a.age < b.age; });
```

What Can We Pass as a Behavior? A Functor

```
// Functor sorting users by age
struct SortByAge {
    bool operator()(const User& a, const User& b) const {
        return a.age < b.age;
    }
};

...
sort_users(users, SortByAge{});
...
```

What Can We Pass as a Behavior? A Polymorphic Object

```cpp
// Interface for user comparison
class UserPredicate {
public:
    virtual ~UserPredicate() = default;
    virtual bool compare(const User& a, const User& b) const = 0;
};

// Concrete implementation of UserPredicate to sort by age
class SortByAge : public UserPredicate {
public:
    bool compare(const User& a, const User& b) const override
    {
        return a.age < b.age;
    }
};

...
sort_users(users, SortByAge{});
...
```

What about formal parameter types?

```
void sort_users(std::vector<User>& users, ??? cmp)
{
    // uses cmp as criterion to sort users
}
```

The Good Ol' Function Pointer

```
using CompFun = bool(const User&, const User&);

void sort_users(std::vector<User>& users, CompFun cmp)
{
    // uses cmp as criterion to sort users
}
```

You can pass:

- a free function
- a stateless lambda (implicitly convertible)

Note:

Member function pointers aren't useful here: they bind the behavior to a specific instance.

Polymorphic Interface

```
// Interface for user comparison
class UserPredicate {
public:
    virtual ~UserPredicate() = default;
    virtual bool compare(const User& a, const User& b) const = 0;
};


void sort_users(std::vector<User>& users, const UserPredicate& cmp)
{
    // uses cmp.compare as criterion to sort users
}
```

- Enables runtime polymorphism
- Supports stateful behaviors
- Requires lifetime management (not necessarily heap allocation)

Templates (and Concepts)

```
template<std::strict_weak_order<const User&, const User&> Compare>
void sort_users(std::vector<User>& users, Compare cmp)
{
    // uses cmp as criterion to sort users
}
```

- Supports any **callable** type — something that provides operator () (functor, lambda, free function, etc.)
- Zero-cost abstraction (Enables **inlining and optimization**)
- Behavior fixed at **compile time** (no runtime substitution)
- Hard to **store or reuse later** — type depends on the callable
- Must be defined in header file
- May increase **code size**

std::function

```cpp
void sort_users(
    std::vector<User>& users,
    const std::function<bool(const User&, const User&)>& cmp)
{
    // uses cmp as criterion to sort users
}
```

- Holds any callable (lambda, functor, function pointer)
- Uniform, copyable, storable type
- Can be changed or reassigned at runtime
- Slight overhead, no inlining

std::variant + std::visit

```
struct ByAge  { bool compare(const User& a, const User& b) { return a.age < b.age;   } };
struct ByName { bool compare(const User& a, const User& b) { return a.name < b.name; } };
struct ById   { bool compare(const User& a, const User& b) { return a.id < b.id;     } };

using Comparison = std::variant<ByAge, ByName, ById>;

void sort_users(std::vector<User>& users, Comparison cmp)
{
    std::visit(
        [&](auto&& comp) {
            // use comp.compare(a,b) to sort users
        },
        cmp
    );
}
```

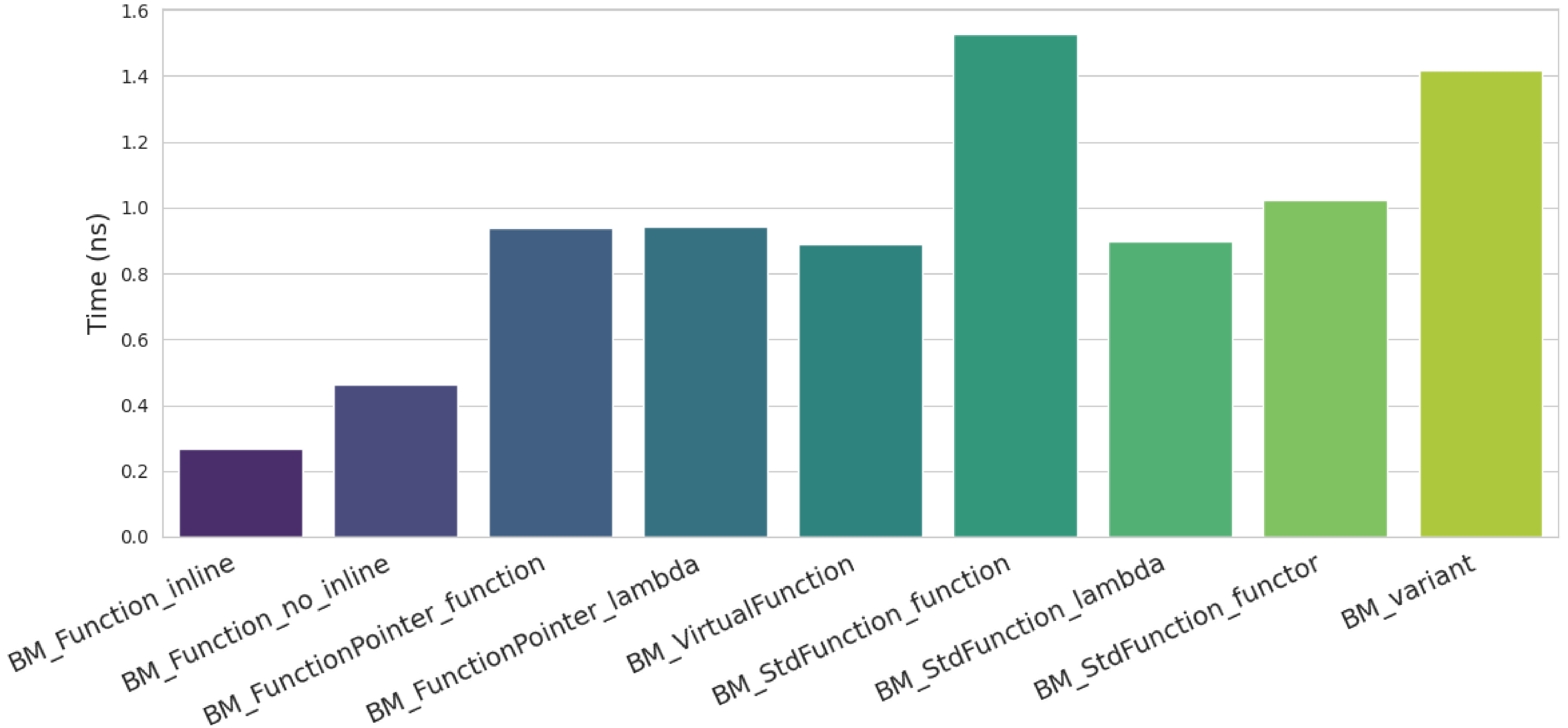
- std::visit performs a **type-safe runtime dispatch**
- Behavior can't be extended at runtime (closed set)
- Usually implemented with a jump table

	Free function	Lambda	Functor	Polymorphic Object	std::variant
Function pointer	✓	Only stateless			
Template	✓	✓	✓	✓	✓
std::function	✓	✓	✓	✓	✓
Interface ptr/ref				✓	
std::variant					✓

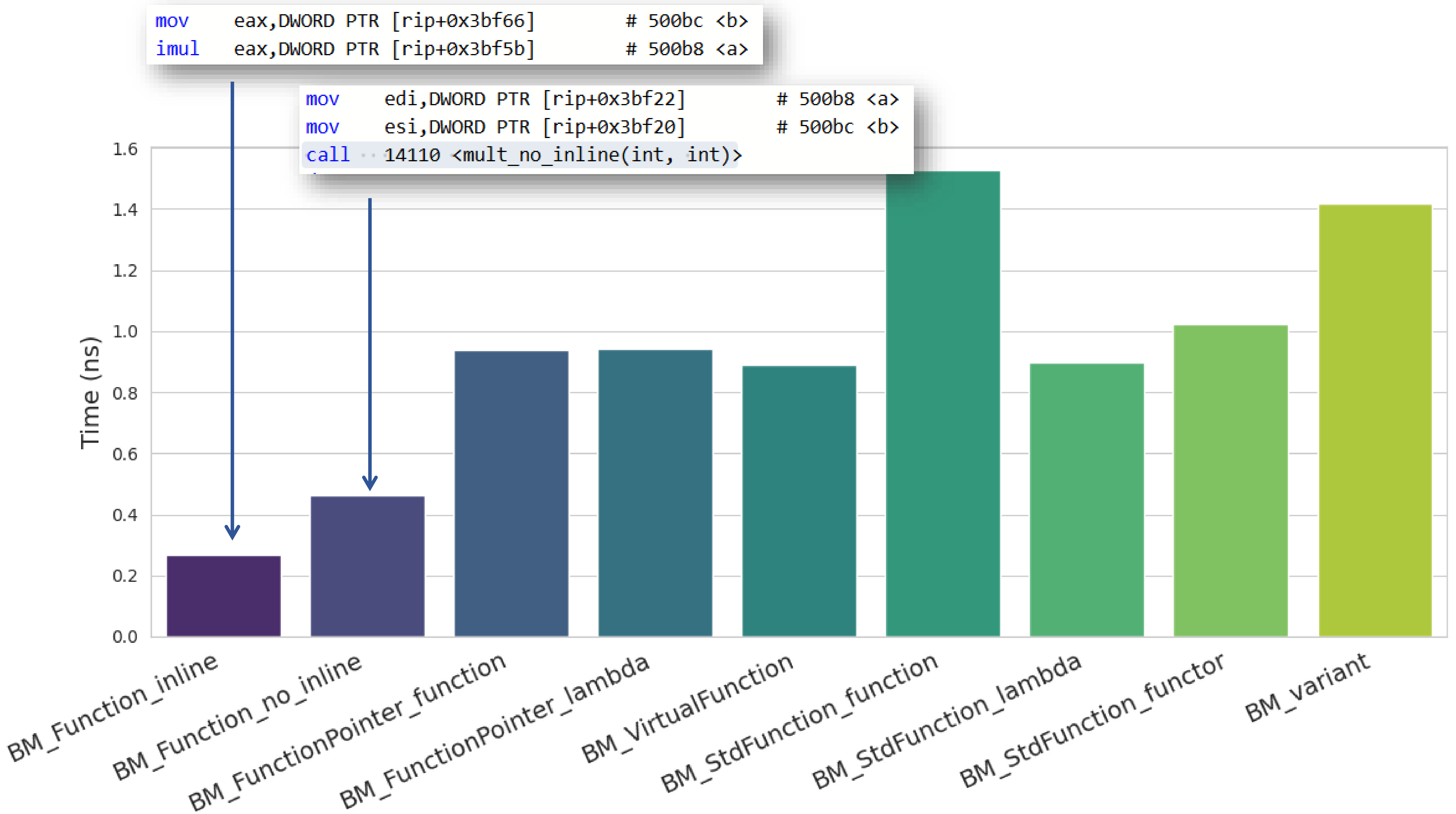
Performance … *sigh*

Do you promise to not take the following results too seriously and as qualitative results only?

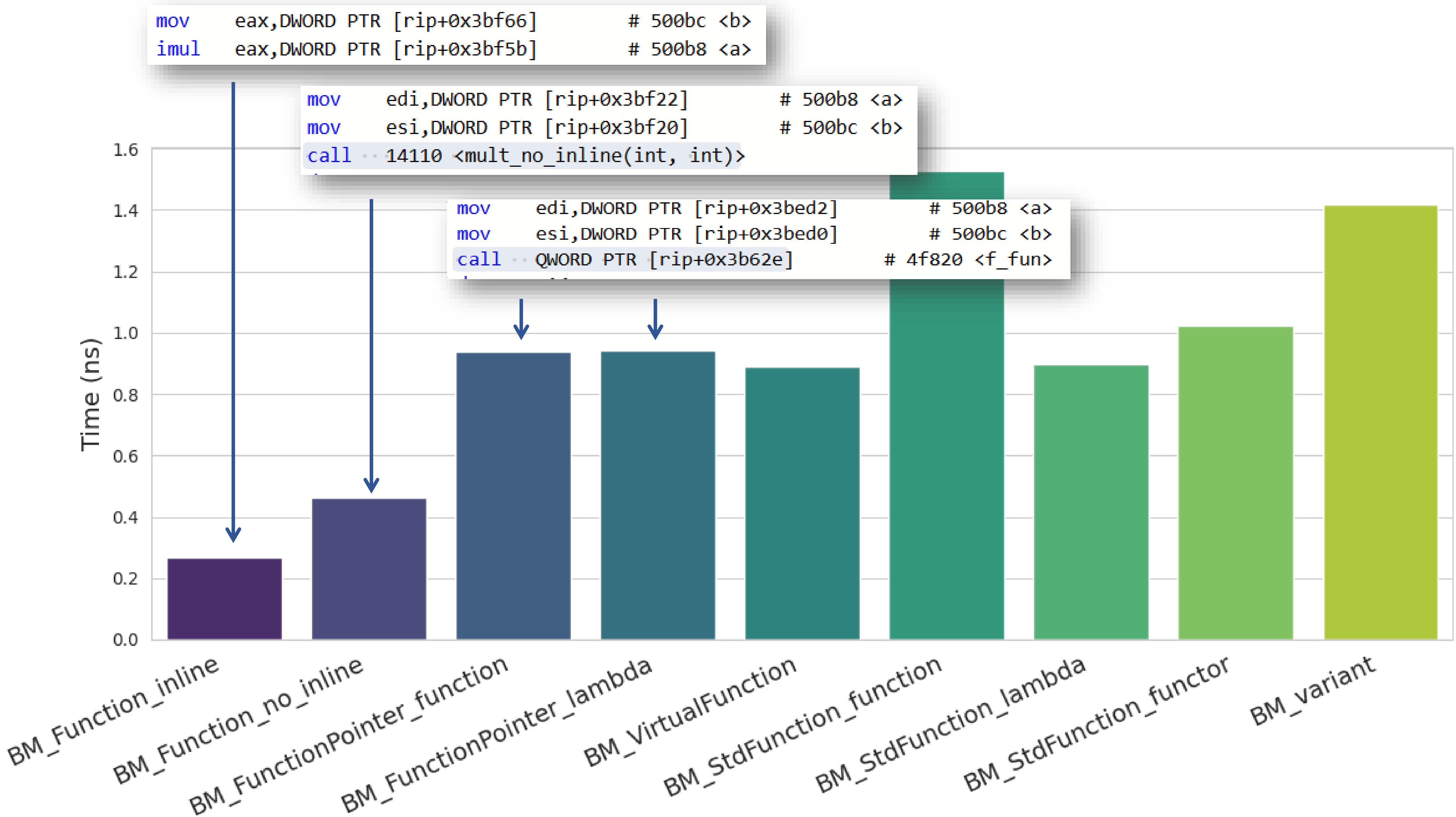
	Free function	Lambda	Functor	Polymorphic Object	std::variant
Function pointer	✓	Only stateless			
Template	✓	✓	✓	✓	✓
std::function	✓	✓	✓	✓	✓
Interface ptr/ref				✓	
std::variant					✓

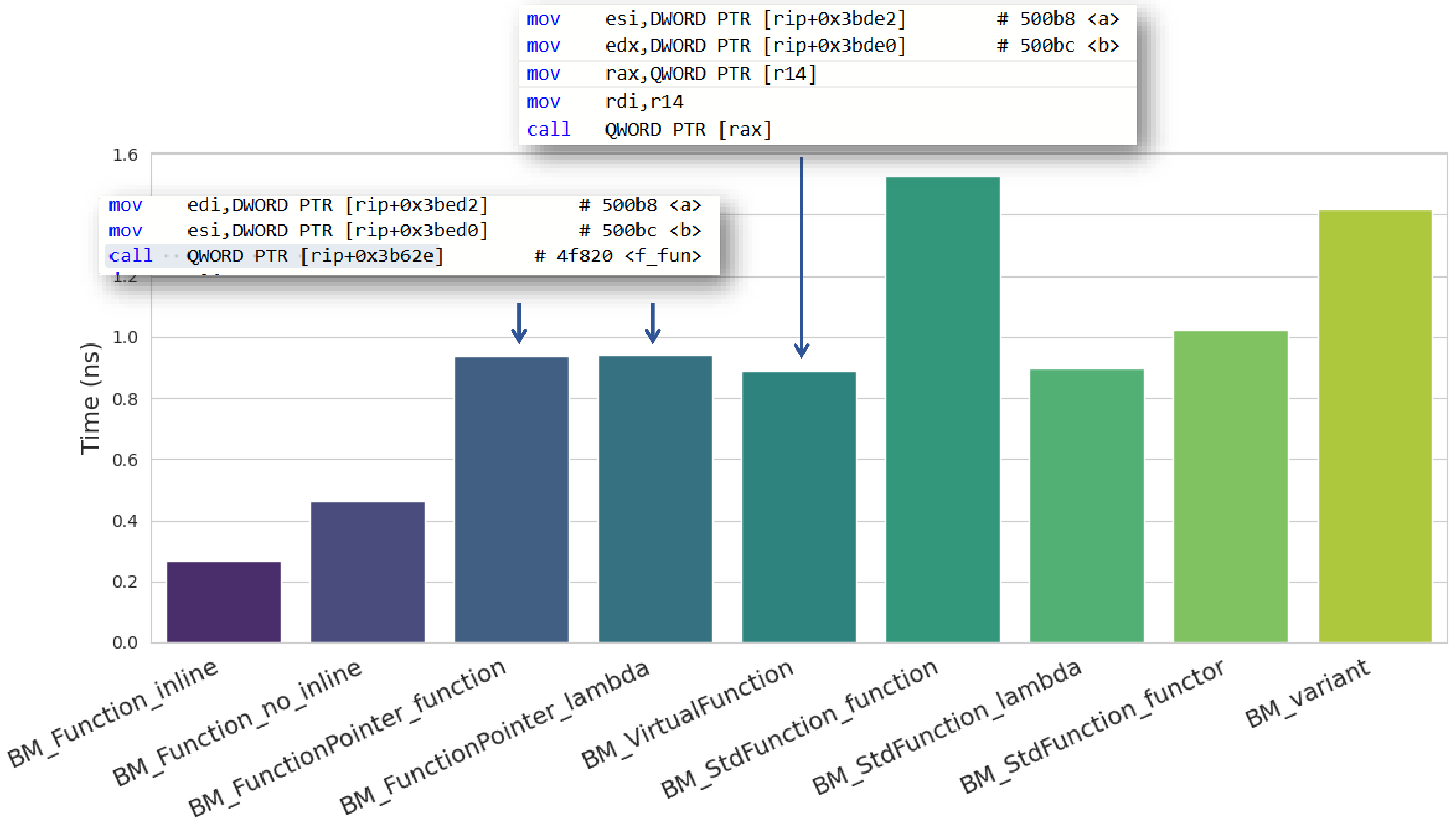


Time (ns)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
BM_Function_inline
BM_Function_no_inline
BM_FunctionPointer_function
BM_FunctionPointer_lambda
BM_VirtualFunction
BM_StdFunction_function
BM_StdFunction_lambda
BM_StdFunction_functor
BM_variant



mov eax,DWORD PTR [rip+0x3bf66] # 500bc <b>
imul eax,DWORD PTR [rip+0x3bf5b] # 500b8 <a>
mov edi,DWORD PTR [rip+0x3bf22] # 500b8 <a>
mov esi,DWORD PTR [rip+0x3bf20] # 500bc <b>
call 14110 <mult_no_inline(int, int)>
Time (ns)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
BM_Function_inline
BM_Function_no_inline
BM_FunctionPointer_function
BM_FunctionPointer_lambda
BM_VirtualFunction
BM_StdFunction_function
BM_StdFunction_lambda
BM_StdFunction_functor
BM_variant


mov eax,DWORD PTR [rip+0x3bf66] # 500bc <b>
imul eax,DWORD PTR [rip+0x3bf5b] # 500b8 <a>
mov edi,DWORD PTR [rip+0x3bf22] # 500b8 <a>
mov esi,DWORD PTR [rip+0x3bf20] # 500bc <b>
call 14110 <mult_no_inline(int, int)>
mov edi,DWORD PTR [rip+0x3bed2] # 500b8 <a>
mov esi,DWORD PTR [rip+0x3bed0] # 500bc <b>
call QWORD PTR [rip+0x3b62e] # 4f820 <f_fun>
Time (ns)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
BM_Function_inline
BM_Function_no_inline
BM_FunctionPointer_function
BM_FunctionPointer_lambda
BM_VirtualFunction
BM_StdFunction_function
BM_StdFunction_lambda
BM_StdFunction_functor
BM_variant



mov esi,DWORD PTR [rip+0x3bde2] # 500b8 <a>
mov edx,DWORD PTR [rip+0x3bde0] # 500bc <b>
mov rax,QWORD PTR [r14]
mov rdi,r14
call QWORD PTR [rax]
mov edi,DWORD PTR [rip+0x3bed2] # 500b8 <a>
mov esi,DWORD PTR [rip+0x3bed0] # 500bc <b>
call QWORD PTR [rip+0x3b62e] # 4f820 <f_fun>
Time (ns)
1.6
1.0
0.8
0.6
0.4
0.2
0.0
BM_Function_inline
BM_Function_no_inline
BM_FunctionPointer_function
BM_FunctionPointer_lambda
BM_VirtualFunction
BM_StdFunction_function
BM_StdFunction_lambda
BM_StdFunction_functor
BM_variant

Where is this case?

```
template <typename F>
void foo(F callback)
{
    // do something useful and provide result
    callback(result);
}
```

Where is this case?

```cpp
template <typename F>
void foo(F callback)
{
    // do something useful and provide result
    callback(result);
}
```

... it depends:

```cpp
void bar(int r)
{
    cout << r;
}

foo(bar);
```

```cpp
struct Bar {
    void operator()(int r) const
    { cout << r; }
};

foo(Bar{});
```

```cpp
foo([](int r) { cout << r; });
```

```cpp
void bar(int r)
{
  cout << r;
}

foo(bar);
```

```cpp
template<>
void foo<void (*)(int)>(void (*cb)(int))
{
  // calculate result
  cb(result);
}
```

```cpp
struct Bar {
  void operator()(int r) const
  { cout << r; }
};

foo(Bar{});
```

```cpp
template<>
void foo<Bar>(Bar cb)
{
  // calculate result
  cb.operator()(result);
}
```

```cpp
foo([](int r) { cout << r; });
```

```cpp
template<>
void foo<__lambda_28_7>(__lambda_28_7 cb)
{
  // calculate result
  cb.operator()(result);
}
```

```
void bar(int r)
{
  cout << r;
}

foo(bar);
```

```
template<>
void foo<void (*)(int)>(void (*cb)(int))
{
  // calculate result
  cb(result);
}
```

```
call     rdx
```

```
struct Bar {
  void operator()(int r) const
  { cout << r; }
};

foo(Bar{});
```

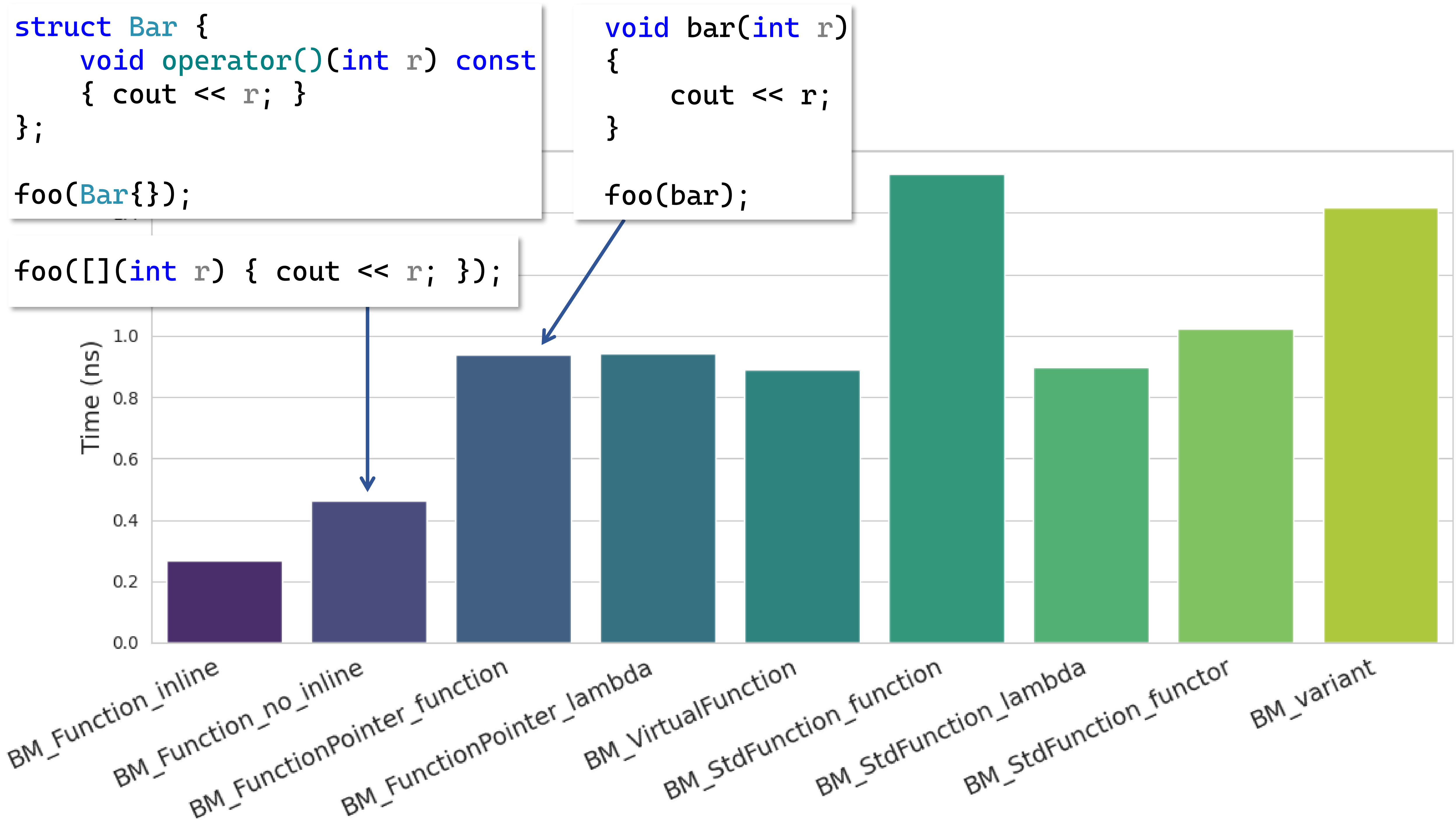
```
template<>
void foo<Bar>(Bar cb)
{
  // calculate result
  cb.operator()(result);
}
```

```
call     Bar::operator()(int) const
```

```
foo([](int r) { cout << r; });
```

```
template<>
void foo<__lambda_28_7>(__lambda_28_7 cb)
{
  // calculate result
  cb.operator()(result);
}
```

```
call     main::'lambda'(int)::operator()(int) const
```

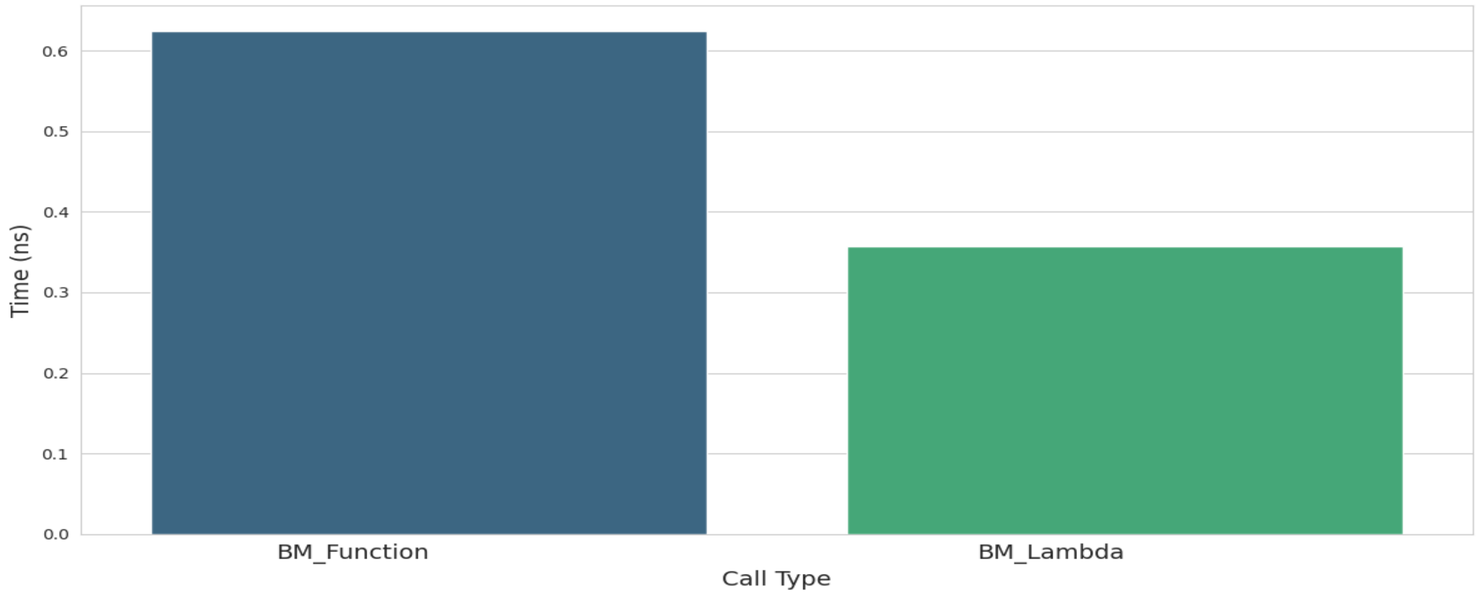

struct Bar {
void operator()(int r) const
{ cout << r; }
};
foo(Bar{});
void bar(int r)
{
cout << r;
}
foo(bar);
foo([](int r) { cout << r; });
Time (ns)
0.0
0.2
0.4
0.6
0.8
1.0
BM_Function_inline
BM_Function_no_inline
BM_FunctionPointer_function
BM_FunctionPointer_lambda
BM_VirtualFunction
BM_StdFunction_function
BM_StdFunction_lambda
BM_StdFunction_functor
BM_variant

When a lambda beats a function pointer…

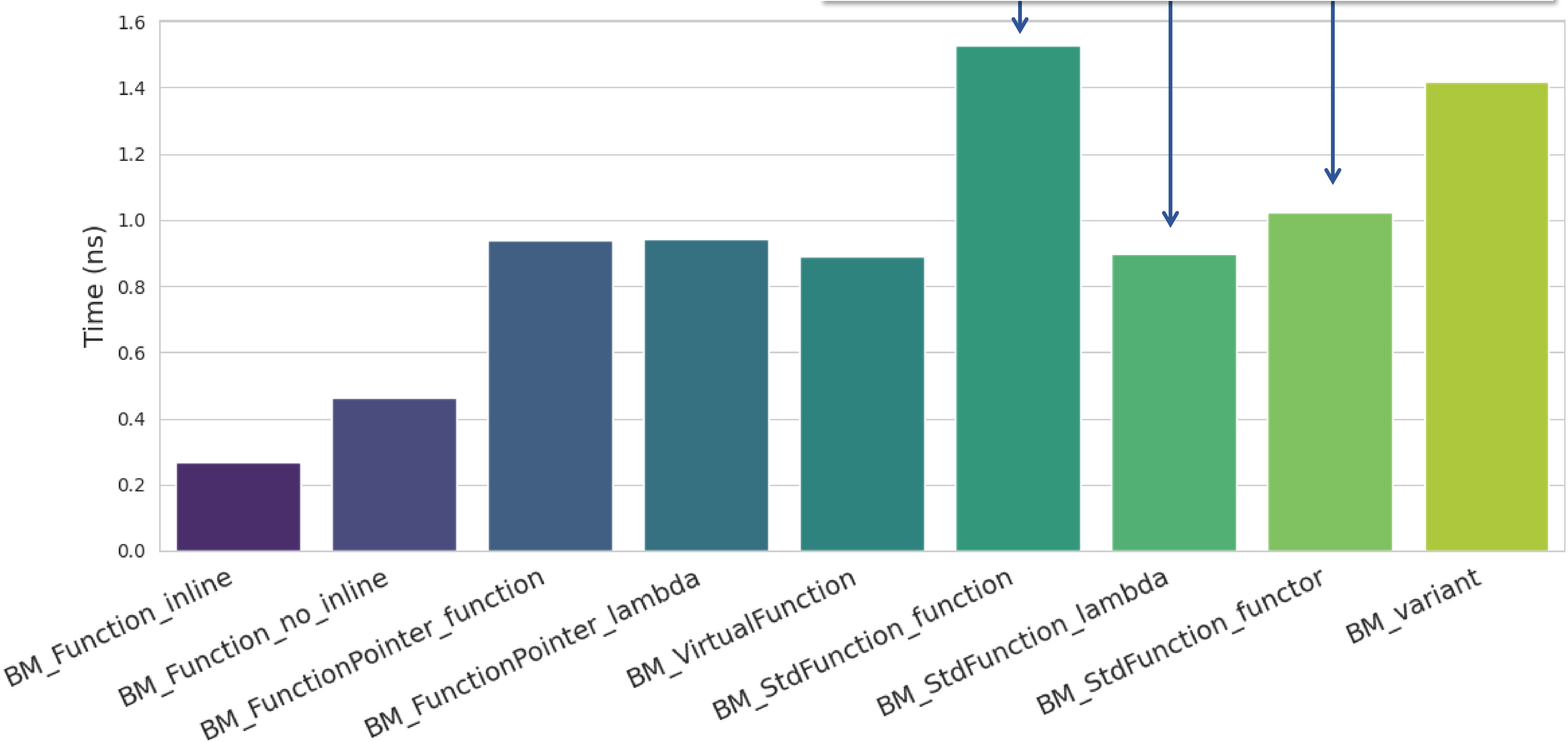
```
template<typename F>
void exec(F&& f) {
    std::forward<F>(f)();
}
```

```
void bar() {}

exec(bar);
```

```
auto foo = []() {};

exec(foo);
```



- Call to std::function::operator()
- Type-erasure dispatch through a virtual function
- **Template instantiation for the original callable type**
- The invoker calls the actual callable



std::visit jump table

```
std::visit(visitor, var);
```

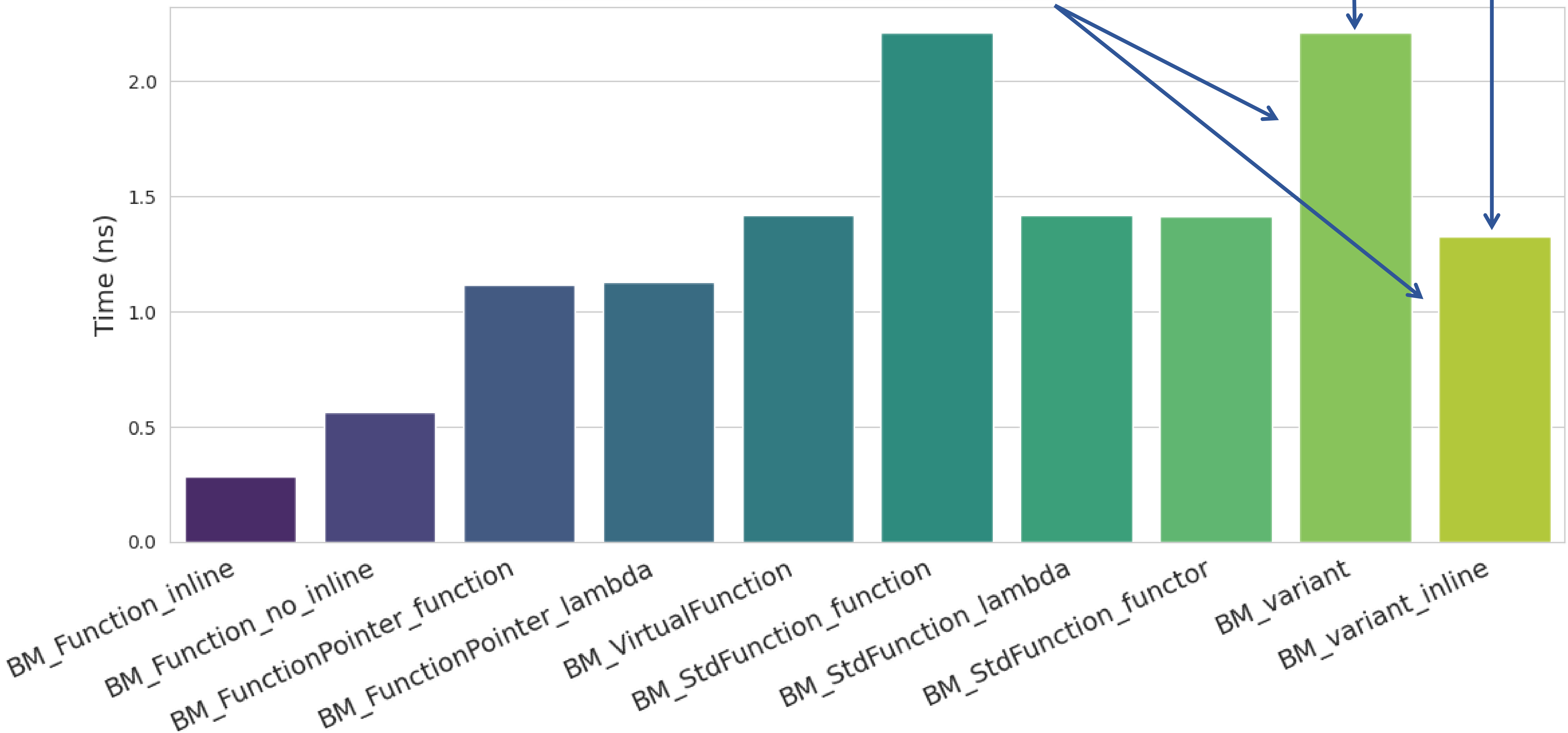
```
static constexpr auto table[] = {
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<A>(var)); },
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<B>(var)); },
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<C>(var)); }
};
table[var.index()](visitor, var);
```

```cpp
struct Circle   { void draw() { /* ... */ } };
struct Square   { void draw() { /* ... */ } };
struct Triangle { void draw() { /* ... */ } };

std::variant<Circle, Square, Triangle> obj;

std::visit(
    [&](auto&& shape) { shape.draw(); }, // visitor
    obj // variant
);
```

```cpp
static constexpr auto table[] = {
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<Circle>(var)); },
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<Square>(var)); },
    [](auto& vis, auto& var) -> decltype(auto) { vis(std::get<Triangle>(var)); }
};
auto visitor = [&](auto&& shape) { shape.draw(); };
table[obj.index()](visitor, obj);
```

- Check the index stored in the std::variant
- Jump to the right case in the jump table
- Setup the operands
- Call the function (can be inlined)

```
movzx  eax,BYTE PTR [rsp+0xf]
lea    rsi,[rsp+0xe]
lea    rdi,[rsp+0xd]
call   QWORD PTR [rax*8+0x424340]
```



Micro-benchmarks: what they really tell us?

Influenced by:

- Library optimizations (std::function, std::visit, …)
- Compiler optimizations (inlining, devirtualization, …)
- Hardware effects (cache, branch prediction, …)

The second should not be part of the benchmark

Micro-benchmarks measure the benchmark itself, not your application!

Real optimization = **profile** the real system

Performance can vary with compilers, versions, flags, CPUs, load, execution path

What is stable are the **design properties** of C++ constructs

Speed fades.
Design stays.

Keeping a partial state of computation

Free function

- Global or static data, singleton & c (booh!)
- C idiom: function ptr + void* parameter

Lambda

- Captures local variables (by copy / ref)
- Init-capture and mutable

Functor

- Keeps state in member variables

Polymorphic object

- Derived class stores its own state

std::variant

- Each alternative carries its own state

Scattering vs. Gathering (AKA Functions vs. Objects)

Functions/lambdas:

single behavior (how to do one thing)

Objects, functors, polymorphic objects, or std::variant:

multiple related behaviors (how this kind of things behaves)

Objects can **share state** and **group coherent logic**.

Guideline:
If behaviors belong together, **keep them together** — wrap them into a single object instead of scattering separate functions.

Passing Logic vs. Passing Behavior

With **functions** or **lambdas**, you pass a **piece of logic to execute**.

With **polymorphic objects**, **functors**, or **std::variant**, you pass **a value that *embodies* the behavior**.

Case 1 (easy): throw-away policy

Use policy immediately

```cpp
class UserRepository {
public:
    template <typename Policy>
    void process_users(Policy policy)
    {
        // use policy here, e.g., apply policy to process users
    }
};
```

Easy, just a method template

Case 2 (hard): long-lived policy

Store policy for later use

```
class UserRepository {
public:
    template <typename Policy>
    void set_policy(Policy p)
    {
        // store the policy to later use, uhm...
        policy = p;
    }

    void use_policy()
    {
        // use the stored policy
    }
private:
    ??? policy;
};
```

Solution #1: class template

```
template <typename Policy>
class UserRepository {
public:
    explicit UserRepository(Policy p) : policy(std::move(p)) {}
    void use_policy()
    {
        // use the stored policy
    }
private:
    Policy policy;
};

int main()
{
    UserRepository users{
        [](){ /* do something useful */ }
    }; // C++17 type deduction from constructor arguments
    users.use_policy();
}
```

Solution #1: class template

```
template <typename Policy>
class UserRepository {
public:
    explicit UserRepository(Policy p) :
    void use_policy()
    {
        // use the stored policy
    }
private:
    Policy policy;
};

int main()
{
    UserRepository users{
        [](){ /* do something useful */ }
    }; // C++17 type deduction from constru
    users.use_policy();
}
```

- ✅ Zero-cost abstraction
- ✅ Any callable type
- ❌ Policy fixed at compile time
- ❌ Each policy produce a different type

Solution #2: Alexandrescu policy-based design

```cpp
// Main class parametrized by policies
template <typename LoggerPolicy, typename StoragePolicy>
class MyService : private LoggerPolicy, private StoragePolicy {
public:
    void save(const std::string& data) {
        this->log("Saving data: " + data);   // logging policy
        this->store(data);                    // storage policy
    }
};
```

Solution #2: Alexandrescu policy-based design

```cpp
// Main class parametrized by policies
template <typename LoggerPolicy, typename StoragePolicy>
class MyService : private LoggerPolicy, private StoragePolicy {
public:
    void save(const std::string& data) {
        this->log("Saving data: " + data);    // logging policy
        this->store(data);                    // storage policy
    }
};
```

Not really about passing behavior

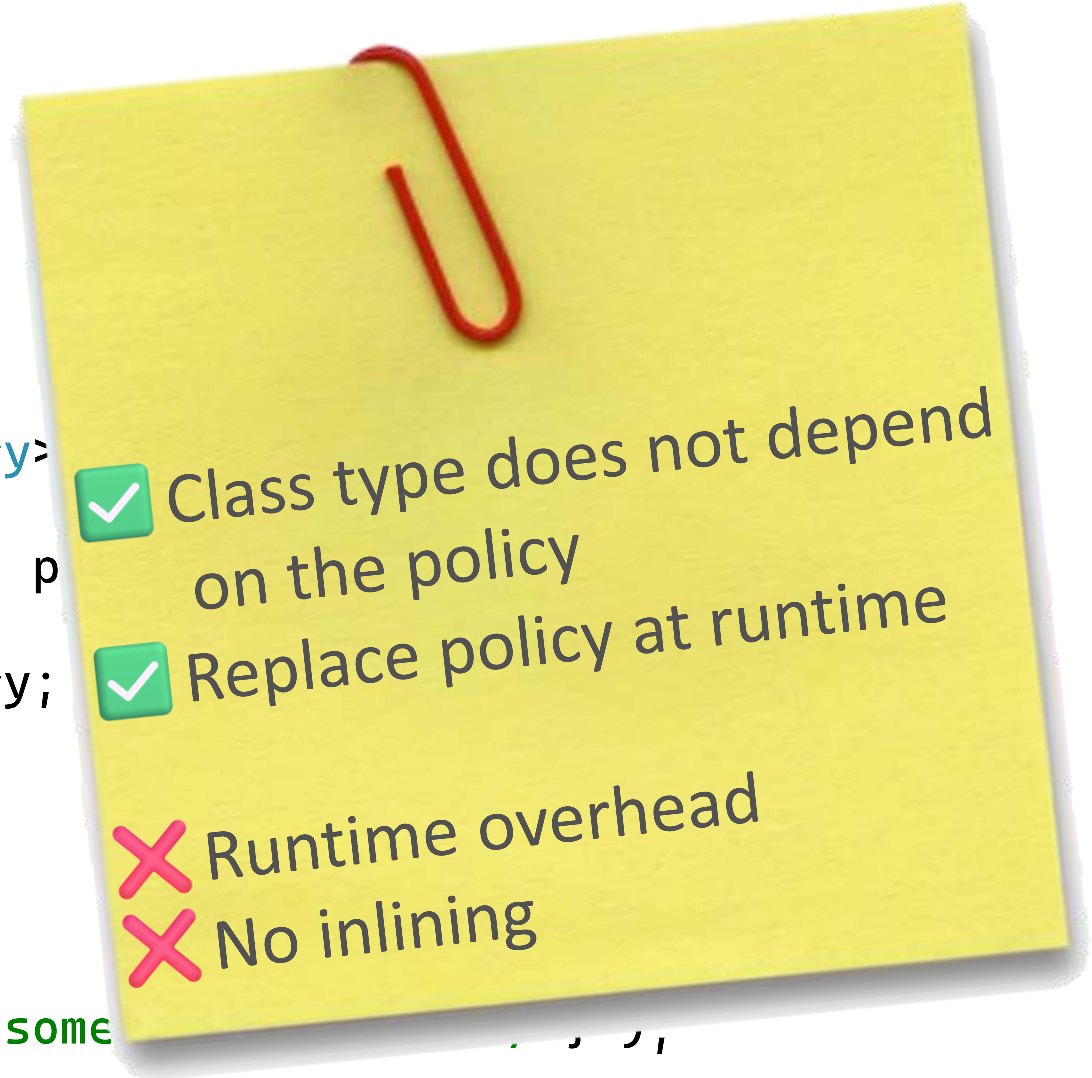
But **composing behaviors at compile time**

Solution #3: type erasure

```
class UserRepository {
public:
    template <typename Policy>
    void set_policy(Policy&& p)
    {
        policy = std::forward<Policy>(p);
    }
    void use_policy() { if (policy) policy(); }
private:
    std::function<void(void)> policy;
};

int main()
{
    UserRepository users;
    users.set_policy( []() { /* do something useful */ } );
    users.use_policy();
}
```

Solution #3: type erasure

```
class UserRepository {
public:
    template <typename Policy>
    void set_policy(Policy&& p)
    {
        policy = std::forward<Policy>
    }
    void use_policy() { if (policy) p
private:
    std::function<void(void)> policy;
};

int main()
{
    UserRepository users;
    users.set_policy( []() { /* do some
    users.use_policy();
}
```



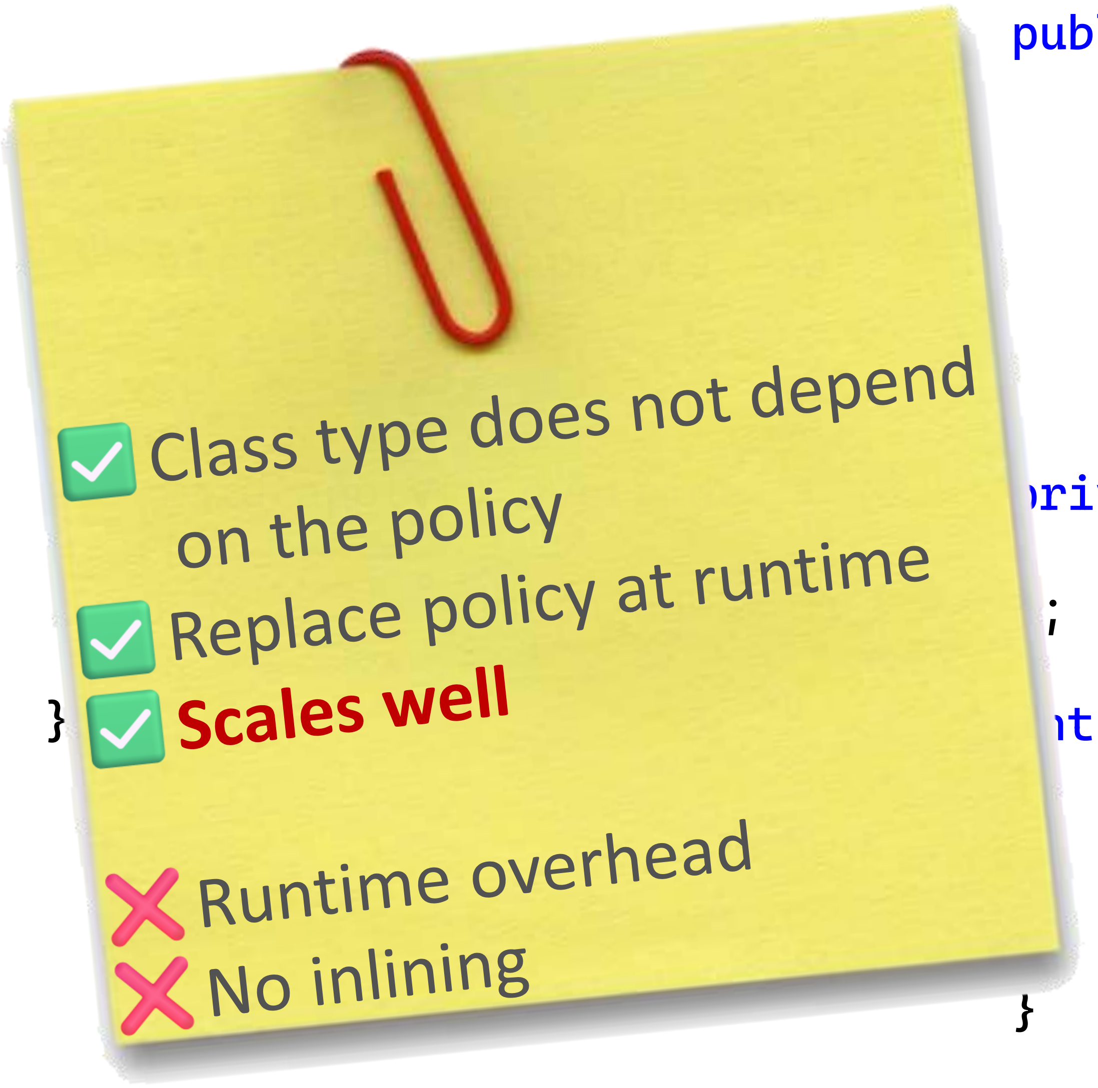
Solution #3: type erasure (the oldest one ☺)

```cpp
struct Policy {
    virtual ~Policy() = default;
    virtual void apply() = 0;
};

struct PrintPolicy : Policy {
    void apply() override
    {
        // do something useful
    }
};
```

GoF Strategy Pattern

```cpp
class UserRepository {
public:
    void set_policy(std::unique_ptr<Policy> p)
    {
        policy = std::move(p);
    }

    void use_policy() { if (policy) policy->apply(); }

private:
    std::unique_ptr<Policy> policy;
};

int main()
{
    UserRepository users;
    users.set_policy(std::make_unique<PrintPolicy>());
    users.use_policy();
}
```

Solution #3: type erasure (the oldest one ☺)




```
class UserRepository {
public:
    void set_policy(std::unique_ptr<Policy> p)
    {
        policy = std::move(p);
    }

    void use_policy() { if (policy) policy->apply(); }

private:
    std::unique_ptr<Policy> policy;
};

int main()

    UserRepository users;
    users.set_policy(std::make_unique<PrintPolicy>());
    users.use_policy();
}
```

Storing the behavior

It depends on *how* your interface accepts the behavior

- Function pointer
- Interface reference or pointer
- std::function
- std::variant

  All can store behaviors directly

- Template

  You need some form of type erasure to store it or make the whole class a template (deciding everything at compile-time)

Run-time substitution vs. compile-time definition

When is Run-Time Change Essential?

- Configuration is user-dependent (e.g., choosing from a config file which algorithm to use).
- Flexibility is required (run-time loaded plugins, drivers, rule engines).
- You must react to variable conditions (e.g., if the network is slow -> change the caching policy; if the train enters a tunnel -> change the communication strategy).
- You want to test/swap behaviors without rebuilding (e.g., in embedded or mission-critical systems where recompiling isn’t always feasible).

With a template interface you cannot change the behavior at run-time

Runtime Extensibility

Template class and std::variant:

- Closed Set of Types at Compile-Time
- Can’t use in plugin-based or dynamic architectures.

Template propagation

Template propagation

```cpp
class Foo {
    std::function<void(void)> policy;
};

class FooClient {
    Foo foo; // no dependency from policy
};
```

Class using std::function or interface

- The dependency is hidden inside the class
- Users don’t need to know what behavior it contains – they only depend on its interface
- **Only the creator of Foo** — the one who injects the behavior — needs to know which behavior is used.

Template propagation

Template class Bar<T>

The dependency is exposed in the type.

Anyone using Bar must either:

- Explicitly know which T is used (impractical — policies leak all over the code), or
- Be a template themselves (cascade effect: parameters multiply — everything becomes a header)

```
template <typename Policy>
class Bar { /* ... */ };

// template class
template <typename Policy>
class BarClient1 {
    Bar<Policy> bar;
};

// depends on ConcretePolicy
class BarClient2 {
    Bar<ConcretePolicy> bar;
};
```

Template propagation

```cpp
class Foo {
    std::function<void(void)> policy;
};

class FooClient {
    Foo foo; // no dependency from policy
};
```

```cpp
template <typename Policy>
class Bar { /* ... */ };

// template class
template <typename Policy>
class BarClient1 {
    Bar<Policy> bar;
};

// depends on ConcretePolicy
class BarClient2 {
    Bar<ConcretePolicy> bar;
};
```

Templates propagate dependencies at compile-time
Interfaces and std::function contain them

Summary

Use template if:

- Performances needed AND
- Confined in a subsystem

Else

- Use polimorphism or std::function

A hybrid is common:

- Use templates internally for performance-critical parts.
- Use type-erasure at architectural boundaries to avoid propagating templates everywhere.

What we’ve explored

- Why *passing behaviors* matters
- The main techniques C++ gives us
- How they work under the hood
- Performance insights
- And a glimpse into design consequences

What to take away

Before writing code, think about the alternatives.
Don’t pick a technique just because it’s *faster* —
performance depends on context, and you’ll never really know until you measure *your own* code.

Remember

C++ is *multi-paradigm*.
Use all the tools it gives you.
Don’t build your entire house out of glass — not even the foundations.

See? No UML.
That was hard.

References

Me: daniele.pallastrelli@gmail.com

Me: @DPallastrelli

Github: http://github.com/daniele77

Web: softwareexploring.blogspot.com