What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance

M. Anton Ertl*

TU Wien

Abstract. In recent years C compiler writers have taken the attitude that tested and working production (i.e., *conforming* according to the C standard) C programs are buggy if they contain undefined behavior, and they feel free to compile these programs (except benchmarks) in a way that they no longer work. The justification for this attitude is that it allows C compilers to optimize better. But while these "optimizations" provide a factor 1.017 speedup with Clang-3.1 on SPECint 2006, for non-benchmarks it also has a cost: if we go by the wishes of the compiler maintainers, we have to "fix" our working, conforming C programs; this requires a substantial effort, and can result in bigger and slower code. The effort could otherwise be invested in source-level optimizations by programmers, which can have a much bigger effect (e.g., a factor > 2.7 for Jon Bentley's traveling salesman program). Therefore, optimizations that assume that undefined behavior does not exist are a bad idea not just for security, but also for the performance of non-benchmark programs.

1 Introduction

Compiler writers are sometimes surprisingly clueless about programming. For example, the first specification for Fortran states: "no special provisions have been included in the FORTRAN system for locating errors in formulas" and "FORTRAN should virtually eliminate coding and debugging", and this approach to error checking made it into the finished product; there were also no programmer-defined functions in the original design, but that was fixed before release [Bac81].

More recently, people have meant one of several different programming languages when they wrote about C; for clarity, we will use different names for these programming languages:

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

C⋆ A language (or family of languages) where language constructs correspond directly to things that the hardware does. E.g., ∗ corresponds to what a hardware multiply instruction does. In terms of the C standard, *conforming programs* are written in C⋆.

"C" The subset of C⋆ that excludes all undefined behavior according to the C standard. In C-standard terms, programs written in "C" are *stricty conforming programs*.

C_{bench} A subset of C⋆ and (slight) superset of "C" that includes the benchmarks considered relevant by the compiler maintainers (e.g., the SPEC CPU benchmarks).

We look at the differences between C⋆ and "C" in Section 2.

Production programmers typically think in C⋆ when programming, and as a result, they program in C⋆, so most production code is not written in "C" (see Section 2). The cluelessness of many recent C compiler maintainers is that they officially support only "C" and officially feel free to compile any source code that performs undefined behavior into arbitrary machine code,[1] even programs that were tested and worked as intended with earlier versions of the same compiler. As John Regehr puts it: "A sufficiently advanced compiler is indistinguishable from an adversary."[2] Inofficially, these compilers support C_{bench}, but that is of little benefit to other programs.

Unlike the authors of the first Fortran compiler, the current C compiler maintainers stubbornly insist on their view.[3] The reason for this seems to be that they evaluate their work through benchmark results of a certain set of benchmarks; by only having to compile these benchmark programs as intended, they want to give their optimizer freedom to produce better benchmark results.

Better benchmark numbers alone are a weak justification for not compiling production programs as intended, so the C compiler maintainers also claim that these "optimizations" give speedups for other programs. However, that would require programmers to convert their C⋆ programs to "C" first, a process which can produce worse code (Section 3), and more importantly, requires an effort that would be much more effective if directed at source-level optimizations. We look at the performance benefits of source-level optimizations in Section 4 and compare it to the differences seen from "optimizations" based on undefined behavior.

Section 5 discusses what compiler writers, standards committees, programmers, and researchers can and should do about these issues.

2 The difference between C⋆ and "C"

Both languages have the same syntax and the same static semantics; the difference is in the run-time semantics.

[1] The classical intimidation was "may format your hard drive", but recently "make demons fly out of your nose" (in short: *nasal demons*) seems to be more popular.

[2] http://blog.regehr.org/archives/970

[3] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

C⋆ maps language elements to corresponding hardware features, and is consistent about this, at least on the platform. The actual behavior at run-time may be different between platforms, lead to an exception, or worse, overwriting of an unrelated data structure (and it is thinkable, although very improbable, that this eventually results in a formatted hard disk on some platforms), but can always be explained by a sequence of hardware steps corresponding to the source program.

"C" is a subset of C⋆ that tries to specify what is portable between different hardware platforms (including some that have died out) and between different C compilers. Therefore it specifies that the behavior of many language elements under some circumstances is *undefined*, implementation-defined[4], or similar. The original idea was that C compilers implement C⋆, and that undefined behavior gives them wiggle room to choose an efficient hardware instruction; e.g., for `<<` use the hardware's shift instruction, and differences between different architectures for some parameters result in not defining the behavior in these cases. But in recent years, compiler maintainers have gone beyond that and "optimized" programs based on the assumption that undefined behavior does not happen, in ways that do not correspond to any direct mapping from language element to the actual hardware; e.g., they "optimize" a bounded loop into into an infinite loop.

There are 203 undefined behaviors listed in appendix J of the C11 standard (up from 191 in C99). And these are not just obscure corner cases that do not occur in real programs, on the contrary, it is likely that most, if not all, production programs exhibit undefined behavior; even in GCC and LLVM itself (i.e., the pinnacles of the church of "C"), undefined behior has been found even when just compiling an empty C or C++ program with optimizations turned off.[5] Standards conformance was a requirement to be considered for inclusion in SPEC CPU 2006, yet 6 out of 9 C programs perform C99-undefined integer operations [DLRA12], and these are not the only undefined behaviors around by far.

2.1 Optimization⋆ and "Optimization"

An optimization is a program transformation that preserves the observable behavior of the program and hopefully results in a program that consumes fewer resources (run-time and code size are typical metrics).

There are a large number of effective optimizations that can be used on C⋆ programs (called optimizations⋆ in the following), e.g., strength reduction, inlining, or register allocation. A simple example would be to optimize a multiplication by 5 into a `lea` instruction on the AMD64 architecture.

GCC and LLVM have been adding "optimizations" based on undefined behavior in "C". These work by assuming that the program exhibits no undefined

[4] And apparently the implementation is allowed to define the implementation-defined behavior as undefined.

[5] http://blog.regehr.org/archives/761

behavior, and deriving various "facts" from this assumption, e.g., about values of variables, propagates these "facts" throughout the program, and uses them in other places for "optimizations". An example is the following function from the SPEC benchmark 464.h264ref:

```
int d[16];

int SATD (void)
{
  int satd = 0, dd, k;
  for (dd=d[k=0]; k<16; dd=d[++k]) {
    satd += (dd < 0 ? -dd : dd);
  }
  return satd;
}
```

This was "optimized" by a pre-release of gcc-4.8 into the following infinite loop:

```
SATD:
.L2:
jmp .L2
```

What happened? The compiler assumed that no out-of-bounds access to d would happen, and from that derived that k is at most 15 after the access, so the following test `k<16` can be "optimized" to `1` (true), resulting in an endless loop. Then the compiler sees that the return is now unreachable, that satd is dead, that dd is dead, and k is dead, and optimizes the rest away.

The GCC maintainers subsequently disabled this optimization for the case occuring in SPEC,[6] demonstrating that, inofficially, GCC supports C_{bench}, not "C".

This kind of "optimization" certainly changes the observable behavior, but its advocates defend it by saying that programs broken by these "optimizations" have been buggy all along. Given the widespread occurence of undefined behavior in production code, this means that the compiler maintainers feel free to compile pretty much every production program in a way that it behaves differently than intended and different from the code produced by an earlier version of the same compiler and tested successfully.

If undefined behaviour is so widespread, why do we notice code broken by "optimizations" only occasionally? Undefined behavior is a run-time property, and only a small portion of these occurs in a way that can be turned into (compile-time) "optimization". Many of of these cases are checks for special cases that do not occur in most executions, such as a check for a buffer overflow; so "optimizing" such checks away may go unoticed unless the program tests for

[6] It still strikes for other programs, see gcc bug 66875.

these specific checks; and such testing may be hard to achieve in larger programs, because 100% test coverage is hard to achieve.

Wang et al. [WZKSL13] have written a static program analyzer that tries to find code that may be "optimized" away in "C", but not optimized* away in C*. They found that 3,471 packages out of 8,575 packages in Debian Wheezy contain a total of about 70,000 such pieces of code (as far as their checker could determine). In most cases "optimizing" these pieces of code away would result in code different from what the programmer intended (programmers rarely write code that they intend to be optimized away). These numbers are pretty alarming, but probably far lower than the number of undefined behaviors and packages containing them.

More on optimization* Actually I was a little bit too cavalier about optimizations* not changing the observable behaviour of C* programs. It is actually possible to write programs in C* where any change in the generated code produces an observable difference, e.g., a program that outputs the bytes in its object code.

But C*programmers would not complain about that. They generally don't expect this level of stability. After all, the bytes in the object code change every time there is a change in the program, e.g., due to a bug fix or a new feature.

What they do expect is, in the first order, the direct results of language elements must not change if they are observable (i.e., influence output or exceptions/signals). So optimization such as strength reduction, dead code elimination, or jump optimization are fine.

Register allocation can change the results of accesses to uninitialized variables. This is also accepted by C* programmers: they don't rely on the values of uninitialized variables, because these values often change during maintenance even in the absence of register allocation.[7]

Of course, "optimization" defenders like to tell stories about bug reports about changes in behavior due to, e.g., changed values of uninitialized variables when optimization is turned on, implying that there is no difference between optimization* and "optimization". But my guess is that in most such cases the bug reporters were not aware that the reason for the breakage is an uninitialized variable, and once they are aware of that, they are likely to change the program to avoid using such values, because the program would also be likely to break on maintenance.

And these kinds of reports are not that frequent: Of the 25 bug reports for gcc components rtl-optimization and tree-optimization resolved or closed between 2015-07-01 and 2015-07-16, three were marked as invalid, and all three were due to "optimizations", none due to optimizations*; Bug 66875 was very similar to the SATD example shown above (but the bug reporter accepted that his program is buggy due to the out-of-bounds array access), and Bugs 66804 and 65709 are both due to gcc using aligned rather than unaligned instructions

[7] Alternatively, a compiler striving for a more deterministic behaviour could initialize all uninitialized variables to a fixed value, at a small cost in performance.

when autovectorizing code with unaligned accesses on the AMD64 architecture (which normally does not impose alignment restrictions).

"Optimization" and benchmarks Compiler maintainers justify "optimizations" with performance, and they have benchmark results to prove it; actually, not really (see below). But if they had such benchmark results, would they really prove anything? There are significant differences between production programs and benchmarks:

- Production programs are maintained, including performance tuning by programmers if desired.
- Benchmark programs are fixed, and are normally not changed anymore. Therefore they cannot benefit from further source-level optimizations by programmers.
- Benchmark programs are fixed, and therefore exempt from the policy of compiler maintainers that it is ok to break code with undefined behavior, as we have seen from the released gcc not breaking `SATD()` from SPECint. Benchmark programs are not rewritten to eliminate undefined behaviors as compiler maintainers demand of non-benchmark programs, and therefore do not suffer from the worse code that such rewrites can cause (see Section 3).

Therefore, even if "optimizations" produce speedups for benchmarks, that does not say anything about the performance effect of enabling "optimizations" on production code.

But do "optimizations" actually produce speedups for benchmarks? Despite frequent claims by compiler maintainers that they do, they rarely present numbers to support these claims. E.g., Chris Lattner (from Clang) wrote a three-part blog posting[8] about undefined behavior, with most of the first part devoted to "optimizations", yet does not provide any speedup numbers. On the GCC side, when asked for numbers, one developer presented numbers he had from some unnamed source from IBM's XLC compiler, not GCC; these numbers show a speedup factor 1.24 for SPECint from assuming that signed overflow does not happen (i.e., corresponding to the difference between `-fwrapv` and the default on GCC and Clang).

Fortunately, Wang et al. [WCC+12] performed their own experiments compiling SPECint 2006 for AMD64 with both gcc-4.7 and clang-3.1 with default "optimizations" and with those "optimizations" disabled that they could identify, and running the results on a on a Core i7-980. They found speed differences on two out of the twelve SPECint benchmarks: 456.hmmer exhibits a speedup by 1.072 (GCC) or 1.09 (Clang) from assuming that signed overflow does not happen. For 462.libquantum there is a speedup by 1.063 (GCC) or 1.118 (Clang) from assuming that pointers to different types don't alias. If the other benchmarks don't show a speed difference, this is an overall SPECint improvement by a factor 1.011 (GCC) or 1.017 (Clang) from "optimizations".

[8] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

2.2 The intended meaning of programs

Why do programmers program in C*, not in "C"? Programmers are usually not language lawyers, nor should they be required to. They usually learn a programming language by reading introductory books, by looking at programs written in the language, and by trying out what the compiler does for various programs, and build a relatively simple model from that, possibly incorporating other knowledge (e.g., about hardware).

GCC maintainers have claimed that they don't know what the intended meaning of programs with undefined behaviour is. However, their own compiler is evidence against this claim. It compiles exactly the code intended by the programmer unless it sees enough of the program to derive "facts" that are then used for "optimization". E.g., in the SATD example, if the compiler did not know the number of elements of d (e.g., because d is defined in a different compilation unit), the compiler would not "optimize" the code and would compile it exactly as intended.

And that's what programmers expect: In the normal case a read from an array (even an out-of-bounds read) performs a load from the address computed for the array element; the programmer expects that load to produce the value at that address, or, in the out-of-bounds case, it may also lead to a segmentation violation (Unix), general protection fault (Windows), or equivalent; but most programmers do not expect it to "optimize" a bounded loop into an endless loop.

The expectations of programmers are reinforced, and partially formed, by the behaviour of compilers. Most of the time programmers experience the compiler without "optimizations" kicking in.

So, contrary to claims by "optimization" advocates[9], the compiler does not need psychic powers to determine the intent of the programmer in case of undefined behaviour, it knows the intent already.

A common assumption is that this expected behaviour corresponds to the behaviour when turning off optimization. That is often, but not always the case. In particular, at least some versions of GCC "optimize" `x+1<=x` even with `-O0`.

The programmers' model may or may not include some caveats (such as different sizes of pointers between 32-bit and 64-bit platforms, or that one should avoid unaligned accesses on some hardware), but not all the fine details of undefined behavior in the C standard. And a programmer who programs for just one platform will often not feel compelled to heed portability caveats even if he knows them. So, we cannot exclude non-portable code from C*.

What is more likely to not be in C* is non-maintainable assumptions (such as the values of uninitialized variables). Still, be very cautious before introducing new optimizations that rely on the assumption that a certain usage is unmaintainable and therefore won't occur.

Coming back to the psychic powers question, while the compiler cannot know which undefined behaviours a particular program exhibits, it is safe to use a conservative approximation (i.e. C*). One can ask whether the optimization under

[9] news:<h02dnSk5W8n4p57OnZ2dnUVZ_qSdnZ2d@supernews.com>

consideration would break things that are likely to be stable across maintenance, or one can use the large corpus of working free software to get an idea which undefined behaviours do occur in production programs. Deriving such knowledge from bug reports is also an option, if all else fails; e.g., if there are several bug reports on alignment problems from autovectorization, obviously unaligned accesses are used by programmers on architectures that support them, and therefore autovectorization should by default use SIMD instructions without alignment restrictions on these architectures.

2.3 Security

I have read arguments that use security to justify compiling programs with undefined behavior different from what was intended. While this paper focuses on the performance and optimization claims, this section discusses this argument briefly.

The argument tends to go something like this: Because there are out-of-bounds array accesses that are security vulnerabilities (in particular, buffer overflow vulnerabilities), and out-of-bounds array accesses are undefined behaviors, it is good for security to compile undefined behaviors differently from the expectations of the programmer. There may be the assumption in this argument that programmers are encouraged to produce fewer out-of-bounds array accesses if compilers do that.

That argument is wrong for several reasons:

- Not all vulnerabilities perform undefined behavior (e.g., privilege escalation or SQL injection).
- Undefined behavior is just as hard to find as vulnerabilities (both usually only show up at run-time for certain corner-case inputs), so encouraging programmers to eliminate undefined behaviors will not make it easier to find vulnerabilities. For a given amount of effort, it is more likely that programmers will find more vulnerabilities if they focus on vulnerabilities than if they also have to look out for undefined behavior.
- Buffer overflow vulnerabilities will typically not be "optimized" into code different from what the programmer intended, because the compiler usually cannot statically determine for such code that the out-of-bounds access happens. If the compiler can determine it, reporting it to the programmer makes much more sense than to "optimize" the code.
- Not all undefined behaviors result in a vulnerability; on the contrary, some security checks perform undefined behavior, and have been "optimized" away [WCC+12,WZKSL13]. There the "optimizing" compiler created a vulnerability that was not present in the C* source code.

2.4 Expressive power

One might think that one can express in "C" all that one can express in C*, but to my surprise that's not the case. So while we are doing a detour from the performance theme of the rest of the paper, let us look at an example of that.[10]

Given that C was designed as a systems programming language, implementing a simple function like `memmove()` in terms of lower-level constructs should be possible in C, e.g, as follows:

```
void *memmove(void *dest, const void *src, size_t n) {
  if (dest<src)
    memcpy_pos_stride(dest,src,n);
  else
    memcpy_neg_stride(dest,src,n);
}
```

where `memcpy_pos_stride()` copies the lower-addressed bytes of `src` before higher-addressed bytes, and `memcpy_neg_stride()` copies the higher-addressed bytes before the lower-addressed ones (I believe that these helper functions can be implemented in "C", but I also believed that of `memmove()` before looking closely into it).

However, this implementation is C*, but not "C", because `p<q` is not defined in "C" if p and q point to different objects. Apparently the reason is to allow more efficient implementation of these operations on segmented architectures (by comparing only the offsets). However, it is likely that both variants above work in C* on such platforms, too, because the result of the comparison does not matter if the pointers point to different objects/segments (in that case either memcpy variant will be correct).

However, a "C" compiler might assume that src and dest point to the same object, propagate that "knowledge" to all users of `memmove()`, and perform various "optimizations" based on that, even on hardware with a flat address space.

Another `memmove()` implementation is based on `malloc()`ing an intermediate copy, but `malloc()` can fail, whereas the standard `memmove()` cannot.

3 Code quality

Your production program is not a benchmark, so compiler maintainers demand that you change your program into a "C" program. Let's see what this can do to the code quality of a simple example.

We want to determine whether a variable x of some signed integer type is the smallest value that type can hold. A succinct way to express this in C* (and in Java) is `x-1>=x`. This may seem puzzling if you think about these types as

[10] For a longer discussion, see news:<2015May1.155805@mips.complang.tuwien.ac.at> ff.

mathematical integers, but note that these are bounded types; if x is the smallest value, then `x-1` cannot be smaller. Note that this does not depend on the signed number representation; some machines may produce an exception on underflow, but, when x is the smallest value, none produces a value that will produce false for this comparison. However, gcc assumes that signed integer underflow does not happen, and uses this assumption to "optimize" this test to always produce false.[11]

So how do we rewrite this into "C"? If we assume that the type of x is `long`, we can write `x==LONG_MIN`.[12] Let us look at the code quality. For the `x-1>=x` variant, we use the gcc option `-fwrapv` that some (but not all) versions of gcc offer (as non-default option) to allow compiling many programs with signed integer overflows as intended. On AMD64 gcc-5.2.0 produces:

```
x-1>=x
48 8d 47 ff                     lea -0x1(%rdi),%rax
48 39 c7                        cmp %rax,%rdi
7f 06                           jg ...

x==LONG_MIN
48 b8 00 00 00 00 00 00 00 80 movabs $0x8000000000000000,%rax
48 39 c7                        cmp %rax,%rdi
75 05                           jne ...
```

Three instructions each, but the "C" variant takes 15 bytes (both with `-O3` and `-Os` (size optimization)), whereas the C* variant takes 9. Another way to implement this check would be

```
48 ff cf                        dec %rdi
71 05                           jno ...
```

This takes two instructions and 5 bytes. An optimizing compiler would ideally produce this machine code from all ways of expressing this check; that would be an optimization*, and a particularly useful one, because one cannot express this code more directly in C (the overflow flag is not a feature in the C programming language).

Another example[13] is the implementation of 32-bit rotation: The straightforward implementation in C*is `(x<<n)|(x>>32-n)` and compiles to the intended code in current compilers but performs undefined behavior in "C" when $n = 0$.[14] A "C" version with the obvious zero check generates additional instructions on both GCC and Clang. The blog entry also looks at another variant

[11] This even happens for `-O0` on, e.g., gcc-4.1.2 on Alpha.

[12] The type of x in the actual program may be a different signed integer type, resulting in much more code to handle all these possible types, while `x-1>=x` is independent of the type.

[13] From http://blog.regehr.org/archives/1063

[14] Presumably the reason for the undefined behaviour is that the machine instruction for `x>>32` produces x on some CPUs and 0 on others; note that either behavior produces the correct result for this idiom.

`(x<<n)|(x>>(-n&31))`,[15] which gcc managed to compile into a `rol`, but Clang didn't and produced code that was longer by four instructions.

So, converting C* code to "C" can lead to worse code even on those compilers whose maintainers say that we should do the conversion. You don't see this effect on benchmarks, because the benchmarks are not changed to eliminate undefined behaviour.

4 Source-level optimizations

Programmers can be very effective at improving the performance of code, and in particular, can do things that the compiler cannot do. Here we first look at three examples.

4.1 SPECint

Wang et al. [WCC+12] did not just present the difference that "optimizations" make for SPECint numbers (Section 2.1), they also looked at the reasons for these performance differences, and found two small source-level changes that made the less "optimizing" compiler variants produce just as fast code as the default compilers.

For 456.hmmer, the problem is that an `int` index is sign-extended in every iteration of an inner loop unless the sign-extension is "optimized" away by assuming that the int does not wrap around. The source-level solution is to use an address-length or unsigned type for the index; Wang et al. used `size_t`, and the slowdown from disabling "optimizations" went away for 456.hmmer.

For 462.libquantum, the problem is a loop-invariant load in the inner loop that could not be moved out of the loop without assuming strict aliasing, because there is a memory store (to a different type) in the loop. The source-level solution is to move that memory access out of the loop. Wang et al. did that, and the slowdown from disabling "optimizations" went away for 462.libquantum. Note that, if the store was to the same type as the load, "optimization" could not move the invariant load out, while source-level optimization still can.

Also note that you just need to look at the inner loops to find and perform such optimization opportunities, while you have to work through your whole program to convert it to "C", plus you may incur slowdowns from the conversion.

4.2 Jon Bentley's Traveling Salesman programs

In "Writing Efficient Programs" [Ben82], Jon Bentley used a relatively short traveling-salesman program that uses a greedy (non-optimal) algorithm as running example for demonstrating various optimizations at the source code level; each optimization step resulted in a new program. The programs in the book were written in Pascal. I transliterated them to C in 2001, keeping the original optimizations and a Pascal-like style (arrays instead of explicit pointer arithmetics),

[15] `n` needs to be unsigned in order for this variant to be "C".

except that I did not perform Bentley's step 7 of switching from floating point to integer arithmetics. This results in the programs `tsp1`...`tsp9` (but without `tsp7`).[16]

Given the Pascal-based style, I expect that these programs benefit from optimizations like strength reduction and induction variable elimination more than many other C programs. They are probably also closer to "C" than many other C programs for the same reason; to test this, I compiled these programs with `gcc-5.2.0 -m32 -fsanitize=undefined -fsanitize-undefined-trap-on-error`, and they ran through (but note that these runs may still perform undefined behaviors that the checker does not check, or for other inputs).

Experimental setup: We used several compiler versions and different options:

gcc We used gcc versions 2.7.2.3 (1997), egcs-1.1.2 (1999, shortly before gcc-2.95), and gcc-5.2.0 (2015). The earlier versions already "optimize" `x-1>=x` (with no way to disable this "optimization"), but overall probably perform much less "optimization" than gcc-5.2.0; e.g., egcs-1.1.2 has an option `-fstrict-aliasing`[17], but (unlike gcc-5.2.0) does not enable it by default, and gcc-2.7.2.3 does not even have such an option. In addition to "optimizations", hopefully gcc also added optimizations* in these 18 years. We use the `-m32` option for gcc-5.2.0 to produce IA-32 code, because the earlier compilers do not produce code for the AMD64 architecture that was only introduced in 2003.

Clang/LLVM We also use clang 3.5 (2014) for breadth of coverage. We use `-m32 -mno-sse` for comparability with the gcc results which also produce code for IA-32 without SSE.

`-fno...` In addition to using `-O3` and default optimizations (including "optimizations") we also compiled with `-O3` but disabled as many "optimizations" as we could identify: For gcc-5.2.0 we used `-fno-aggressive-loop-optimizations -fno-unsafe-loop-optimizations -fno-delete-null-pointer-checks -fno-strict-aliasing -fno-strict-overflow -fno-isolate-erroneous-paths-dereference -fwrapv`, for clang 3.5 we use `-fno-strict-aliasing -fno-strict-overflow -fwrapv`.[18] These compilers still might perform various "optimizations" that are not covered with these flags, but that is as close to turning these compilers into C* compilers as we can get.

`-O0` It is often suggested to turn off optimization in order to get a C* compiler. While this does not work (some gcc versions "optimize" `x-1>=x` even at `-O0`), we tried `-O0` to see how much it hurts performance.

[16] http://www.complang.tuwien.ac.at/anton/lvas/effizienz/tsp.html

[17] With strict aliasing the compiler assumes roughly that pointers to different types do not point to the same object.

[18] For clang I used the subset of the gcc options that clang accepts, because I could not find documentation on any such options.

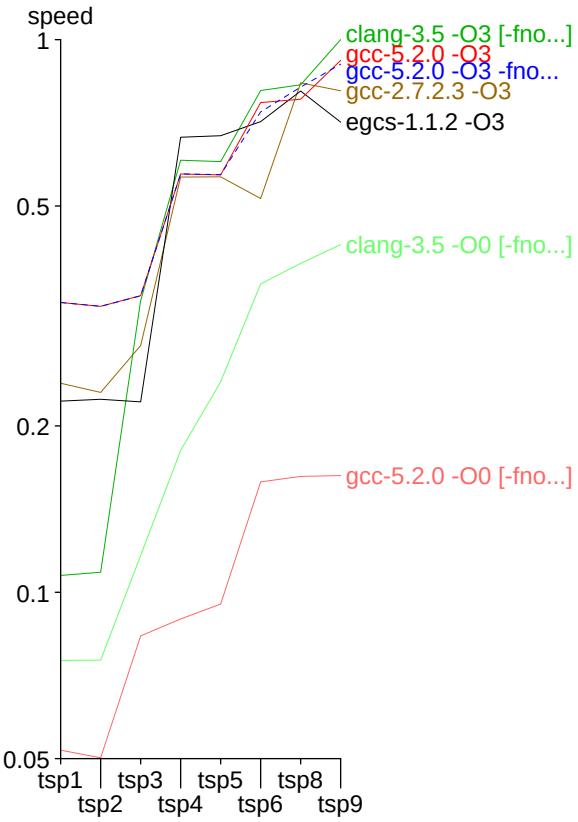

Fig. 1. Performance across different compilers of Jon Bentley's Traveling Salesman program variants for visiting 5000 cities

We ran the resulting binaries on a Core i3-3227U (Ivy Bridge), with 5000 cities. We measured the run-time in cycles using CPU performance counters with `perf stat -r 100 -e cycles`; this runs the program 100 times and reports "average" (probably arithmetic mean) and standard deviation; the standard deviation for our measurements was at most 0.62%.

Figure 1 shows the results. In cases where the binaries where the same with and without the `-fno...` options, one line is shown for both, with the label containing `[-fno...]`.

Source-level optimization speedup: Overall, the source-level optimizations provide a good speedup (starting at a factor 2.7 between tsp1 and tsp9 for `gcc-5.2.0 -O3`) across all compilers and options, with some optimization steps having little effect on performance, while others have a larger effect. This also disproves claims that compiler optimizers are now so good that they make source-

level optimization unnecessary; on the contrary, compilers do not perform most of these source-level optimizations from a 33-year old book. If compilers performed these optimizations themselves, we would expect the `-O3` lines to be flat, but in fact the speedups from source-level optimizations provide similar speedups to the `-O3`-compiled versions as to the `-O0`-compiled versions; only the optimization from tsp4–tsp5 (inlining one function) is performed by the compilers with `-O3`, resulting in horizontal line segments for this step. Section 4.4 looks at why source-level optimization is effective.

"Optimization" speedup: We first compare the binaries generated with and without the `-fno...` options. For `clang-3.5 -O3`, `clang-3.5 -O0` and `gcc-5.2.0 -O0`, all of the programs are compiled to the same binary code without and with `-fno...` options. So for these compilers show only one set of results. For `gcc-5.2.0 -O3`, there are no differences in the binaries for tsp1...tsp3, but there are for tsp4...tsp9, so we measure both sets of binaries for `gcc-5.2.0 -O3`.

For tsp4/5 (and of course for tsp1–3, where there is no difference in the binaries) the code generated by `gcc-5.2.0 -O3` has the same speed as the code compiled with the `-fno...` options; for tsp6 it is faster by a factor of 1.04, for tsp8 it is *slower* by a factor of 1.05, and for tsp9 it is faster by a factor of 1.02. So disabling these "optimizations" has only a minor and inconsistent effect on performance.

`-O0` vs. `-O3`: By contrast disabling both "optimization" and optimization* with `-O0` has a dramatic effect on performance, especially for gcc (factor 5.6 for tsp9). So using `-O0` is not a good suggestion as a C* compiler if you care for performance: In addition to still performing some "optimizations", it also produces slow code.

Other results: Every compiler has some program for which it is fastest: gcc-2.7.2.3 is fastest for tsp8, egcs-1.1.2 is fastest for tsp4/5, gcc-5.2.0 is fastest for tsp1–3, and clang-3.5 is fastest for tsp6 and tsp9. Overall, the performance with `-O3` is remarkably close given the 18 years span the gcc versions; Proebsting's law tongue-in-cheekly predicts a factor of 2 between gcc-2.7.2.3 and 5.2.0.

The slow speed of `clang-3.5 -O3` for tsp1/2 is probably due to differences in `sqrt()` implementation, because tsp1/2 have a large number of calls to `sqrt()`, while that number is much smaller for tsp3–tsp9.

4.3 Gforth

Gforth is an implementation of the Forth programming language. Its virtual machine was implemented as a threaded-code interpreter starting in 1992 [Ert93]; already that version heavily used GNU C extensions to achieve better performance than is possible in standard C* (let alone "C"). And while it is extremely far from being a strictly conforming (and thus portable) program according to

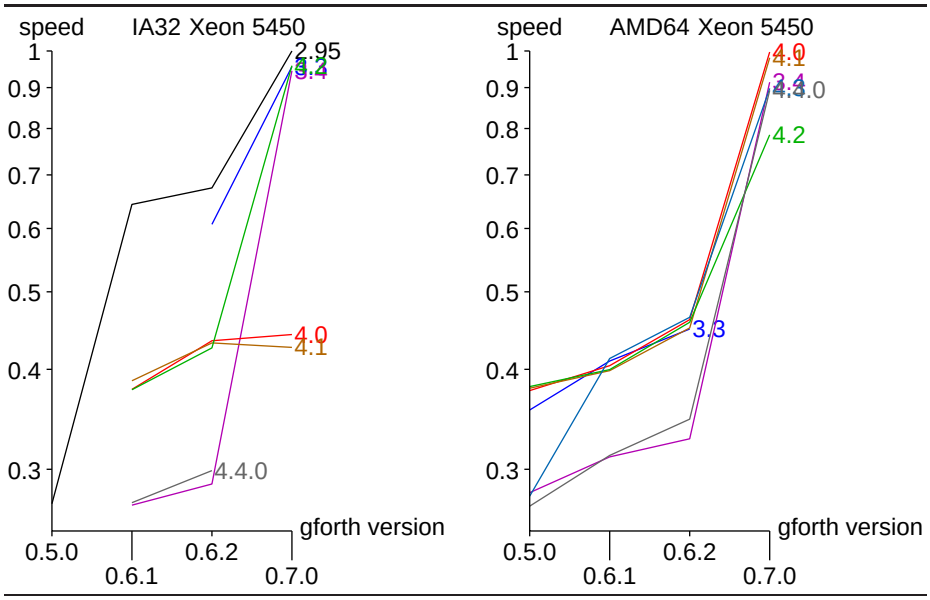

Fig. 2. Gforth performance with different GCC versions

the C standard, it is pretty portable: e.g., we tested Gforth 0.7.0 on eight different architectures, five operating systems, and up to nine gcc versions per architecture.

In 2009 we compared the performance of different Gforth versions across different CPUs and different gcc versions [Ert09], and below we discuss the results as relevant for the present paper. We measured different Gforth versions, from Gforth 0.5.0 (2000) to 0.7.0 (2008) compiled with various GCC versions from 2.95 (1999) to 4.4.0 (2009).

We ran five application benchmarks on a 3GHz Xeon E5450, each one three times per configuration, taking the median of the three runs and the geometric mean over these inividual benchmark results. The data we present here is the same as in Figures 8 and 9 of the Gforth performance paper [Ert09], but we present it differently.

Figure 2 shows 32-bit and 64-bit[19] results for different Gforth and GCC versions. As you can see, the source-level optimizations between Gforth 0.5.0 (2000) and 0.7.0 (2008) provide a speedup by a factor > 2.6 on most compiler versions. Some compiler versions don't perform as well as others, either because a source-level optimization was disabled[20], or because of some compiler-specific

[19] For AMD64, generic code is used up to 0.6.2, special AMD64 support was added only in 0.7.0.

[20] Gforth checks whether the assumptions used by the optimization hold, and falls back to older techniques if they do not.

problem (like bad register allocation: gcc-4.0 and -4.1 on IA32); for details, see the original paper [Ert09].

We also prototyped an optimization that moves Gforth further into JIT compiler territory, with less per-target effort than required by a conventional JIT compiler [EG04], and this optimization produced a median speedup of 1.32 on an Athlon, and 1.52 on a PPC 7400.

We had plans to turn this optimization into a production feature, but eventually realized that the GCC maintainers only care about certain benchmarks and treat production programmers as adversaries that deserve getting their code broken. Given that the Gforth source code is extremely far from "C", and this optimization would have taken it even further from "C", we dropped the plans for turning this feature into a production feature. So, in this instance, the focus on "C" and "optimizations" by the C compiler maintainers resulted in less performance overall.

The current GCC and Clang maintainers suggest that we should convert programs to "C". How would that turn out for Gforth?

We could not use explicit register allocation, and therefore lose much of the performance advantage of gforth-0.7.0. More importantly, we have to drop dynamic superinstructions, and, since everything else builds on that, that would throw us back to Gforth-0.5.0 performance. In addition, we would have to drop threaded code, so we would lose even more performance. We could get back a little performance by implementing static superinstructions and static stack caching in a new way (without dynamic superinstructions), but overall I expect a slowdown by a factor > 3 compared to gforth-0.7.0 from these changes alone.

Gforth is outside "C" in many other respects, in particular in its data and memory model. It is unclear to me how that part could be turned into efficient "C" code, but it certainly will not increase performance.

Changing the code to "C" would not just reduce performance by a lot, but also require a huge effort. Instead of spending that effort to make Gforth slower, we are considering switching to native code compilation, and getting rid of C as much as possible. Machine code, while not as portable as C used to be, offers the needed expressive power, reliability, and stability, that gcc used to give us, but no longer does.

4.4 Why are programmers effective?

Gcc-5.2.0 does not perform most of the optimizations that Bently performed in his 33 years old book. Why?

One big advantage that the programmers have is that they have to satisfy the requirements document (or the specification) of the program, while the compiler uses the source code as specification and has to keep to that. The source code overspecifies the program, so the compiler cannot perform the same optimizations that the programmer can. For example, tsp8 does not produce the same stdout output as tsp6, so a compiler could not perform that change.

Another advantage that programmers have is that they can have a better understanding of the design of the program, and therefore can perform trans-

formations that the compiler cannot perform because it cannot determine that the transformation is safe or profitable to perform. As an example, in Gforth we have numbers for the VM instructions in the image file format, and replace them with code addresses when loading the image file, eliminating the need to do the code address lookup at run-time; compilers do not do that, because they cannot prove that the instruction numbers are not used in any other way.

Finally, programmers can apply optimizations for idioms for which compiler maintainers do not develop optimizations because they do not occur frequently enough in "relevant" code (i.e., their benchmarks).

Of course, source-level optimization costs in development, and may also cost in maintenance. However, the recommendation by "optimization" advocates of "fixing" or "sanitizing" your code (i.e., converting it to "C") also costs in development and in maintenance.

As can be seen here, the optimization by programmers is far more effective than "optimization" by compilers, so it makes more sense to have a compiler with only optimization* and invest the development budget for optimization in source-level optimization rather than "sanitizing". In particular, in source-level optimization you can concentrate on the parts of the programs relevant to performance, and stop when the benefit/cost ratio of further optimizations becomes too small, while "sanitizing" has to cover the whole program, as any undefined behavior in the program allows nasal demons, or "optimizations", to happen.

5 Perspectives

Given the state of things, what should be done?

5.1 Compilers

Compiler maintainers should change their attitude: Programs that work with previous versions of the compiler on the same platform are not buggy just because they exhibit undefined behaviour; after all, they are conforming C programs. So the compiler should be conservative and disable "optimizations" by default, in particular new or enhanced "optimizations". If you really want the default to be "C", then at least provide a single, stable option for disabling "optimizations"; in this way, a C* program compiled with that option will also work with the next version of the compiler.

Also, there is still a lot of improvement possible in terms of optimizations*, and it would be a good idea to shift effort from "optimizations" to optimizations*.

I do not see a good reason for having "optimizations" at all, but if a compiler writer wants to implement them, it would be a good idea to be able to warn when "optimizations" actually have an effect. Contrary to what Chris Lattner claims[21], this is not that hard: Just keep track of both facts* and "facts". When

[21] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html

eventually generating code, if the code generated based on just facts* would be different from the code generated based on "facts", provide a warning. Providing a good explanation for the warning may be hard, but that's just an indication of how unintuitive the "optimization" is.

5.2 Standards

The compiler maintainers try to deflect from their responsibility for the situation by pointing at the C standard. But the C standard actually takes a very different position from what the compiler maintainers want to make us believe. In particular, the C99 rationale[22] [C03] states:

> **C code can be non-portable.** Although it strove to give programmers the opportunity to write truly portable programs, the C89 Committee did not want to force programmers into writing portably, to preclude the use of C as a "high-level assembler": the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between strictly conforming program and conforming program (§4).
>
> **Keep the spirit of C.** The C89 Committee kept as a major goal to preserve the traditional spirit of C. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like:
>
> – Trust the programmer.
> – Don't prevent the programmer from doing what needs to be done.
> – Keep the language small and simple.
> – Provide only one way to do an operation.
> – Make it fast, even if it is not guaranteed to be portable.
>
> The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be how the target machine's hardware does it[23] rather than by a general abstract rule.

Still, the C standard blesses compilers like GCC and Clang that intentionally compile only strictly conforming C programs as intended as conforming implementations. There is the hope that market forces will drive compilers towards higher quality-of-implementation; unfortunately, that does not seem to work out: The compiler maintainers' perception of quality-of-implementation is based on benchmark results, and this perception has led to the current situation; and since both GCC and Clang maintainers take the same view of non-"C" programs, and

[22] The C11 rationale has not been published yet.

[23] By contrast, "what the hardware does" seems to be an insult in GCC circles: news:<07qdndj_37WIDp7OnZ2dnUVZ_tmdnZ2d@supernews.com>.

there is no competition from an optimizing* C* compiler in the free-software world, there is little that market forces can do in that area.

So, while the standard is not responsible for the current situation (the maintainers of these C compilers are), the C standards committee might still tighten the standard such that it cannot be abused by compiler maintainers in this way.

Instead of not specifying behavior (or explicitly specifying undefined behavior) for many cases where different C* implementations may produce different results, the standard should enumerate the possible results in enough detail to discourage "optimizing" compilers.

Pascal Cuoq, Matthew Flat and John Regehr suggested such an approach[24]: They want to define a friendly dialect of C by working through the list of undefined behaviours in the C11 standard, and reduce the amount of undefinedness, e.g., by replacing "has undefined behaviour" with "results in an unspecified value". This is definitely going in the right direction. There will still remain a gap between the way programmers think about the language construct and the way the friendly C specification describes it, but if the friendly C specification is tight enough to rule out "optimizations", the remaining gap may still be a source of joy for language lawyers, but harmless for practical programmers. "Unspecified value" may not be quite tight enough, though, because it does not give a specified result for, e.g., `x-x` when x is uninitialized.

5.3 Programmers

The attitude of compiler maintainers puts programmers in a tough situation. The next version of the compiler could break their currently-working program. The simplest way to deal with that is to stick with the compiler version that works with your program. You may need to keep the old binaries, or compile the old sources with a newer compiler (I have built gcc-2.7.2.3 with gcc-4.8 for this paper).

If you want to make your program work for newer versions of the compiler (e.g., because you want to port to an architecture that is not supported by the old version), you can use the flags that define some of the undefined behaviours (such as those listed for our experiments in Section 4.2). You can also see if lowering the optimization level helps.

As a long-term perspective, you could also decide to switch to another programming language. Unfortunately, there are not that many languages available that occupy the C* niche: a low-level language that can be used as portable replacement for assembly language; in particular, C ate all its brethren in the Algol family (e.g., Bliss, BCPL). Forth is available, but is probably unattractive to most programmers coming from Algol-like languages. C-- [JRR99] was intended as a portable assembly language, but seems to have been relegated to an internal component of the Glasgow Haskell Compiler.

Instead of using a portable assembly language, you can go for real assembly language; in earlier times C was less cumbersome and therefore more attractive,

[24] http://blog.regehr.org/archives/1180

but that has changed. In contrast to C as currently implemented, assembly language is rock-solid. The disadvantage of assembly language is that it is not portable, but the number of different architectures in general-purpose computers has fallen a lot since the 1990s, so you may be able to make do with just one or two.

You probably don't want to write everything in assembly language, but most code in higher-level languages. There are a number of newer Algol-family languages that are slightly higher-level than C used to be that may be appropriate for the higher-level language part, e.g., Rust, D, and Go. I do not have experience with these languages, nor have I examined the specification and the attitude of the compiler maintainers, so unfortunately I cannot make any recommendations here.

Our perspective for Gforth is to use Forth as high-level language and assembly/machine language for the parts that cannot be done in Forth (for these parts we currently use GNU C).

5.4 Tools and Research Opportunities

Just as the existence of "C" compilers has spawned tools and research papers on finding undefined behaviors and code that may be "optimized" away although it should not, a focus on C* compilers and source-level optimization could lead to tools for finding source-level optimization opportunities.

E.g., consider the source-level optimizations Wang et al. [WCC+12] used on SPECint: To get rid of sign extensions, a tool could perform run-time checks to see if using `long` instead of `int` produces a different result, and if not, could then suggest to the programmer to change the type of inner-loop induction variables accordingly. A tool could also check whether a load in an inner loop always produces the same result during a run, and, if so, suggest to the programmer to move the load out of the loop manually; the source-level optimization also works in cases where "optimization" does not work because there is a store in the loop to the same type as the load.

The research questions are what other optimizations can be supported by such tools, and how effective they are. Of course, the possible optimizations are not limited to those that are possible by converting to "C" and enabling "optimizations".

6 Related work

Complications generate interesting research questions, even if they are unnecessary complications.

Wang et al. [WCC+12] describe a number of "optimizations" by "C" compilers, and give examples where the "optimizations" were not optimizations, but led to C* code not being compiled as intended; most of the examples are for security vulnerabilities caused by "optimizations". It also gives performance results showing that "optimizations" give a very small speedup even for SPECint.

There is work on finding undefined behavior with run-time checks, e.g., for integer computations [DLRA12]. Of particular interest for the current work are the empirical results of applying these checks to various programs, in particular, how widespread these undefined behaviors are.

Not all undefined behaviors can be exploited by "optimizations", and another paper by Wang et al. [WZKSL13] describes a tool for finding code that may be "optimized" (but not optimized*) away. In addition, they describe a number of security vulnerabilities caused by "optimizations" and also gives empirical results on how many program fragments are "optimized" away in how many programs.

7 Conclusion

"Optimizations" based on assuming that undefined behavior does not happen buy little performance even for SPECint benchmarks (1.7% speedup with Clang-3.1, 1.1% with GCC-4.7), and incur large costs: Most production programs perform undefined behaviors and removing them all requires a lot of effort, and may cause worse code to be produced for the program. Moreover, a number of security checks have been "optimized" away, leaving the affected programs vulnerable.

If you are prepared to invest that much effort in your program for performance, it is much better to invest it directly in source-level optimization instead of in removing undefined behavior. E.g., just two small source-level changes give the same speedups for SPECint as the "optimizations"; we also presented examples of source-level optimizations that buy speedup factors > 2.6.

Compiler writers should disable "optimizations" by default, or should at least give the programmers a single flag to disable them all and that also disables new "optimizations" in future compiler versions. A focus on optimizations* and on supporting source-level optimizations better would also be welcome. Finally, programs that work on a version of your compiler are conforming C programs and they are not buggy just because they perform undefined behavior.

References

Bac81. John Backus. The history of Fortran I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981. 1

Ben82. Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982. 4.2

C03. *Rationale for International Standard—Programming Languages—C*, revision 5.10 edition, 2003. 5.2

DLRA12. Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering (ICSE)*, 2012. 2, 6

EG04. M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004. 4.3

Ert93. M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993. 4.3

Ert09. M. Anton Ertl. A look at Gforth performance. In *25th EuroForth Conference*, pages 23–31, 2009. 4.3

JRR99. Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999. 5.3

WCC+12. Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Asia-Pacific Workshop on Systems (APSYS'12)*, 2012. 2.1, 2.3, 4.1, 5.4, 6

WZKSL13. Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013. 2.1, 2.3, 6