A furtive fumble in Hard-Core Obscenity: the misuse of Template Meta-Programming to implement micro-optimisations in HFT.

J.M.McGuiness\textsuperscript{1}

\textsuperscript{1}Count-Zero Limited

ACCU London, 2016
1 Background
   - HFT & Low-Latency: Issues
   - C++ is THE Answer!
   - Oh no, C++ is just NOT the answer!
   - Optimization Case Studies.

2 Examples
   - Performance quirks in compiler versions.
   - Static branch-prediction: use and abuse.
   - Switch-statements: can these be optimized?
   - Perversions: Counting the number of set bits. “Madness”
   - The Effect of Compiler-flags.
   - Template Madness in C++: extreme optimization.

3 Conclusion
HFT & Low-Latency: Issues

- HFT & low-latency 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.
- Apologies if you know this already!
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!

J.M. McGuiness
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!

C++ is THE Answer!
Oh no, C++ is just NOT the answer!
Optimization Case Studies.
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!

---

J.M.McGuiness
Knuth, Amdahl: I spurn thee!
HFT & Low-Latency: Issues

- HFT & low-latency 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.

- Apologies if you know this already!
Like its predecessor C, C++ can be very low-level:

- Enables the intimacy required between software & hardware.
- Assembly output tuned directly from C++ statements.

Yet C++ is high-level: complex abstractions readily modeled.

Has increasingly capable libraries:

- E.g. Boost.
- Especially C++11, 14 & up-coming 17 standards.

I shall ignore other languages, e.g. D, Functional-Java, etc.

(garbage-collection kills performance, not low-enough level.)
C++ is THE Answer!

- Like its predecessor C, C++ can be very low-level:
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

- Yet C++ is high-level: complex abstractions readily modeled.

- Has increasingly capable libraries:
  - E.g. Boost.
  - Especially C++11, 14 & up-coming 17 standards.

- I shall ignore other languages, e.g. D, Functional-Java, etc.
  - (garbage-collection kills performance, not low-enough level.)
C++ is THE Answer!

- Like its predecessor C, C++ can be very low-level:
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

- Yet C++ is high-level: complex abstractions readily modeled.

- Has increasingly capable libraries:
  - E.g. Boost.
  - Especially C++11, 14 & up-coming 17 standards.

- I shall ignore other languages, e.g. D, Functional-Java, etc.
  - (garbage-collection kills performance, not low-enough level.)
C++ is THE Answer!

- Like its predecessor C, C++ can be very low-level:
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

- Yet C++ is high-level: complex abstractions readily modeled.

- Has increasingly capable libraries:
  - E.g. Boost.
  - Especially C++11, 14 & up-coming 17 standards.

- I shall ignore other languages, e.g. D, Functional-Java, etc.
  - (garbage-collection kills performance, not low-enough level.)
C++ is THE Answer!

- Like its predecessor C, C++ can be very low-level:
  - Enables the intimacy required between software & hardware.
  - Assembly output tuned directly from C++ statements.

- Yet C++ is high-level: complex abstractions readily modeled.

- Has increasingly capable libraries:
  - E.g. Boost.
  - Especially C++11, 14 & up-coming 17 standards.

- I shall ignore other languages, e.g. D, Functional-Java, etc.
  - (garbage-collection kills performance, not low-enough level.)
Oh no, C++ is NOT just the answer!

- There is more to low-latency 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), etc.
  - 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++, icc?
    - Focus: g++ C++11 & 14, some results for clang v3.8 & icc.
Oh no, C++ is NOT just the answer!

- There is more to low-latency 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), etc.
  - 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++, icc?
    - Focus: g++ C++11 & 14, some results for clang v3.8 & icc.
Oh no, C++ is NOT just the answer!

- There is more to low-latency 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), etc.
  - 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++, icc?
    - Focus: g++ C++11 & 14, some results for clang v3.8 & icc.

Knuth, Amdahl: I spurn thee!
There is more to low-latency 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), etc.

- 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++, icc?

  Focus: g++ C++11 & 14, some results for clang v3.8 & icc.
Optimization Case Studies.

- Despite the above, we choose to use C++,
  - which we will need to optimize.

- Optimizing C++ is not trivial, some examples shall be provided [1]:
  - Performance quirks in compiler versions.
  - Static branch-prediction: use and abuse.
  - Switch-statements: can these be optimized?
  - Counting the number of set bits.
  - Extreme templating: the case of memcpy().
Despite the above, we choose to use C++,

- which we will need to optimize.

Optimizing C++ is not trivial, some examples shall be provided [1]:

- Performance quirks in compiler versions.
- Static branch-prediction: use and abuse.
- Switch-statements: can these be optimized?
- Counting the number of set bits.
- Extreme templating: the case of memcpy().
Despite the above, we choose to use C++,

which we will need to optimize.

Optimizing C++ is not trivial, some examples shall be provided [1]:

- Performance quirks in compiler versions.
- Static branch-prediction: use and abuse.
- Switch-statements: can these be optimized?
- Counting the number of set bits.
- Extreme templating: the case of memcpy().
Performance quirks in compiler versions.

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

Example code, using -O3 -march=native:

```c
#include <string.h>
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 - v5.3.0:**

```c
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
```

- Notice how the instructions are better scheduled in the newer version, with no use of the stack.
Comparison of code generation in icc & clang.

**icc v13.0.1:**

```avr asm
foo():
movaps src(%rip), %xmm0 #8.3
movaps %xmm0, dest(%rip) #8.3
movl 16+src(%rip), %eax #8.3
movl %eax, 16+dest(%rip) #8.3
ret #9.1
dest:
src:
.byte 48
XXXsnipXXX
.byte 73
.byte 0
```

**clang 3.5.0 & 3.8.0:**

```avr asm
foo(): # @foo()
movaps src(%rip), %xmm0
movaps %xmm0, dest(%rip)
movl $4802631, dest+16(%rip) # imm=0x494847
retq
dest:
.zero 20
src:
.asciz "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

- Notice fewer instructions, but use of the stack - increases pressure on the cache, and the necessary memory-loads.

J.M.M ScGuiness

Knuth, Amdahl: I spurn thee!
Performance quirks in compiler versions.
Static branch-prediction: use and abuse.
Switch-statements: can these be optimized?
Perversions: Counting the number of set bits. “Madness”
The Effect of Compiler-flags.
Template Madness in C++: extreme optimization.

Does this matter in reality?

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

J.M.M<>Guiness
Knuth, Amdahl: I spurn thee!
Static branch-prediction: use and abuse.

- Which comes first? The `if()` `bar1()` or the `else` `bar2()`?
  - 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()?
  - 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()?
  - 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:

```c
extern void bar1();
extern void bar2();
void foo(bool i) {
    if (i) bar1();
    else bar2();
}
```
Generated Assembler using g++ v4.8.2, v4.9.0, v5.1.0 & v5.3.0

at -O0 & -O1:

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

At -O2 & -O3:

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

● **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 & CLANG v3.8.0

**ICC at -O2 & -O3:**

```assembly
foo(bool):
   testb %dil, %dil #5.7
   je ..B1.3 # Prob 50% #5.7
   jmp bar1() #6.2
..B1.3:   # Preds
   ..B1.1
   jmp bar2()
```

**CLANG at -O1, -O2 & -O3:**

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

- 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!
Background
Examples
Conclusion

Performance quirks in compiler versions.
Static branch-prediction: use and abuse.
Switch-statements: can these be optimized?
Perversions: Counting the number of set bits. “Madness”
The Effect of Compiler-flags.
Template Madness in C++: extreme optimization.

Use \_\_builtin\_expect\(i, 1\) in g++ for consistency.

- **BUT**: Adding \_\_builtin\_expect\(i, 1\) to the dtor of a stack-based string caused a slowdown in g++ v4.8.5!

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.

J.M.McGuiness
Knuth, Amdahl: I spurn thee!
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!
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!
Contiguous labels cause a jump-table to be created.
g++ v5.3.0 -O3 generated code.

Without `__builtin_expect()`:

```assembly
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()
```

With `__builtin_expect()`:

```assembly
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()
```

- Identical - it has no effect; icc & clang are likewise unmodified.
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”?
Compare various Implementations and their Performance using `-O3 -std=c++14`.

- A perennial favourite of interviews! Sooooo tedious...
- The obvious implementation:

```cpp
 constexpr inline __attribute__((const))
 unsigned long result() noexcept(true) {
    const uint64_t num=843678937893;
    unsigned long count=0;
    do {
       if (LIKELY(num&1)) {
          ++count;
       }
    } while (num >>= 1);
    return count;
}
```

Assembler:

```
 movabsq $843678937893, %rax
 .L2:
    movq %rax, %rsi
    shrq %rax
    andl $1, %esi
    addq %rsi, %rcx
    subl $1, %edx
    jne .L2
    movq %rcx, k(%rip)
    xorl %eax, %eax
    ret
```
Part 1: Now using templates to unroll the loop.

The template implementation:

```cpp
template<
    uint8_t Val,
    class BitSet>
struct unroller : unroller<Val-1, BitSet>;

template<class T, T... args>
struct array_t;

template<unsigned long long Val>
struct shifter;

template<typename Val, 
    template<typename> class Fn,
    unsigned long long... bitmasks>
struct gen_bitmasks;

struct count_setbits {
    constexpr static element_type
    result() noexcept(true) { 
        unsigned long num = 843678937893;
        return unroller_t::result(num);
    }
};
```

Assembler:

```
movq $22, k(%rip)
xorl %eax, %eax
ret
```

- Outrageous templating has enabled constexpr!
Part 2: Now using assembly.

The asm `POPCNT` implementation;

```c
#include <stdint.h>
inline uint64_t result() noexcept(true) {
    const uint64_t num=843678937893;
    uint64_t count=0;
    __asm__ volatile ("POPCNT %1, %0;"
    :="r"(count)
    :"r"(num)
    );
    return count;
}
```

Assembler:

```
movabsq $843678937893, %rax
POPCNT %rax, %rax;
xorl %eax, %eax
ret
```

- Contrary to popular belief: inlining happens, despite the `__asm__` block.
- Result has to be dynamically computed.

J.M.McGuiness

Knuth, Amdahl: I spurn thee!
Part 2: Now using builtins.

The \texttt{\_\_builtin\_popcount\_l1} implementation; -mpopcnt:

```c
#include <stdint.h>
constexpr inline __attribute__((const))
inline uint64_t result(uint64_t num)
noexcept(true) {
    const uint64_t num=843678937893;
    return \_\_builtin\_popcount\_l1(num);
}
```

Assembler:

```
movq $22, k(%rip)
xorl %eax, %eax
ret
```

- Note how the builtin enables the result to be computed at compile-time, without that template malarky.
- But requires a suitable ISA.
Performance quirks in compiler versions.
Static branch-prediction: use and abuse.
Switch-statements: can these be optimized?
Perversions: Counting the number of set bits. “Madness”
The Effect of Compiler-flags.
Template Madness in C++: extreme optimization.

Does this matter in reality?

- Very variable performance: the latest g++ (v5.1.0 & v5.3.0, with kernels v4.1.15 & v4.4.6) is a disaster!

J.M.McGuiness
Knuth, Amdahl: I spurn thee!
Counting set bits: conclusion.

- Know thine architecture:
  - Without the right tools for the job, one has to work very hard with complex templates.
  - With the right architecture, and compiler, much more simple code can use builtins.

- One can use assembler, and it will be fast.
  - But not as fast as builtins as compilers can replace code with constants!

- Review your code when updating hardware & compiler.
The Curious Case of `memcpy()` and SSE.

Examined with various compilers with -O3 -std=c++14.

```cpp
__attribute__((aligned(256))) const char s[] =
    "And for something completely different."
char d[sizeof(s)];
void bar1() {
    std::memcpy(d, s, sizeof(s));
}
```

- Because copying is VERY common.
- Surely compilers simply “get this right”? 
Assembly output from \texttt{g++ v4.9.0-5.3.0}.

- \texttt{mavx} has no effect.

\begin{verbatim}
bar1():
  movabsq $2338053640979508801, %rax
  movq %rax, d(%rip)
  movabsq $7956005065853857651, %rax
  movq %rax, d+8(%rip)
  movabsq $7308339910637985895, %rax
  movq %rax, d+16(%rip)
  movabsq $7379539555062146420, %rax
  movq %rax, d+24(%rip)
  movabsq $13075866425910630, %rax
  movq %rax, d+32(%rip)
  ret

d:
  .zero 40
\end{verbatim}

- Surely use SSE? All other options had no effect.
No -mavx.

bar1(): # @bar1()
    movabsq $13075866425910630, %rax
    movq %rax, d+32(%rip)
    movaps s+16(%rip), %xmm0
    movaps %xmm0, d+16(%rip)
    movaps %xmm0, %xmm0
    movaps %xmm0, d(%rip)
    retq

    d:
        .zero 40

    s:
        .asciz "And for something completely different."

With -mavx.

bar1(): # @bar1()
    vmovaps s(%rip), %ymm0
    vextractf128 $1, %ymm0, d+16(%rip)
    movabsq $13075866425910630, %rax
    movq %rax, d+32(%rip)
    vmovaps %xmm0, d(%rip)
    vzeroupper
    retq

    d:
        .zero 40

    s:
        .asciz "And for something completely different."

Note how the SSE registers are now used, unlike g++, although same number of instructions.
Performance quirks in compiler versions.
Static branch-prediction: use and abuse.
Switch-statements: can these be optimized?
Perversions: Counting the number of set bits. "Madness"
The Effect of Compiler-flags.
Template Madness in C++: extreme optimization.

Assembly output from icc v13.0.1 -std=c++11.

No -mavx.

bar1():
  movaps s(%rip), %xmm0 #205.3
  movaps %xmm0, d(%rip) #205.3
  movaps 16+s(%rip), %xmm1 #205.3
  movaps %xmm1, 16+d(%rip) #205.3
  movq 32+s(%rip), %rax #205.3
  movq %rax, 32+d(%rip) #205.3
  ret #206.1

d:
  s:
    .byte 65
    .byte 0

With -mavx.

bar1():
  vmovups 16+s(%rip), %xmm0 #205.3
  vmovups %xmm0, 16+d(%rip) #205.3
  movq 32+s(%rip), %rax #205.3
  movq %rax, 32+d(%rip) #205.3
  vmovups %xmm1, 16+s(%rip) #205.3
  vmovups %xmm1, d(%rip) #205.3
  ret #206.1

d:
  s:
    .byte 65
    .byte 0

• Like clang, the SSE registers are used, but a totally different schedule.

Like clang, the SSE registers are used, but a totally different schedule.

J.M.McGuiness

Knuth, Amdahl: I spurn thee!
Let’s go Mad...

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

Examined with various compilers with `-O3 -std=c++14 -mavx`.

```cpp
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.**

```assembly
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
```

**v5.1.0-5.3.0.**

```assembly
bar():
    pushq %rbp
    vmovdqa .LC1(%rip), %ymm0
    movabsq $13075866425910630, %rax
    movq %rax, d+32(%rip)
    movq %rsp, %rbp
    pushq %r10
    vmovdqa %ymm0, d(%rip)
    vzeroupper
    popq %r10
    popq %rbp
    ret

d:
    .zero 40

.LC1:
    .quad 2338053640979508801
    .quad 7956005065853857651
    .quad 7308339910637985895
    .quad 7379539555062146420
```

• v4.9.0 is excellent, but 5.3.0 went mad!!!

J.M.McGuiness

Knuth, Amdahl: I spurn thee!
Judicious use of micro-optimized templates can provide a performance enhancement.
Performance quirks in compiler versions.
Static branch-prediction: use and abuse.
Switch-statements: can these be optimized?
Perversions: Counting the number of set bits. “Madness”
The Effect of Compiler-flags.
Template Madness in C++: extreme optimization.

Again, does this matter?

- No statistical difference, but g++ code-gen was indifferent:
  - Excellent optimizations confounded by choice of compiler.
  - Tried clang v3.5.0, but does not compile - not all are equal.
The impact of compiler version on performance.

Comparison of stack-string ctor and dtor performance.
Comparison of stack-string ctor, dtor and assignment performance.
Comparison of stack-string ctor, dtor and replace performance.

Mean_rate_(operations/sec).

Error-bars: % average deviation.

Building...

Warning! Different y-scales.
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 is **very** sensitive to the version of compiler.
  - Choosing the right compiler is hard: re-optimizations are hugely costly without good tests.
  - The g++ 5.3.0 with ABI11 is in progress: appalling results...

- Outlook:
  - No one compiler appears to be best - choice is crucial.
  - Newer versions of clang have not been investigated.
For Further Reading

http://libjmmcg.sf.net/

Jeff Andrews
*Branch and Loop Reorganization to Prevent Mispredicts*

Agnar Fog
*The microarchitecture of Intel, AMD and VIA CPUs*
http://www.agner.org/optimize/microarchitecture.pdf
For Further Reading

**ARM11 MPCore Processor Technical Reference Manual**
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/ch06s02s03.html

Prof. Bhargav C Goradiya, Trusit Shah
*Implementation of Backward Taken and Forward Not Taken Prediction Techniques in SimpleScalar*

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