Performance of std::expected with Monadic Operations

VITALY FANASKOV

About me

Vitaly Fanaskov

Principal software engineer at reMarkable

Work on C++ frameworks and libraries

Agenda

- Error handling in general
- Briefly about exceptions and std::expected
- Important implementation details of monadic operations
- Common use cases and benchmarks
- Practical recommendations

Error handling

Error handling methods

- Return codes
- Function arguments
- Exceptions
- Optional and Result/Either/Expected

Different error handling methods

[Error handling in C++ - Vitaly Fanaskov - NDC Techtown 2022]

What and why?

- Exceptions
- std::expected

- Similar flexibility
- Similar use cases
- The closest alternatives

Briefly about exceptions

Definition

- Unplanned event
- Occurs during program execution
- Creates an exception object

- The object is handed to the runtime system (*throw*)
- The runtime system is looking for exception handler
- Exception handler is found (*catch*) or terminate/crash if not

Exception safety

- No-throw (always succeeds)
- Strong (keeps the same state as before an exception thrown)
- Basic (no resources are leaked, objects' invariants are intact)
- No exception safety (memory corruption, leaks, inconsistency, etc.)

Try-catch example

```
const std::vector vec{1, 2, 3};

try {
    std::println("The fourth element: {}", vec.at(3));
} catch (const std::out_of_range& e) {
    std::println("Error: {}", e.what());
}

// Error: vector::_M_range_check: __n (which is 3) ≥ this→size() (which is 3)
```

Throw an exception

```
template <class E>
E elementByIndex(const std::vector<E>& vec, std::size_t index)
{
   if (index ⩾ vec.size()) {
       throw std::out_of_range(
           std::format(
               "Error: index (which is {}) out of range (which is [0, {})).",
               index,
               vec.size()));
   }

   return vec[index];
}
```

Briefly about std::expected

Basic example of using std::expected

```
#include <expected>



std::expected<int, Error> expectedBox;
expectedBox = 42;

std::println("The value is: {}", expectedBox.value());
```

std::expected as a return value

When an operation can fail and we need to know why:

```
std::expected<Widget, WidgetError> loadWidget()
{
    // If error
    return std::unexpected(WidgetError{ /* ... */ });

    // Actual result
    return Widget{ /* ... */ };
}
```

Process std::expected

```
void loadWidget()
{
   if (const auto widgetBox = getNewWidget(); widgetBox.has_value()) {
       const auto widget = widgetBox.value();
       // Do something with the widget ...
   } else {
       const auto error = widgetBox.error();
       // Handle the error ...
   }
}
```

Process std::expected with monadic operations

```
getWidget()
    .and_then([](const auto &widget) -> std::expected<Widget, WidgetError> {
        // Do something with the widget ...
        return widget;
    })
    .transform([](const auto &widget) -> ID { return widget.id(); })
    .or_else([](const auto& error) -> std::expected<ID, WidgetError> {
        log(error);
        // Possibly recover and/or cleanup ...
        // Return value or error ...
    })
```

Available monadic operations

```
transform(F f)
    f: Value → ValueOut
transform_error(F f)
    f: Error → ErrorOut


and_then(F f)
    f: Value → expected<ValueOut, Error>
or_else(F f)
    f: Error → expected<Value, ErrorOut>
```

For more information

[Monadic Operations in Modern C++: A Practical Approach - Vitaly Fanaskov - CppCon 2024]

Important implementation details of std::expected

Types

- `template <class E> class unexpected`
- `template <class E> class bad_expected_access`
- `template <class T, class E> class expected`

Data structure

```
union {
    T val;
    E unex; // NOTE: not unexpected<E>!
};
bool has_val;
```

Data structure

```
union {
    T val;
    E unex; // NOTE: not unexpected<E>!
};
bool has_val;
```

Simplified “and_then”

```
template <class F>
auto and_then(F&& f) &
{
    using U = std::invoke_result_t<F, T&>; // + compile-time checks
    if (has_val) {
        return std::invoke(std::forward<F>(f), val);
    } else {
        return U{unex};
    }
}
```

Simplified “and_then”

```
template <class F>
auto and_then(F&& f) &
{
    using U = std::invoke_result_t<F, T&>; // + compile-time checks
    if (has_val) {
        return std::invoke(std::forward<F>(f), val);
    } else {
        return U{unex};
    }
}
```

Simplified “transform”

```
template <class F>
auto transform(F&& f) &
{
    using U = std::invoke_result_t<F, T&>;
    using R = expected<U, E>;

    if (has_val) {
        return R{std::invoke(std::forward<F>(f), val)};
    } else {
        return R{unex};
    }
}
```

Simplified “transform”

```
template <class F>
auto transform(F&& f) &
{
   using U = std::invoke_result_t<F, T&>;
   using R = expected<U, E>;

  if (has_val) {
      return R{std::invoke(std::forward<F>(f), val)};
   } else {
      return R{unex};
   }
}
```

Simplified “or_else”

```
template <class F>
auto or_else(F&& f) &
{
    using U = std::invoke_result_t<F, E&>; // + compile-time checks
    if (has_val) {
        return U{val};
    } else {
        return std::invoke(std::forward<F>(f), unex);
    }
}
```

Simplified “or_else”

```
template <class F>
auto or_else(F&& f) &
{
   using U = std::invoke_result_t<F, E&>; // + compile-time checks
   if (has_val) {
      return U{val};
   } else {
       return std::invoke(std::forward<F>(f), unex);
   }
}
```

Simplified “transfrom_error”

```
template <class F>
auto transform_error(F&& f) &
{
   using U = std::invoke_result_t<F, E&>;
   using R = expected<T, U>;

   if (has_val) {
       return R{val};
   } else {
       return R{std::invoke(std::forward<F>(f), unex)};
   }
}
```

Simplified “transfrom_error”

```
template <class F>
auto transform_error(F&& f) &
{
   using U = std::invoke_result_t<F, E&>;
   using R = expected<T, U>;

  if (has_val) {
     return R{val};
   } else {
       return R{std::invoke(std::forward<F>(f), unex)};
   }
}
```

Potential issues

- Extra “has_val” field
- “if” inside all operations
- Potentially: copies of value or error
- Potentially: result type changes

Compare use cases

Compare implementation

```
double sqrt(double v)
{
   if (v < 0.0) {
       throw std::invalid_argument(
           "Negative number");
   } else {
       return std::sqrt(v);
   }
}
```

```
using Result =
   std::expected<double, std::string>;

Result sqrt(double v) noexcept
{
   if (v < 0.0) {
      return std::unexpected(
         "Negative number");
   } else {
      return std::sqrt(v);
   }
}
```

Some computation

```
double calc(double v)
{
   return log(sqrt(log(v)));
}
```

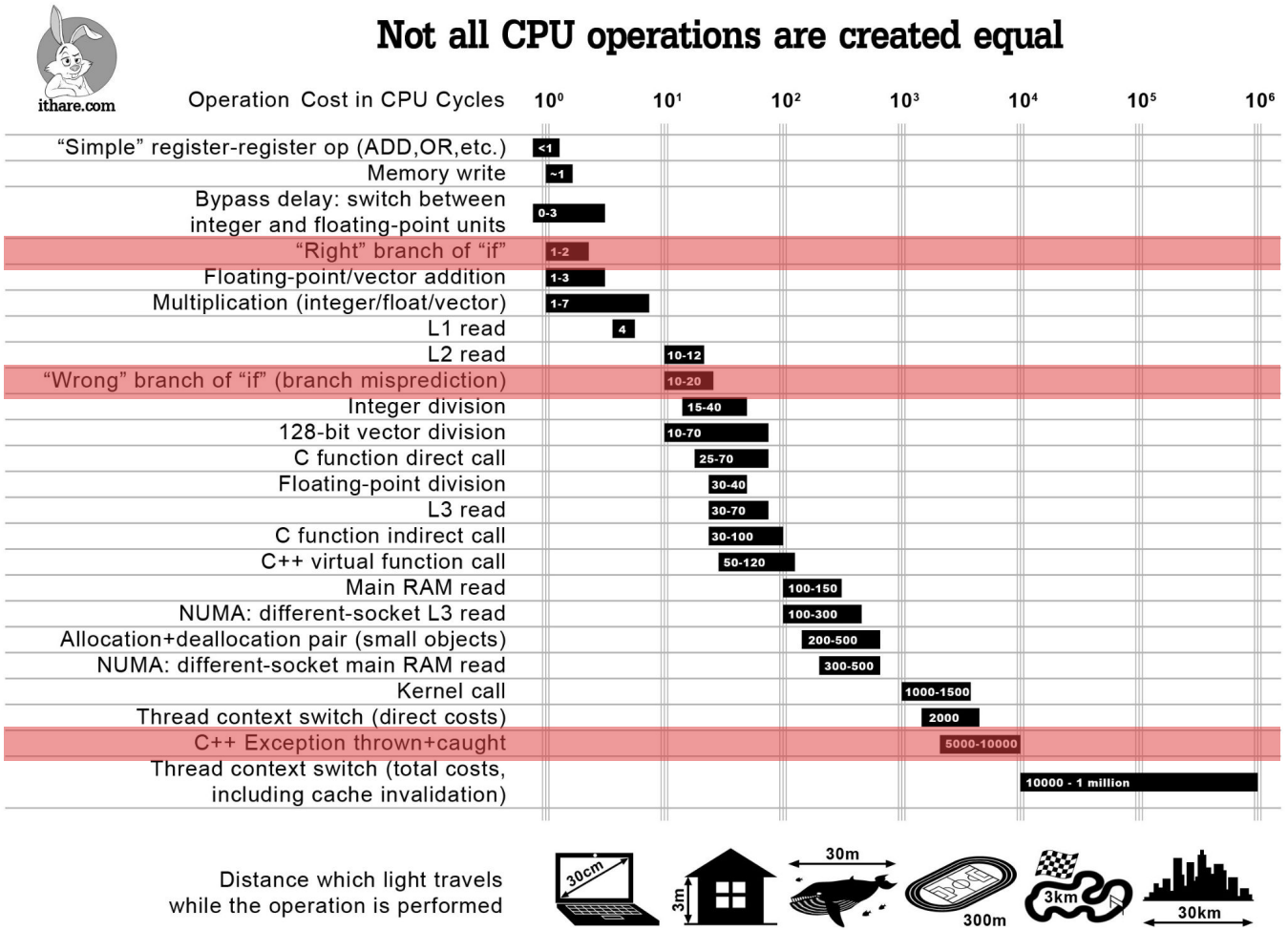
```
Result calc(double v) noexcept
{
   return log(v)
      .and_then(&sqrt)
      .and_then(&log);
}

//+ two hidden "ifs"
```

Handle errors

```
try {
   const auto result =
      with_exceptions::calc(125);
   std::println("Result: {}", result);
} catch (const std::invalid_argument& e) {
   std::println("Error: {}", e.what());
}
```

```
const auto result =
   with_expected::calc(125);
if (result.has_value()) {
   std::println(
      "Result: {}", result.value());
} else {
   std::println(
      "Error: {}", result.error());
}
```

Some reference data


ithare.com
Not all CPU operations are created equal
Operation Cost in CPU Cycles
10⁰
10¹
10²
10³
10⁴
10⁵
10⁶
"Simple" register-register op (ADD,OR,etc.)
<1
Memory write
~1
Bypass delay: switch between integer and floating-point units
0-3
"Right" branch of "if"
1-2
Floating-point/vector addition
1-3
Multiplication (integer/float/vector)
1-7
L1 read
4
L2 read
10-12
"Wrong" branch of "if" (branch misprediction)
10-20
Integer division
15-40
128-bit vector division
10-70
C function direct call
25-70
Floating-point division
30-40
L3 read
30-70
C function indirect call
30-100
C++ virtual function call
50-120
Main RAM read
100-150
NUMA: different-socket L3 read
100-300
Allocation+deallocation pair (small objects)
200-500
NUMA: different-socket main RAM read
300-500
Kernel call
1000-1500
Thread context switch (direct costs)
2000
C++ Exception thrown+caught
5000-10000
Thread context switch (total costs, including cache invalidation)
10000 - 1 million
Distance which light travels while the operation is performed
30cm
3m
30m
300m
3km
30km

Operations cost

Operation	**Cost in CPU cycles**
“Right” branch of “if”	1-2
“Wrong” branch of “if” (branch misprediction)	10-20
C++ exception thrown+caught	5000-10000

Why are exceptions so slow?

P2544R0: C++ exceptions are becoming more and more problematic

- The exceptions are allocated in dynamic memory.
- The table driven unwinder logic used by modern C++ compilers grabs a global mutex to protect the tables from concurrent changes.

Let’s do some benchmarks!

Do your own benchmarks!

- Build type (release or release with debug info mode)
- Certain toolchain
- Hardware
- Use cases
- Data types
- Number of elements/operations

Hardware config

```
CPU:
     model name      : Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
     cpu MHz         : 3600.000
     cache size      : 16384 KB

RAM (x4):
     Size: 16 GB
     Speed: 3200 MT/s
     Configured Voltage: 1.2 V
RAM timings:
     AA-RCD-RP-RAS (cycles) as DDR4-1866             13-13-13-31
     AA-RCD-RP-RAS (cycles) as DDR4-1600             11-11-11-27
```

How to compare

- Pick a relatively "light" operation
- Perform several operations in sequence (10, 100, 1000, 10000)
- Run without errors
- Run with an error (at the beginning, middle, and end)
- Check std::expected or catch an exception

Operation code

```
Data operation(Data obj)
{
   if (wrongStep(obj)) {
       throw /* ... */;
   } else {
       incOp(obj.counter);
       return obj;
   }
}
```

```
std::expected/* ... */ operation(Data obj) noexcept
{
   if (wrongStep(obj)) {
       return /* ... */;
   } else {
       incOp(obj.counter);
       return obj;
   }
}
```

```
/* ... */obj = operation(obj);
/* ... */
```

```
operation(obj).and_then(&operation).
/* ... */
```

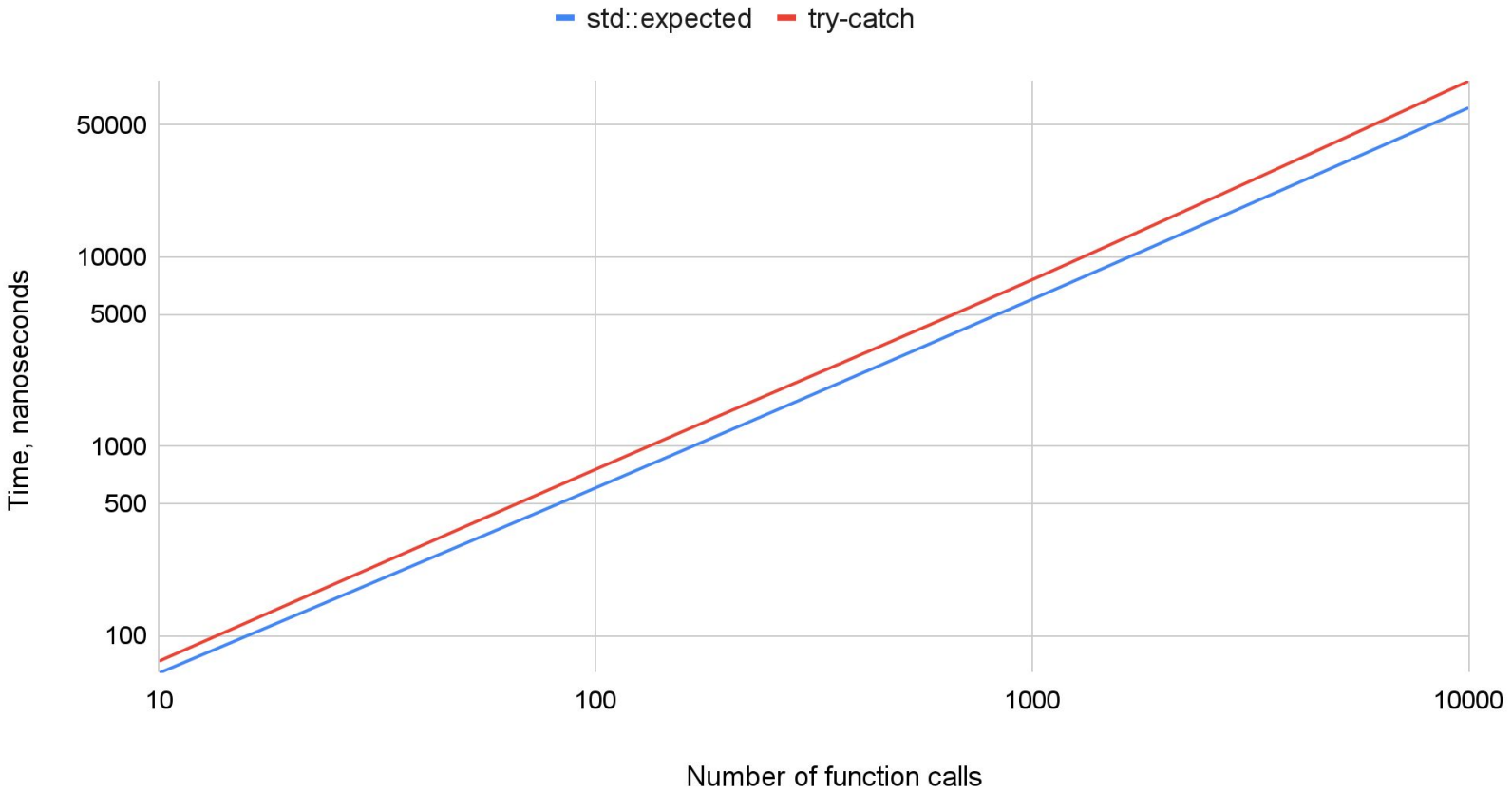
Close to declarative software design

- N monadic operations

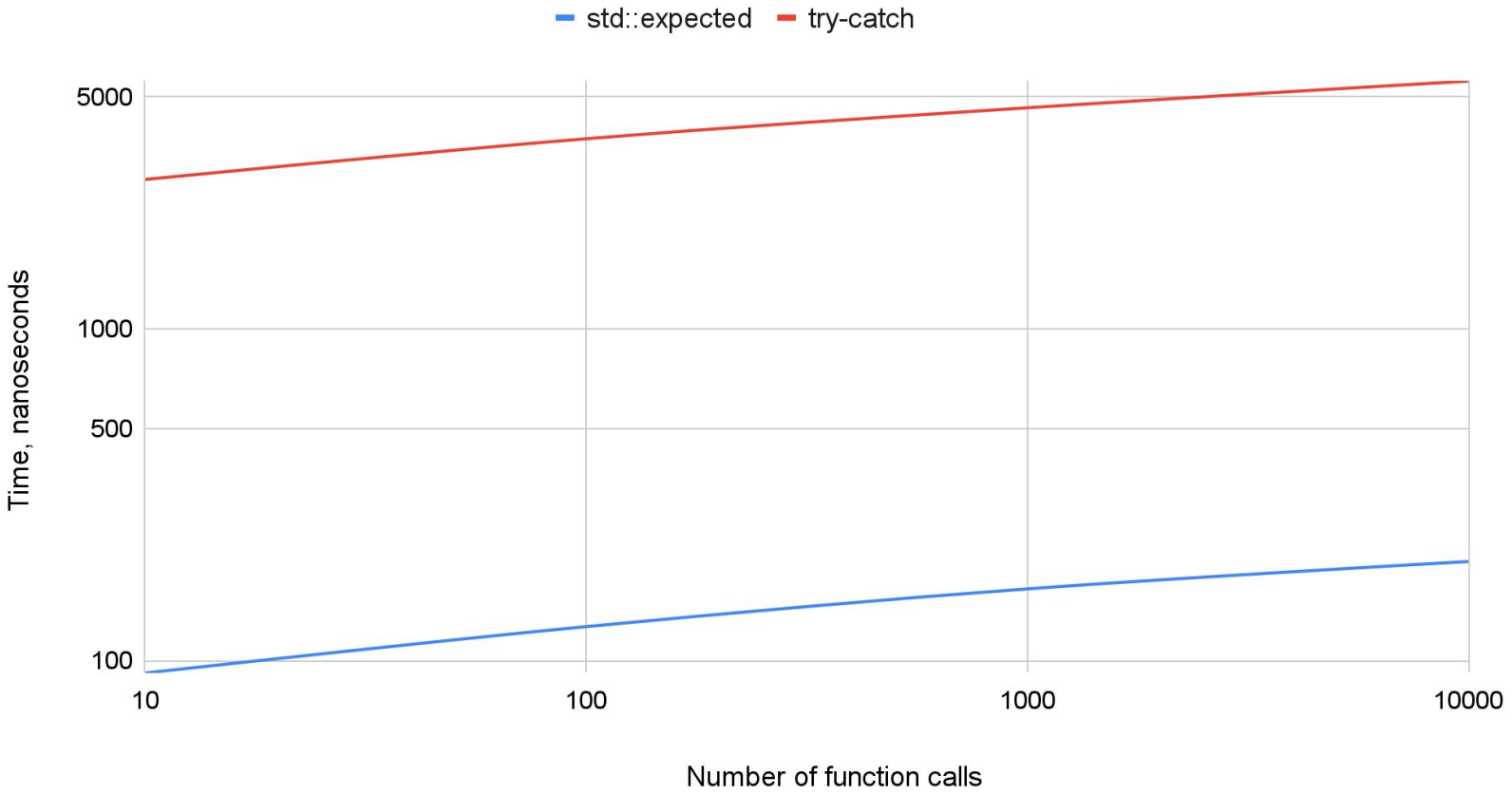
Compared to:

- 1 throw
- 1 catch

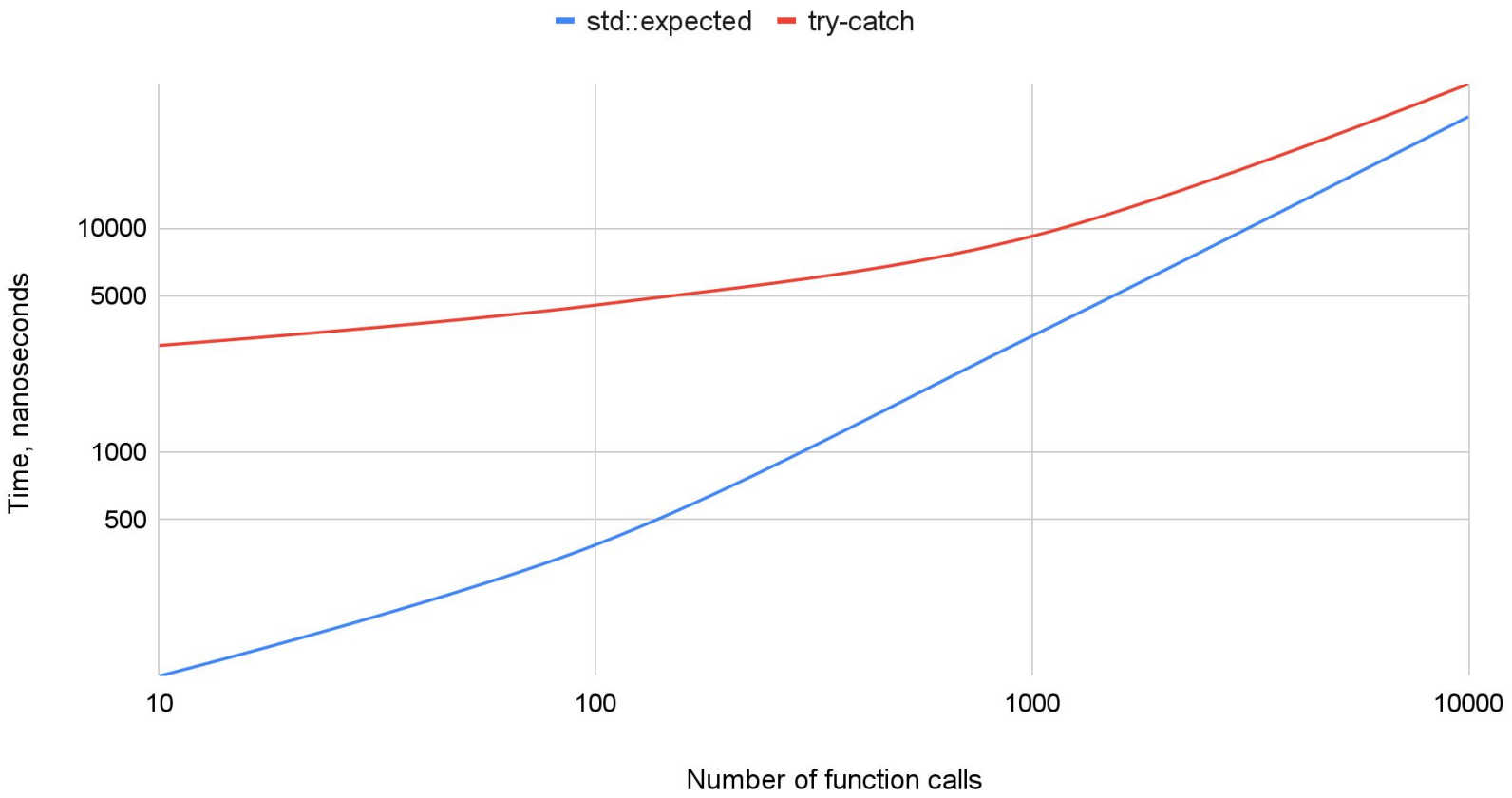
Simple run (no errors)



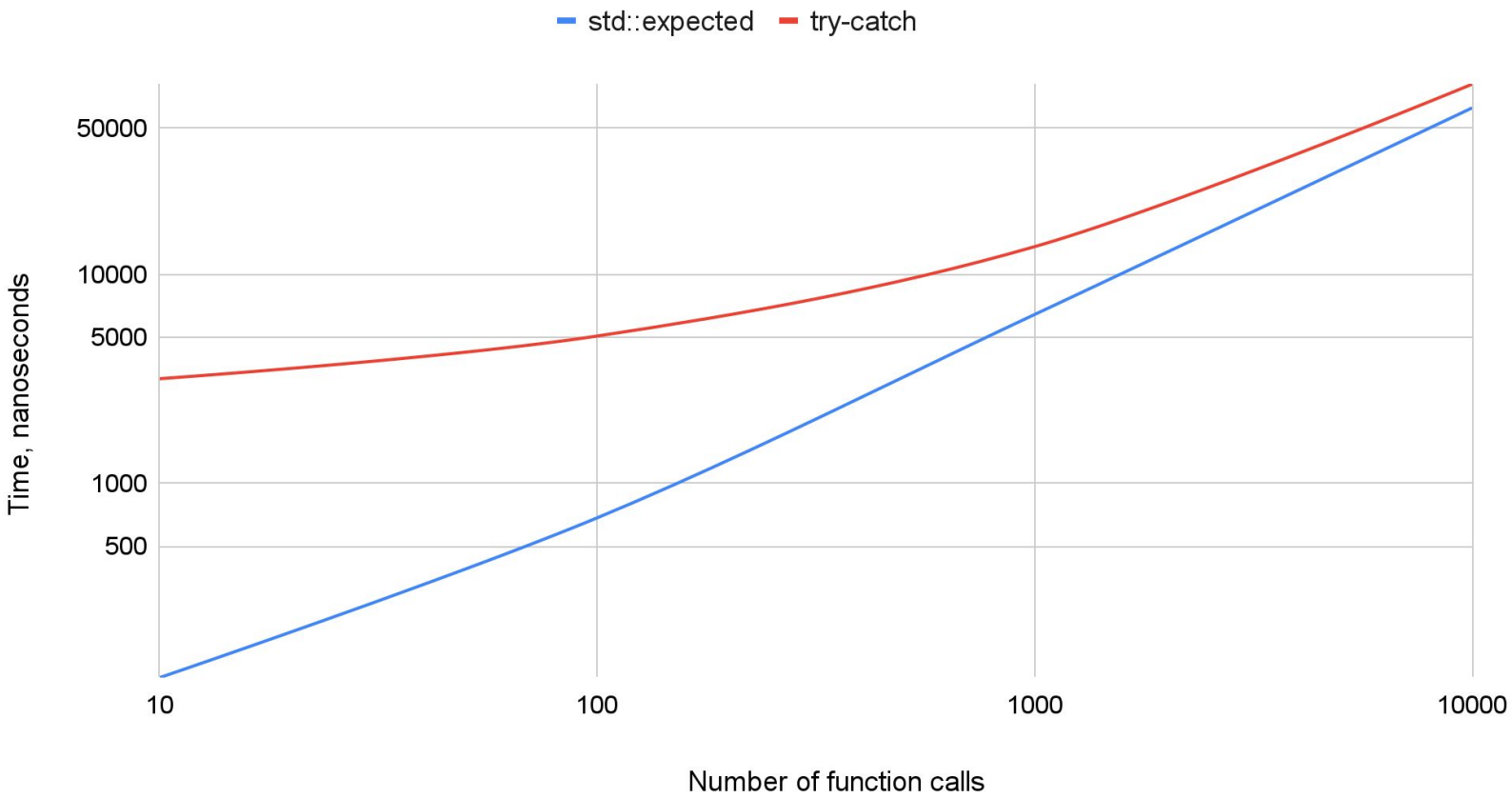
An error at the very beginning



An error in the middle



An error at the end



Conclusions

- In general, code with monadic operations is faster than with exceptions
- If an error happens at the beginning of the calculations, then code with monadic operations is significantly faster
- For a call chain of approx. less than 500 operations, monadic operations work at least two times faster than exceptions

What do we do with the extra copy/move operations?

A simple operation

```
using Result =
   std::expected<Noisy, std::string>;

Result operation()
{
   return Noisy{};
}

// ...

const auto result = operation();
```

```
Ctor: 0x7ffc99fddda7
Move ctor: 0x7ffc99fdddc8
Dtor: 0x7ffc99fddda7
Dtor: 0x7ffc99fdddc8
```

and_then/transform operation

```
std::ignore = result
   .and_then(
      [](const Noisy& noisy) → Result {
         doSmth(noisy);
         return noisy;
   })
   .and_then(
      [](const Noisy& noisy) → Result {
         doSmth(noisy);
         return noisy;
   });
```

```
Do smth: 0x7fffb9c65828
Copy ctor: 0x7fffb9c657d8
Do smth: 0x7fffb9c657d8
Copy ctor: 0x7fffb9c65800
Dtor: 0x7fffb9c65800
Dtor: 0x7fffb9c657d8
```

Can we just move?

```
std::ignore = result.and_then(
   [](Noisy&& noisy) → Result {
      doSmth(noisy);
      return noisy;
});
```

Compilation error!

Explicit move or a temporary object

```
std::ignore = std::move(result).and_then(
   [](Noisy&& noisy) → Result {
      doSmth(noisy);
      return noisy;
}).and_then(
   [](Noisy&& noisy) → Result {
      doSmth(noisy);
      return noisy;
})
```

```
Do smth: 0x7fff8da77a68
Move ctor: 0x7fff8da77a18
Do smth: 0x7fff8da77a18
Move ctor: 0x7fff8da77a40
Dtor: 0x7fff8da77a40
Dtor: 0x7fff8da77a18
```

Transfer an error

```
using AnotherResult =
   std::expected<std::string, Noisy>;

AnotherResult result = /* error */;

std::ignore = std::move(result)
   .and_then([](const std::string& o) → AnotherResult {
      return o;
   })
   .and_then([](const std::string& o) → AnotherResult {
      return o;
   })
   .and_then([](const std::string& o) → AnotherResult {
      return o;
   });
```

Move ctor: 0x7fff55df36b8

Move ctor: 0x7fff55df36e0

Move ctor: 0x7fff55df3708

Conclusions

- Copy/move operations for value/error will be performed on each step
- Highly unlikely will they be optimised out

Practical recommendations

Use "in-place" constructors

```
Result operation()
{
   return Result{std::in_place,};
}
```

```
Ctor: 0x7fffc82707b0
Dtor: 0x7fffc82707b0
```

```
Result operation()
{
   return Result{std::unexpect,};
}
```

Use smart-pointers or references

```
std::expected<std::unique_ptr<Noisy>, std::string> operation()
{
   return std::make_unique<Noisy>();
}
```

```
std::expected<std::reference_wrapper<Noisy>, std::string> operation(Noisy& noisy)
{
   doSmth(noisy);
   return std::ref(noisy);
}
```

Use simple data structures

```
struct Noisy
{
   int field1{};
   double field2{};
   std::string field3{};
   Foo field4{};
};
```

- Just a structure
- With “regular” fields only
- You can simply pass by value in many cases

Thank you!