High-level effect handlers in C++

Sam Lindley

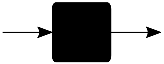
The University of Edinburgh

9th December 2022

(joint work with Dan Ghica, **Maciej Piróg**, and Marcos Maroñas Bravo)

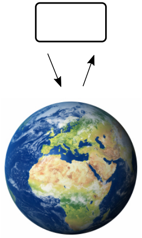
Effects

Programs as black boxes (Church-Turing model)?



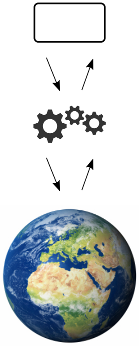
Effects

Programs must interact with their context



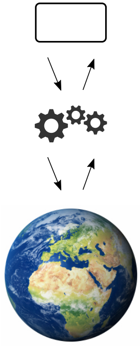
Effects

Programs must interact with their context



Effects

Programs must interact with their context

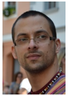


Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

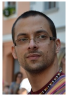
Typically ad hoc and hard-wired

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

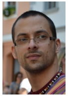
Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Composable and **customisable** user-defined interpretation of effects in general

Effect handlers

 Gordon Plotkin  Matija Pretnar

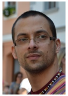
Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Composable and **customisable** user-defined interpretation of effects in general

Give programmer direct access to **context**

(c.f. resumable exceptions, monads, delimited control)

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Composable and **customisable** user-defined interpretation of effects in general

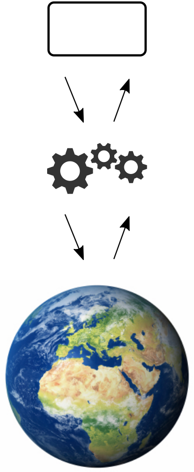
Give programmer direct access to **context**

(c.f. resumable exceptions, monads, delimited control)

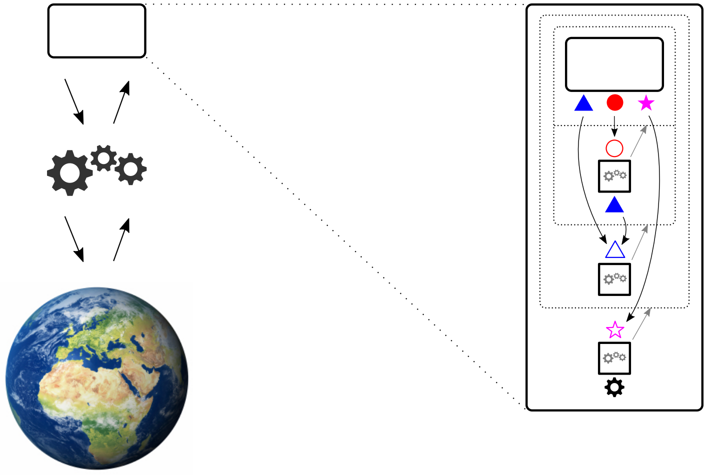
Growing industrial interest

GitHub	`semantic`	Code analysis library (> 25 million repositories)
	React	JavaScript UI library (> 2 million websites)
Uber	Pyro	Statistical inference (10% ad spend saving)

Effect handlers as composable user-defined operating systems



Effect handlers as composable user-defined operating systems



Example 1: state

Effect interface

```
template <typename S>
struct Put : eff::command<> {
  S newState;
};

template <typename S>
struct Get : eff::command<S> { };
```

Example 1: state

Effect interface

```
template <typename S>
struct Put : eff::command<> {
  S newState;
};

template <typename S>
struct Get : eff::command<S> { };
```

Wrapper functions

```
template <typename S>
void put(S s) {
  eff::invoke_command(Put<S>{{}, s});
}

template <typename S>
S get() {
  return eff::invoke_command(Get<S>{});
}
```

Example 1: state

Effect interface

```
template <typename S>
struct Put : eff::command<> {
  S newState;
};

template <typename S>
struct Get : eff::command<S> { };
```

User computation

```
int inc() {
  put(get<int>() + 1);
  return get<int>();
}
```

Wrapper functions

```
template <typename S>
void put(S s) {
  eff::invoke_command(Put<S>{{}, s});
}

template <typename S>
S get() {
  return eff::invoke_command(Get<S>{});
}
```

Example 1: state

Effect interface

```
template <typename S>
struct Put : eff::command<> {
  S newState;
};

template <typename S>
struct Get : eff::command<S> { };
```

User computation

```
int inc() {
  put(get<int>() + 1);
  return get<int>();
}
```

Wrapper functions

```
template <typename S>
void put(S s) {
  eff::invoke_command(Put<S>{{}, s});
}

template <typename S>
S get() {
  return eff::invoke_command(Get<S>{});
}
```

- Command invocation searches stack for a matching handler (c.f. exceptions)

Example 1: state

Handler

```
template <typename Answer, typename S>
class Stateful : public eff::handler<Answer, Answer, Put<S>, Get<S>> {
public:
  Stateful(S initialState) : state(initialState) { }
private:
  S state;
  Answer handle_command(Put<S> p, eff::resumption<Answer()> r) override {
    state = p.newState;
    return std::move(r).resume();
  }
  Answer handle_command(Get<S>, eff::resumption<Answer(S)> r) override {
    return std::move(r).resume(state);
  }
  Answer handle_return(Answer a) override { return a; }
};
```

Example 1: state

Handler

```
template <typename Answer, typename S>
class Stateful : public eff::handler<Answer, Answer, Put<S>, Get<S>> {
public:
  Stateful(S initialState) : state(initialState) { }
private:
  S state;
  Answer handle_command(Put<S> p, eff::resumption<Answer()> r) override {
    state = p.newState;
    return std::move(r).resume();
  }
  Answer handle_command(Get<S>, eff::resumption<Answer(S)> r) override {
    return std::move(r).resume(state);
  }
  Answer handle_return(Answer a) override { return a; }
};
```

- By default RTTI is used to safely match a command invocation with a handler

Example 1: state

```
int main() {
  std::cout << eff::handle<Stateful<int, int>>(inc, 100);
}
```

Example 1: state

```
int main() {
  std::cout << eff::handle<Stateful<int, int>>(inc, 100);
}
```

101

Example 2: lightweight threads

Effect interface and wrapper functions

```
struct Yield : eff::command<> { };
struct Fork : eff::command<> {
  std::function<void()> proc;
};

void yield() {
  eff::invoke_command(Yield{});
}
void fork(std::function<void()> proc) {
  eff::invoke_command(Fork{{}, proc});
}
```

Example 2: lightweight threads

Effect interface and wrapper functions

```
struct Yield : eff::command<> { };
struct Fork : eff::command<> {
  std::function<void()> proc;
};

void yield() {
  eff::invoke_command(Yield{});
}
void fork(std::function<void()> proc) {
  eff::invoke_command(Fork{{}, proc});
}
```

User computation

```
void mainThread() {
  std::cout << "M1 "; fork([=]() {std::cout << "A1 "; yield(); std::cout << "A2 "});
  std::cout << "M2 "; fork([=]() {std::cout << "B1 "; yield(); std::cout << "B2 "});
  std::cout << "M3 ";
}
```

Example 2: lightweight threads

Handler

```
using Res = eff::resumption<void()>;
class Scheduler : public eff::handler<void, void, Yield, Fork> {
public:
  static void Start(std::function<void()> f) {
    queue.push_back(eff::wrap<Scheduler>(f));
    while (!queue.empty()) {
      Res resumption = std::move(queue.front());
      queue.pop_front();
      std::move(resumption).resume();
    }
  }
private:
  static std::list<Res> queue;
  void handle_command(Yield, Res r) override {
    queue.push_back(std::move(r));
  }
  void handle_command(Fork f, Res r) override {
    queue.push_back(std::move(r));
    queue.push_back(eff::wrap<Scheduler>(f.proc));
  }
  void handle_return() override { }
};
```

Example 2: lightweight threads

```
int main() {
  Scheduler::Start(mainThread);
}
```

Example 2: lightweight threads

```
int main() {
  Scheduler::Start(mainThread);
}
```

M1 M2 M3 A1 B1 A2 B2

Example 3: flat handlers and plain commands

Handler

```
template <typename Answer, typename S>
class Stateful : public eff::flat_handler<Answer, eff::plain<Put<S>>, eff::plain<Get<S>>> {
public:
  Stateful(S initialState) : state(initialState) { }
private:
  S state;
  void handle_command(Put<S> p) override {
    state = p.newState;
  }
  S handle_command(Get<S>) override {
    return state;
  }
};
```

Example 3: flat handlers and plain commands

Handler

```
template <typename Answer, typename S>
class Stateful : public eff::flat_handler<Answer, eff::plain<Put<S>>, eff::plain<Get<S>>> {
public:
  Stateful(S initialState) : state(initialState) { }
private:
  S state;
  void handle_command(Put<S> p) override {
    state = p.newState;
  }
  S handle_command(Get<S>) override {
    return state;
  }
};
```

- A flat handler automatically includes an identity return clause
- A plain command clause automatically invokes the resumption in tail position

Example 4: lightweight threads with a kill command

```
struct Kill : eff::command<> { };

class Scheduler : public eff::handler<void, void, Yield, Fork, eff::no_resume<Kill>> {
  // ...
  void handle_command(Kill) override { }
};
```

Example 4: lightweight threads with a kill command

```
struct Kill : eff::command<> { };

class Scheduler : public eff::handler<void, void, Yield, Fork, eff::no_resume<Kill>> {
  // ...
  void handle_command(Kill) override { }
};
```

- A no resume command clause cannot invoke the resumption

Example 5: actors

Effect interface and wrappers

```
using Pid = std::shared_ptr<std::queue<std::any>>;
struct Spawn : eff::command<> {
  std::function<void()> body;
};
struct Self : eff::command<Pid> { };
struct Send : eff::command<> {
  Pid p;
  std::any msg;
};
struct Receive : eff::command<std::any> { };
Pid spawn(std::function<void()> body) {
  return eff::invoke_command(Spawn({}, body));
}
Pid self() {
  return eff::invoke_command(Self{});
}
template <typename T> void send(Pid p, T msg) {
  eff::invoke_command(Send({}, p, msg));
}
template <typename T> T receive() {
  return std::any_cast<T>(eff::invoke_command(Receive{}));
}
```

Example 5: actors

Handler

```
template <typename Answer>
class Act : public eff::flat_handler<Answer, eff::plain<Spawn>, eff::plain<Self>,
      eff::plain<Send>, eff::plain<Receive>> {
  Pid handle_command(Self) override {
    return get<Pid>();
  }
  Pid handle_command(Spawn s) override {
    auto mailbox = std::make_shared<std::queue<std::any>>();
    fork([=]() { eff::handle<Stateful<void, Pid>>(s.body, mailbox); });
    return mailbox;
  }
  void handle_command(Send s) override {
      s.p->push(s.msg);
  }
  std::any handle_command(Receive) override {
    auto mailbox = get<Pid>();
    while (mailbox->empty()) { yield(); }
    auto msg = mailbox->front();
    mailbox->pop();
    return msg;
  }
}
```

Example 6: async/await

Futures

```
struct GenericFuture {
  std::vector<eff::resumption<void()>> awaiting;
};
template <typename T>
class Future : public GenericFuture {
  std::optional<T> value;
  ...
};
```

Example 6: async/await

Futures

```
struct GenericFuture {
  std::vector<eff::resumption<void()>> awaiting;
};
template <typename T>
class Future : public GenericFuture {
  std::optional<T> value;
  ...
};
```

Effect interface and wrapper

```
struct Await : eff::command<> {
  GenericFuture* future;
};
template <typename T>
T await(Future<T>* future) {
  if (*future) { return *(future->value); }
  eff::invoke_command(Await{{}, future});  // Suspend until the value is ready
  return future->Value();
}
```

Example 6: async/await

Futures

```
struct GenericFuture {
  std::vector<eff::resumption<void()>> awaiting;
};
template <typename T>
class Future : public GenericFuture {
  std::optional<T> value;
  ...
};
```

Effect interface and wrapper

```
struct Await : eff::command<> {
  GenericFuture* future;
};
template <typename T>
T await(Future<T>* future) {
  if (*future) { return *(future->value); }
  eff::invoke_command(Await{{}, future});  // Suspend until the value is ready
  return future->Value();
}
```

- Command is monomorphic but its wrapper is polymorphic

Example 7: generators

Effect interface and wrapper

```
template <typename T>
struct Yield : eff::command<> {
  T value;
};
template <typename T>
void yield(int64_t label, T x) {
  eff::static_invoke_command(label, Yield<T>{{}, x});
}
```

Example 7: generators

Effect interface and wrapper

```
template <typename T>
struct Yield : eff::command<> {
  T value;
};
template <typename T>
void yield(int64_t label, T x) {
  eff::static_invoke_command(label, Yield<T>{{}, x});
}
```

- The label identifies a handler by name
- Static command invocation asserts that we know the exact type of the handler

Example 7: generators

```cpp
template <typename T> struct GenState;
template <typename T> using Result = std::optional<GenState<T>>;

template <typename T>
struct GenState {
  T value;
  eff::resumption<Result<T>()> resumption;
};

template <typename T>
class GeneratorHandler : public eff::handler<Result<T>, void, Yield<T>> {
  Result<T> handle_command(Yield<T> y, eff::resumption<Result<T>()> r) override {
    return GenState<T>{y.value, std::move(r)};
  }
  Result<T> handle_return() override {
    return {};
  }
};
```

Example 7: generators

```
template <typename T> class Generator {
public:
  Generator(std::function<void(std::function<void(T)>)> f) {
    auto label = eff::fresh_label();
    result = eff::handle<GeneratorHandler<T>>(label, [f, label](){
      f([label](T x) { yield<T>(label, x); });
    });
  }
  T Value() const {
    if (!result) { throw std::out_of_range("Generator::Value"); }
    return result.value().value;
  }
  bool Next() {
    if (!result) { throw std::out_of_range("Generator::Next"); }
    result = std::move(result->resumption).resume();
    return result.has_value();
  }
  explicit operator bool() const {
    return result.has_value();
  }
private:
  Result<T> result = {};
};
```

Example 7: generators

Using generators

```
int main()
{
  Generator<int> naturals([](auto yield) {
    int i = 1;
    while (true) { yield(i++); }
  });

  for (int i = 0; i < 100; i++) {
    std::cout << naturals.Value() << std::endl;
    naturals.Next();
  }
}
```

Example 7: generators

Using generators

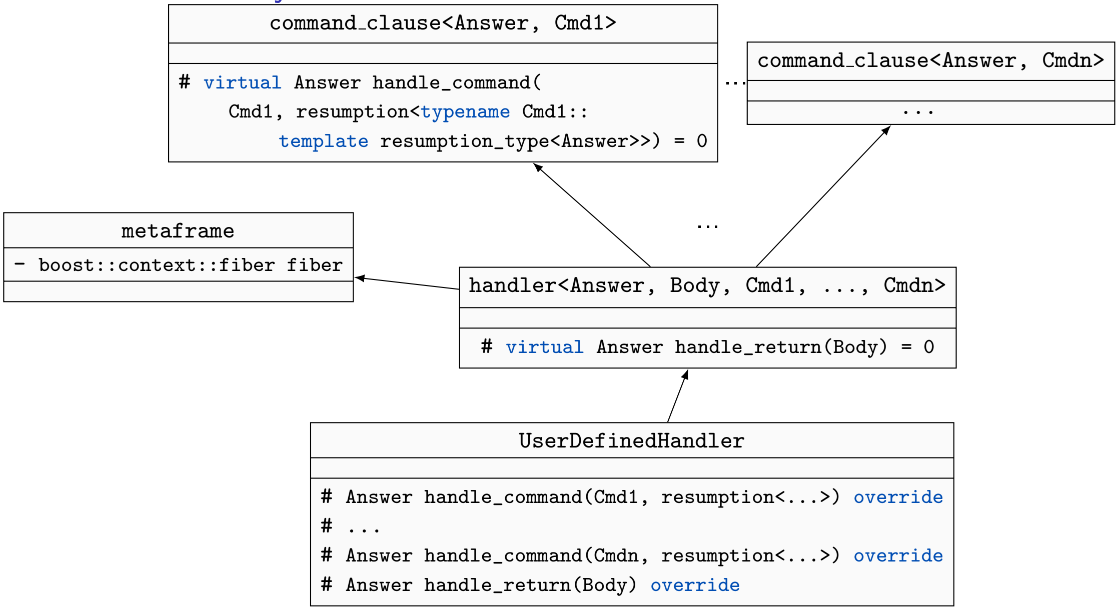
```
int main()
{
  Generator<int> naturals([](auto yield) {
    int i = 1;
    while (true) { yield(i++); }
  });

  for (int i = 0; i < 100; i++) {
    std::cout << naturals.Value() << std::endl;
    naturals.Next();
  }
}
```

- Effects and handlers entirely encapsulated by the generator class
- Named handler used internally ensures that different generators do not interfere

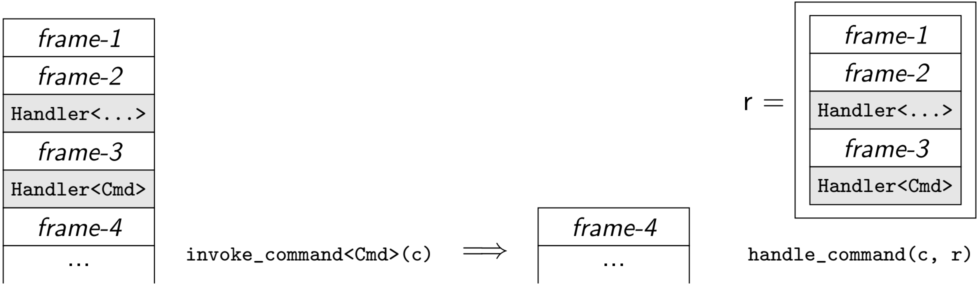
Design summary

- affine deep handlers
- commands (operations) are classes
- handlers are classes parameterised by commands they handle
- both anonymous and named handlers (as in previous talk)
- flat handlers (identity return clause)
- plain handler clauses (tail-resumptive)
- no resume handler clauses (exceptions)

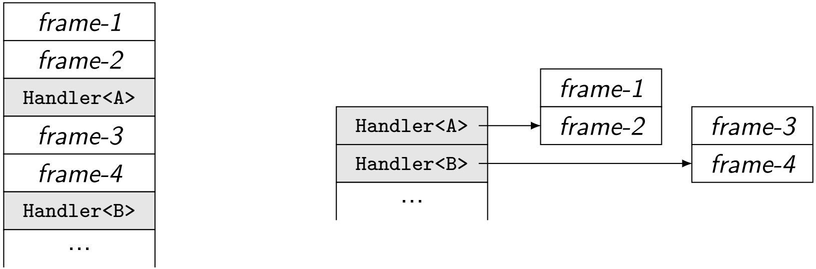
Inheritance hierarchy for handlers



Invoking a command



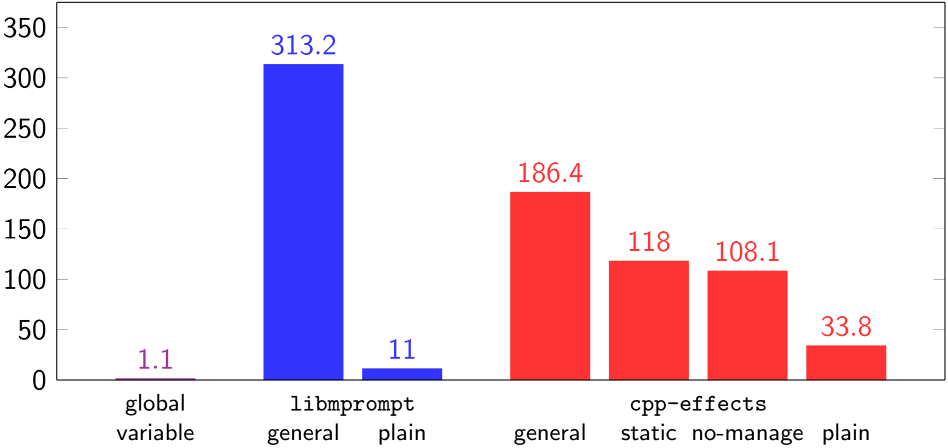
Flat stack versus nested stack with pointers to stacklets (fibers)



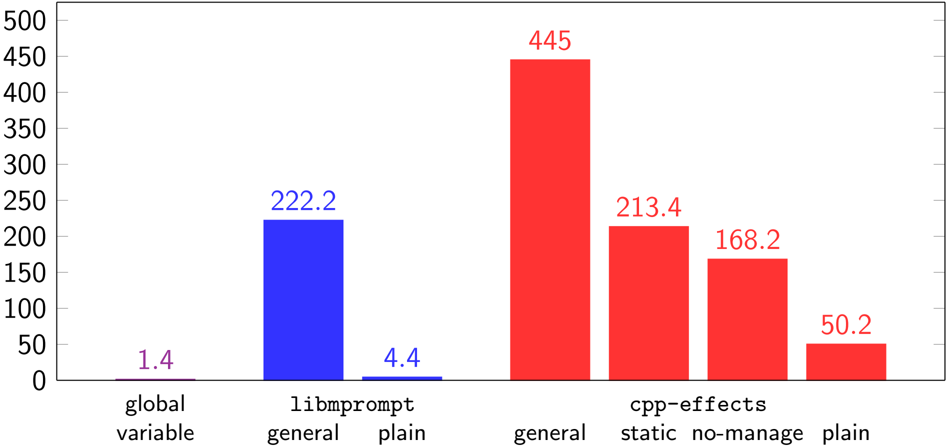
Implementation

- backend — `boost.context` fibers
- nested stack (one stacklet / fiber per handler)
- pre-allocation of resumptions
- reference counting
- move constructors as a crude alternative to substructural types
- no manage optimisation (when handler and resumptions do not escape)

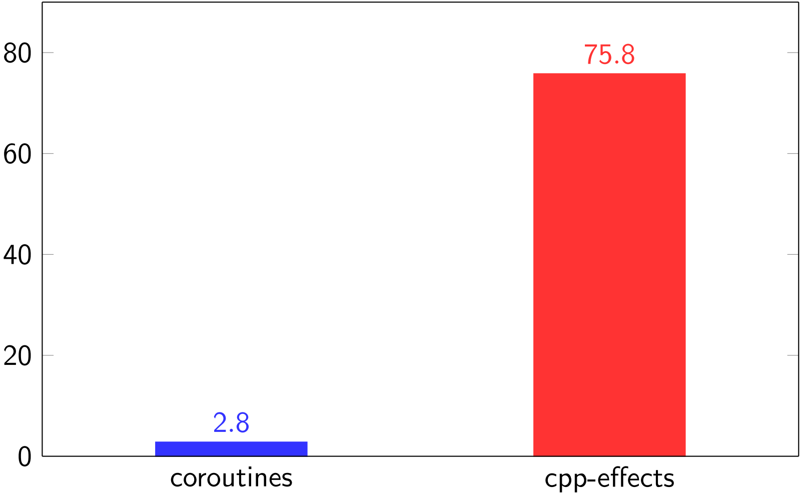
State using Clang natively (average time per iteration in ns)



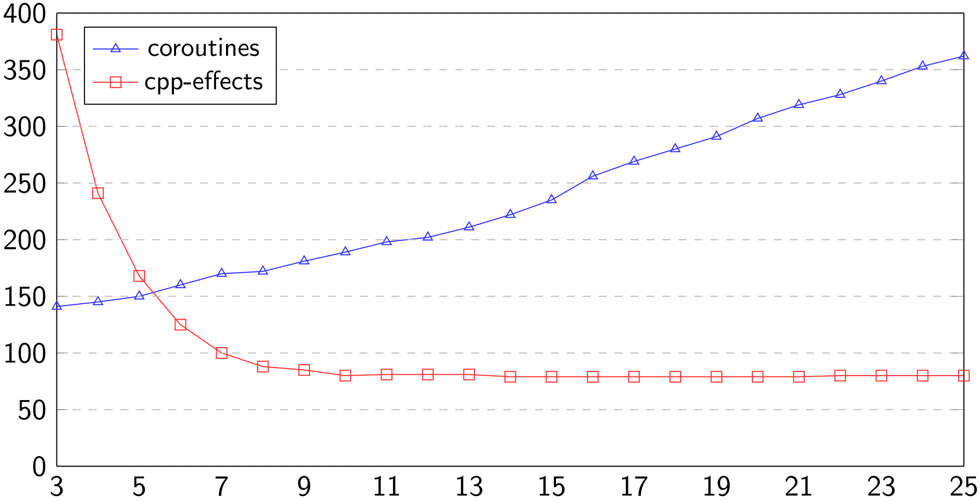
State using GCC in Docker (average time per iteration in ns)



Generating a number (in ns)



Recursive tree traversal (ns per node)



https://github.com/maciejpirog/cpp-effects