ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/370198510

Template Meta-State Machines, Shannon, Madness...

Presentation · April 2023

DOI: 10.13140/RG.2.2.22841.65125

CITATIONS
0

READS
76

1 author:

Jason McGuiness
Count-Zero Limited

20 PUBLICATIONS **6** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project DSELs for Parallel Architectures View project

Project High performance techniques for HFT & low-latency trading. View project

All content following this page was uploaded by Jason McGuiness on 22 April 2023.

The user has requested enhancement of the downloaded file.

Template Meta-State Machines, Shannon, Madness...

J.M.M^c^Guiness[1]

[1]accuconf2023@hussar.me.uk

ACCU Conference, Bristol, 2023

Version: 73fd1

Outline

A very simple piece of code...

Why is this instruction sequence so interesting?

GOTO considered harmful[1]:

```
CALL QWORD PTR [BASE+STRIDE*STATE]
# STATE input, branch-target is Transition.
```

- Because it is at the heart of a state machine.
  - This may be generated as an `if-else` chain.
  - But this could impact the branch-predictor.
- The instruction timing for a computed-`GOTO` is excellent in modern super-scalar architectures vs mis-prediction of a branch:
  - 1-2 clock cycles vs $O(20)$ estimated from the depth of the pipeline ≈ 20 stages[2], the cost of a pipeline restart.

How to compute the STATE?

- We shall leave aside the computation of the `BASE` and the `STRIDE` as they are more easy to compute.
- The `STATE` may be more complex: because this may be random, not sequential.
    - Thus a perfect, preferably minimal hash should be generated.
    - *Information Entropy, elucidated by Shannon, means there is no general solution...*
    - Persevere: generation of the hash shall be attempted despite those issues...

Problem statement.

1. We shall generate computed-`GOTO`s.
2. No amount of effort shall be sacrificed to attain point 1, above.

Methodology for the investigation.

1. We need to identify a suitable code-base for this:
    1. In this case I chose the FIX-to-MIT translator[3] as the contained meta state-machines are suitable.
    (Upon which I have previously much presented.)
2. Modifications to the code-base shall be made to permit comparative testing.
    1. Computed-`GOTO` vs. a "naïve" meta state-machine that has been implemented with `if-else` chains.
    2. The results shall be statistically significant.
    3. *Using computed-`GOTO`s shall have ramifications.*
        - Which shall be discussed, later.

Results of the investigation.

1. Histograms of the `if-else` vs computed-`GOTO` performance shall be presented.
   1. And discussion of these in detail...
2. A detailed, subjective review of the code that had to be added:
   1. including reviewing the comprehensibility, maintainability and impact on compile-time.
3. Finally, conclusions shall be presented drawn from these analyses.

A low-latency, HFT FIX-to-MIT/BIT translator?

- In trading systems one has this simplified process:
  - Messages sent from a client to an exchange.
    - This involves a state machine.
  - Messages sent from an exchange back to a client.
    - Likewise this includes a state machine.
- Both state machines are on the hot path.
- My numerous previous presentations regarding low-latency optimisations and HFT [3] in C++ motivated this investigation regarding writing the *fastest* meta state-machine possible...
  - To such an extreme that there would *no longer* be statistically significant performance differences!
- Why, oh why?
  - The adventure of writing the code to implement this was beyond anything one may ever wish to sensibly do...

BEWARE!

- This is not a talk about how to optimise code that has not yet been optimised!

  Premature optimisation is the root of all evil[4]!

  - *This talk certainly verges on that...*

- Ensure that one has run one's preferred profiler, etc, beforehand.
  - Heed Amdahl's Law[5].
- Code should be: comprehensible, maintainable & compile reasonably quickly.
  - The code presented here, in my opinion, verges on failing all of those requirements!

The impact of the runtime STATE.

`constrained_override_type`

- The target address is all that is known about the destination object, the `Transition`.
- Specifically, run-time `STATE` shall cause a jump to the related `Transition::process(...)`:
  - in a *generic* manner if a *generic* state machine is to be used,
  - effectively that `STATE` should be an index operation into a collection of `Transitions`,
  - generality shall be provided by generating wrappers for `Transition::process(...)`,
    - the types of the parameters to `process(...)` must be recovered to permit overloading resolution,
  - the state machine that was implemented was based upon the Boost "Meta State Machine"[6].

The transitions - BASE & STRIDE.

(Inspired by discussions with Vladimir Arnost.)
`unordered_tuple`

We shall need a collection of `Transitions` into which the `STATEs` index.

- It was chosen to implement this as a buffer of `alignas(STRIDE) std::array<std::byte,...>` into which the `Transitions` shall be placement-`new`'d.
- Thus the `BASE` & `STRIDE` may be computed at compile-time.
- This collection will require a suitable `operator[]` indexed by the input `STATE`.
    - The `STATEs` shall need hashing... More later...

Beware the angle-bracket, my son...

Much has been written on the pitfalls relating to unconstrained abuse of template meta-programming in C++, amongst such luminaries as:

- "Henney's Hypothesis"[7]:

  > *For each additional template parameter, the potential number of users is halved.*

  - I have many of these, in some cases unbounded sets...
- "Template Metaprogramming Made Easy (Huh?)" by Bartosz Milewski[8]:

  > ... *Big part of it is that C++ templates are rather ill suited for metaprogramming, to put it mildly.* ...

This presentation contains many such sins...

- *You have been warned...*

Schematic design of computed-GOTO meta state-machine.

computed-GOTO `meta_state_machine`

`StatesTransitionTable` = `unordered_tuple` provides `operator[](...)` using `perfect_hash` to call (one of the) `constrained_override_type::abstract_base_type::process(...)`

row contains a `constrained_override_type::final_concrete_type` which wraps a call to `abstract_base_type::process(...)`

`concrete_type` wraps a `Transition::signatures_types` and provides the override for `process(...)`, calls the contained `Transition::process(...)`

`concrete_type` inherits recursively for each `Transition::signatures_types` & provides the...

...

Terminal `concrete_type` inherits from `abstract_base_type` & provides the...

`abstract_base_type` inherits from `abstract_type_unroller`

`abstract_type_unroller` wraps a `Transition::signatures_types` & provides the `virtual process(...)=0`

`abstract_type_unroller` inherits recursively for each `Transtition::signatures_types` & provides the...

...

Terminal `abstract_type_unroller` inherits from `ultimate_base_type` & provides the...

...

More rows...

The unordered_tuple.

The purpose of this class is to contain all of the transitions:

Schematic of unordered_tuple.

```
template<class TransitionBase, class Hasher, class... Transitions>
class unordered_tuple {
public:
    constexpr size_t max_size = ...;
    constexpr size_t stride = ...;
    TransitionBase &operator[](state_type state) {
        auto base = wrapped_transitions.data();
        auto offset = Hasher(state)*stride;
        return reinterpret_cast<TransitionBase &>(base+offset);
    }
private:
    alignas(stride)
    std::array<std::byte, max_size> wrapped_transitions{};
};
```

Notes: `unordered_tuple`.

`TransitionBase`: base class common to all `Transitions`. Shall be supplied by `constrained_override_type` as `abstract_base_type`.

`Hasher`: hashing algorithm applied to the set of `STATE`s contained in the `Transitions`. More details later...

`Transitions`: The set of `Transitions` to be contained. Each of which has a copy of the `STATE` as a value. Each `Transition` wrapped by a `constrained_override_type` from the `meta_state_machine`.

`wrapped_transitions`: The set of `Transitions` is placement-`new`'d into a buffer of suitable alignment & size. Placed with a stride of `STRIDE`.

Why constrained_override_type?

Warning it is just syntactic sugar!

Consider a naïve if-else implementation of a state machine:

example if-else implementation.

```
state_type msm::process(state_type state, Params... &&p) {
    if (state == NEW_ORDER) {
        return new_order_transition.process(p...);
    } else if (state == ORDER_CANCEL) {
...
}
```

vs:

example computed-GOTO implementation.

```
state_type msm::process(state_type state, Params... &&p) {
    return transitions[state].process(p...);
}
```

Discussion of why `constrained_override_type`?

- In the `if-else` implementation we know the exact transition to call, thus the type and number of parameters at compile-time.
- Whereas in the computed-`GOTO` implementation some form of interface class must be called, as the `STATE` is only known at run-time.
  - Therefore the `process()` method must be provided by some form of base class.
  - But the parameters to it may vary or be of disparate types...!
  - C++ does not support virtual, template member-functions! One cannot write:

virtual, templated member-functions - no!

```
struct foo {
    template<class... Params> virtual void bar(Params... p);
};
```

Purposes: `constrained_override_type`.

1. Supply a suitable abstract base class for each of the `Transitions` with suitable declarations of pure-virtual `process()` methods obtained from each particular `Transition`.
2. From those base classes, aggregate them in an inheritance chain to compute a `TransitionBase` abstract base class for the `unordered_tuple` `TransitionBase` class.
3. Create wrappers for each `Transition` so that the wrapper may inherit from the abstract base class so generated and the `Transition` it will wrap. These wrappers may then be passed to the `unordered_tuple`.
4. In reality this is overly general for these needs, but I wanted to generalise... Oops...

Notes: `constrained_override_type`.

As C++ does not yet support reflection, the previous points 1 & 3 imply that each `Transition` must supply some kind of type that lists the return type and parameter types for each declaration of `process()` within each `Transition`, termed `ProcessFns` in the following examples.

Components: constrained_override_type.

We need to generate the pure-virtual methods with the correct declarations for the ProcessFns of the Transition:

Schematic of abstract_type_unroller - base case.

```
template<class RetT, class... Params>
struct abstract_type_unroller<member_function_type<RetT, Params...>>
: ultimate_base_type {
    virtual RetT process(Params ...&&p) const noexcept(false) = 0;
    virtual RetT process(Params ...&&p) noexcept(false) = 0;
};
```

Components: constrained_override_type.

Now generate the remaining pure-virtual methods for the ProcessFns of the Transition:

Schematic of abstract_type_unroller - generate the rest.

```
template<class RetT, class... Params, class... ProcessFns>
... abstract_type_unroller<member_function_type<RetT, Params...>,
    ProcessFns...> : abstract_type_unroller<ProcessFns...> {
    using base_type::base_type;
    using base_type::process;
    virtual RetT process(Params ...&&p) const noexcept(false) = 0;
    virtual RetT process(Params ...&&p) noexcept(false) = 0;
};
```

Declare a type that shall be the abstract base class containing all the pure-virtual methods:

```
using abstract_base_type = abstract_type_unroller<Transitions...>;
using TransitionBase = abstract_base_type;
```

Components: constrained_override_type.

We now need a class that has an is-a Transition for which the ProcessFns may be called:

Schematic of concrete_type - base case.

```
template<class abstract_base_type, class Transition, class RetT,
class... Params>
struct concrete_type<abstract_base_type, Transition,
member_function_type<RetT, Params...>>
: Transition, abstract_base_type {
    template<class... Args>
    explicit concrete_type(Args &&...args) noexcept(...)
    : Transition(std::forward<Args>(args)...), abstract_base_type() {
    }
    RetT process(Params &&...p) const noexcept(...) override {
        return this->Transition::process(std::forward<Params>(p)...);
    }
    RetT process(Params &&...p) noexcept(...) override {
        return this->Transition::process(std::forward<Params>(p)...);
    }
};
```

Components 1/2: constrained_override_type.

Now generate the rest for all of the Transitions:

Schematic of concrete_type - generate the rest.

```
template<class abstract_base_type, class Transition, class RetT,
class... Params, class... ProcessFns>
struct concrete_type<abstract_base_type,
member_function_type<RetT, Params...>, ProcessFns...>
: concrete_type<abstract_base_type, Transition, ProcessFns...> {
    using
base_t = concrete_type<abstract_base_type, Transition, ProcessFns...>;
    using base_t::base_t;
    using base_t::process;
    RetT process(Params &&...p) const noexcept(...) override {
        return this->Transition::process(std::forward<Params>(p)...);
    }
    RetT process(Params &&...p) noexcept(...) override {
        return this->Transition::process(std::forward<Params>(p)...);
    }
};
```

Components 2/2: constrained_override_type.

Declare an equivalent type that shall be the concrete class containing all the overridden wrappers for the `Transition::process()` methods:

```
template<class T>
struct finalizer final : T {
    using T::T;
};
using final_concrete_type =
finalizer<concrete_type<abstract_base_type, Transitions...>>;
```

Compare to `abstract_base_type`, above.

Reflections 1/2: constrained_override_type.

The `final_concrete_type` may be used inside `unordered_tuple` to supply the `process()` methods that should be called according to the `STATE`s in a type-safe manner.

- Ultimately, the generated `concrete_type` has been inherited-from with the `final` keyword to remove all virtual-function calls by `finalizer<T>`. The fact that all of the types are fully defined before use is also vital for this.
- We have now recovered the type information for the return and parameter types for each of the `Transition::process()`'s, so they may be called safely.

Reflections 2/2: `constrained_override_type`.

- This requires that each `Transition` defines `signatures_types (ProcessFns)` that contains the return and parameter types for each `Transition::process()` declared:

```
using signatures_types = std::pair<
    return_type,    // N.B. all process() methods return the same type.
    std::tuple<
        std::tuple<first_parameter_for_first_overload, ...>,
        std::tuple<first_parameter_for_second_overload, ...>
    >
>;
```

Introducing schematic of: meta_state_machine.

```
template<class StateTransitionTable>
struct machine {
    using states_type = typename StateTransitionTable::states_type;
    template<class... Args> machine(Args &&...args) noexcept(...);
    template<class... Params> states_type
    process(states_type state, Params &&...p) const noexcept(...) {
        return table[state].process(std::forward<Params>(p)...);
    }
    template<class... Params> states_type
    process(states_type state, Params &&...p) noexcept(...) {
        return table[state].process(std::forward<Params>(p)...);
    }
    StateTransitionTable table;
};
```

- The chief item of interest is the `StateTransitionTable`: it combines the `unordered_tuple` & `constrained_override_type`.
- U.B. would happen if the wrong `Transition` were called via a valid, but mis-computed hash of a `STATE`...

Schematic details of: StateTransitionTable.

```
template<class...  Transitions>
using StateTransitionTable = unordered_tuple<
    states_type,
    wrapped_first_row_t::abstract_base_type,
    perfect_hash<get_state_as_hash<Transitions>::value...>,
    wrapped_first_row_t::template final_type<Transition>,
    detuple_make_rows<
        Transitions,
        make_row_wrappers_t<Transitions::signatures_types...>::type
    >::type...
>;
```

Notes for: StateTransitionTable.

abstract_base_type: introduced earlier in the discussion regarding constrained_override_type.

perfect_hash: the generated hasher that shall be discussed in the next section...

make_row_wrappers_t: a meta-function that extracts the details from signatures_types for use by...

detuple_make_rows: wraps the ProcessFns in each Transition in the StateTransitionTable with a constrained_override_type:

```
template<Transition>
using detuple_make_row =
constrained_override_type::result_type<Transition>;
```

Schematic exchange to client: `meta_state_machine`.

```
using StateTransitionTable = rows<
    row<
        ServerHeartbeat_t::static_type,
        just_send_to_client<ClientHeartbeat_t>,    // Example
Transition.
        ClientHeartbeat_t::static_type
    >,
    row<
        ExecutionReport_t::static_type,
        ExecutionReportResponse,    // Example Transition.
        ExecutionReportResponse::exit_values
    >,
    ...,
    row<
        MsgTypes_t::MatchAll,
        send_specified_msg<Reject_t>,    // Example Transition.
        Reject_t::static_type
    >
>;
```

Note that the definitions of `rows` and `row` have been omitted for brevity, see [3] for more details.

The hashing algorithm: requirements.

- *We know the discreet set of* `STATE`*s a posteriori: it is part of the MIT specifications[9].*
- A perfect hash is required, preferably minimal.
- The generated hash-function must have as few instructions as possible.
  - Existing hash-generators such as `gperf`[10] do not fit all of the above criteria.
    - The generated hash functions are many lines of assembly, we require $\ll 20$ instructions.
- We need to create a generator for this highly-optimised hash algorithm.
  - Minor relaxation: the hashing need not be stable in the sense of order of the `STATE`s in the domain.
- Generated hash-function shall be checked to ensure no collisions.

The hashing algorithm: requirements.

- *We know the discreet set of* `STATE`*s a posteriori: it is part of the MIT specifications[9].*
- A perfect hash is required, preferably minimal.
- The generated hash-function must have as few instructions as possible.
  - Existing hash-generators such as `gperf`[10] do not fit all of the above criteria.
    - The generated hash functions are many lines of assembly, we require $\ll 20$ instructions.
- We need to create a generator for this highly-optimised hash algorithm.
  - Minor relaxation: the hashing need not be stable in the sense of order of the `STATE`s in the domain.
- Generated hash-function shall be checked to ensure no collisions.

The xor_modulo_hash algorithm: design.

(Inspired by discussions with Dr. Richard A. Harris[11], deceased.)

```
size_t xor_modulo_hash(state_type state, uint64_t seed,
size_t denominator) {
    return (state ^ seed) % denominator;
}
```

- ^ operation: the seed shuffles the bits in the STATE so in a sense adds extra entropy to the algorithm.
  - Brute-force search attempts to find an acceptable seed (enumerates domain of all seeds).
- % operation: constrains the output of the algorithm, also:
  - Range is [0,...,denominator), i.e. how minimal hash shall be.
  - Experimentation: denominator must be odd.
  - Strength reduction: optimise potentially very slow %.
- Fixed {seed, denominator} instantiates a hash function, written to a C++ header-file.

Generating the seed and denominator, 1/3.

- Surprisingly *slow*: on my AMD Ryzen 9 3900 at over 4GHz it can take over 15 minutes.
  - No surprise as brute-force!
  - Searching for the `seed` is embarrassingly parallel:
    - Developed `find_first_of(...)` a data-parallel algorithm as part of PPD[12]: it searches the input domain $[start, ..., end)$ to find a suitable `seed` for a suitable `predicate`.

Generating the seed and denominator, 2/3.

- May *never* find a suitable `seed` with a sufficiently small `denominator` for the set of input `STATE`s.
  1. A perfect, minimal hash $\forall$ sets of `STATE`s input does not exist.
     - The entropy added by the `seed` and `denominator` will be inadequate for the majority of inputs.
     - Shannon: there is only so much information in the inputs and a very constrained amount of entropy that may be added in the `xor_modulo_hash` algorithm.
     - One may need a different algorithm.
     - For the same `denominator` the smallest `seed` was used.
     - If a `seed` of 0 found we should use that optimisation!
  2. Experimentation revealed that usually a `denominator` $> (n+1)$ was required, where n is the number of `STATE`s input, i.e. a classic space-time trade-off.
     - Collision-free hash has been guaranteed by design as the `unordered_tuple` has no collision-avoidance mechanism.

Generating the seed and denominator, 3/3.

1. Experience indicates that ∀ inputs required, solutions were always found.
2. ***The build may fail because generating the hash for the set of*** `STATE`***s may fail!***
3. Note that the compiler is very unlikely to generate this for us:
    1. The set of `STATE`s are effectively random numbers: the MIT specifications have been bizarre in this regard.
        - Oh! Would it be moot if exchanges just used $\mathbb{N}$atural numbers...
    2. Usually the transformation goes something like: `if-else` to `switch`, then analyse the `switch` to see if a computed-`GOTO` may be generated (only simple cases) or use bisection then cluster then table or recurse otherwise `if-else`.
    3. Avoids "there ain't no such thing as a general, minimal, and perfect hash" by "bisection with divide-and-conquer".

Test 1: Micro-benchmark: methodology.

- Implements a cut-down version of the FIX-to-MIT/BIT Translator (more later) client-to-exchange, meta state-machine:
  - Has all of the correct values for the `STATE`s,
  - Same number of `Transitions`, but highly simplified.
- A comparison of the performance of `if-else` vs `computed-GOTO` was made.
  - The selection of the input `STATE` was randomized to try and defeat the branch-predictor.
- Unless otherwise noted, g++ v12.2.1 and clang++ v15.0.1 were used.

Test 1: Micro-benchmark: methodology.

- Implements a cut-down version of the FIX-to-MIT/BIT Translator (more later) client-to-exchange, meta state-machine:
  - Has all of the correct values for the `STATE`s,
  - Same number of `Transitions`, but highly simplified.
- A comparison of the performance of `if-else` vs `computed-GOTO` was made.
  - The selection of the input `STATE` was randomized to try and defeat the branch-predictor.
- Unless otherwise noted, g++ v12.2.1 and clang++ v15.0.1 were used.

Test 2: A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
  - listens to socket (the client-side) for FIX-format[13] messages,
  - sends & receives binary-protocol MIT/BIT-format[9] messages via a server-side socket.
  - Two tests:
    1. In-order: one sends then receives the response repeatedly.
    2. Out-of-order: sends all orders, then waits to receive all the responses.
- Uses Boost.ASIO, but numerous other optimisations, including the above, used SSE2 & higher instructions.
- Neither a Solarflare card nor OpenOnload drivers were used.
  - Would have reduced context-switches.
- Previous presentations have more detail [14].

Test 2: A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
  - listens to socket (the client-side) for FIX-format[13] messages,
  - sends & receives binary-protocol MIT/BIT-format[9] messages via a server-side socket.
  - Two tests:
    1. In-order: one sends then receives the response repeatedly.
    2. Out-of-order: sends all orders, then waits to receive all the responses.
- Uses Boost.ASIO, but numerous other optimisations, including the above, used SSE2 & higher instructions.
- Neither a Solarflare card nor OpenOnload drivers were used.
  - Would have reduced context-switches.
- Previous presentations have more detail [14].

Test 2: A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
  - listens to socket (the client-side) for FIX-format[13] messages,
  - sends & receives binary-protocol MIT/BIT-format[9] messages via a server-side socket.
  - Two tests:
    1. In-order: one sends then receives the response repeatedly.
    2. Out-of-order: sends all orders, then waits to receive all the responses.
- Uses Boost.ASIO, but numerous other optimisations, including the above, used SSE2 & higher instructions.
- Neither a Solarflare card nor OpenOnload drivers were used.
  - Would have reduced context-switches.
- Previous presentations have more detail [14].

Test 2: More details.

- A FIX New Order message is sent to a socket,
  - translated to MIT/BIT native binary format,
    - sent over sockets to a basic simulator,
    - which responds with a fill,
  - translated back to a FIX fill message.
- Sent back to the client.
- Computer was quiescent; `numactl` was not used, threads were pinned.
  - Highly optimised kernel, Gentoo/Linux.
    - Lap-brick: Single AMD Ryzen 9 3900, 12 physical cores, $\geq$ 4.0GHz, DDR4 RAM & NVMe storage.

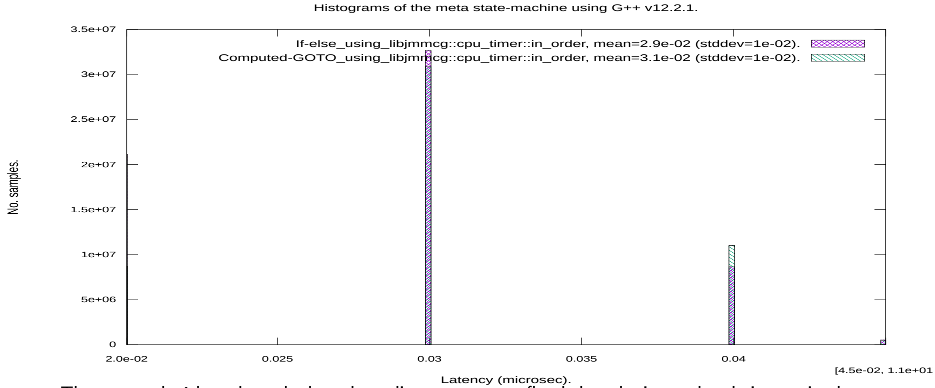
Test 2: More details.

- A FIX New Order message is sent to a socket,
  - translated to MIT/BIT native binary format,
    - sent over sockets to a basic simulator,
    - which responds with a fill,
  - translated back to a FIX fill message.
- Sent back to the client.
- Computer was quiescent; `numactl` was not used, threads were pinned.
  - Highly optimised kernel, Gentoo/Linux.
    - Lap-brick: Single AMD Ryzen 9 3900, 12 physical cores, $\geq$ 4.0GHz, DDR4 RAM & NVMe storage.

Test 2: More details.

- A FIX New Order message is sent to a socket,
  - translated to MIT/BIT native binary format,
    - sent over sockets to a basic simulator,
    - which responds with a fill,
  - translated back to a FIX fill message.
- Sent back to the client.
- Computer was quiescent; `numactl` was not used, threads were pinned.
  - Highly optimised kernel, Gentoo/Linux.
    - Lap-brick: Single AMD Ryzen 9 3900, 12 physical cores, $\geq$ 4.0GHz, DDR4 RAM & NVMe storage.

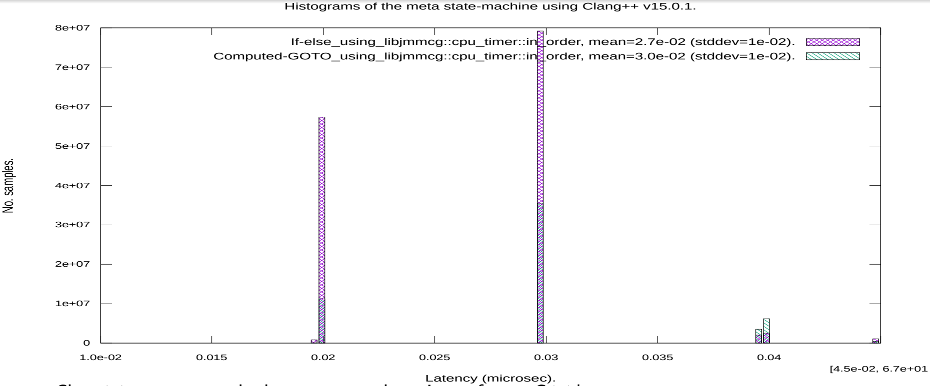
Test 1: Micro-benchmark: results G++ ($\leq 2\%$ MAD[15]).



There are only 4 branches: the branch-predictor never gets flooded as the internal code is too simple.

- The BTB has ≈ 32 slots - so never flooded, works beautifully.
- When used, the predictor was very accurate $\geq 98\%$: modern predictors also use Markov chains.

Test 1: Micro-benchmark: results Clang++ ($\leq 2\%$ MAD).



Clang++ generates code that appears to be twice as fast as G++!

- Some issue in the code-generation by G++?
- Pipeline hazards affecting the outcome?
    - 4 conditionals in the `Transitions`, so can be accommodated by the hardware.

Test 2: `if-else`: G++ generated code.

```
405:  movzx eax,WORD PTR [rbp-0x2b2]    #    unroll::msm.process(state, p...);
40c:  cmp ax,0x44    #    return state==Transition1::start
? transition1.process(p...) : Transition2::process(p...);
410:  jne 997
416:  call Transition2::process(p...)
997:  cmp ax,0x46    #    return state==Transition3::start
? transition3.process(p...) : Transition4::process(p...);
99b:  jne 9ab
99d:  call Transition4::process(p...)
9ab:  cmp ax,0x47
9af:  jne 9ba
9b5:  call Transition1::process(p...)
9ba:  call Transition3::process(p...)
```

The compiler generated the expected jumps.

Test 2: computed-GOTO: G++ generated code.

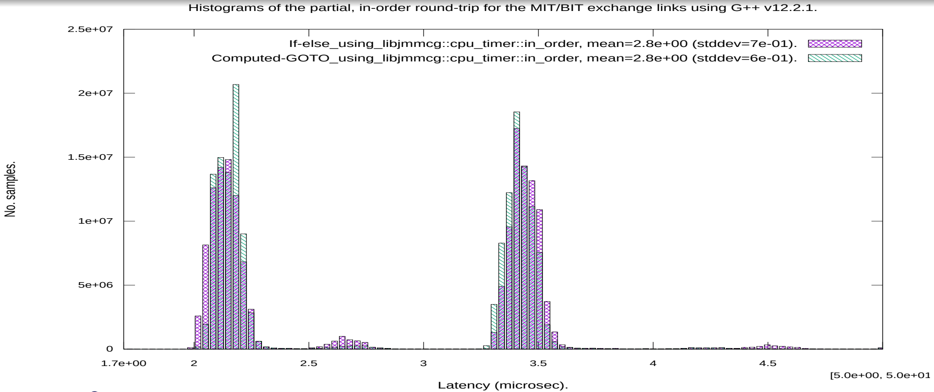
```
3c5:  movzx eax,WORD PTR [rbp-0x2c6]    #    hash::msm.process(state, p...);
3cc:  mov ecx,0xcccccccd
3d1:  mov BYTE PTR [rbp-0x2a8],0x0
3d8:  mov DWORD PTR [rbp-0x2a4],0x44    #    state
3e2:  mov rsi,r15
3e5:  vmovdqa xmm3,XMMWORD PTR [rbp-0x330]
3ed:  vmovdqa xmm2,XMMWORD PTR [rbp-0x310]
3f5:  mov edx,eax
3f7:  vmovdqa xmm1,XMMWORD PTR [rbp-0x320]
3ff:  vmovdqa xmm0,XMMWORD PTR [rbp-0x340]
407:  imul rdx,rcx
40b:  shr rdx,0x22
40f:  lea edx,[rdx+rdx*4]    #    Address computations from wrappers?
412:  sub eax,edx
414:  lea rdx,[rbp-0x2c7]    #    Address computations from wrappers?
41b:  shl rax,0x5    #    denominator - strength reduction!
41f:  vmovdqa XMMWORD PTR [rbp-0x150],xmm3
427:  vmovdqa XMMWORD PTR [rbp-0x1b0],xmm2
42f:  lea rdi,[rbx+rax*1+0x40]    #    Address computations from wrappers?
434:  vmovdqa XMMWORD PTR [rbp-0x190],xmm1
43c:  vmovdqa XMMWORD PTR [rbp-0x130],xmm0
444:  mov rax,QWORD PTR [rdi]
447:  call QWORD PTR [rax+0x18]    #    The computed-GOTO! As a virtual-method call...
```

Test 2: computed-`GOTO`: reflections re. generated code.

The compiler generated the expected computed-`GOTO`:

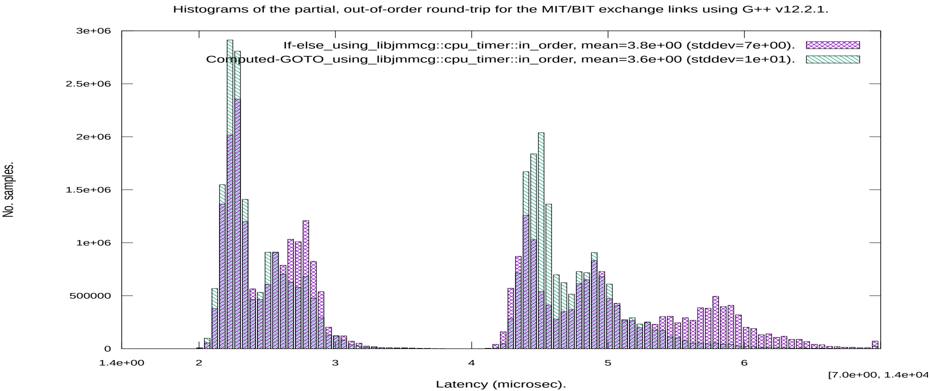
- Disappointing that the code-generation takes so many assembly instructions.
- Lots of code-motion and inlining makes the calls to hashing and indexing make the assembly complex to decipher...
- The multiple `lea` instructions may be causing AGIs...
  - Causing pipeline stalls: inadequate de-virtualisation?
- Recall we need this to be $\ll 20$ clock-cycles, which this is unlikely to be.
- Godbolt[16] could not be used.
  - Instead `objdump -drwCS -Mintel` with a lot of editing.

Test 2: `if-else` vs computed-`GOTO`: histogram, in-order G++ ($\leq 5\%$ MAD).



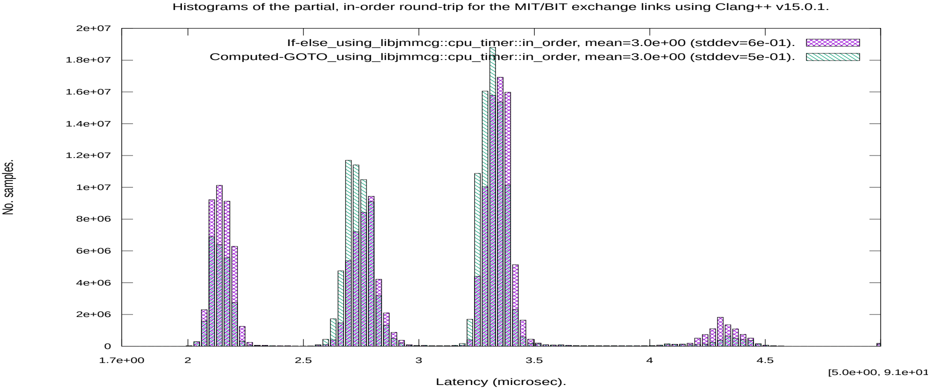
- Kurtosis crucial: usual statistics fail us - need the histogram to tell all.
- Computed-`GOTO` out-performs `if-else`, despite curious assembly generated.

Test 2: `if-else` vs computed-`GOTO`: histogram, out-of-order G++ ($\leq 7\%$ MAD).



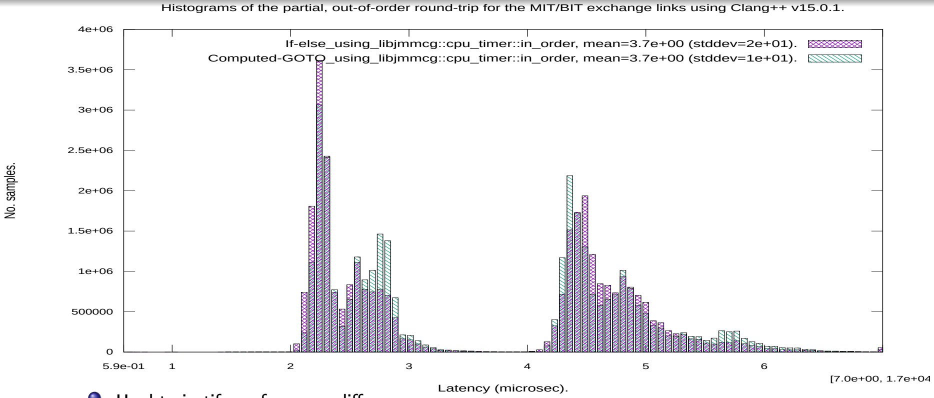
- Computed-`GOTO` out-performs `if-else`, outliers for the latter a serious problem for algos.

Test 2: `if-else` vs computed-`GOTO`: histogram, in-order Clang++ ($\leq 4\%$ MAD).



- The roughly 750nsec resonance, likely due to hardware hazards, is most curious.

Test 2: `if-else` vs computed-`GOTO`: histogram, out-of-order Clang++ ($\leq 5\%$ MAD).



- Hard to justify performance difference.
- The consistent difference between G++ and Clang++ deserves better analysis.

Test 2: Reflections: Performance differences.

- Very small improvements: micro-optimisation "noise floor" hit!
  - Computed-`GOTO`: the generated assembly looks suspicious...
  - Yet it outperforms `if-else` from evidence in histograms.
- The FIX-to-MIT/BIT-Translator test had the same input `STATE`, so executed the same `Transition`:
  - Ideal for a branch-predictor, as a highly predictable conditional-jump!
  - Surprising computed-`GOTO` *beat it*!
  - Researchers at Intel, AMD, etc, etc work very hard to improve the branch-predictor...
- `objdump` has issues generating the disassembly...

Reflections: Code complexity, maintainability and compile-time performance.

- The complexity of the code is outrageous:
  - It looks very much like write-once code: maintainability is lost.
  - Required: a modern compiler that supports C++ very well.
    - g++ v12.2.1 & clang++ v15.0.1 used.
  - The compile-time was not significantly increased...
    - The translation unit takes over 6 minutes to compile on my lap-brick.
    - Less powerful computers could take 10s of minutes...
- Over 10Gb of RAM is required to compile the translation unit.
  - Limits parallelisation of the build...
- The build scripts for `cmake` become much more complex...

Reflections: Issues regarding `xor_modulo_hash` algorithm.

- The hash function must be generated first to permit inlining, which serialises the build.
  - Requires parallelisation or is excessively slow.
- The hash may fail to be generated:
  - Meta state-machine cannot be generated! ***Code simply will not compile!***
  - Critical issue in production code - ***never*** rely on luck to compile one's code...!
  - Use `if-else` or another; $\ll 20$ clock cycles to run: more sophisticated algorithms may have excessive run-times vs `if-else`. Common algorithms e.g. `std::hash`, Hsieh[17] or Murmur2[18]: too slow for our purpose.
- Speculation: more `STATE`s and their values may make generation of the algorithm more likely to fail.
  - Currently `STATE`s ≤ 7 in any of the meta state-machines.

Conclusions. 1/2.

- Exorbitant effort taken to achieve little result?
  - In terms of speed-up: yes.
    - Suspicious assembly-generation may have reduced the performance.
- Managed to force the compiler to generate the computed-`GOTO` with C++:
  - *Heroic effort:* implemented library to recover type-information for `Transition::process(...)` : minimal intrusiveness for users.
  - *Madness*: implemented a data-parallel algorithm to attempt (yes, *Shannon*) to compute minimal, perfect hashes for a high-performance hash.
  - An extremely high-performance *Template Meta-Progammed* state-machine has been implemented as a library[3].

Conclusions. 1/2.

- Exorbitant effort taken to achieve little result?
  - In terms of speed-up: yes.
    - Suspicious assembly-generation may have reduced the performance.
- Managed to force the compiler to generate the computed-`GOTO` with C++:
  - *Heroic effort:* implemented library to recover type-information for `Transition::process(...)` : minimal intrusiveness for users.
  - *Madness*: implemented a data-parallel algorithm to attempt (yes, *Shannon*) to compute minimal, perfect hashes for a high-performance hash.
  - An extremely high-performance *Template Meta-Progammed* state-machine has been implemented as a library[3].

Conclusions, 2/2.

- Choice of micro-optimisation investigated would have been vital:
  Premature optimisation is the root of all evil[4]!
  - The techniques that had to be used may serve as a warning to others...
- I did this so that you would not have to!
- My sincere thanks to:
  - Jon Chesterfield for his patience and tolerance: he listened to me talk about this since 2017...
  - My reviewers: Paul Evans & Vladimir Arnost.
- For more information on methodology or notes, please contact:
  accuconf2023@hussar.me.uk

Conclusions, 2/2.

- Choice of micro-optimisation investigated would have been vital:

  Premature optimisation is the root of all evil[4]!

  - The techniques that had to be used may serve as a warning to others...
- I did this so that you would not have to!
- My sincere thanks to:
  - Jon Chesterfield for his patience and tolerance: he listened to me talk about this since 2017...
  - My reviewers: Paul Evans & Vladimir Arnost.
- For more information on methodology or notes, please contact: *accuconf2023@hussar.me.uk*

Conclusions, 2/2.

- Choice of micro-optimisation investigated would have been vital:

  Premature optimisation is the root of all evil[4]!

  - The techniques that had to be used may serve as a warning to others...
- I did this so that you would not have to!
- My sincere thanks to:
  - Jon Chesterfield for his patience and tolerance: he listened to me talk about this since 2017...
  - My reviewers: Paul Evans & Vladimir Arnost.
- For more information on methodology or notes, please contact: ***accuconf2023@hussar.me.uk***

An outstanding success: introducing Dr. Cassio Neri & Prof. Lorenz Schneider...

- An outstanding micro-optimisation regarding date-conversions:
  - "Euclidean affine functions and their application to calendar algorithms"[19]
- Basically optimises the conversion between the Gregorian calendar and Unix Epoch-based offsets.
- Used in all *nix kernels, `libc`, `<chrono>`, Microsoft .Net (including C♯, etc), Android...
  - Literally billions of installations...
- The most successful micro-optimisation I have *ever* heard of!

For Further Reading I

- Dijkstra, E. W. "Go To statement considered harmful, Commm." ACzI/I, II (3) (1968): 147-148, https://doi.org/10.1145/988056.988069
- https://agner.org/optimize/instruction_tables.ods
- http://libjmmcg.sf.net/
- https://hans.gerwitz.com/2004/08/12/premature-optimization-is-the-root-of-all-evil.html
- Gustafson, J.L. (2011). "Amdahl's Law." In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA, https://doi.org/10.1007/978-0-387-09766-4_77

For Further Reading II

- https://www.boost.org/doc/libs/1_81_0/libs/msm/doc/HTML/index.html
- https://www.oreilly.com/library/view/extended-stl-volume/9780321305503/ch14.html
- Bartosz Milewski "Template Metaprogramming Made Easy (Huh?)", https://bartoszmilewski.com/2009/09/08/template-metaprogramming-made-easy-huh/
- http://www.borsaitaliana.it/borsaitaliana/gestione-mercati/migrazionemillenniumit-mit/mit203nativetradinggatewayspecification.en_pdf.htm
- https://www.gnu.org/software/gperf/

For Further Reading III

- https://web.archive.org/web/20221208124607/https://thusspakeak.com/
- McGuiness, J., Egan, C., "A Domain-Specific Embedded Language for Programming Parallel Architectures.", DCABES 2013, https://www.researchgate.net/publication/340902083_A_Domain-Specific_Embedded_Language_for_Programming_Parallel_Architectures/references
- https://fiximate.fixtrading.org

For Further Reading IV

- McGuiness, J., "A Performance Analysis of a Simple Trading System...", CPPNow, Aspen, 2019, https://www.researchgate.net/publication/340926245_A_Performance_Analysis_of_a_Simple_Trading_System
- "http://mathbits.com/MathBits/TISection/Statistics1/MAD.html
- https://gcc.godbolt.org
- http://www.azillionmonkeys.com/qed/hash.html
- https://simonhf.wordpress.com/2010/09/25/murmurhash160/

For Further Reading V

Neri, C., Schneider, L., "Euclidean affine functions and their application to calendar algorithms", Software Practice and Experience 53(1), December 2022, http://dx.doi.org/10.1002/spe.3172, https://www.researchgate.net/publication/365981828_Euclidean_affine_functions_and_their_application_to_calendar_algorithms

View publication stats