Self-move and no-op: Survey of current libraries and hardware

Document Number: **d2229 R0 – draft 6**
Date: 2021-01-16
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, LEWG, LWG

Abstract

This paper reports the results of a survey of existing practice related to self-move:

On current `std::` implementations, what does self-move do for various standard types?

On current processors, what is the worst case cost for guaranteeing that self-move is a no-op, namely the cost of a brute-force `if(this!=&rhs)` identity test branch?

Contents

0 Overview

0.1 Background and motivation

Self-copy and self-swap have long been required to be no-ops in the C++ standard library and in popular guidance books. This follows “do as the ints do,” and avoids data loss in the case of self-copy or self-swap. Self-move, however, is only required to be a no-op for some types in the C++ standard library (see §1).

While discussing C++ Core Guidelines pull request #1606, the group revisited the concern that guaranteeing self-move to be a no-op could be expensive. In the worst case, providing this guarantee would require adding an `if(this!=&rhs)` identity test to a move assignment operator, and we know branches can be expensive due to mispredictions and pipeline stalls. However, it turned out that no one we knew had a benchmark we could cite that measured the overhead of that branch on move. I started to write such a benchmark to measure the cost, but kept failing to measure any even as I kept trying more examples on more hardware. Along the way, I also learned that there was disagreement among experts about what the standard library requires today, and divergence among implementations. So, after spending more time than I expected on the research but having learned some useful things, I’m documenting the results in this paper.

0.2 Goals and scope

This paper reports data from two surveys I performed of existing practice related to self-move:

§1 In the standard library and its implementations, what does self-move do for various types?

§2 On current processor hardware, what is the cost of the worst case for guaranteeing that self-move is a no-op, namely performing a brute-force `if(this!=&rhs)` identity test branch?

This paper does not discuss whether self-move can happen in “good” code, whether data loss should be considered acceptable if self-move does happen, or whether self-move should be required to be a no-op.

0.3 Acknowledgments

Thank you to all of the following for feedback on drafts of this paper and other contributions below.

Thank you to Arthur O’Dwyer and Bjarne Stroustrup for asking to revisit the question of self-move, and pointing out the divergence between the C++ Core Guidelines and the standard library, in Guidelines pull request #1606.

Special thanks to Howard Hinnant, the lead proposer and world expert on move semantics, for the discussion of potential costs that led to this paper and for the benchmark examined in §2.3, and thank you to the following for their insights into specific implementations of the standard library and hardware: Marshall Clow, Andrew Kaylor, Erich Keane, Billy O’Neal, Ville Voutilainen, and Jonathan Wakely.

Thank you to the following for kindly running the §2.3 benchmark on a variety of CPU and GPU hardware I did not have direct access to: Christian Ceelen, Niall Douglas, Jonathan Henson, David Olsen, Billy O'Neal, Kervin Peguero, Hubert Tong, and Jan Wilmans.

Thank you to the following for additional review and comments: Ben Craig, Olivier Giroux, Bryce Lelbach, and Sean Parent.

Finally, thank you to Matt Godbolt, Fred Tingaud, Eelis van der Weegen and their supporters for Godbolt.org, quick-bench.com, and eel.is/c++draft.

1 Survey of the standard library and implementations: What self-copy/move does for `std::` types

1.1 What self-**copy** does for `std::` types

This section documents which standard library implementations implement self-copy as a no-op.

The test code is available at https://godbolt.org/z/WEh9bc. That link includes the results for libstdc++ and libc++. For MSVC I ran the test on my machine, and validated with the maintainers that the no-op observations for all types were real for all values, not just artifacts that happened to work for the values in this particular test.

Self-move of…	**ISO C++ requires**	**GCC 10.1 libstdc++**	**Clang 10.0 libc++**	**MSVC 16.6.2 STL**
`unique_ptr`	*invalid*	*n/a*	*n/a*	*n/a*
`shared_ptr`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`string (short)`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`string (long)`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`vector`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`list`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`forward_list`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`set`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`map`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`unordered_set`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`unordered_map`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)

Note:

- In [tab:container.req], we require the postcondition that lhs == rhs.
- In [tab:cpp17.copyassignable], we require the same, but phrase it as “lhs is equivalent to rhs, the value of rhs is unchanged.”

1.2 What self-**move** does for `std::` types

This section documents which standard library implementations implement self-move as a no-op.

The test code is available at https://godbolt.org/z/UmEsxe. That link includes the results for libstdc++ and libc++. For MSVC I ran the test on my machine, and validated with the maintainers that the no-op observations for all types were real for all value, not just artifacts that happened to work for the values in this particular test.

Self-move of…	**ISO C++ requires**	**GCC 10.1 libstdc++**	**Clang 10.0 libc++**	**MSVC 16.6.2 STL**
`unique_ptr`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`shared_ptr`	no-op (orig value)	no-op (orig value)	no-op (orig value)	no-op (orig value)
`string (short)`		empty	empty	no-op (orig value)
`string (long)`		empty	empty	no-op (orig value)
`vector`		empty	empty	no-op (orig value)
`list`	valid object	heap corruption	empty	no-op (orig value)
`forward_list`	***(see Note)***	empty	empty	no-op (orig value)
`set`		empty	empty	no-op (orig value)
`map`		empty	empty	no-op (orig value)
`unordered_set`		empty	empty	no-op (orig value)
`unordered_map`		empty	empty	no-op (orig value)

Note:

- ISO C++ requires that self-move assignment of all standard library types is always at least defined behavior that leaves a valid object, and for certain types (notably smart pointers) requires it to be a no-op. [res.on.arguments]/1.3 states a basic library-wide requirement (implicitly 'unless otherwise specified') that rvalue reference parameters must not alias. We do 'specify otherwise' in [tab:cpp17.moveassignable]: The first row, "If `t` and `rv` do not refer to the same object…," says that they might alias and assigns a post-condition when they don't. The second row states a postcondition that applies whether or not `t` and `rv` refer to the same object, which defines self-move's behavior. — Making this clearer is the subject of active LWG issues 2468 and 2839.

Additional notes:

- libstdc++ reports that they are moving to update all libstdc++ types to make self-move be defined behavior that leaves a valid object.

2 Survey of execution costs: Worst-case cost of guaranteeing self-move is a no-op

This section aims to evaluate the worst-case cost of guaranteeing that self-move is a no-op, by injecting a brute-force `if(this!=&rhs)` identity test branch on every move, without awareness of the implementation details (so the test may be redundant) and without any attempt at optimization (e.g., `[[likely]]`), and measuring the overhead.

2.1 FAQs

2.1.1 Q: "These results feel too regular. My intuition is the branch was optimized away because *<thing compiler might know about test case>*?"

No, the branch is in the generated code. See the provided disassembly showing the branch instruction, and/or the `volatile` variations which are forced to carry the branch (plus an extra gratuitous memory access) and have the same performance. While experience is valuable, it's important not to rely on intuition about performance, and I have tried to limit myself to empirical measurement of real existing implementations and only test cases proposed by the people who have the performance concerns.

2.1.2 Q: "It's hard to measure tiny differences. Could these be dominated by *<instruction alignment or microarchitectural feature>*?"

If the only measurable differences really are on the order of instruction alignment issues, we have already answered whether the branch itself is a cost on the order of a pipeline stall with "no."

Most experiments saw no measurable difference for the no-op test on any platform. In two experiments there was a measurable difference, in both cases found to be caused by instruction alignment (see notes in §2.2.1 and §2.3.1), and in both cases happened to make case with the branch slightly faster (not slower).

2.2 Wrapping `std::` types to brute-force self-move no-op guarantee

This section tests the cost of adding a brute-force `if(this!=&rhs)` check around existing libstdc++ and libc++ `std::` types' move assignment operators. We measure a loop of `vector<T>::insert(middle,x)` to exercise `T` move operations with virtually no other work; note this also never does self-move and so the added test is pure overhead. `T` is a given `std::` type used directly or wrapped to add additional work as follows.

1. **`std_type` directly:** Tests the unmodified type `std_type` as currently implemented.
2. **`noop<std_type>`:** Wraps `std_type` to add a brute-force `if(this!=&rhs)` self-move assignment check:

```
template<class T>
struct noop {
  T t;
  noop(const T& t_) : t{t_} { }
  noop(const noop& rhs) : t{rhs.t} { }
  noop(noop&& rhs) : t{std::move(rhs.t)} { }
  noop& operator=(const noop& rhs) { t = rhs.t; return *this; }
  noop& operator=(noop&& rhs) { if (this != &rhs) t = std::move(rhs.t); return *this; }
};
```

3. noop_volatile<std_type>: Identical to noop, except adds a volatile read to ensure the identity check must emit a branch regardless of optimizations (note this tests against a static volatile variable instead of this, and so introduces the overhead of a load from an additional memory location not present in the other tests):

```
void* volatile xyzzy;

template<class T>
struct noop_volatile {
  T t;
  noop_volatile(const T& t_) : t{t_} { }
  noop_volatile(const noop_volatile& rhs) : t{rhs.t} { }
  noop_volatile(noop_volatile&& rhs) : t{std::move(rhs.t)} { }
  noop_volatile& operator=(const noop_volatile& rhs) { t = rhs.t; return *this; }
  noop_volatile& operator=(noop_volatile&& rhs)
                                    { if (xyzzy != &rhs) t = std::move(rhs.t); return *this; }
};
```

4. nomove<std_type>: Wraps std_type to disable move and force copy always, as a sanity check to check whether, and by how much, the other tests are still faster than a deep copy:

```
template<class T>
struct nomove {
  T t;
  nomove(const T& t_) : t{t_} { }
  nomove(const nomove& rhs) : t{rhs.t} { }
  nomove& operator=(const nomove& rhs) { t = rhs.t; return *this; }
  // no move
};
```

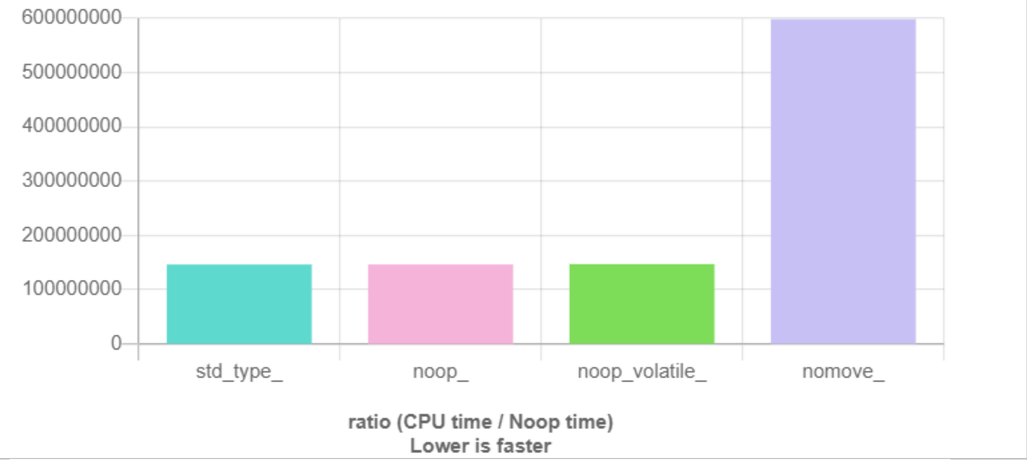
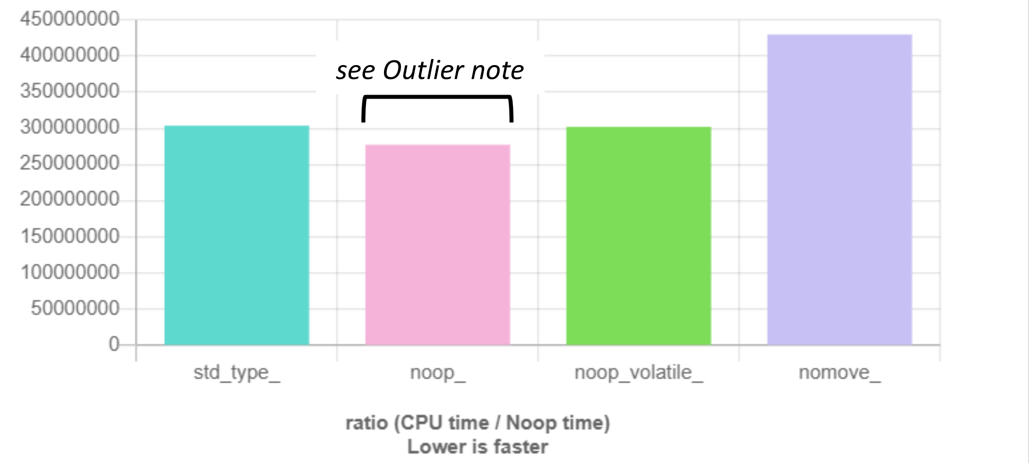
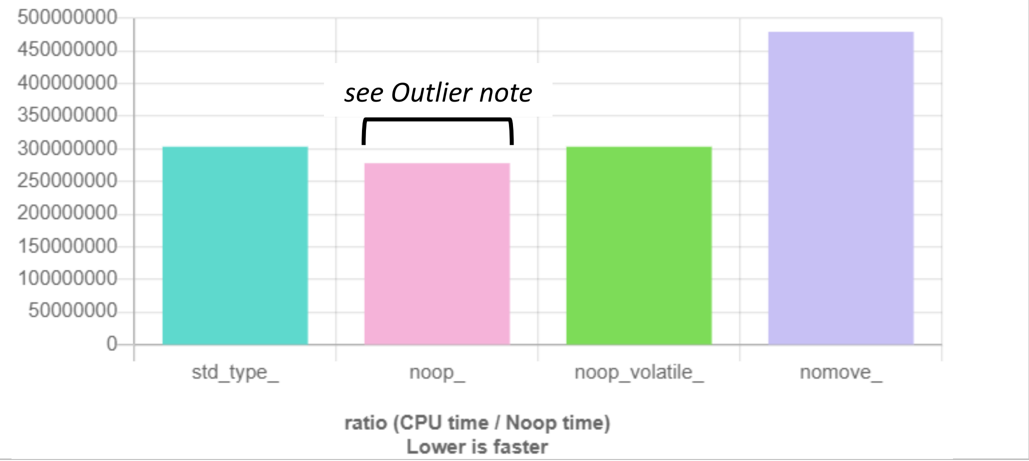
Each test was run on quick-bench.com on an Intel AVX2-capable chip (unspecified model, but AVX2 means 2013 or newer) built with each of the following:

- Clang 9.0 -O2 with libc++
- Clang 9.0 -O2 with libstdc++
- GCC 9.2 -O2 with libstdc++

Disassembly note: You can view the disassembly for each linked test via the Compiler Explorer button → and then in the disassembly search for "xyzzy" to quickly find the identity test branch for the volatile case, to confirm the branch is in the generated code.

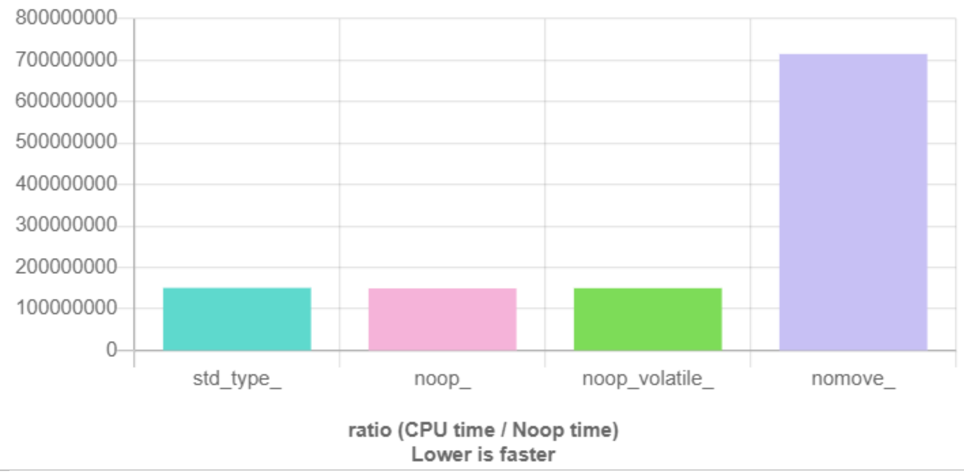
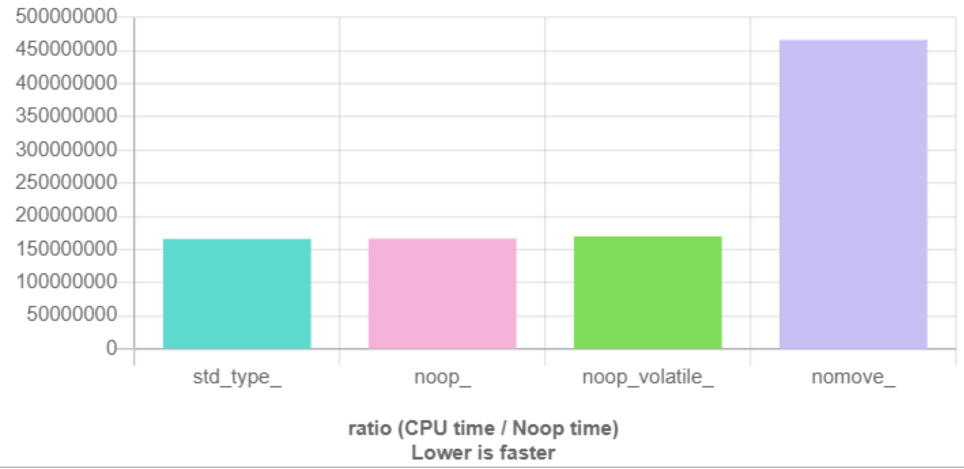
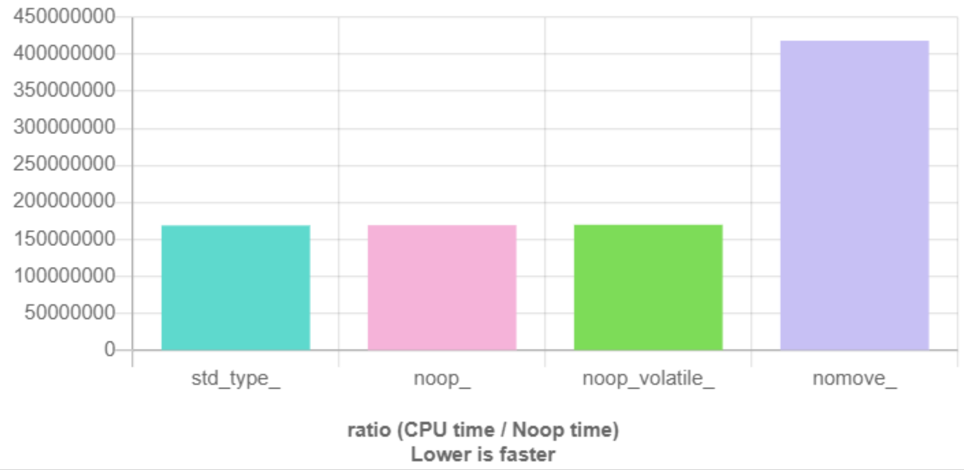
"Outlier investigation" notes: In one test, noop was faster than direct use of std_type which I considered surprising and warranted understanding, so I did further investigation and added a note to document the results with the example. The cause appears to be instruction alignment issues, where adding a few nop instructions removed the performance difference.

2.2.1 `std::string` (small size, 5 characters)

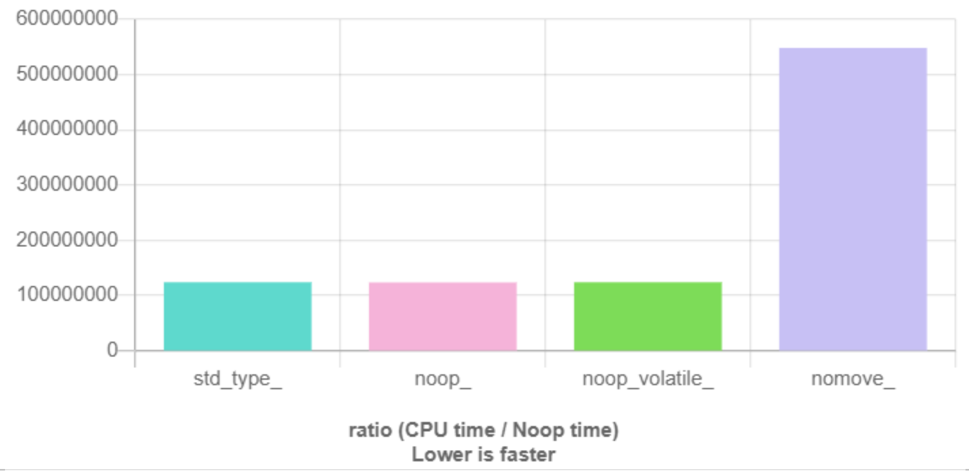
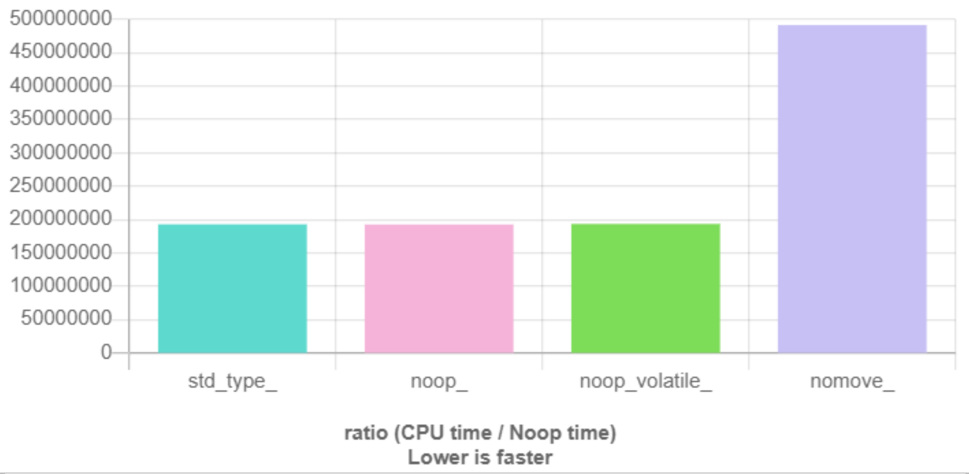
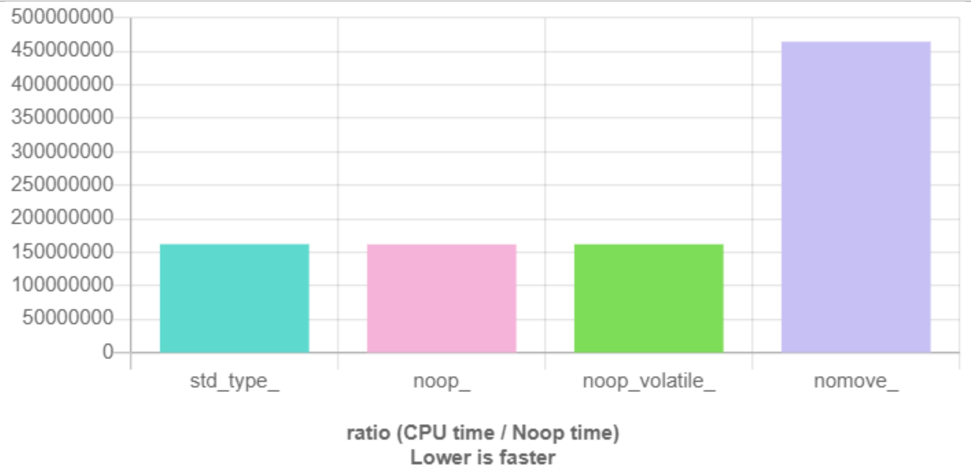
Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

Outlier investigation: For this small `string` case only, with libstdc++ only (built with either compiler), noop is consistently slightly faster. Because the performance difference is stable, I conjectured that it was due to code generation and instruction-cache effects, where the extra self-move test instructions might happen to improve other instruction alignment or locality as a side effect. **Test and resolution:** I added a few nop instructions just before the noop identity test, and that did in fact remove the performance difference — see Clang / libstdc++ **plus 4 nops** and GCC 9.2 / libstdc++ **plus 1 nop** which are identical to the above tests except adding 4 and 1 nop instructions respectively. This also emphasizes that the performance differences we are measuring are so small that they are dominated by instruction alignment issues, not a misprediction pipeline stall or a memory access.

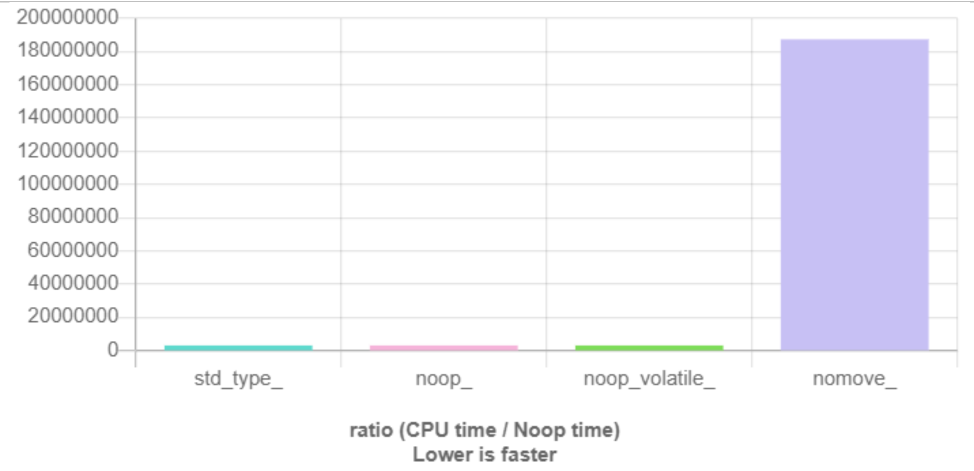
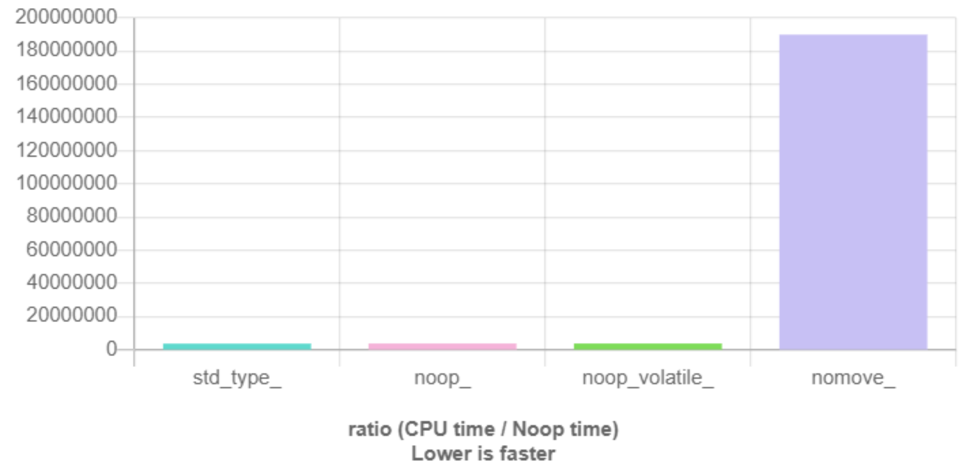
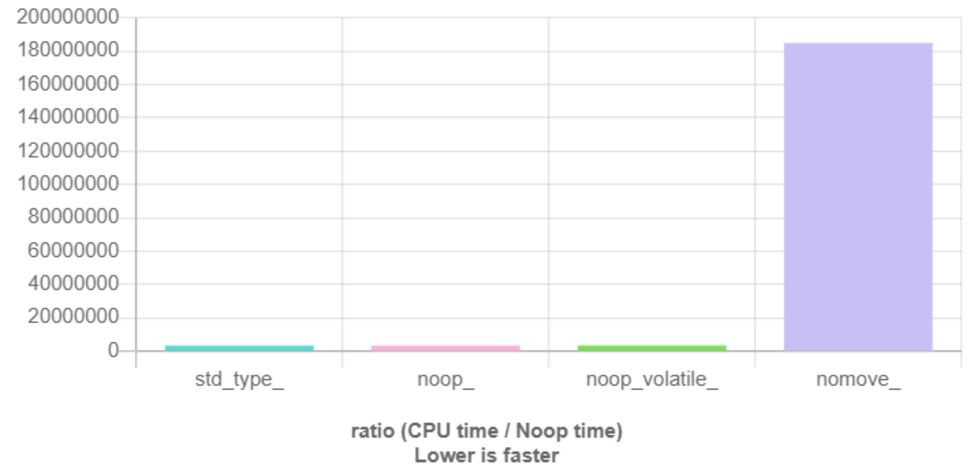
2.2.2 `std::string` (larger size, 30 characters)

Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

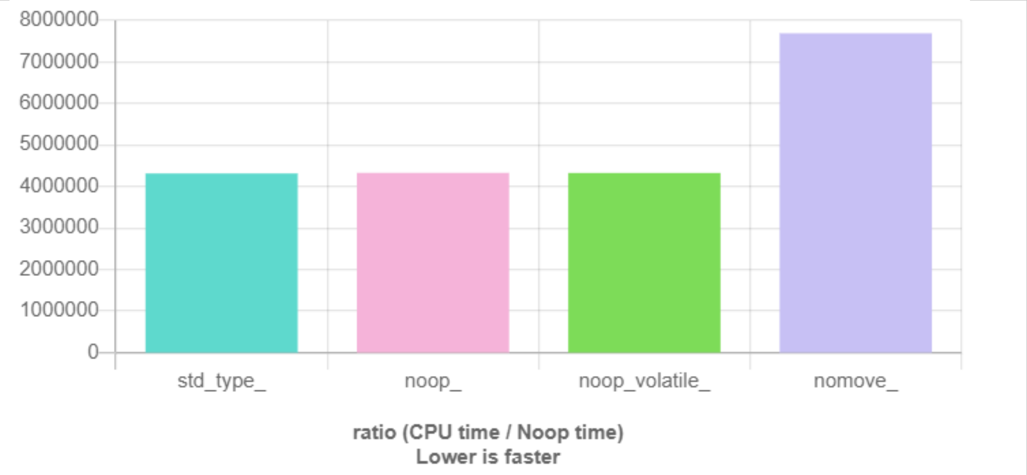
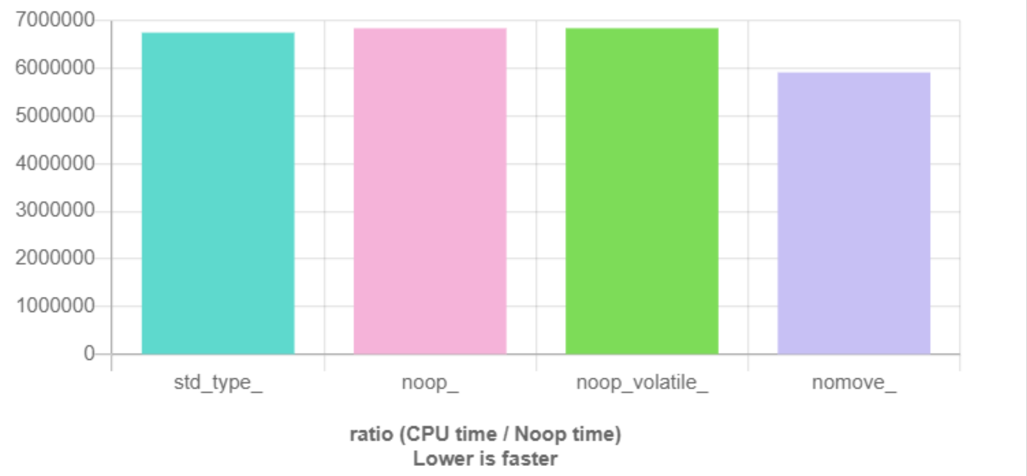
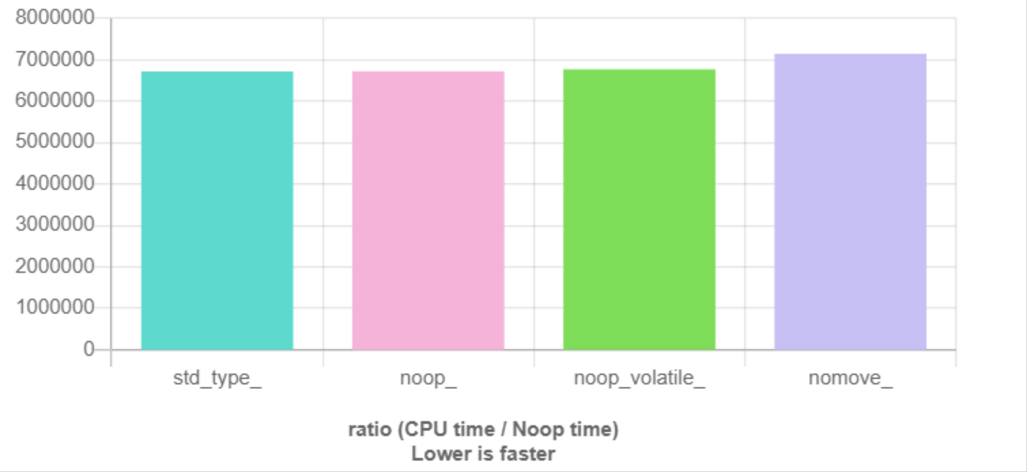
2.2.3 `std::vector<int>` (tiny size, 5 elements)

Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

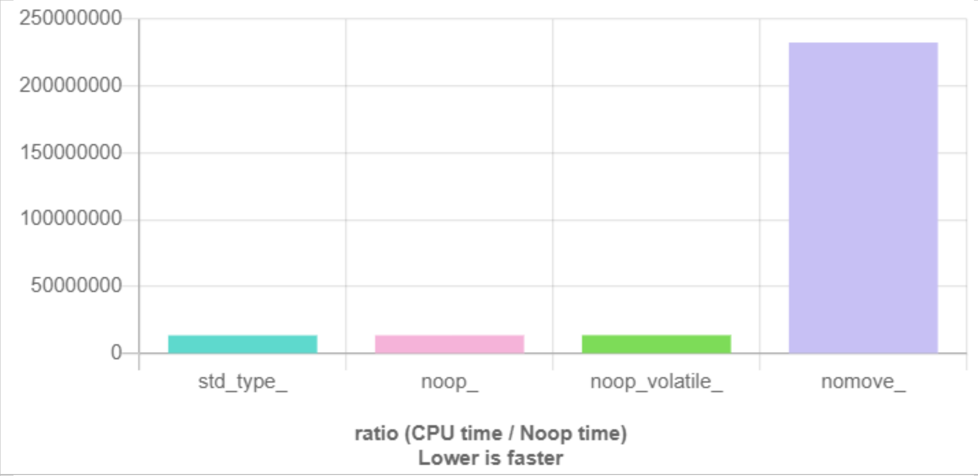
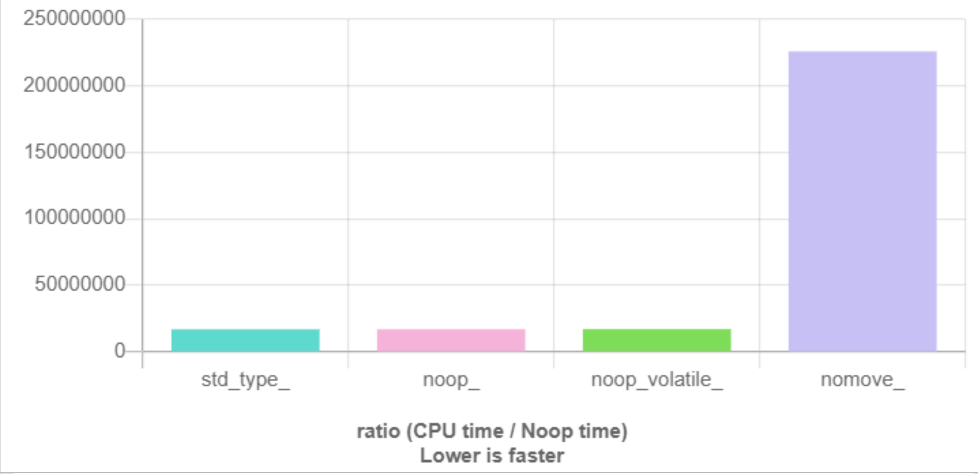
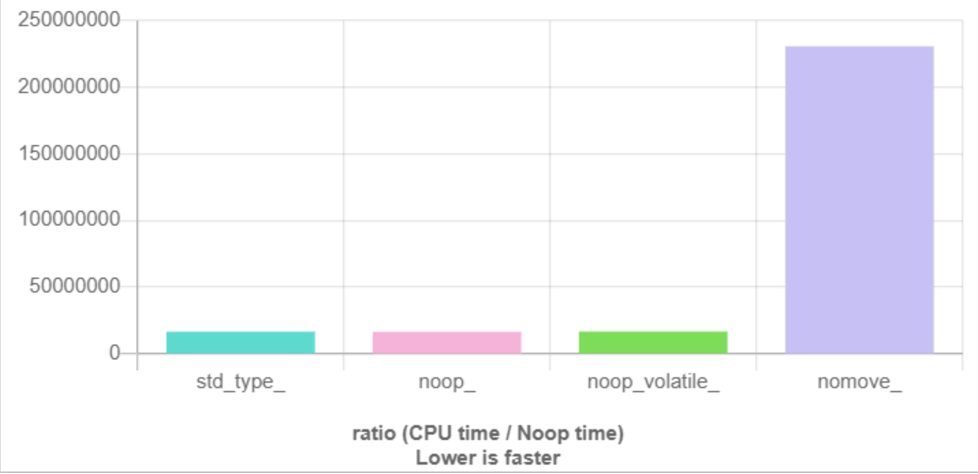
2.2.4 `std::vector<int>` (small size, 100 elements)

[Clang 9.0 -O2 with libc++]()	
[Clang 9.0 -O2 with libstdc++]()	
[GCC 9.2 -O2 with libstdc++]()	

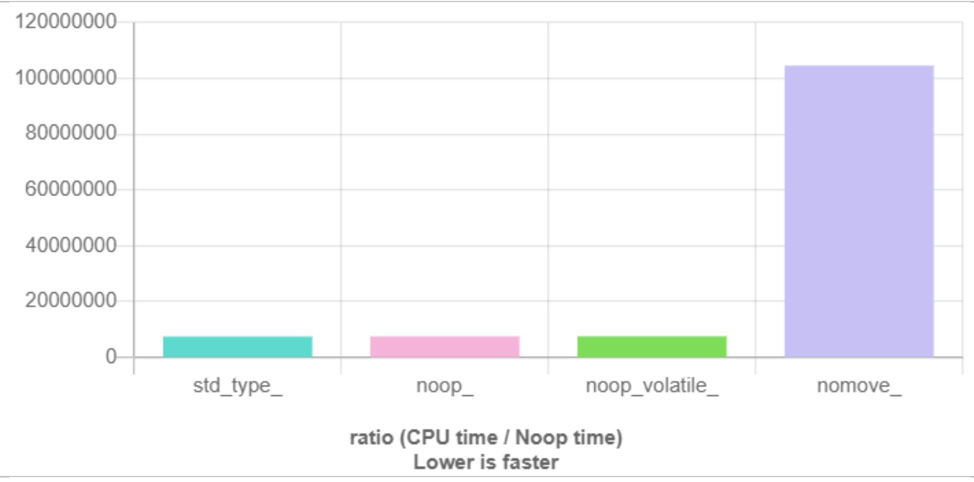
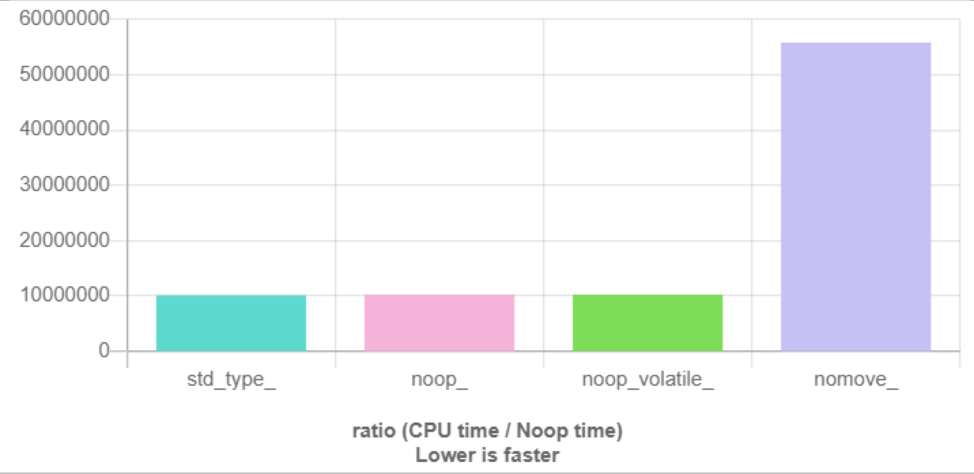
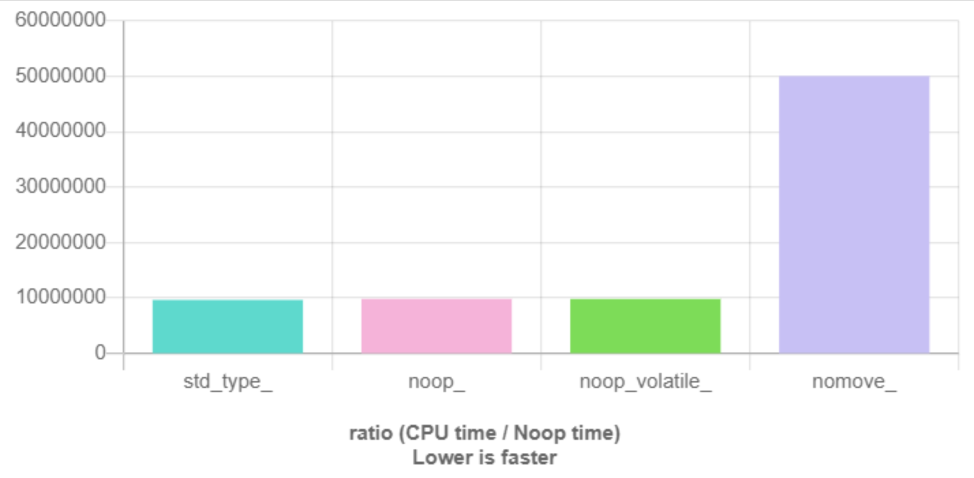
2.2.5 `std::list<int>` (tiny size, 5 elements)

Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

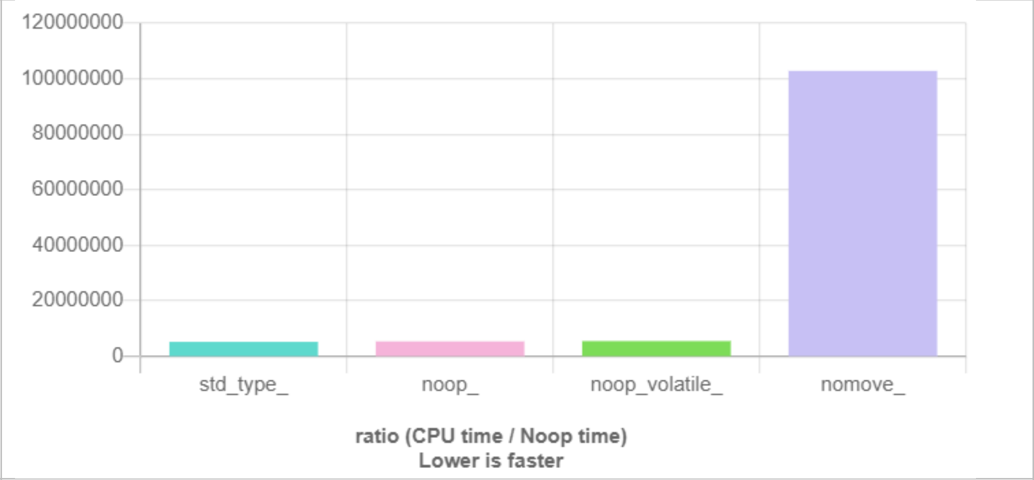
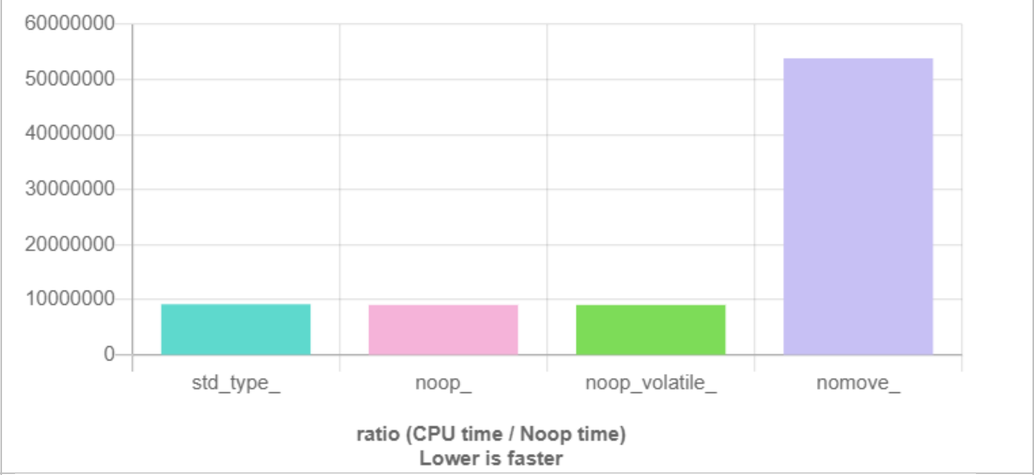
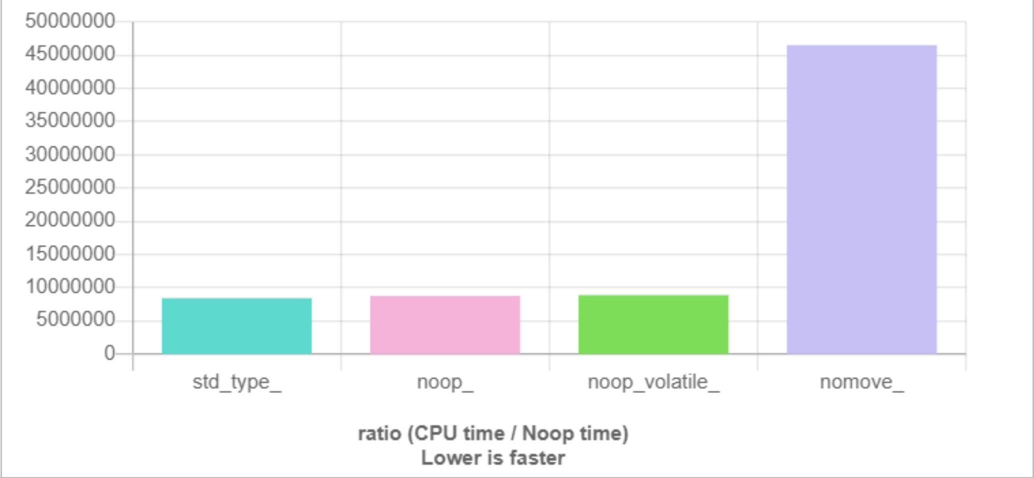
2.2.6 `std::list<int>` (small size, 100 elements)

[Clang 9.0 -O2 with libc++]()	
[Clang 9.0 -O2 with libstdc++]()	
[GCC 9.2 -O2 with libstdc++]()	

2.2.7 `std::map<int, long long>` (tiny size, 10 elements)

Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

2.2.8 `std::set<int>` (tiny size, 10 elements)

Clang 9.0 -O2 with libc++	
Clang 9.0 -O2 with libstdc++	
GCC 9.2 -O2 with libstdc++	

2.3 Benchmarking move for a simple resource-owning type

Because `std::` types can be complex, Howard Hinnant suggested benchmarking a simple resource-owning type `A` that owns one dynamically allocated `int`, and differ only in move assignment. I tested four variations of `A`:

1. `not_noop::A`: Tests a move assignment that is not a no-op for self-move but leaves the object in a valid state:

```
A& operator=(A&& a) noexcept {
    delete data_;
    data_ = nullptr;
    data_ = a.data_;
    a.data_ = nullptr;
    return *this;
}
```

2. `noop::A`: Tests the same move assignment operator, except adds a brute-force `if(this!=&rhs)` identity check to guarantee a no-op for self-move instead of a redundant `nullptr` assignment:

```
A& operator=(A&& a) noexcept {
    if (this != &a) {
        delete data_;
        data_ = a.data_;
        a.data_ = nullptr;
    }
    return *this;
}
```

3. `weird::A`: Like `noop`, except changes the branch condition to something that will not be always `true` or always `false` — we should expect that about half the time it avoids doing any work at all (which leaks memory, but it's `weird`), so the best case in a large run is that this could be up to 50% faster than `not_noop` if (a) we avoid the actual move work half the time and (b) the branch is low-cost (e.g., if allocation patterns produce sequential addresses, this branch can encounter alternating short runs of all-true or all-false which is friendly to predictors):

```
A& operator=(A&& a) noexcept {
    if ((size_t)this & 0x0100) {
        delete data_;
        data_ = a.data_;
        a.data_ = nullptr;
    }
    return *this;
}
```

4. `nomove::A`: Tests a copy-only type as a sanity check, where instead of move assignment we have only:

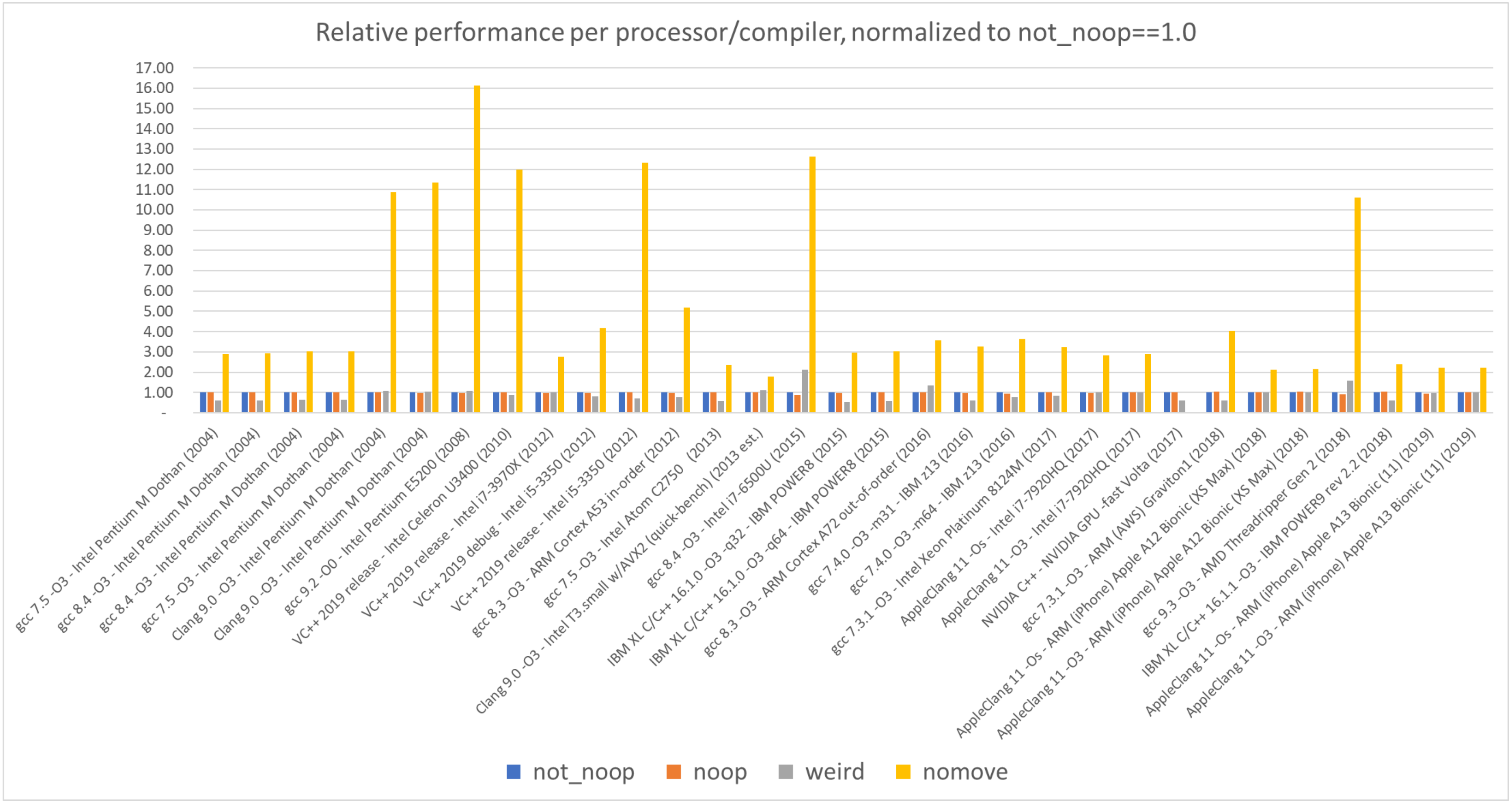
```
A& operator=(const A& a) {
    if (this != &a) {
        delete data_;
        data_ = new int(*a.data_);
    }
    return *this;
}
```

2.3.1 Results across processors: AMD, Apple, ARM, IBM, Intel, NVIDIA

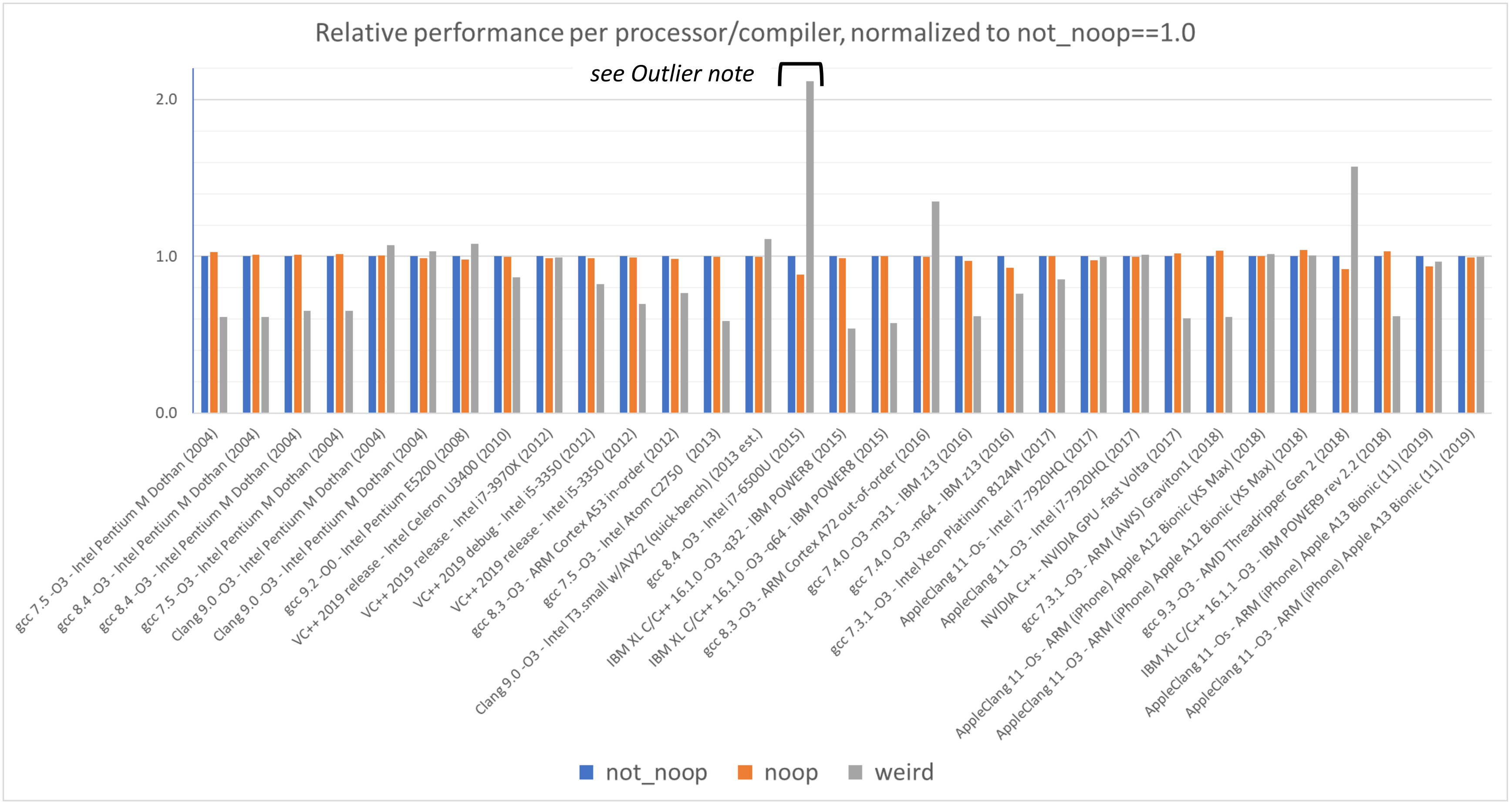
With the gracious assistance of several helpers and companies, I ran this benchmark code (GitHub) on a variety of processors, including

- of a variety of vintages (released from 2004 to 2019, shown in that order for lack of another interesting order)
- phone, tablet, desktop, datacenter, and mainframe (e.g., A13 iPhone to Graviton datacenter server to z13 mainframe)
- in-order and out-of-order processors (e.g., ARM Cortex A53 in-order and A72 out-of-order)
- “beefy” and “underpowered/netbook” parts (e.g., AMD Threadripper and Intel Atom)
- CPUs and a GPU (the latter has no predictor)

and using a variety of compilers, and normalized the results to `not_noop==1.0`. On all combinations, `nomove` was worst, as expected.



Removing `nomove` so that the differences in the others are easier to see:

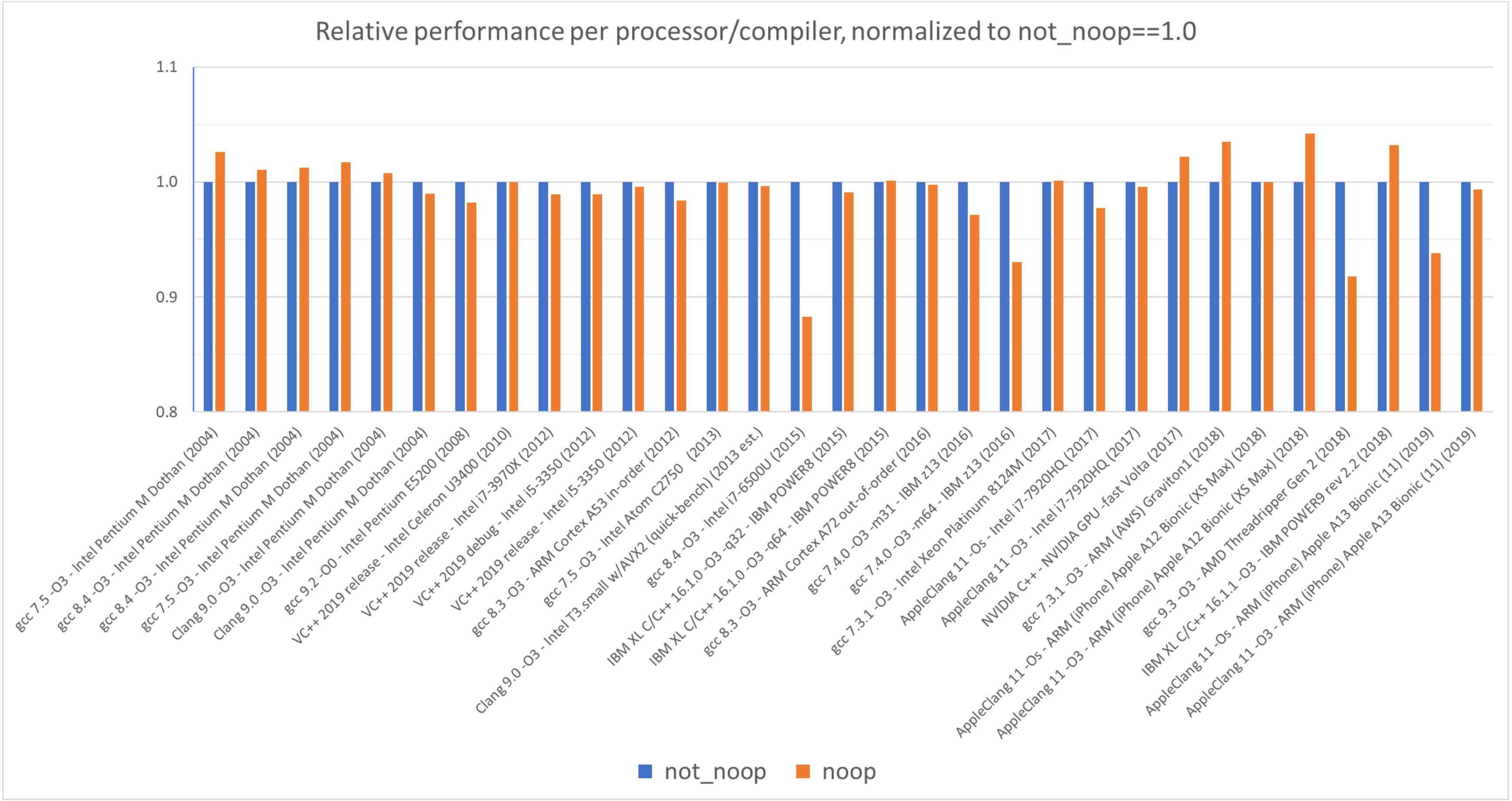


Now we can better see that `weird` does in fact vary from approximately twice as fast (by doing no work and just leaking the `int` half the time) to mostly-small overheads.

Outlier investigation: The notable outlier was the two runs on the Intel i7-6500U processor, where `weird` was unusually (consistently 2.1×) slower than any other processor compared to `not_noop`. **Test and resolution:** I consulted Andrew Kaylor at Intel, who conjectured that the Skylake conditional jump boundary issue, which affected some i7-6500U chips, likely slowed down the `weird` case, but not the `not_noop` case. We checked the processor ID of the machine on which this test was performed, and confirmed that it was one of the affected chips, and that there was not another likely explanation for a difference that big for `weird`. (For the same CPU, the small speedup of the `noop` case compared to `not_noop` was likely due to some other microarchitectural difference, not the conditional jump boundary issue; see the analysis of a similar small speedup in §2.2.1.)

Finally, removing both `nomove` and `weird` and magnifying the scale so that the differences between `not_noop` and `noop` are easier to see.

Recall that the `not_noop` case's entire body is one `delete` of an `int*` followed by three `int*` pointer assignments.



The `noop` with the branch was faster as often as it was slower, because the tests did not control for microarchitectural effects (e.g., TLB, instruction cache) which are the only observable cost and vary for the code generation of any function; the branch itself had no costs on the scale of a pipeline stall on any tested hardware.

2.3.2 Sample generated code for `not_noop` vs `noop`

Finally, for reference this section includes the generated code on various compilers for `not_noop` vs `noop`, with hand-annotated highlights for the instruction deltas.

This is mainly interesting to confirm that all do have a branch in the `noop` generated code.

For illustration purposes, it uses this sample:

```
void test(A& x, A& y) {
    x = std::move(y);
}
```

and documents the generated code using:

- Clang 9.0 -O3
- Clang 5.0 -O3
- GCC 10.1 -O3
- GCC 8.1 -O3
- MSVC 19.24 -O3

Finally, for completeness this section ends with a three-way side-by-side comparison of `not_noop`, `no_op`, and `weird` using GCC 8.1 -O3.

2.3.2.1 Clang 9.0 -O3

```
not_noop::test(not_noop::A&, not_noop::A&):


    push    r14
    push    rbx
    push    rax
    mov     r14, rsi
    mov     rbx, rdi
    mov     rdi, qword ptr [rdi]
    test    rdi, rdi
    je      .LBB0_2
    call    operator delete(void*)
.LBB0_2:
    mov     qword ptr [rbx], 0
    mov     rax, qword ptr [r14]
    mov     qword ptr [rbx], rax
    mov     qword ptr [r14], 0
    add     rsp, 8
    pop     rbx
    pop     r14

    ret
```

```
noop::test(noop::A&, noop::A&):
    cmp     rdi, rsi
    je      .LBB1_4
    push    r14
    push    rbx
    push    rax
    mov     r14, rsi
    mov     rbx, rdi
    mov     rdi, qword ptr [rdi]
    test    rdi, rdi
    je      .LBB1_3
    call    operator delete(void*)
.LBB1_3:

    mov     rax, qword ptr [r14]
    mov     qword ptr [rbx], rax
    mov     qword ptr [r14], 0
    add     rsp, 8
    pop     rbx
    pop     r14
.LBB1_4:
    ret
```

2.3.2.2 Clang 5.0 -O3

```
not_noop::test(not_noop::A&, not_noop::A&):
    push    r14
    push    rbx
    push    rax
    mov     r14, rsi
    mov     rbx, rdi


    mov     rdi, qword ptr [rbx]
    test    rdi, rdi
    je      .LBB0_2
    call    operator delete(void*)
.LBB0_2:
    mov     qword ptr [rbx], 0
    mov     rax, qword ptr [r14]
    mov     qword ptr [rbx], rax
    mov     qword ptr [r14], 0

    add     rsp, 8
    pop     rbx
    pop     r14
    ret
```

```
noop::test(noop::A&, noop::A&):
    push    r14
    push    rbx
    push    rax
    mov     r14, rsi
    mov     rbx, rdi
    cmp     rbx, r14
    je      .LBB1_4
    mov     rdi, qword ptr [rbx]
    test    rdi, rdi
    je      .LBB1_3
    call    operator delete(void*)
.LBB1_3:

    mov     rax, qword ptr [r14]
    mov     qword ptr [rbx], rax
    mov     qword ptr [r14], 0
.LBB1_4:
    add     rsp, 8
    pop     rbx
    pop     r14
    ret
```

2.3.2.3 gcc 10.1 -O3

```
not_noop::test(not_noop::A&, not_noop::A&):


    push    rbp
    mov     rbp, rsi
    push    rbx
    mov     rbx, rdi
    sub     rsp, 8
    mov     rdi, QWORD PTR [rdi]
    test    rdi, rdi
    je      .L2
    mov     esi, 4
    call    operator delete(void*, unsigned long)
.L2:
    mov     QWORD PTR [rbx], 0
    mov     rax, QWORD PTR [rbp+0]
    mov     QWORD PTR [rbx], rax
    mov     QWORD PTR [rbp+0], 0
    add     rsp, 8
    pop     rbx
    pop     rbp
    ret
```

```
noop::test(noop::A&, noop::A&):
    cmp     rsi, rdi
    je      .L15
    push    rbp
    mov     rbp, rdi
    push    rbx
    mov     rbx, rsi
    sub     rsp, 8
    mov     rdi, QWORD PTR [rdi]
    test    rdi, rdi
    je      .L10
    mov     esi, 4
    call    operator delete(void*, unsigned long)
.L10:

    mov     rax, QWORD PTR [rbx]
    mov     QWORD PTR [rbp+0], rax
    mov     QWORD PTR [rbx], 0
    add     rsp, 8
    pop     rbx
    pop     rbp
    ret
.L15:
    ret
```

2.3.2.4 gcc 8.1 -O3

```
not_noop::test(not_noop::A&, not_noop::A&):


    push    rbp
    mov     rbp, rsi
    mov     esi, 4
    push    rbx
    mov     rbx, rdi

    sub     rsp, 8
    mov     rdi, QWORD PTR [rdi]
    call    operator delete(void*, unsigned long)
    mov     QWORD PTR [rbx], 0
    mov     rax, QWORD PTR [rbp+0]
    mov     QWORD PTR [rbx], rax
    mov     QWORD PTR [rbp+0], 0
    add     rsp, 8
    pop     rbx
    pop     rbp
    ret
```

```
noop::test(noop::A&, noop::A&):
    cmp     rsi, rdi
    je      .L7
    push    rbp
    mov     rbp, rdi

    push    rbx
    mov     rbx, rsi
    mov     esi, 4
    sub     rsp, 8
    mov     rdi, QWORD PTR [rdi]
    call    operator delete(void*, unsigned long)

    mov     rax, QWORD PTR [rbx]
    mov     QWORD PTR [rbp+0], rax
    mov     QWORD PTR [rbx], 0
    add     rsp, 8
    pop     rbx
    pop     rbp
    ret
.L7:
    ret
```

2.3.2.5 MSVC 19.24 -O3

```
not_noop::A & not_noop::A::operator=(not_noop::A &&)
$LN3:
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    sub     rsp, 56 ; 00000038H



    mov     rax, QWORD PTR this$[rsp]
    mov     rax, QWORD PTR [rax]
    mov     QWORD PTR $T1[rsp], rax
    mov     edx, 4
    mov     rcx, QWORD PTR $T1[rsp]
    call    void operator delete(void *,unsigned __int64)
    mov     rax, QWORD PTR this$[rsp]
    mov     QWORD PTR [rax], 0
    mov     rax, QWORD PTR this$[rsp]
    mov     rcx, QWORD PTR a$[rsp]
    mov     rcx, QWORD PTR [rcx]
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR a$[rsp]
    mov     QWORD PTR [rax], 0

    mov     rax, QWORD PTR this$[rsp]
    add     rsp, 56 ; 00000038H
    ret     0
```

```
noop::A & noop::A::operator=(noop::A &&)
$LN4:
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    sub     rsp, 56 ; 00000038H
    mov     rax, QWORD PTR a$[rsp]
    cmp     QWORD PTR this$[rsp], rax
    je      SHORT $LN2@operator
    mov     rax, QWORD PTR this$[rsp]
    mov     rax, QWORD PTR [rax]
    mov     QWORD PTR $T1[rsp], rax
    mov     edx, 4
    mov     rcx, QWORD PTR $T1[rsp]
    call    void operator delete(void *,unsigned __int64)
    mov     rax, QWORD PTR this$[rsp]


    mov     rcx, QWORD PTR a$[rsp]
    mov     rcx, QWORD PTR [rcx]
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR a$[rsp]
    mov     QWORD PTR [rax], 0
$LN2@operator:
    mov     rax, QWORD PTR this$[rsp]
    add     rsp, 56 ; 00000038H
    ret     0
```

2.3.2.6 gcc 8.1 -O3 with “weird”

No branch	Identity branch	“Weird” randomish branch
`not_noop::test(not_noop::A&, not_noop::A&):`	`noop::test(noop::A&, noop::A&):`	`weird::test(weird::A&, weird::A&&):`
	`    cmp     rsi, rdi`	`    test    edi, 256`
	`    je      .L7`	`    jne     .L17`
		`    ret`
		`.L17:`
`    push    rbp`	`    push    rbp`	`    push    rbp`
`    mov     rbp, rsi`	`    mov     rbp, rdi`	`    mov     rbp, rsi`
`    mov     esi, 4`		`    mov     esi, 4`
`    push    rbx`	`    push    rbx`	`    push    rbx`
`    mov     rbx, rdi`	`    mov     rbx, rsi`	`    mov     rbx, rdi`
	`    mov     esi, 4`	
`    sub     rsp, 8`	`    sub     rsp, 8`	`    sub     rsp, 8`
`    mov     rdi, QWORD PTR [rdi]`	`    mov     rdi, QWORD PTR [rdi]`	`    mov     rdi, QWORD PTR [rdi]`
`    call    operator delete(void*, unsigned long)`	`    call    operator delete(void*, unsigned long)`	`    call    operator delete(void*, unsigned long)`
`    mov     QWORD PTR [rbx], 0`		
`    mov     rax, QWORD PTR [rbp+0]`	`    mov     rax, QWORD PTR [rbx]`	`    mov     rax, QWORD PTR [rbp+0]`
`    mov     QWORD PTR [rbx], rax`	`    mov     QWORD PTR [rbp+0], rax`	`    mov     QWORD PTR [rbx], rax`
`    mov     QWORD PTR [rbp+0], 0`	`    mov     QWORD PTR [rbx], 0`	`    mov     QWORD PTR [rbp+0], 0`
`    add     rsp, 8`	`    add     rsp, 8`	`    add     rsp, 8`
`    pop     rbx`	`    pop     rbx`	`    pop     rbx`
`    pop     rbp`	`    pop     rbp`	`    pop     rbp`
`    ret`	`    ret`	`    ret`
	`.L7:`	
	`    ret`	

3 Conclusions

Everyone understands branches can be expensive. However, all branches are not created equal: Some are very expensive, some are less expensive, some are intermittently expensive depending on the data being processed…

… and some appear to be free. It appears that the `if(this!=&rhs)` branch is free in practice on all tested processors, because it has special properties. These properties appear to be:

- **The branch virtually always takes the same path.** All predictors, even on weak hardware, easily predict branches correctly that have runs of equal truth values, after seeing the first handful of answers that go the same way. This is especially true of nearly-always-true and nearly-always-false branches.
- **The branch condition uses only memory load results that are already used in the taken path, and that already dominate the memory cost of the function.** The `this` and `rhs` arguments are not complex expressions, they are two raw pointer values that are already hot in registers/stack from the call site as part of the function call stack frame, where even if those are speculated their values are immediately required to execute the rest of the function body which contains further loads/stores to members of both objects that are address-dependent on the pointers' values. So the branch condition relies on strictly less information (the pointers' inequality) than is already required to execute the function body (their specific values and their dependent loads/stores), and the function's cost with or without the branch is already dominated by the cost of those memory operations. This means that, by construction, the presence of the branch cannot incur a pipeline stall or other memory effect that is not already being incurred anyway.
- **The pointers-equal path is empty.** On CPUs the empty branch fits easily into the instruction cache and the branch target buffer, so a processor that runs both speculatively will be able to fit them alongside each other. On GPUs the instruction stream is still linear, just a couple of instructions longer.

Because of these qualitative properties by construction, the measured free cost of this test appears to be not an accident, but a property we can expect in principle on all available processors.

———

This paper reports failure to measure overhead for the worst case cost of guaranteeing that self-move is a no-op, namely a brute-force identity test on available hardware and/or any standard library implementation. This paper provides a set of runnable microbenchmarks for further experimentation, so others can try to discover measurable overhead for some type on some hardware. However, interpreting performance measurements is notoriously difficult, and a change that has no observable local cost could have macro costs when added throughout a system; this paper did not measure adding the test throughout a large program.

Conclusion 1. Branches with the three characteristics above — with one alternative empty, virtually always taking the same path, and testing a condition that uses only memory locations that will be used anyway by the nonempty path — should be considered cost-free on all known hardware.

Conclusion 2. Language design decisions should be decided on their other merits, but not considering concern about potential performance cost of a branch for an identity test (e.g., as currently required for copy assignment) or not-null test (e.g., as currently required for `delete` and `free`). Those tests should be considered free.