Man Against Compiler

Alexander Nasonov

August 10, 2019

About Myself

- NetBSD contributor.

About Myself

- NetBSD contributor.
  - Author of bpfjit.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.
- Day job in HFT @ XMM SWAP LTD.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.
- Day job in HFT @ XMM SWAP LTD.
  - Heavily templatised C++ (ugh!).

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.
- Day job in HFT @ XMM SWAP LTD.
  - Heavily templatised C++ (ugh!).
  - "The winner takes it all" performance optimisations.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.
- Day job in HFT @ XMM SWAP LTD.
  - Heavily templatised C++ (ugh!).
  - "The winner takes it all" performance optimisations.
  - DSL mini-compiler.

About Myself

- NetBSD contributor.
  - Author of bpfjit.
  - Contributor of small changes, mostly C and a tiny bit of Lua.
  - Maintainer of a bunch of pkgsrc packages.
- Day job in HFT @ XMM SWAP LTD.
  - Heavily templatised C++ (ugh!).
  - "The winner takes it all" performance optimisations.
  - DSL mini-compiler.
- Not a compiler guy.

Compilers

- gcc 6.3 and newer.

Compilers

- gcc 6.3 and newer.
- clang 8.0.

Compilers

- gcc 6.3 and newer.
- clang 8.0.
- CompCert (for the presentation only).

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.
  - Faster than gcc -O0.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.
  - Faster than gcc -O0.
- Not completely free software.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.
  - Faster than gcc -O0.
- Not completely free software.
  - Copyright by INRIA and AbsInt.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.
  - Faster than gcc -O0.
- Not completely free software.
  - Copyright by INRIA and AbsInt.
  - Free download for non-commercial use.

CompCert (version 3.5)

- Formally-verified optimising compiler of (almost all of) the C99.
  - Automated proof of every optimisation pass.
  - Targets PowerPC, ARM, RISC-V and X86.
  - Faster than gcc -O0.
- Not completely free software.
  - Copyright by INRIA and AbsInt.
  - Free download for non-commercial use.
  - Some files are dual licenced under GPL v2/v3.

CompCert (cont)

- CompCert is NOT:

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.
  - The compiler assumes that a user program is well defined.

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.
  - The compiler assumes that a user program is well defined.
  - All bets are off when undefined behaviour is *triggered*.

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.
  - The compiler assumes that a user program is well defined.
  - All bets are off when undefined behaviour is *triggered*.
  - Given a formally proved equivalence of optimisation passes, undefined behaviour can be tracked from source down to assembly (?).

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.
  - The compiler assumes that a user program is well defined.
  - All bets are off when undefined behaviour is *triggered*.
  - Given a formally proved equivalence of optimisation passes, undefined behaviour can be tracked from source down to assembly (?).
  - Correspondence is often not 1:1, though. E.g. function arguments can be evaluated in any order.

CompCert (cont)

- CompCert is NOT:
  - It doesn't prove your programs!
  - It doesn't prove that your program is well defined.
- Undefined behaviour.
  - The compiler assumes that a user program is well defined.
  - All bets are off when undefined behaviour is *triggered*.
  - Given a formally proved equivalence of optimisation passes, undefined behaviour can be tracked from source down to assembly (?).
  - Correspondence is often not 1:1, though. E.g. function arguments can be evaluated in any order.
  - Interpreter mode to catch undefined behaviour at runtime.

Part I

Lets see some assembly.

Trivial Program

The most trivial C program.

```
int main() {
   return 1;
}
```

Trivial Program (cont)

Intel x64 assembly generated by CompCert:

```
48 83 ec 08              sub    rsp,0x8
48 8d 44 24 10           lea    rax,[rsp+0x10]
48 89 04 24              mov    QWORD PTR [rsp],rax
b8 01 00 00 00           mov    eax,0x1
48 83 c4 08              add    rsp,0x8
c3                       ret
```

Trivial Program (cont)

Intel x64 assembly generated by CompCert:

```
48 83 ec 08              sub    rsp,0x8
48 8d 44 24 10           lea    rax,[rsp+0x10]
48 89 04 24              mov    QWORD PTR [rsp],rax
b8 01 00 00 00           mov    eax,0x1
48 83 c4 08              add    rsp,0x8
c3                       ret
```

- Prologue.

Trivial Program (cont)

Intel x64 assembly generated by CompCert:

```
48 83 ec 08              sub    rsp,0x8
48 8d 44 24 10           lea    rax,[rsp+0x10]
48 89 04 24              mov    QWORD PTR [rsp],rax
b8 01 00 00 00           mov    eax,0x1
48 83 c4 08              add    rsp,0x8
c3                       ret
```

- Prologue.
- Epilogue.

Trivial Program (cont)

Intel x64 assembly generated by CompCert:

```
48 83 ec 08              sub    rsp,0x8
48 8d 44 24 10           lea    rax,[rsp+0x10]
48 89 04 24              mov    QWORD PTR [rsp],rax
b8 01 00 00 00           mov    eax,0x1
48 83 c4 08              add    rsp,0x8
c3                       ret
```

- Prologue.
- Epilogue.
- Function body.

Trivial Program (cont)

Both gcc and clang produce more optimised binary:

```
b8 01 00 00 00          mov     eax,0x1
c3                      ret
```

Trivial Program (cont)

You can't make it any smaller ... Or can you?

Division by Zero

```
int main() {
   return 1  ;
}
```

Division by Zero

```
int main() {
    return 1/0;
}
```

Division by Zero

```
int main() {
   return 1/0;
}
```

Compilers generate division-by-zero warning.

```
In function 'main':
warning: division by zero [-Wdiv-by-zero]
 int main() { return 1/0; }
```

Division by Zero (cont)

Intel x64 assembly generated by gcc 7.4.0.

```
0f 0b                    ud2 ; __builtin_trap()
```

Division by Zero (cont)

Intel x64 assembly generated by gcc 7.4.0.

```
0f 0b                    ud2 ; __builtin_trap()
```

Main takeaway:

- It crashes loud and clear when an undefined behaviour is *triggered*.

Division by Zero (cont)

Intel x64 assembly generated by clang 8.0.0.

```
; NO mov eax,0x1
c3                              ret
```

Division by Zero (cont)

Intel x64 assembly generated by clang 8.0.0.

```
; NO mov eax,0x1
c3                              ret
```

Main takeaways:

- clang silently generates non ABI conformant function as if it was declared 'void main()'.

Division by Zero (cont)

Intel x64 assembly generated by clang 8.0.0.

```
; NO mov eax,0x1
c3                              ret
```

Main takeaways:

- clang silently generates non ABI conformant function as if it was declared 'void main()'.
- Fancy debugging similar problems?

Loop with signed arithmetic

```
int foo(int n, int r) {
   for (; n != INT_MIN; n++) {
     r = n + r / 2;
   }
   return r;
}
```

Loop with signed arithmetic

```
int foo(int n, int r) {
   for (; n != INT_MIN; n++) {
     r = n + r / 2;
   }
   return r;
}
```

- Signed division by 2 is less performant (correction after a shift for negative values).

Loop with signed arithmetic

```
int foo(int n, int r) {
   for (; n != INT_MIN; n++) {
     r = n + r / 2;
   }
   return r;
}
```

- Signed division by 2 is less performant (correction after a shift for negative values).
- Undefined behaviour for most inputs.

Loop with signed arithmetic

```
int foo(int n, int r) {
    for (; n != INT_MIN; n++) {
      r = n + r / 2;
    }
    return r;
}
```

- Signed division by 2 is less performant (correction after a shift for negative values).
- Undefined behaviour for most inputs.
- Compilers aren't smart enough to abuse it fully.

Loop with signed arithmetic

```
int foo(int n, int r) {
   for (; n != INT_MIN; n++) {
     r = n + r / 2;
   }
   return r;
}
```

- Signed division by 2 is less performant (correction after a shift for negative values).
- Undefined behaviour for most inputs.
- Compilers aren't smart enough to abuse it fully.
- But they will get smarter.

Loop with signed arithmetic (cont)

Reverse engineered assembly.

```
int foo_gcc(int n, int r) {
    return r;
}

int foo_clang(int n, int r) {
    for (; n != INT_MIN; ) {}
    return r;
}
```

Loop with signed arithmetic (cont)

Reverse engineered assembly.

```
int foo_gcc(int n, int r) {
   return r;
}

int foo_clang(int n, int r) {
   for (; n != INT_MIN; ) {}
   return r;
}
```

- Both are valid (remember, all bets are off!).

Loop with signed arithmetic (cont)

Reverse engineered assembly.

```
int foo_gcc(int n, int r) {
   return r;
}

int foo_clang(int n, int r) {
   for (; n != INT_MIN; ) {}
   return r;
}
```

- Both are valid (remember, all bets are off!).
- In the following slides assume n != INT_MIN.

Loop with signed arithmetic (cont)

Rewrite the loop to avoid an undefined behaviour in the loop condition.

```
while (1) {
   r = n + r / 2;
     if (n == INT_MAX)
       break;
     n++;
}
```

Loop with signed arithmetic (cont)

Rewrite the loop to avoid an undefined behaviour in the loop condition.

```
while (1) {
   r = n + r / 2;
     if (n == INT_MAX)
       break;
     n++;
}
```

- It doesn't look as nice as the original idiom.

Loop with signed arithmetic (cont)

Rewrite the loop to avoid an undefined behaviour in the loop condition.

```
while (1) {
   r = n + r / 2;
     if (n == INT_MAX)
       break;
     n++;
}
```

- It doesn't look as nice as the original idiom.
- Undefined behaviour in the body is still possible.

Loop with signed arithmetic (cont)

Rewrite the loop to avoid an undefined behaviour in the loop condition.

```
while (1) {
    r = n + r / 2;
      if (n == INT_MAX)
        break;
      n++;
}
```

- It doesn't look as nice as the original idiom.
- Undefined behaviour in the body is still possible.
- Alternatively, pass *-fwrapv* option to gcc and clang.

Loop with signed arithmetic (cont)

Table: timeit foo(1, 1)

gcc	2.78
CompCert	2.81
clang	4.09

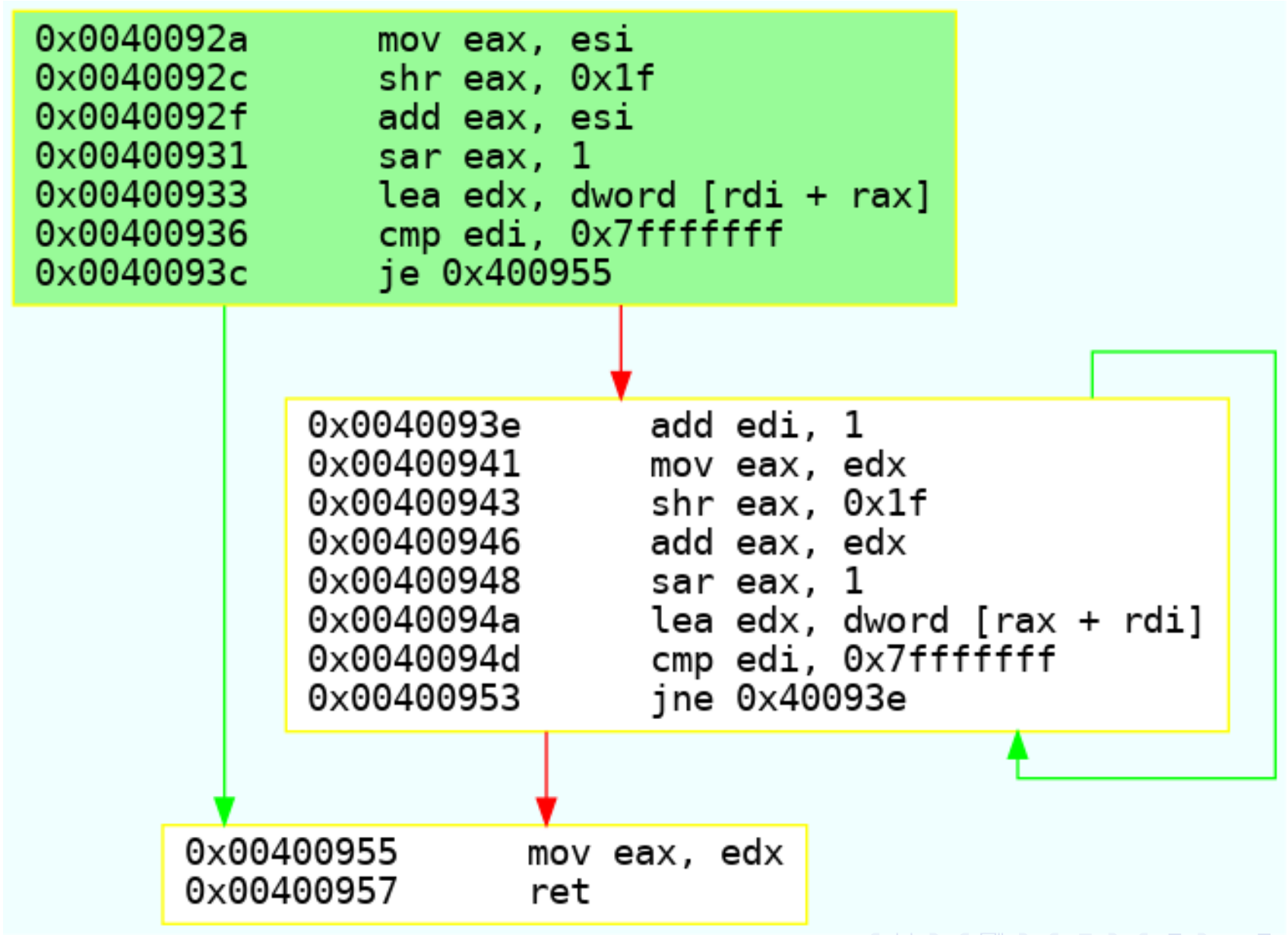
Loop with signed arithmetic (cont)

Use radare2 RE tool to draw basic blocks.

```
$ r2 -A ./a.out
> s sym.foo
> agfd > bb.dot
> exit
$ dot -Tpng -o bb.png bb.dot
```

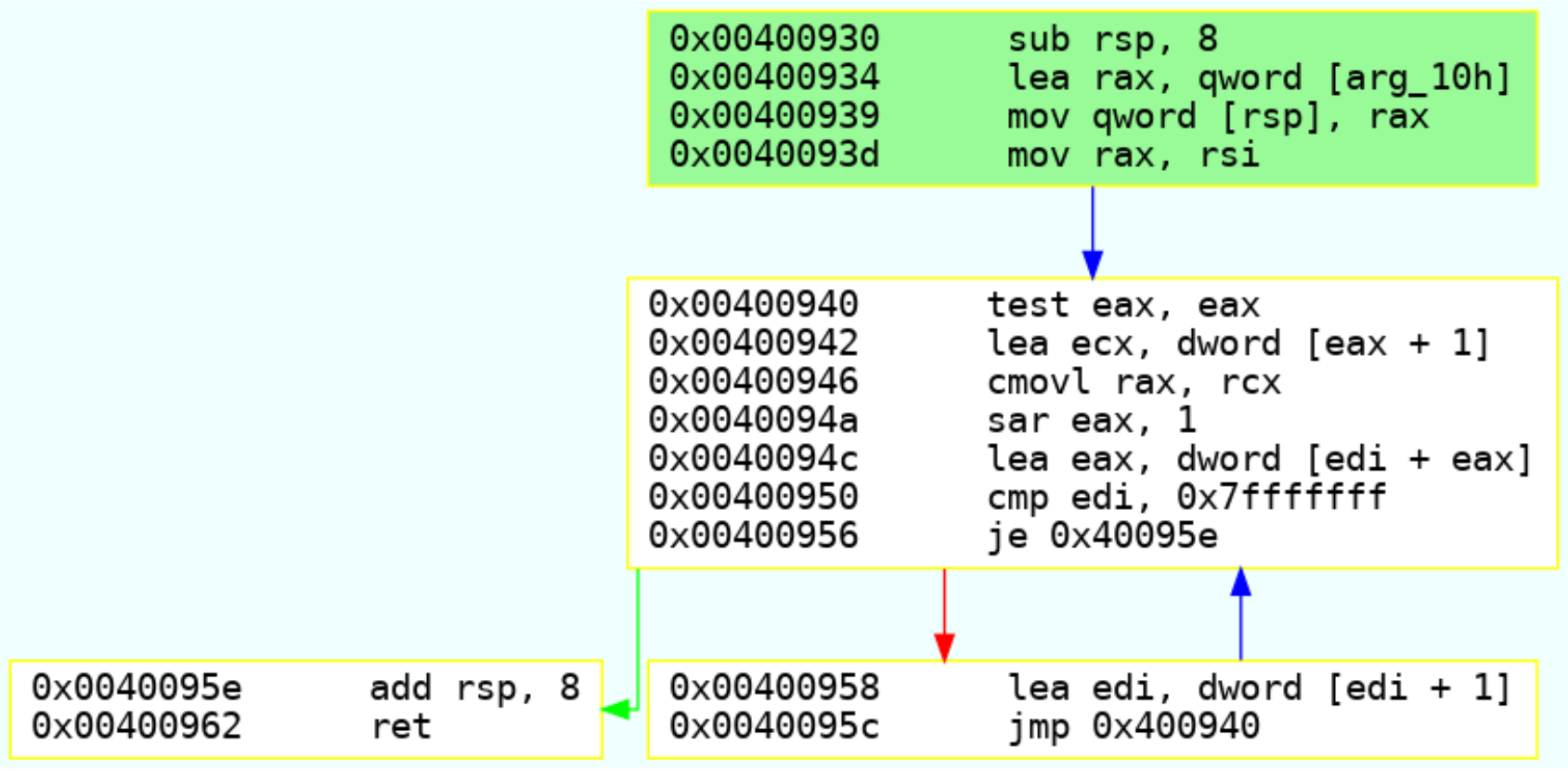
Loop with signed arithmetic (cont)

Basic blocks of gcc code (2.78 seconds).



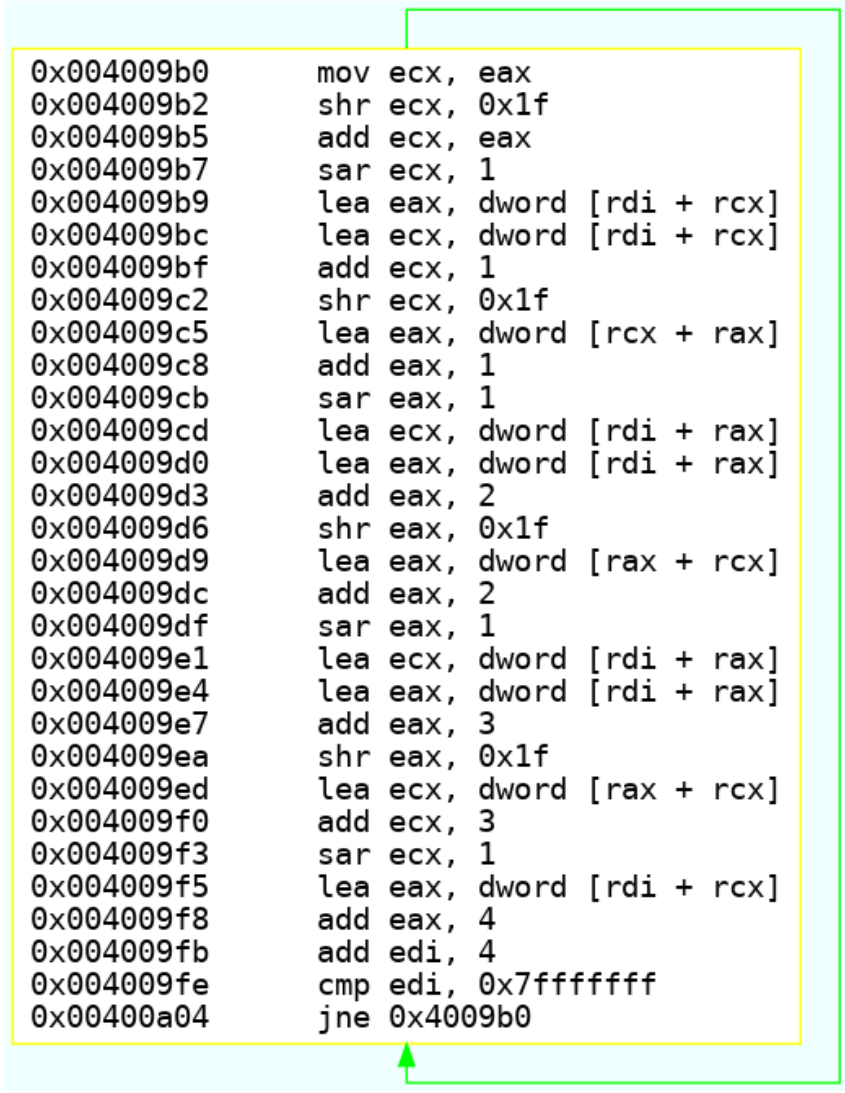
Loop with signed arithmetic (cont)

Basic blocks of CompCert code (2.81 seconds).



Loop with signed arithmetic (cont)

Main loop of clang code (4.09 seconds).



```
0x004009b0      mov ecx, eax
0x004009b2      shr ecx, 0x1f
0x004009b5      add ecx, eax
0x004009b7      sar ecx, 1
0x004009b9      lea eax, dword [rdi + rcx]
0x004009bc      lea ecx, dword [rdi + rcx]
0x004009bf      add ecx, 1
0x004009c2      shr ecx, 0x1f
0x004009c5      lea eax, dword [rcx + rax]
0x004009c8      add eax, 1
0x004009cb      sar eax, 1
0x004009cd      lea ecx, dword [rdi + rax]
0x004009d0      lea eax, dword [rdi + rax]
0x004009d3      add eax, 2
0x004009d6      shr eax, 0x1f
0x004009d9      lea eax, dword [rax + rcx]
0x004009dc      add eax, 2
0x004009df      sar eax, 1
0x004009e1      lea ecx, dword [rdi + rax]
0x004009e4      lea eax, dword [rdi + rax]
0x004009e7      add eax, 3
0x004009ea      shr eax, 0x1f
0x004009ed      lea ecx, dword [rax + rcx]
0x004009f0      add ecx, 3
0x004009f3      sar ecx, 1
0x004009f5      lea eax, dword [rdi + rcx]
0x004009f8      add eax, 4
0x004009fb      add edi, 4
0x004009fe      cmp edi, 0x7fffffff
0x00400a04      jne 0x4009b0
```

Building a parallel reality

```
int bar(int n, int r) {
    if (n < 0 || r < 0) return foo(n, r);

    while (1) {
      r = n + (r >> 1); // was r = n + r / 2
      if (n == INT_MAX)
        break;
      n++;
    }
}
```

Building a parallel reality

```
int bar(int n, int r) {
    if (n < 0 || r < 0) return foo(n, r);

    while (1) {
      r = n + (r >> 1); // was r = n + r / 2
      if (n == INT_MAX)
        break;
      n++;
    }
}
```

- Non-negative n and r never turn negative ...

Building a parallel reality

```
int bar(int n, int r) {
    if (n < 0 || r < 0) return foo(n, r);

    while (1) {
      r = n + (r >> 1); // was r = n + r / 2
      if (n == INT_MAX)
        break;
      n++;
    }
}
```

- Non-negative n and r never turn negative ...
- in our parallel universe without signed overflows.

Building a parallel reality

```
int bar(int n, int r) {
    if (n < 0 || r < 0) return foo(n, r);

    while (1) {
      r = n + (r >> 1); // was r = n + r / 2
      if (n == INT_MAX)
        break;
      n++;
    }
}
```

- Non-negative n and r never turn negative ...
- in our parallel universe without signed overflows.
- Thus, division by 2 can be replaced with a shift.

Building a parallel reality (cont)

Table: timeit foo(1, 1), timeit bar(1, 1)

gcc	2.78	1.39
CompCert	2.81	1.39
clang	4.09	2.11

Building a parallel reality (cont)

Table: timeit foo(1, 1), timeit bar(1, 1)

gcc	2.78	1.39
CompCert	2.81	1.39
clang	4.09	2.11

- Impressive speed-up for a simple trick.

Building a parallel reality (cont)

Table: timeit foo(1, 1), timeit bar(1, 1)

gcc	2.78	1.39
CompCert	2.81	1.39
clang	4.09	2.11

- Impressive speed-up for a simple trick.
- This guy did what?!?!?!

Building a parallel reality (cont)

- It's like a proof by contradiction ...

Building a parallel reality (cont)

- It's like a proof by contradiction ...
- but you can stop at any time.

Building a parallel reality (cont)

- It's like a proof by contradiction ...
- but you can stop at any time.
- Start with a contradicting assumption that your program is well defined ...

Building a parallel reality (cont)

- It's like a proof by contradiction ...
- but you can stop at any time.
- Start with a contradicting assumption that your program is well defined ...
- build a corpus of "facts" until you find an obvious contradiction.

Building a parallel reality (cont)

- It's like a proof by contradiction ...
- but you can stop at any time.
- Start with a contradicting assumption that your program is well defined ...
- build a corpus of "facts" until you find an obvious contradiction.
- But usually, compilers give up and turn "facts" into code.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.
- Thus, r / 2 can only be 0.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.
- Thus, r / 2 can only be 0.
- This can be rewritten as r <2.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.
- Thus, r / 2 can only be 0.
- This can be rewritten as r <2.
- Rewind to the previous iteration and see that

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.
- Thus, r / 2 can only be 0.
- This can be rewritten as r <2.
- Rewind to the previous iteration and see that
- INT_MAX - 1 + r' / 2 <2.

Building a parallel reality (cont)

Lets finish our proof by contradiction.

- Recall that n and r are non-negative.
- n is equal to INT_MAX on the last iteration.
- Thus, r / 2 can only be 0.
- This can be rewritten as r <2.
- Rewind to the previous iteration and see that
- INT_MAX - 1 + r' / 2 <2.
- Contradiction.

Building a parallel reality (cont)

Optimised to "death" version.

```
int bar(int n, int r) {
   if (n >= 0 && r >= 0)
     __builtin_trap(); // oder return r;
   return foo(n, r);
}
```

Undefined behaviour can format your disk

From Krister Walfridsson's blog.

```
typedef int (*Function)();

static Function Do;

static int EraseAll() { return system("rm -rf /"); }

void NeverCalled() { Do = EraseAll; }

int main() { return Do(); }
```

Undefined behaviour can format your disk (cont)

```
;-- main:
(fcn) sym.main 10
  int sym.main (int argc, char **argv, char **envp);
; CALL XREF from entry0 (0x4008c8)
0x004009b0      mov edi, str.rm__rf     ; section..rodata ; 0x4009ce ; "rm -rf /"
0x004009b5      jmp sym.imp.system
```

Undefined behaviour can format your disk (cont)

```
;-- main:
(fcn) sym.main 10
  int sym.main (int argc, char **argv, char **envp);
; CALL XREF from entry0 (0x4008c8)
0x004009b0      mov edi, str.rm__rf     ; section..rodata ; 0x4009ce ; "rm -rf /"
0x004009b5      jmp sym.imp.system
```

- The compiler assumes that *Do* can't be a NULL pointer when called.

Undefined behaviour can format your disk (cont)

```
;-- main:
(fcn) sym.main 10
  int sym.main (int argc, char **argv, char **envp);
; CALL XREF from entry0 (0x4008c8)
0x004009b0      mov edi, str.rm__rf     ; section..rodata ; 0x4009ce ; "rm -rf /"
0x004009b5      jmp sym.imp.system
```

- The compiler assumes that *Do* can't be a NULL pointer when called.
- Therefore, it must have been set to *EraseAll* by a prior call of *NeverCalled*.

Undefined behaviour can format your disk (cont)

```
;-- main:
(fcn) sym.main 10
  int sym.main (int argc, char **argv, char **envp);
; CALL XREF from entry0 (0x4008c8)
0x004009b0      mov edi, str.rm__rf     ; section..rodata ; 0x4009ce ; "rm -rf /"
0x004009b5      jmp sym.imp.system
```

- The compiler assumes that *Do* can't be a NULL pointer when called.
- Therefore, it must have been set to *EraseAll* by a prior call of *NeverCalled*.
- But our program doesn't call *NeverCalled*!

Undefined behaviour can format your disk (cont)

```
;-- main:
(fcn) sym.main 10
  int sym.main (int argc, char **argv, char **envp);
; CALL XREF from entry0 (0x4008c8)
0x004009b0      mov edi, str.rm__rf     ; section..rodata ; 0x4009ce ; "rm -rf /"
0x004009b5      jmp sym.imp.system
```

- The compiler assumes that *Do* can't be a NULL pointer when called.
- Therefore, it must have been set to *EraseAll* by a prior call of *NeverCalled*.
- But our program doesn't call *NeverCalled*!
- Unfortunately, clang isn't clever enough to figure it out.

Part II - The death of optimizing compilers

The death of optimizing compilers.

Part II - The death of optimizing compilers

The death of optimizing compilers.
by Daniel J. Bernstein, aka djb.

The death of optimizing compilers

- Modern computers are fast.

The death of optimizing compilers

- Modern computers are fast.
- Modern processors are JIT compilers.

The death of optimizing compilers

- Modern computers are fast.
- Modern processors are JIT compilers.
- Most code is cold or even only run once.

The death of optimizing compilers

- Modern computers are fast.
- Modern processors are JIT compilers.
- Most code is cold or even only run once.
- IO/memory is a bottleneck when running cold code.

Examples of hand-optimised assembly

- Interpeter loop in LuaJIT.

Examples of hand-optimised assembly

- Interpeter loop in LuaJIT.
- Check LuaJIT's mailing archives for Mike's Pall posts.

Examples of hand-optimised assembly

- Interpeter loop in LuaJIT.
- Check LuaJIT's mailing archives for Mike's Pall posts.
- See also a blog post by Peter Cawley (corsix) about optimising CPython.

Examples of hand-optimised assembly

- Interpeter loop in LuaJIT.
- Check LuaJIT's mailing archives for Mike's Pall posts.
- See also a blog post by Peter Cawley (corsix) about optimising CPython.
- Cryptographic code (OpenSSL, CloudFlare).

Examples of hand-optimised assembly

- Interpeter loop in LuaJIT.
- Check LuaJIT's mailing archives for Mike's Pall posts.
- See also a blog post by Peter Cawley (corsix) about optimising CPython.
- Cryptographic code (OpenSSL, CloudFlare).
- 0x80.pl website by Wojciech Mula.

Domain-Specific specialised code

- In software.

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.
  - Mathematical libraries.

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.
  - Mathematical libraries.
  - Password crackers (John The Ripper, cathash).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.
  - Mathematical libraries.
  - Password crackers (John The Ripper, cathash).
  - Cryptography (crypto-mining, HSM).

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.
  - Mathematical libraries.
  - Password crackers (John The Ripper, cathash).
  - Cryptography (crypto-mining, HSM).
  - Video/audio processing.

Domain-Specific specialised code

- In software.
  - Mathematical libraries (ATLAS/BLAS, linpack, ML/AI, etc).
  - Regular expressions (hyperscan).
  - Cryptographic libraries.
  - Video/audio processing (rendering, codecs etc).
  - Language runtimes (Go, LuaJIT, etc).
  - Misc code (libdivide, etc).
- In hardware.
  - Mathematical libraries.
  - Password crackers (John The Ripper, cathash).
  - Cryptography (crypto-mining, HSM).
  - Video/audio processing.
  - Packet forwarding/inspection.

Part III - Other languages

Other languages

Byte-code based languages.

- General considerations.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.
  - There is still space for UB but it's much narrower.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.
  - There is still space for UB but it's much narrower.
- Java.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.
  - There is still space for UB but it's much narrower.
- Java.
  - Java seems to be UB free (based on StackOverflow answers).

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.
  - There is still space for UB but it's much narrower.
- Java.
  - Java seems to be UB free (based on StackOverflow answers).
  - It has *implementation-defined* and *unspecified* behaviour.

Other languages

Byte-code based languages.

- General considerations.
  - Generally safe languages.
  - Bytecode instructions are simple with clear inputs, outputs and effects.
  - When a native code generated off a bytecode, it's usually equivalent to interpetation.
  - There is still space for UB but it's much narrower.
- Java.
  - Java seems to be UB free (based on StackOverflow answers).
  - It has *implementation-defined* and *unspecified* behaviour.
  - Wrap-around for signed arithmetic.

Other languages

- Go

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".
  - Out of bound access and null pointer access panic in a controlled way.

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".
  - Out of bound access and null pointer access panic in a controlled way.
  - Shifting by more than a width is well defined.

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".
  - Out of bound access and null pointer access panic in a controlled way.
  - Shifting by more than a width is well defined.
  - *INT_MIN/-1* is well defined.

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".
  - Out of bound access and null pointer access panic in a controlled way.
  - Shifting by more than a width is well defined.
  - *INT_MIN/-1* is well defined.
- Rust

Other languages

- Go
  - "Go has much less undefined behavior than C/C++" - Ian Lance Taylor.
  - Ian gives one example on UB and it's a race condition.
  - "Signed integers wrap on overflow, which reduces the scope of loop optimization".
  - Out of bound access and null pointer access panic in a controlled way.
  - Shifting by more than a width is well defined.
  - *INT_MIN/-1* is well defined.
- Rust
  - If there is a UB, it's a bug!

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.
  - More performance driven.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.
  - More performance driven.
  - Don't seem to care about undefined behaviour.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.
  - More performance driven.
  - Don't seem to care about undefined behaviour.
  - Some even abuse it.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.
  - More performance driven.
  - Don't seem to care about undefined behaviour.
  - Some even abuse it.
- Different goals complicate compilers.

Conclusion

C/C++ seems to be driven by opposing forces.

- InfoSec paranoia.
  - C/C++ are very popular but aren't safe languages.
  - Extra-protection doesn't harm.
  - Typically, it only introduces 1-2% slowdown.
  - But it accumulates.
- The standard committee and compilers nerds.
  - More performance driven.
  - Don't seem to care about undefined behaviour.
  - Some even abuse it.
- Different goals complicate compilers.
  - One part of the compiler abuses UB but another part might be a runtime detection of UB!

Links

- My company www.xmmswap.com
- LuaJIT www.luajit.org
- CompCert compcert.inria.fr
- Peter Cawley www.corsix.org
- Wojciech Mula www.0x80.pl
- Shafik Yaghmour shafik.github.io