A Performance Analysis of a Simple Trading System...

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

[1]Count-Zero Limited

BCS APSG, London, 2019

Outline

HFT & Low-Latency: Issues

- HFT & low-latency trading are performance-critical, obviously:
  - provides edge in the market over competition, faster is better.
- Is not rocket-science:
  - Not safety-critical: it's not aeroplanes, rockets nor reactors!
  - Perverse: to be truly fast is to do nothing!
  - It is message passing, copying bytes
    - perhaps with validation, aka risk-checks.
- It requires low-level control:
  - Of the hardware & software that interacts with it intimately.
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

HFT & Low-Latency: Issues

- HFT & low-latency trading are performance-critical, obviously:
  - provides edge in the market over competition, faster is better.
- Is not rocket-science:
  - Not safety-critical: it's not aeroplanes, rockets nor reactors!
  - Perverse: to be truly fast is to do nothing!
  - It is message passing, copying bytes
    - perhaps with validation, aka risk-checks.
- It requires low-level control:
  - Of the hardware & software that interacts with it intimately.
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

HFT & Low-Latency: Issues

- HFT & low-latency trading are performance-critical, obviously:
  - provides edge in the market over competition, faster is better.
- Is not rocket-science:
  - Not safety-critical: it's not aeroplanes, rockets nor reactors!
  - Perverse: to be truly fast is to do nothing!
  - It is message passing, copying bytes
    - perhaps with validation, aka risk-checks.
- It requires low-level control:
  - Of the hardware & software that interacts with it intimately.
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

HFT & Low-Latency: Issues

- HFT & low-latency trading are performance-critical, obviously:
  - provides edge in the market over competition, faster is better.
- Is not rocket-science:
  - Not safety-critical: it's not aeroplanes, rockets nor reactors!
  - Perverse: to be truly fast is to do nothing!
  - It is message passing, copying bytes
    - perhaps with validation, aka risk-checks.
- It requires low-level control:
  - Of the hardware & software that interacts with it intimately.
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

Much more to low-latency software than just C++:

- Hardware needs to be considered:
  - multiple-processors (one for O/S, one for the gateway),
  - bus per processor; cores dedicated to tasks,
  - network infrastructure (including co-location for highest performance), etc.
  - *And any bugs that may be found...*
  - Software issues confound:
    - which O/S, not all distributions are equal,
    - tool-set support is necessary for rapid development,
    - configuration needed: c-groups/isolcpu, performance tuning.
- Not all compilers, or even versions, are equal...
  - Which is faster clang, g++ or icc?
    - Focus: g++, mainly C++17, also clang v4 - v7 & some icc.

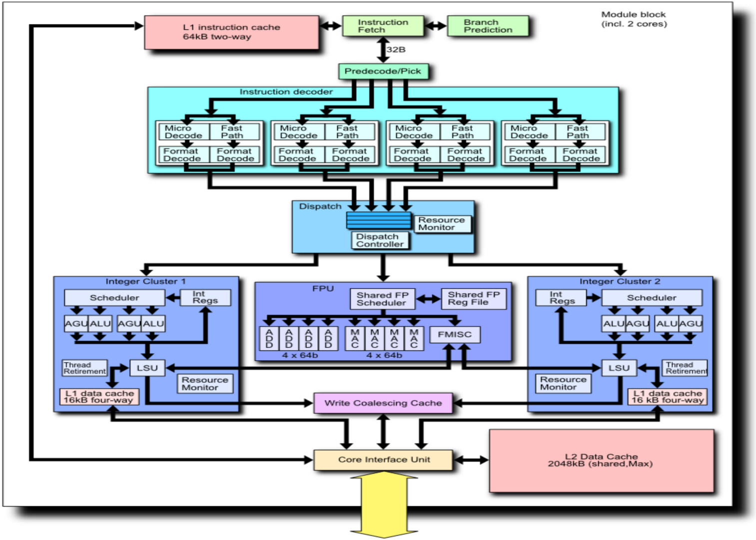
Much more to low-latency software than just C++:

- Hardware needs to be considered:
  - multiple-processors (one for O/S, one for the gateway),
  - bus per processor; cores dedicated to tasks,
  - network infrastructure (including co-location for highest performance), etc.
  - *And any bugs that may be found...*
  - Software issues confound:
    - which O/S, not all distributions are equal,
    - tool-set support is necessary for rapid development,
    - configuration needed: c-groups/isolcpu, performance tuning.
- Not all compilers, or even versions, are equal...
  - Which is faster clang, g++ or icc?
    - Focus: g++, mainly C++17, also clang v4 - v7 & some icc.

Much more to low-latency software than just C++:

- Hardware needs to be considered:
    - multiple-processors (one for O/S, one for the gateway),
    - bus per processor; cores dedicated to tasks,
    - network infrastructure (including co-location for highest performance), etc.
    - *And any bugs that may be found...*
    - Software issues confound:
        - which O/S, not all distributions are equal,
        - tool-set support is necessary for rapid development,
        - configuration needed: c-groups/isolcpu, performance tuning.
- Not all compilers, or even versions, are equal...
    - Which is faster clang, g++ or icc?
        - Focus: g++, mainly C++17, also clang v4 - v7 & some icc.

AMD Bulldozer, circa 2013.



...And the BUGS: "The Spectre of Meltdown": An Overview.

- Meltdown [8]:
  - Extremely briefly: "Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations ... Out-of-order execution is an indispensable performance feature..."
- Spectre [9], amongst other variants:
  - Extremely briefly: "Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary."
- Billions of devices affected, incl. Intel ***& AMD architectures***.
- Mitigation via kernel patches is critical to avoid attack (verified using [10]).

Optimization Case Studies.

- Despite the above, we choose to use C++,
  - which we will need to *optimize*,
  - shall examine influence of compiler, O/S & hardware.
- Optimizing C++: non-trivial; from [1] the examples I chose:
  - Performance quirks in compiler versions. (Warm-up.)
  - Static branch-prediction: use and abuse.
  - Switch-statements: can these be optimized?
  - Extreme templating: the case of `memcpy()`.
  - Put it together: A full FIX-to-MIT/BIT exchange translator.

Optimization Case Studies.

- Despite the above, we choose to use C++,
  - which we will need to *optimize*,
  - shall examine influence of compiler, O/S & hardware.
- Optimizing C++: non-trivial; from [1] the examples I chose:
  - Performance quirks in compiler versions. (Warm-up.)
  - Static branch-prediction: use and abuse.
  - Switch-statements: can these be optimized?
  - Extreme templating: the case of `memcpy()`.
  - Put it together: A full FIX-to-MIT/BIT exchange translator.

Optimization Case Studies.

- Despite the above, we choose to use C++,
  - which we will need to *optimize*,
  - shall examine influence of compiler, O/S & hardware.
- Optimizing C++: non-trivial; from [1] the examples I chose:
  - Performance quirks in compiler versions. (Warm-up.)
  - Static branch-prediction: use and abuse.
  - Switch-statements: can these be optimized?
  - Extreme templating: the case of `memcpy()`.
  - Put it together: A full FIX-to-MIT/BIT exchange translator.

Performance quirks in compiler versions.

- Compilers normally improve with versions, don't they?

Example code, using `-O3 -march=native`:

```
#include <string.h>
static const char src[20]="0123456789ABCDEFGHI";
char dest[20];
void foo() {
    memcpy(dest, src, sizeof(src));
}
```

Comparison of code generation in g++.

v4.4.7:

```
foo():
    movabsq $3978425819141910832, %rdx
    movabsq $5063528411713059128, %rax
    movl $4802631, dest+16(%rip)
    movq %rdx, dest(%rip)
    movq %rax, dest+8(%rip)
    ret
dest:  .zero 20
```

v4.7.3:

```
foo():
    movq src(%rip), %rax
    movq %rax, dest(%rip)
    movq src+8(%rip), %rax
    movq %rax, dest+8(%rip)
    movl src+16(%rip), %eax
    movl %eax, dest+16(%rip)
    ret
dest:  .zero 20
src:
    .string "0123456789ABCDEFGHI"
```

- g++ v4.4.7 schedules the movabsq sub-optimally.
- g++ v4.7.3 does not use any SSE instructions, and uses the stack, so is sub-optimal.

Comparison of code generation in g++.

v4.8.1 - v6.3.0:

```
foo():
    movabsq
$3978425819141910832, %rax
    movl $4802631,
dest+16(%rip)
    movq %rax, dest(%rip)
    movabsq
$5063528411713059128, %rax
    movq %rax,dest+8(%rip)
    ret
dest:   .zero 20
```

v7.0.0 - v7.3.0:

```
foo():
    vmovdqa xmm0, XMMWORD PTR
.LC0[rip]
    mov DWORD PTR
dest[rip+16], 4802631
    vmovaps XMMWORD PTR
dest[rip], xmm0
    ret
dest:   .zero 20
,LC0:
    .quad 3978425819141910832
    .quad 5063528411713059128
```

v8.1.0 - v8.3.0:

```
foo():
    vmovdqa xmm0, XMMWORD PTR
src[rip]
    mov eax, DWORD PTR
src[rip+16]
    vmovaps XMMWORD PTR
dest[rip], xmm0
    mov DWORD PTR
dest[rip+16], eax
    ret
dest:   .zero 20
src:   .string
"0123456789ABCDEFGHI"
```

- g++ v4.8.1-v6.3.0: notice SSE instructions are better scheduled, stack not used.
- g++ v7.0.0-v7.3.0: stack & AVX2 used: sub-optimal;
- g++ v8.1.0-v8.3.0: extra stack accesses: looks worse.
- Very unstable output - highly dependent upon version.

Comparison of code generation in icc & clang.

icc v13.0.1-v17:

```
foo():
    vmovups xmm0, XMMWORD PTR
src[rip]
    vmovups XMMWORD PTR
dest[rip], xmm0
    mov eax, DWORD PTR
16+src[rip]
    mov DWORD PTR
16+dest[rip], eax
    ret
dest:
src:
    .long 858927408
    XXXsnipXXX
    .long 4802631
```

icc v18 - v19:

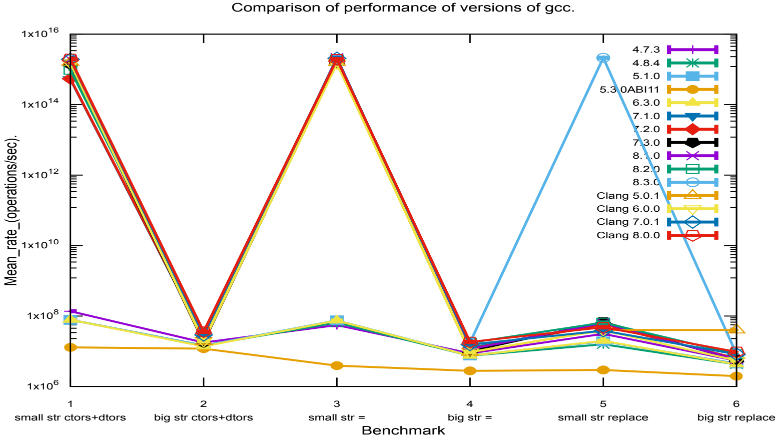
```
foo():
    vmovups xmm0, XMMWORD PTR
src[rip]
    mov eax, DWORD PTR
16+src[rip]
    vmovups XMMWORD PTR
dest[rip], xmm0
    mov DWORD PTR
16+dest[rip], eax
    ret
dest:
src:
    .long 858927408
    XXXsnipXXX
    .long 4802631
```

clang 3.5.0-7.0.1:

```
foo():  # @foo()
    vmovaps src(%rip), %xmm0
    vmovaps %xmm0, dest(%rip)
    movl $4802631,
dest+16(%rip)
    retq
dest:
    .zero 20
src:
    .asciz
"0123456789ABCDEFGHI"
```

- Note fewer instructions, but use of the stack - increases pressure on the data cache, etc with memory-loads.
- clang has very stable output compared to icc & g++.

Does this matter in reality?



- Hope that performance improves with compiler version...
  - This is not always so: there can be significant differences!

Static branch-prediction: use and abuse.

- Which comes first? The `if() bar1()` or the `else bar2()`?
- Intel [2], ARM [4] & AMD differ: older architectures use BTFNT rule [3, 5].
  - Backward-Taken: for loops that jump backwards. (Not discussed in this talk.)
  - Forward-Not-Taken: for `if-then-else`.
  - Intel added the 0x2e & 0x3e prefixes, but no longer used.
  - But super-scalar architectures still suffer costs of mis-prediction & research into predictors is on-going and highly proprietary.
- `__builtin_expect()` was introduced that emitted these prefixes, now just used to guide the compiler.
- The fall-through should be `bar1()`, not `bar2()`!

Static branch-prediction: use and abuse.

- Which comes first? The `if() bar1()` or the `else bar2()`?
- Intel [2], ARM [4] & AMD differ: older architectures use BTFNT rule [3, 5].
  - Backward-Taken: for loops that jump backwards. (Not discussed in this talk.)
  - Forward-Not-Taken: for `if-then-else`.
  - Intel added the 0x2e & 0x3e prefixes, but no longer used.
  - But super-scalar architectures still suffer costs of mis-prediction & research into predictors is on-going and highly proprietary.
- `__builtin_expect()` was introduced that emitted these prefixes, now just used to guide the compiler.
- The fall-through should be `bar1()`, not `bar2()`!

Static branch-prediction: use and abuse.

- Which comes first? The `if() bar1()` or the `else bar2()`?
- Intel [2], ARM [4] & AMD differ: older architectures use BTFNT rule [3, 5].
  - Backward-Taken: for loops that jump backwards. (Not discussed in this talk.)
  - Forward-Not-Taken: for `if-then-else`.
  - Intel added the 0x2e & 0x3e prefixes, but no longer used.
  - But super-scalar architectures still suffer costs of mis-prediction & research into predictors is on-going and highly proprietary.
- `__builtin_expect()` was introduced that emitted these prefixes, now just used to guide the compiler.
- The fall-through should be `bar1()`, not `bar2()`!

So how well do compilers obey the BTFNT rule?

The following code was examined with various compilers:

```
extern void bar1();
extern void bar2();
void foo(bool i) {
    if (i) bar1();
    else bar2();
}
```

Generated Assembler using g++.

v4.8.2-v5.5 & v8.1-v8.3: at -O0 & -O1:

```
foo(bool):
    subq $8, %rsp
    testb %dil, %dil
    je .L2
    call bar1()
    jmp .L1
.L2:
    call bar2()
.L1:
    addq $8, %rsp
    ret
```

v6.1-v7.3: at -O0 & -O1:

```
foo(bool):
    subq $8, %rsp
    testb %dil, %dil
    je .L2
    call bar1()
    jmp .L1
.L2:
    call bar2()
.L1:
    addq $8, %rsp
    ret
```

v4.8.2-v7.3: at -O2 & -O3:

```
foo(bool):
    testb %dil, %dil
    jne .L4
    jmp bar2()
.L4:
    jmp bar1()
```

v8.1-v8.3: at -O2 & -O3:

```
foo(bool):
    testb %dil, %dil
    je .L2
    jmp bar1()
.L2:
    jmp bar2()
```

- ***Oh no!*** g++ switches the fall-through, so one can't ***consistently*** statically optimize branches in g++...[6]

Generated Assembler using ICC v13.0.1-v19.0.1 & Clang v3.8.0-7.0.1.

ICC at -O2 & -O3:

```
foo(bool):
    testb %dil, %dil
    je ..B1.3
    jmp bar1()
..B1.3:
..B1.1
    jmp bar2()
```

Clang at -O1, -O2 & -O3:

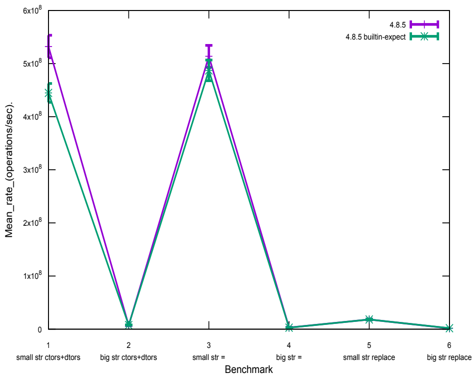
```
foo(bool):
    testb %dil, %dil
    je .LBB0_2
    jmp bar1()
.LBB0_2:
    jmp bar2()
```

- Lower optimization levels still order the calls to `bar[1|2]()` in the same manner, but the code is unoptimized.
- ***BUT at -O2 & -O3 g++ reverses the order of the calls compared to clang & icc!!!***
  - ***Impossible to optimize for g++ and other compilers!***

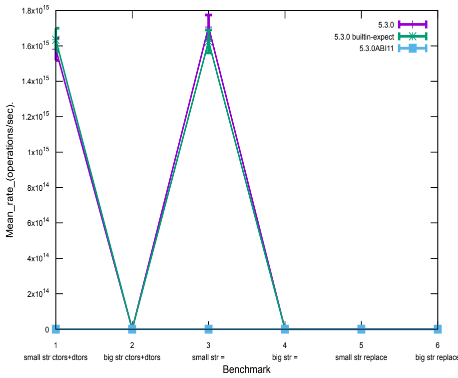
Test `__builtin_expect(i, 1)` with g++ v4.8.5-v5.3.0.

- BUT: Added to the dtor: caused a slowdown in the ctor-dtor test!

Comparison of effect of --builtin-expect using gcc v4.8.5 and -std=c++11.

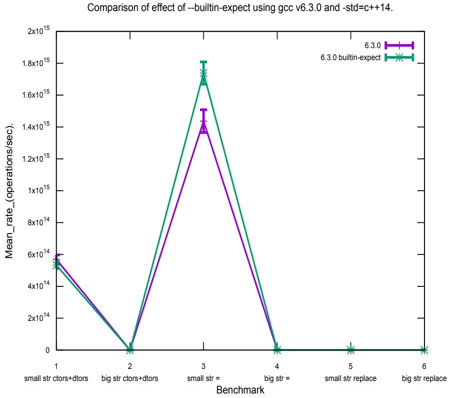


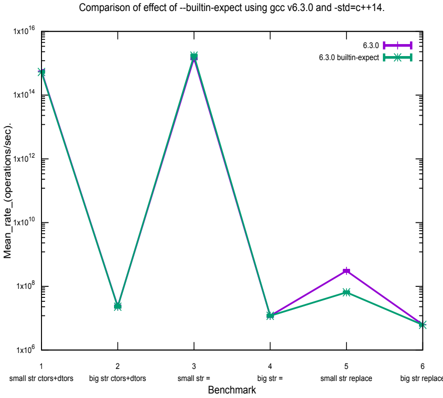
Comparison of effect of --builtin-expect using gcc v5.3.0 and -std=c++14.



Test `__builtin_expect(i, 1)` with g++ v6.3.0.

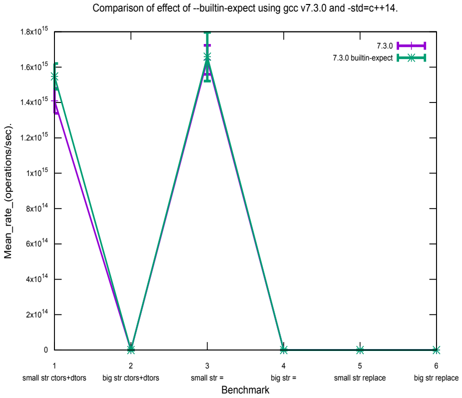
- BUT: Added to the dtor: caused a slowdown in small-string-replace!

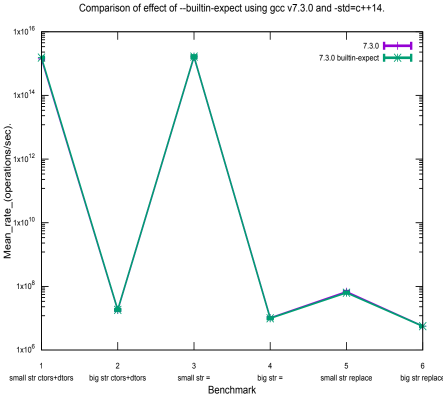




Test __builtin_expect(i, 1) with g++ v7.3.0.

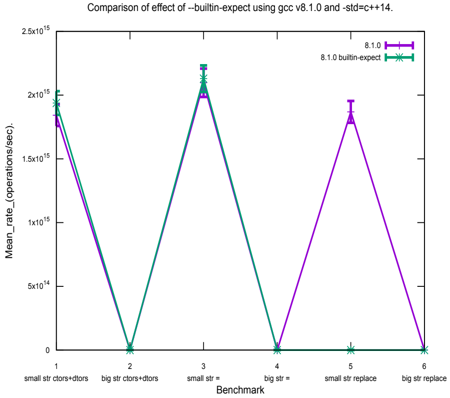
- BUT: Added to the dtor: no statistical effect!

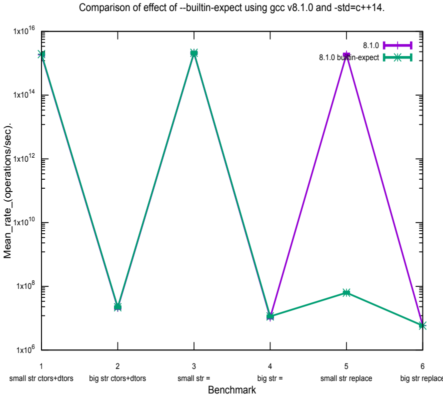




Test __builtin_expect(i, 1) with g++ v8.1.0.

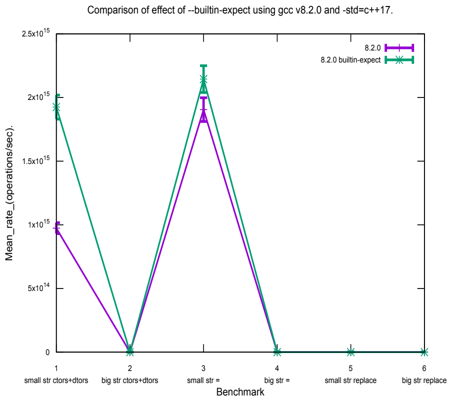
- BUT: Added to the dtor: confounded optimiser in small-string-replace!

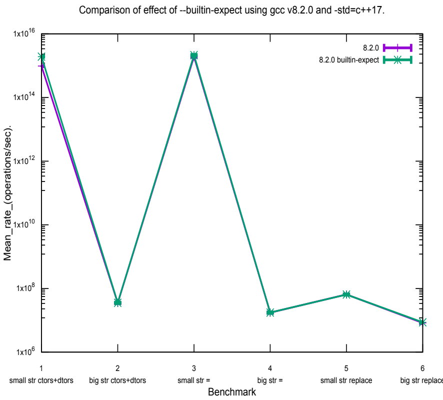




Test __builtin_expect(i, 1) with g++ v8.2.0.

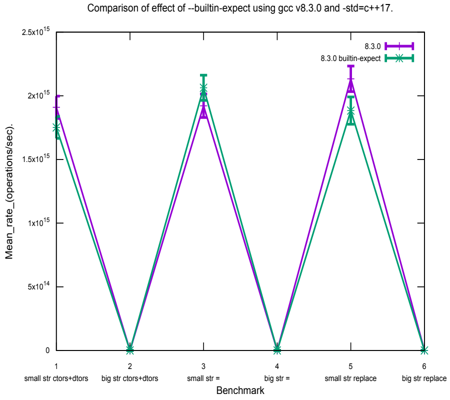
- Note how change in fall-though has no dramatic effect on performance.

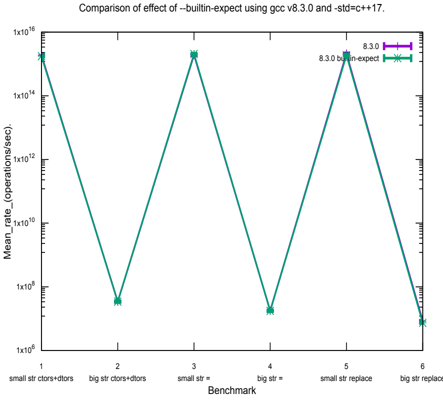




Test `__builtin_expect(i, 1)` with g++ v8.3.0.

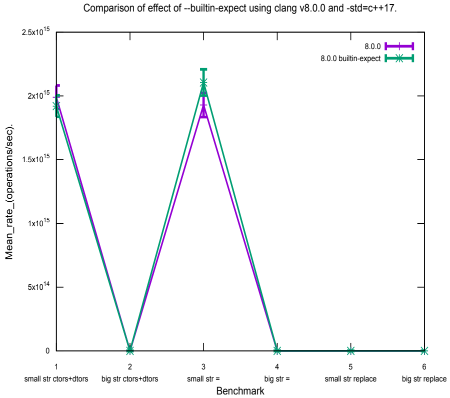
- Note improved optimiser.

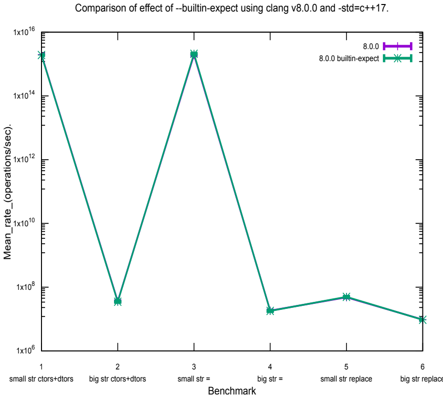




Test `__builtin_expect(i, 1)` with clang v6.0.0 - v8.0.0.

- Highly consistent results.





Does a `switch`-statement have a preferential case-label?

- Common lore seems to indicate that either the first case-label or the `default` are somehow the statically predicted fall-through.
- For non-contiguous labels in clang, g++ & icc this is not so.
    - g++ uses a decision-tree algorithm[7], basically case labels are clustered numerically, and the correct label is found using a binary-search.
        - clang & icc seem to be similar. I shall focus on g++ for this talk.
    - There is no static prediction!

Does a `switch`-statement have a preferential case-label?

- Common lore seems to indicate that either the first case-label or the `default` are somehow the statically predicted fall-through.
- For non-contiguous labels in clang, g++ & icc this is not so.
  - g++ uses a decision-tree algorithm[7], basically case labels are clustered numerically, and the correct label is found using a binary-search.
    - clang & icc seem to be similar. I shall focus on g++ for this talk.
  - There is no static prediction!

Does a `switch`-statement have a preferential case-label?

- Common lore seems to indicate that either the first case-label or the `default` are somehow the statically predicted fall-through.
- For non-contiguous labels in clang, g++ & icc this is not so.
  - g++ uses a decision-tree algorithm[7], basically case labels are clustered numerically, and the correct label is found using a binary-search.
    - clang & icc seem to be similar. I shall focus on g++ for this talk.
  - There is no static prediction!

What does this look like?

Example of simple non-contiguous labels.

```
extern bool bar1();
extern bool bar2();
extern bool bar3();
extern bool bar4();
extern bool bar5();
extern bool bar6();
bool foo(int i) {
    switch (i) {
        case 0:  return bar1();
        case 30:  return bar2();
        case 9:  return bar3();
        case 787:  return bar4();
        case 57689:  return bar5();
        default:  return bar6();
    }
}
```

- Contiguous labels cause a jump-table to be created.

g++ -O3 generated code, `__builtin_expect()` has no effect.

v5.3.0 - v7.3.0:

```
foo(int):
cmpl $30, %edi
je .L3
jg .L4
testl %edi, %edi
je .L5
cmpl $9, %edi
jne .L2
jmp bar3()
L4:
cmpl $787, %edi
je .L7
        cmpl $57689,
%edi
jne .L2
jmp bar5()
.L2:
jmp bar6()
.L7:
jmp bar4()
.L5:
jmp bar1()
.L3:
jmp bar2()
```

v8.1.0 - v8.3.0:

```
foo(int):
cmpl $30, %edi
je .L2
jle .L10
cmpl $787, %edi
je .L7
cmpl $57689, %edi
jne .L6
jmp bar5()
.L10:
testl %edi, %edi
je .L4
        cmpl $9, %edi
jne .L6
jmp bar3()
.L2:
jmp bar2()
.L7:
jmp bar4()
.L6:
jmp bar6()
.L4:
jmp bar1()
```

- Identical - it has no effect; gcc & icc are likewise unmodified.
- But clang v3.8.0 - v7.0.1 *is* affected by `__builtin_expect()` in the expected manner.

An obvious hack:

- One has to hoist the statically-predicted label out in an `if`-statement, and place the switch in the `else`.
    - Modulo what we now know about static branch prediction...Surely compilers simply "get this right"?

Let's go Mad...

- Can ***blatant*** templating make an even faster `memcpy()`?

Examined with various compilers with -O3 -std=c++17 -mavx.

```
template<
    std::size_t SrcSz, std::size_t DestSz, class Unit,
    std::size_t SmallestBuff=min<std::size_t, SrcSz, DestSz>::value,
    std::size_t Div=SmallestBuff/sizeof(Unit), std::size_t Rem=SmallestBuff%sizeof(Unit)
> struct aligned_unroller {
    // ...  An awful lot of template insanity.  Omitted to avoid being arrested.
};
template< std::size_t SrcSz, std::size_t DestSz > inline void constexpr
memcpy_opt(char const (&src)[SrcSz], char (&dest)[DestSz]) noexcept(true) {
    using unrolled_256_op_t=private_::aligned_unroller< SrcSz, DestSz, __m256i >;
    using unrolled_128_op_t=private_::aligned_unroller< SrcSz-unrolled_256_op_t::end,
DestSz-unrolled_256_op_t::end, __m128i >;
    // XXXsnipXXX
    // Unroll the copy in the hope that the compiler will notice the sequence of copies and
optimize it.
    unrolled_256_op_t::result(
        [&src, &dest](std::size_t i) {
            reinterpret_cast<__m256i*>(dest)[i]= reinterpret_cast<__m256i const *>(src)[i];
        }
    );
    // XXXsnipXXX
}
```

Assembly output from g++.

v4.9.0.

```
bar():
    movq s+32(%rip), %rax
    vmovdqa s(%rip), %ymm0
    vmovdqa %ymm0, d(%rip)
    movq %rax, d+32(%rip)
    vzeroupper
    ret
s:  .string "And for
something completely
different."
d:  .zero 40
```

v5.1.0 - v7.3.0.

```
bar():
    vmovups s+32(%rip), %ymm0
    movabsq
$13075866425910630, %rax
    vmovups %ymm0, d+32(%rip)
    movq %rax, d+32(%rip)
    vzeroupper
    ret
d:
s:  .string "And for
something completely
different."
```

v8.1.0 - v8.2.0.

```
bar():
    vmovaps 32+_ZL1s(%rip),
%ymm0
    movabsq
$7310016635654988832, %rax
    vmovaps %ymm0, 32+d(%rip)
    movq %rax, 32+d(%rip)
    movl $3044462, 40+d(%rip)
    vzeroupper
    ret
d:
s:  .string "And for
something completely
different."
```

- All look good apart from the stack usage.

Assembly output from clang v3.8.0-v7.0.1.

Assembler output.

```
.LCPI0_0:
    .long 1718182944
...
    .long 0
bar1():
    vmovaps .LCPI0_0(%rip), %ymm0
    vmovups %ymm0, d+32(%rip)
    movabsq $7310016635654988832, %rax
    movq %rax, d+32(%rip)
    movl $3044462, d+40(%rip)
    vzeroupper
    ret
d:  .zero 44
```

- Judicious use of micro-optimized templates *can* provide a performance enhancement.

Assembly output from icc -mavx & -std=c++11.

icc v13.0.1.

```
bar():
    movl $s, %eax #198.14
    movl $d, %ecx #198.17
    vmovdqu (%rax), %ymm0 #154.44
    vmovdqu %ymm0, (%rcx) #153.37
    movq 32(%rax), %rdx #166.44
    movq %rdx, 32(%rcx) #165.37
    vzeroupper #199.1
    ret #199.1
d:
s:  .byte 65
    ...
    .byte 0
```

icc v16.

```
bar():
    vmovups 32+s(%rip), %ymm0
    movq 32+s(%rip), %rax
    vmovups %ymm0, 32+d(%rip)
    movq %rax, 32+d(%rip)
    vzeroupper
    retq
d:
s:
```

- Use of micro-optimized templates *can* do unexpected things:
    - icc v16 produces good results.

Assembly output from icc -mavx & -std=c++14.

icc v17.

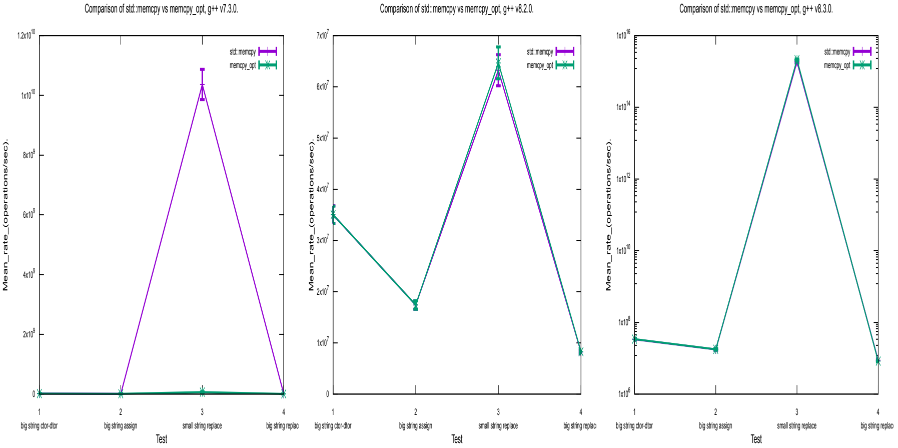
```
bar():
    movl $s, %edi
    movl $d, %esi
    jmp void memcpy_opt<40ul, 40ul>(char
const (&) [40ul], char (&) [40ul])
    vmovups 32(%rdi), %ymm0
    movq 32(%rdi), %rax
    vmovups %ymm0, 32(%rsi)
    movq %rax, 32(%rsi)
    vzeroupper
    ret
d:
s:
```

icc v18 - v19.0.1.

```
bar():
    vmovups 32+s(%rip), %xmm0
    movq 32+s(%rip), %rax
    movl 40+s(%rip), %edx
    vinsertf128 $1, 48+s(%rip), %ymm0,
%ymm1
    vmovups %xmm1, 32+d(%rip)
    vextractf128 $1, %ymm1, 48+d(%rip)
    movq %rax, 32+d(%rip)
    movl %edx, 40+d(%rip)
    vzeroupper
    ret
d:
s:
```

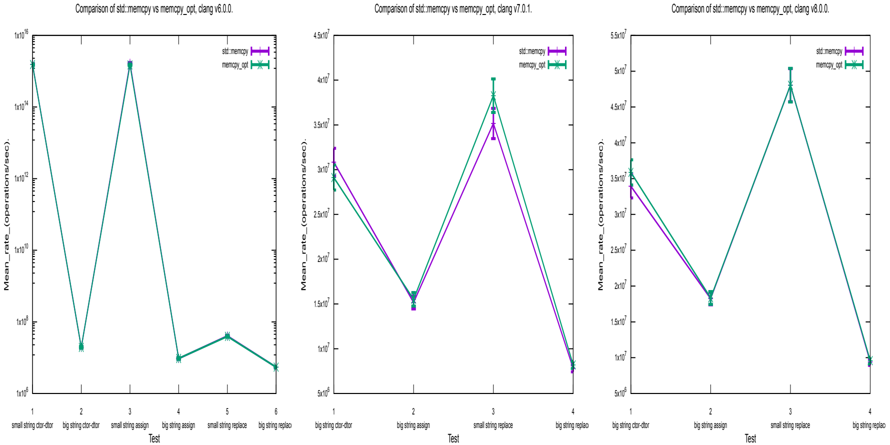
- Use of micro-optimized templates *can* do unexpected things:
  - icc v17 - v19 produce suboptimal results.

Performance: g++ (Warning: changes of scale.)



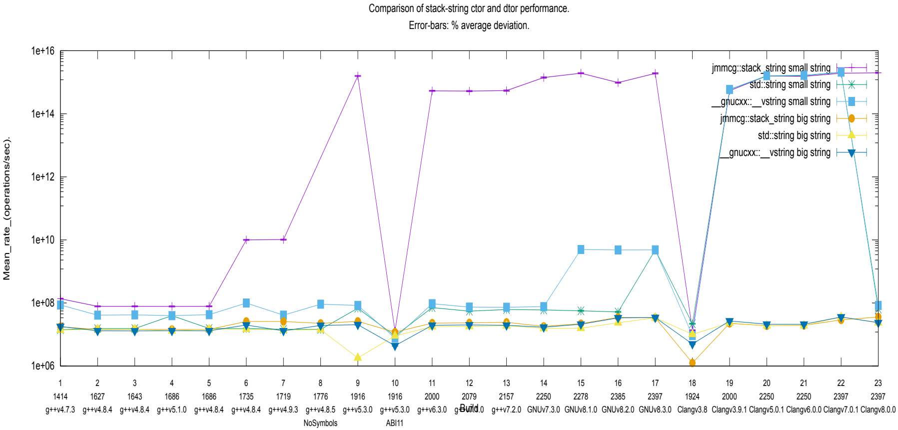
- optimizations confounded by use of the stack?
- Later version managed to optimise out a test case!

Performance: clang (Warning: changes of scale.)



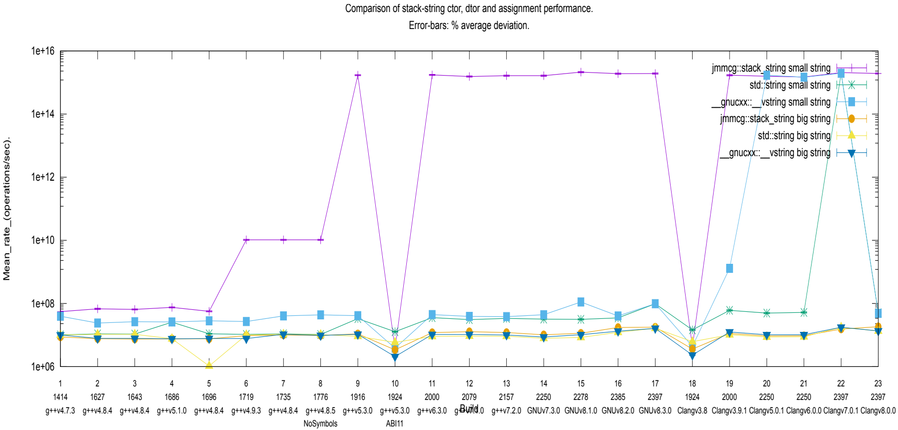
- Performance erratic over versions - poor scheduling?
  - 4180's: very slow vector units.
- More consistent than g++.

Part 1: Compiler version and performance.



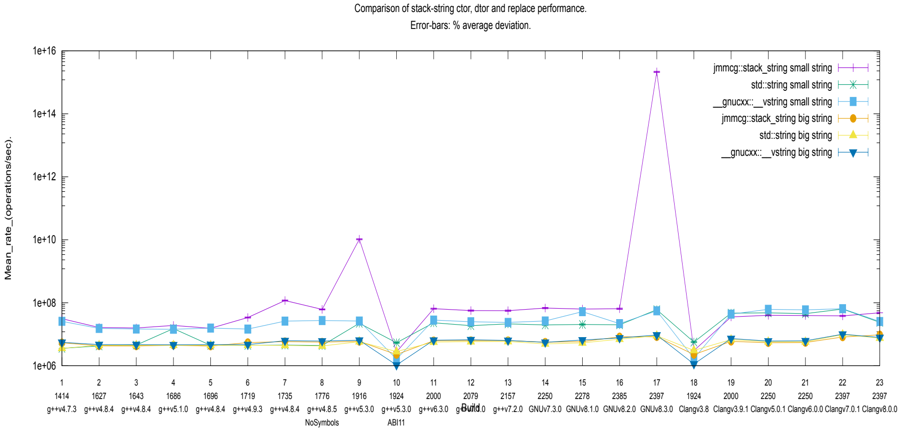
- Note highly inconsistent results.
- Occasional ability of compiler to optimise out test - ideal!

Part 2: Compiler version and performance.



- Clang optimises better than g++.

Part 3: Compiler version and performance.



A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
  - listens to socket (the client-side) for FIX format messages,
  - sends & receives binary-protocol MIT/BIT formats messages via a server-side socket.
- Uses Boost.ASIO, but many many other optimisations including the above used, SSE2 & higher instructions.
- Both Solarflare card & OpenOnload driver were not used.
  - Would have reduced context-switches.

A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
  - listens to socket (the client-side) for FIX format messages,
  - sends & receives binary-protocol MIT/BIT formats messages via a server-side socket.
- Uses Boost.ASIO, but many many other optimisations including the above used, SSE2 & higher instructions.
- Both Solarflare card & OpenOnload driver were not used.
  - Would have reduced context-switches.

A Simple FIX-to-MIT/BIT Translator.

- This translator is a heavily-templated library:
    - listens to socket (the client-side) for FIX format messages,
    - sends & receives binary-protocol MIT/BIT formats messages via a server-side socket.
- Uses Boost.ASIO, but many many other optimisations including the above used, SSE2 & higher instructions.
- Both Solarflare card & OpenOnload driver were not used.
    - Would have reduced context-switches.

Details of the Test.

- 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 both quiet & busy, with & without numactl.
    - Highly optimised kernel.
        - Dual AMD 4180 at 2.6GHz: old, slow (particularly SSE etc).

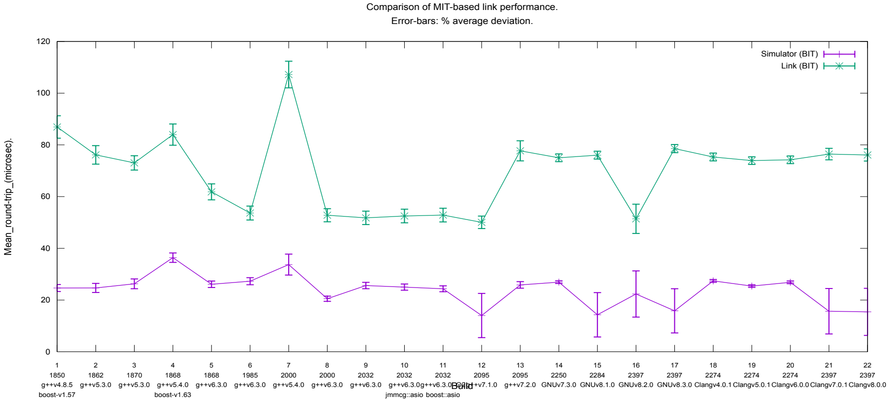
Details of the Test.

- 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 both quiet & busy, with & without numactl.
  - Highly optimised kernel.
    - Dual AMD 4180 at 2.6GHz: old, slow (particularly SSE etc).

Details of the Test.

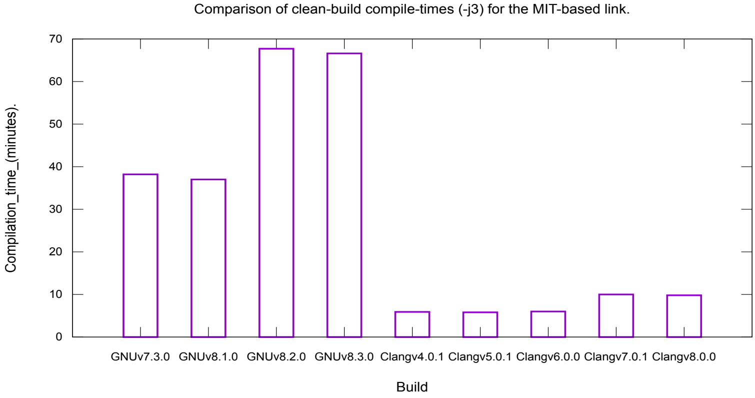
- A FIX New Order message is sent to a socket,
  - translated to MIT/BIT native binary format,
    - sent over sockets to a basic simulator,
    - which responds with a fill,
  - translated back to a FIX fill message.
- Sent back to the client.
- Computer was both quiet & busy, with & without numactl.
  - Highly optimised kernel.
    - Dual AMD 4180 at 2.6GHz: old, slow (particularly SSE etc).

Software optimisations, compiler versions.



- g++ v7.2.0 - v8.3.0, apart from v8.2.0, are much worse.
- clang performans c.f. v7.3.0 - v8.3.0 (not good).

Comparison of compilation times.



- g++ is much slower than clang for heavily templated code.
- Latter versions modified code but consistent result.

Getting Clang to compile...

- `libcxx` fails to link due to ABI.
  - Would need to rebuild all 3rd party - life too short.
- `libstdc++` has issues; clang detects this DR:

`/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/bits/hashtable_policy.h`

"Broken"

```
// Helper type used to detect whether the
hash functor is noexcept.
template<typename _Key, typename _Hash>
struct __is_noexcept_hash :
std::__bool_constant<
    noexcept(declval<const
_Hash&>()(declval<const _Key&>()))
> { };
```

"Fixed"

```
// Helper type used to detect whether the
hash functor is noexcept.
template<typename _Key, typename _Hash>
struct __is_noexcept_hash :
std::__bool_constant<
    false
> { };
```

- Brain-wave! Changed `noexcept(true)` to `noexcept` in a hash functor (bug in clang).

The Clang error novel (edited to fit)...

```
In file included from /usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/bits/hashtable.h:35:
/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/bits/hashtable_policy.h:87:11:  error:
exception specification is not available until end of class definition
    noexcept(declval<const _Hash&>()(declval<const _Key&>()))>
/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/type_traits:144:14:  note:  in
instantiation of template class 'std::__detail::__is_noexcept_hash<security_id_key,
hash_security_id_key>' requested here
    :  public conditional<_B1::value, _B2, _B1>::type
/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/type_traits:154:36:  note:  in
instantiation of template class 'std::__and_<std::__is_fast_hash<hash_security_id_key>,
std::__detail::__is_noexcept_hash<security_id_key, hash_security_id_key> >' requested here
    :  public __bool_constant<!bool(_Pp::value)>
/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/bits/unordered_map.h:46:34:  note:  in
instantiation of template class
'std::__not_<std::__and_<std::__is_fast_hash<hash_security_id_key>,
std::__detail::__is_noexcept_hash<security_id_key, hash_security_id_key> > >' requested here
    typename _Tr = __umap_traits<__cache_default<_Key, _Hash>::value>>
/usr/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include/g++-v7/bits/unordered_map.h:103:15:  note:  in
instantiation of default argument for '__umap_hashtable<security_id_key, int,
hash_security_id_key, std::equal_to<security_id_key>, std::allocator<std::pair<const
security_id_key, int> > >' required here
    typedef __umap_hashtable<_Key, _Tp, _Hash, _Pred, _Alloc> _Hashtable;
```

O/S & Hardware Choices (all used gcc v7.3.0).

- Two of the most commonly-used OSes were examined:
  1. CentOS (common - stock ISO image, not tuned):
     - Used a lot in finance, e.g. merchant banks & hedge funds.
     - A proxy for RedHat, Scientific Linux, etc.
  2. Ubuntu (common - stock ISO image, not tuned):
     - Much used on client desktops, etc.
  3. Gentoo (expert/crafty use):
     - Customised, heavily optimised, striped-down.
- Used overclocked (4.2GHz) Haswell: still in production.
  - Firmware patches not applied.
  - Newer Skylakes are not so heavily tuned to HFT.
- Recall both Solarflare card & OpenOnload driver were not used.
  - Also would reduce impact of mitigations.
  - OpenOnload often not used! (Simplifies deployment/not available for kernel version.)

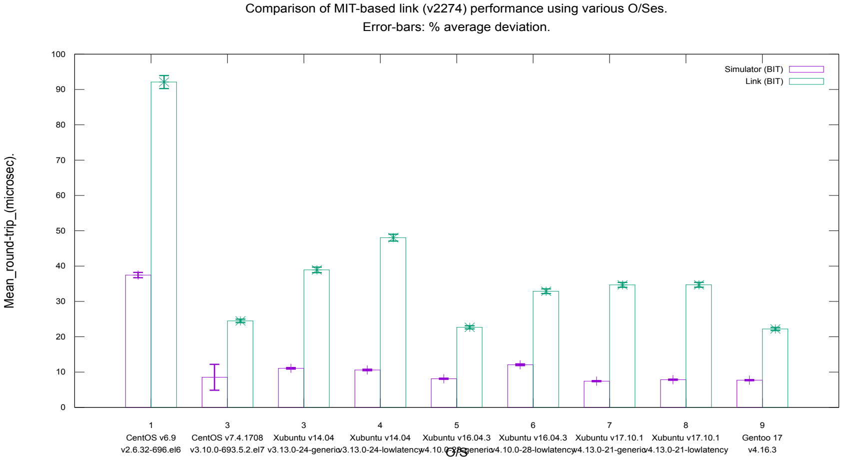
O/S & Hardware Choices (all used gcc v7.3.0).

- Two of the most commonly-used OSes were examined:
  1. CentOS (common - stock ISO image, not tuned):
     - Used a lot in finance, e.g. merchant banks & hedge funds.
     - A proxy for RedHat, Scientific Linux, etc.
  2. Ubuntu (common - stock ISO image, not tuned):
     - Much used on client desktops, etc.
  3. Gentoo (expert/crafty use):
     - Customised, heavily optimised, striped-down.
- Used overclocked (4.2GHz) Haswell: still in production.
  - Firmware patches not applied.
  - Newer Skylakes are not so heavily tuned to HFT.
- Recall both Solarflare card & OpenOnload driver were not used.
  - Also would reduce impact of mitigations.
  - OpenOnload often not used! (Simplifies deployment/not available for kernel version.)

O/S & Hardware Choices (all used gcc v7.3.0).

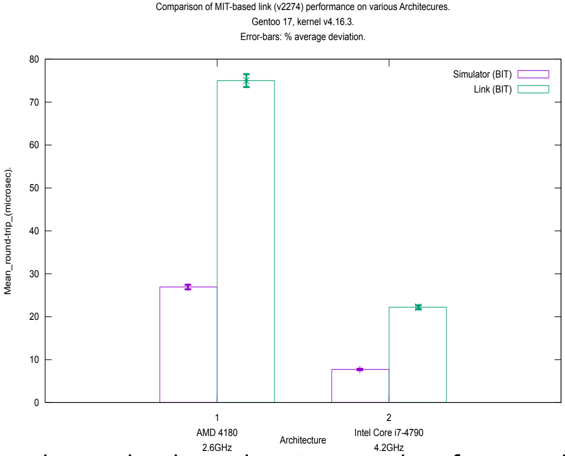
- Two of the most commonly-used OSes were examined:
  1. CentOS (common - stock ISO image, not tuned):
     - Used a lot in finance, e.g. merchant banks & hedge funds.
     - A proxy for RedHat, Scientific Linux, etc.
  2. Ubuntu (common - stock ISO image, not tuned):
     - Much used on client desktops, etc.
  3. Gentoo (expert/crafty use):
     - Customised, heavily optimised, striped-down.
- Used overclocked (4.2GHz) Haswell: still in production.
  - Firmware patches not applied.
  - Newer Skylakes are not so heavily tuned to HFT.
- Recall both Solarflare card & OpenOnload driver were not used.
  - Also would reduce impact of mitigations.
  - OpenOnload often not used! (Simplifies deployment/not available for kernel version.)

Impact of O/S.



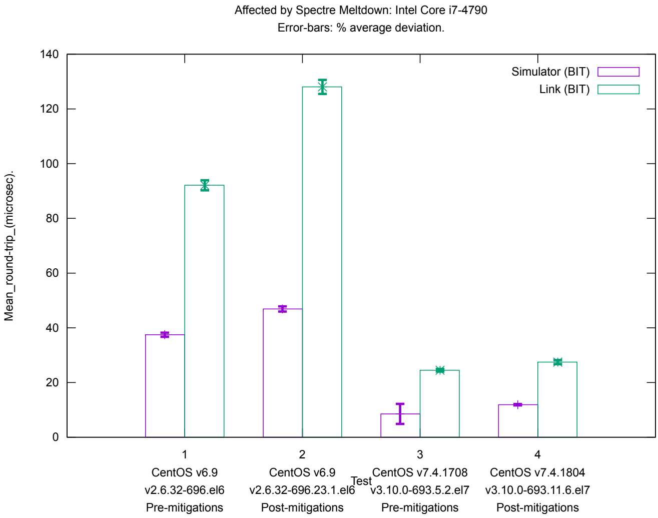
- ***WOW! Major impact on performance!***

Impact of Hardware.

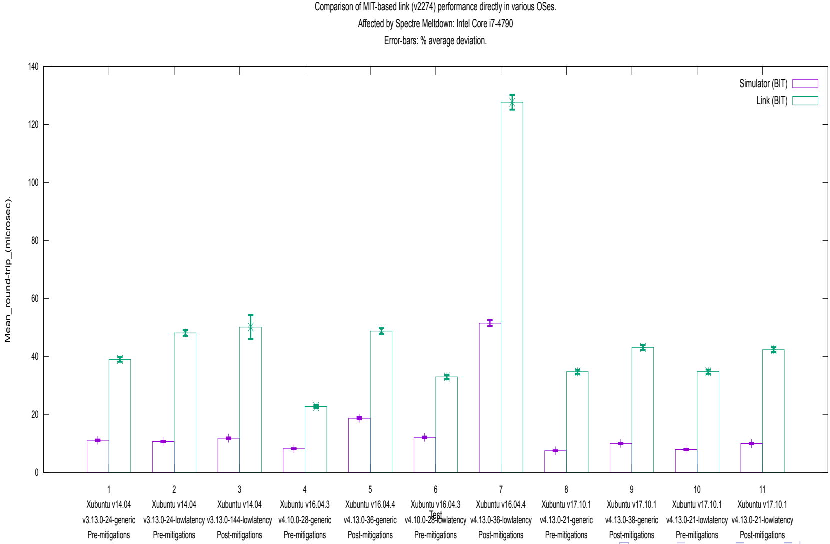


- Expected: new hardware has *improved* performance!
    - More than by clock-speed: better implementation of ISA.
    - ***Equivalent impact to choice of O/S!!!***

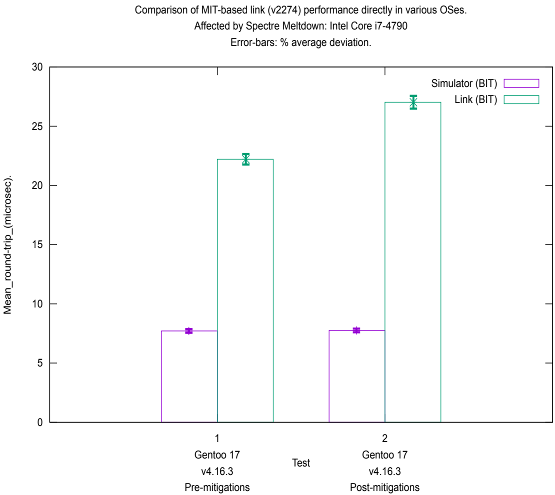
CentOS: Impact of Hardware Bugs.



Xubuntu: Impact of Hardware Bugs.



Gentoo: Impact of Hardware Bugs.



The Situation is so Complex...

- One must profile, profile and profile again - takes a lot of time.
  - Time the critical code; experiment with removing parts.
  - Unit tests vital; record performance to maintain SLAs.
- Highly-tuned code: *sensitive* to versions of compiler & O/S.
  - Choosing the right compiler is hard: re-optimizations are hugely costly without good tests.
  - The g++ v7 & 8-series are slower than v6...
  - Clang has stable performance, slow as g++ v7 & 8-series.
  - Choice of O/S can have ***equivalent impact!***
  - ***Effort spent in massaging code significantly smaller impact than compiler or O/S choice.***
- Outlook:
  - Select hardware, O/S very wisely.
  - No one compiler appears to be best - choice is crucial.

The Situation is so Complex...

- One must profile, profile and profile again - takes a lot of time.
  - Time the critical code; experiment with removing parts.
  - Unit tests vital; record performance to maintain SLAs.
- Highly-tuned code: *sensitive* to versions of compiler & O/S.
  - Choosing the right compiler is hard: re-optimizations are hugely costly without good tests.
  - The g++ v7 & 8-series are slower than v6...
  - Clang has stable performance, slow as g++ v7 & 8-series.
  - Choice of O/S can have ***equivalent impact!***
  - ***Effort spent in massaging code significantly smaller impact than compiler or O/S choice.***
- Outlook:
  - Select hardware, O/S very wisely.
  - No one compiler appears to be best - choice is crucial.

The Situation is so Complex...

- One must profile, profile and profile again - takes a lot of time.
  - Time the critical code; experiment with removing parts.
  - Unit tests vital; record performance to maintain SLAs.
- Highly-tuned code: *sensitive* to versions of compiler & O/S.
  - Choosing the right compiler is hard: re-optimizations are hugely costly without good tests.
  - The g++ v7 & 8-series are slower than v6...
  - Clang has stable performance, slow as g++ v7 & 8-series.
  - Choice of O/S can have ***equivalent impact!***
  - ***Effort spent in massaging code significantly smaller impact than compiler or O/S choice.***
- Outlook:
  - Select hardware, O/S very wisely.
  - No one compiler appears to be best - choice is crucial.

Major Impact on Haswell for this Benchmark...

- Mitigations for Haswell had high impact: CentOS: over 12%, Xubuntu: over 5% performance loss.
  - Application of such mitigations has highly variable impact, how can we trust the mitigations are effective?

- Extremely important to verify performance impact for latency-sensitive applications.
- In this case the solution is firewall, etc & avoid mitigations.
  - FIX looks safe but use of ASCII buffers: ripe for overruns...
  - Note: in this case Xubuntu is 8% faster than CentOS!

- How to demonstrate to regulator this is acceptable? Multiple clients connect to client-broker software? Regulations may require software audit to demonstrate that clients cannot access each other's data.

Major Impact on Haswell for this Benchmark...

- Mitigations for Haswell had high impact: CentOS: over 12%, Xubuntu: over 5% performance loss.
  - Application of such mitigations has highly variable impact, how can we trust the mitigations are effective?

- Extremely important to verify performance impact for latency-sensitive applications.
- In this case the solution is firewall, etc & avoid mitigations.
  - FIX looks safe but use of ASCII buffers: ripe for overruns...
  - Note: in this case Xubuntu is 8% faster than CentOS!

- How to demonstrate to regulator this is acceptable? Multiple clients connect to client-broker software? Regulations may require software audit to demonstrate that clients cannot access each other's data.

Major Impact on Haswell for this Benchmark...

- Mitigations for Haswell had high impact: CentOS: over 12%, Xubuntu: over 5% performance loss.
  - Application of such mitigations has highly variable impact, how can we trust the mitigations are effective?

- Extremely important to verify performance impact for latency-sensitive applications.
- In this case the solution is firewall, etc & avoid mitigations.
  - FIX looks safe but use of ASCII buffers: ripe for overruns...
  - Note: in this case Xubuntu is 8% faster than CentOS!

- How to demonstrate to regulator this is acceptable? Multiple clients connect to client-broker software? Regulations may require software audit to demonstrate that clients cannot access each other's data.

Final Thoughts...

- Discard Amhdal's Law at you peril! (10% of the code takes 90% of the run-time.)
  - Optimising code is *not* a panacea:
    - **it can lead to huge technical debt!**
  - Upgrading hardware is a cheap-and-dirty fix for performance..
  - The choice of O/S has an extremely surprising impact!
    - **It must not be under-estimated.**
- For more information on methodology or notes, please contact: *consultant@count-zero.ltd.uk*
  - Available to discuss options for your specific circumstances.

Final Thoughts...

- Discard Amhdal's Law at you peril! (10% of the code takes 90% of the run-time.)
  - Optimising code is *not* a panacea:
    - **it can lead to huge technical debt!**
  - Upgrading hardware is a cheap-and-dirty fix for performance..
  - The choice of O/S has an extremely surprising impact!
    - **It must not be under-estimated.**
- For more information on methodology or notes, please contact: *consultant@count-zero.ltd.uk*
  - Available to discuss options for your specific circumstances.

For Further Reading I

- http://libjmmcg.sf.net/
- Jeff Andrews
  Branch and Loop Reorganization to Prevent Mispredicts
  https://software.intel.com/en-us/articles/branch-and-loop-reorganization-to-prevent-mispredicts/
- Agner Fog
  The microarchitecture of Intel, AMD and VIA CPUs
  http://www.agner.org/optimize/microarchitecture.pdf
- *ARM11 MPCore Processor Technical Reference Manual*
  http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/ch06s02s03.html

For Further Reading II

- Prof. Bhargav C Goradiya, Trusit Shah
  Implementation of Backward Taken and Forward Not Taken Prediction Techniques in SimpleScalar
  http://ijarcsse.com/docs/papers/Volume_3/6_June2013/V3I6-0492.pdf
- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66573
- Jasper Neumann and Jens Henrik Gobbert
  Improving Switch Statement Performance with Hashing Optimized at Compile Time
  http://programming.sirrida.de/hashsuper.pdf

For Further Reading III

- Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg
  Meltdown.
  `https://arxiv.org/abs/1801.01207`
- Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom
  Spectre Attacks: Exploiting Speculative Execution.
  `https://arxiv.org/abs/1801.01203`
- *Spectre & Meltdown vulnerability/mitigation checker for Linux.*
  `https://github.com/speed47/spectre-meltdown-checker/`