Document: P2646R0
Date: 2022-OCT-15
Project: Programming Language C++
Audience: EWG
Reply-to: Parsa Amini: me@parsaamini.net
Joshua Berne: jberne4@bloomberg.net
John Lakos: jlakos@bloomberg.net

Explicit Assumption Syntax *Can* Reduce *Run Time*

Abstract

Many compilers provide platform-specific *assumption* syntax, such as `__builtin_assume` in Clang or idiomatic use of `__builtin_unreachable()` in GCC. This augmented syntax can then indicate to the compiler that it is allowed but not required to assume that some condition — typically a Boolean-valued expression — is always true. Recently, after due consideration, the `[[assume]]` attribute was formally adopted into the C++ working draft (P1774R8) to provide a facility for expressing such assumptions *portably* in source code. As is well known and easily demonstrated, the use of such *compiler-accessible* assumption constructs can noticeably affect compile times as well as object-code and overall program sizes. On the other hand, some members of the C++ Standards Committee have suggested (wrongly) that modern compilers and CPUs conspire to realize essentially all runtime performance benefits available on modern architectures, thereby obviating use of explicit assumption constructs in source code.

Until recently, empirical studies that (1) demonstrate any meaningful impact of compiler-accessible assumption constructs on run time and (2) are reproducible on modern platforms have not been readily available. In this paper, we first exhibit some potential reasons *why* the use of an assumption attribute *might* reduce run time. We then go on to develop a benchmark framework suitable for demonstrating that using such assumption constructs *can* meaningfully reduce the overall run time of C++ programs. Even without a portable way to express assumptions, we exploit equivalent, platform-specific assumption constructs to reproducibly quantify the sometimes-substantial effects of explicitly stated assumptions on overall run times for today's most current platforms. We conclude that the recent addition of `[[assume]]` (P1774R8) to the C++ working draft will benefit those who aspire to use C++23 to implement software having truly optimal *runtime* performance.

Introduction

Explicit assumption syntax — i.e., language constructs used by the programmer to explicitly inform the compiler of some truth that it might otherwise be unable to deduce — is not new. Such syntax, albeit platform-

specific until now, has existed in C++ for nearly a quarter century.[1] As of Visual Studio 2005, the `__assume` construct had been in multiple prior releases of that compiler. GCC introduced `__builtin_unreachable` in 2010 (GCC version 4.5) about a decade ago, and Clang introduced `__builtin_assume` about four years after that.[2] Such constructs typically achieve their purpose by introducing undefined behavior (UB) into the program when some explicitly stated condition must hold.

For example, consider the `double`-valued function `sqrt`:

```
double sqrt(double x)
    // Return the positive value whose representation multiplied by itself
    // is the closest to `x`; if there's a tie, prefer the smaller one.  The
    // behavior is undefined unless `x >= 0`.
{
    __builtin_assume(x >= 0);  // compiler-accessible assumption construct

    if (x < 0) return 0; // Defined behavior may be elided as `x >= 0` is true.

    // … (remaining implementation elided)
}
```

Providing the platform-dependent `__builtin_assume` assumption construct above allows the compiler to remove the line that would return 0 if `x` were negative, which is correct since the contract states non-negative `x` as a precondition.

Even without explicit syntax, we can still try communicating with the compiler by creating our own, say, `MY_ASSUME(BOOLEAN_VALUE)` macro that, if the provided Boolean expression is false, would immediately invoke some obvious UB (e.g., dereferencing and assigning to a literal null pointer):

```
#define MY_ASSUME(BOOLEAN_VALUE) { if (!(BOOLEAN_VALUE)) *(int*)(0) = 0; }
    // Replace with undefined behavior if `(BOOLEAN_VALUE == false)`; else NOP.

double sqrt(double x)
    // (Return… …smaller one.) The behavior is undefined unless `x >= 0`.
{
    MY_ASSUME(x >= 0);  // home-grown compiler-accessible assumption construct

    if (x < 0) return 0;  // defined behavior *might* still be elided

    // … (remaining implementation elided)
}
```

As it turns out, not all compilers treat all UB equally. In our experiments, for example, MSVC[3] and ICX[4] did *not* optimize based on assignment through a literal null pointer. Clang[5] had almost identical results between assigning

[1] P1774R8
[2] Clang 3.6.0
[3] Microsoft Visual C++ 2022 (MSVC 12.33.31630)
[4] Intel ICX 2022.2.0.20220730
[5] Clang 15.0.1

through a null pointer and use of its `__builtin_assume` intrinsic, whereas GCC[6] seemed to optimize a null reference slightly better than it did when we used the `__builtin_unreachable` idiom:

```
double sqrt(double x)
    // (Return… …smaller one.) The behavior is undefined unless `x >= 0`.
{
    if (x < 0) __builtin_unreachable();   // compiler-accessible assumption (GCC)

    if (x < 0) return 0;  // defined behavior (*might* still be elided)

    // … (remaining implementation elided)
}
```

Moreover, if the compiler can see that calling `sqrt` on a negative value would be UB, the compiler is allowed to use any such inevitable (i.e., cannot-be-bypassed) call to elide code upstream from that call, sometimes referred to as a *time travel optimization*:

```
int negative_y_count = 0;  // count number of negative args to `myFunc`

double myFunc(double y)
    // Do something; `y` must not be negative.
{
    if (y < 0) ++negative_y_count;  // can be elided because, if we get here,
                                    // sqrt(y) will always be reached
    return sqrt(y);
}
```

Anecdotal tales of various meaningful effects of home-grown and implementation-specific compiler-accessible assumption constructs on generated code have long been touted. Historically, however, surprisingly little reproducible empirical data has been available to suggest what effect, if any, such compile-time assumptions might have on overall program run times. Absent compelling empirical data, some committee members have merrily conjectured that assumptions, though they might have an impact on reducing generated code size, will necessarily worsen compile time and, at best, have a negligible impact on run time. Modern compilers typically notice and perform the optimizations available on the current platform. With modern CPU architectures and branch predictors, even when the compiler fails to elide an untaken branch, that branch can nonetheless execute with effectively zero cost if it is well predicted.

One paper, in particular[7], suggests that when enabling compiler-accessible assumption constructs on preconditions, the overall run time of a particular application grew by approximately 1%. This surprising result was questioned during the SG21 meeting in Prague in February 2020. Herb Sutter and John Lakos agreed that Bloomberg would undertake a comprehensive study to

[6] GCC 12.2.0
[7] P2064R0

determine if and to what extent making a precondition a compiler-accessible assumption might affect overall program run times: (1) in which direction, (2) by how much, and (3) characterized by what kinds of coding patterns. The research and results we present in this paper form the beginning of our response to that open question.

With the adoption of portable assumptions[8] to the C++ working paper, understanding the potential impact such assumptions can have on runtime performance now becomes important to all users considering using `[[assume]]`. In this paper, we'll start by discussing several use cases in which explicit assumptions made available to the compiler appear to improve code generation and lead to improved runtime performance. We then choose just one of those use cases and establish a specific experiment that demonstrates — beyond any doubt — that a compiler-accessible assumption *can* and *does* meaningfully affect (invariably reduce) overall runtime performance.

In particular, we'll exhibit reproducible data that, even on the most modern compilers, shows significant potential *runtime* effects of introducing explicit compiler-accessible assumption syntax. This potential runtime benefit will, in turn, underscore the importance of having such crucial semantics now part of the (portable) Standard rather than only as optional, eclectic, inconsistent, nonportable extensions.

THEORETICAL ADVANTAGES

As one might reasonably expect, the more accurate information a compiler has to work with, the more likely it will be able to generate better object code — e.g., code that takes less time to run. One might also presume that analyzing more information could take additional compile time. One could also imagine scenarios where the additional information might result in less generated code, which in pathologically extreme cases might noticeably reduce overall compilation time. Taking this reasoning a step further, less code typically runs faster, so we might conjecture that run times might also be affected.

Although compiler-accessible assumptions sometimes lead to substantially smaller programs, the smaller size is not necessarily reflected in reduced overall run times. In many cases, the reduction in object code results from dead-code elimination. Even when the dead code is not eliminated via an explicit assumption construct, compilers are typically still smart enough to place the presumed unlikely code remotely, and branch predictors often obviate loading the dead code into memory. In such cases, the difference in run time, with and without a compiler-accessible assumption construct, is often nothing more than a single, readily predicted branch, leading to no measurable difference in run times.

[8] P1774R8

In numerous specific cases, however, we might reasonably expect that a well-placed compiler-accessible assumption construct could lead to better-generated code and meaningful reductions in run times, whereas without an explicit assumption construct, the compiler would have no way to deduce such truth and thus could not apply the corresponding optimizations. In what follows, we briefly delineate some distinct ways in which user-provided assumption constructs might significantly impact generated code and thus the prospect of perhaps also achieving some meaningful reduction in run times. In each of these six cases, we will use a local-library-provided `ASSUME(EXP)` macro to indicate to the compiler that `EXP` can be assumed to be (convertible to) `true`.

1. **Eliding conditional branches.** If there is one thing compiler-accessible assumptions are good at, it's removing conditional branches (and associated dead code) where only one direction is supported behavior. Consider a simple function, `branches`, that takes as its only argument a Boolean flag, `firstBranch`, such that the if-branch will be executed if the flag is `true` and the else-branch otherwise:

```
void branches(bool firstBranch)
    // Invoke `doSomething` if `firstBranch` is `true`; otherwise, invoke
    // `doSomethingElse`.
{
    if (firstBranch) {
        doSomething();
    }
    else {
        doSomethingElse();
    }
}
```

   A function with no preconditions is said to have a *wide contract*; otherwise, it has a *narrow contract*. Suppose we know that we will never pass `false` to the function. We can then change the contract from a *wide* to a *narrow* one and assume that the contract will be followed scrupulously:

```
void branches2(bool firstBranch)
    // Invoke `doSomething` if `firstBranch` is `true`; otherwise the behavior
    // is undefined.
{
    ASSUME(firstBranch); // Per English contract: `firstBranch` must be true.

    if (firstBranch) {      // elidable
        doSomething();
    }
    else {
        doSomethingElse();  // elidable
    }
}
```

Since the compiler knows it can assume that `firstBranch` is always true, it can elide both the expression check and the else-block without changing the meaning of a correct program:

```
void elidedBranches2(bool firstBranch)
    // Invoke `doSomething` if `firstBranch` is `true`; otherwise the behavior
    // is undefined.
{
    doSomething();
}
```

This first theoretical case is the one we will analyze deeply in this paper. Note that although we will employ nested `if-else` expressions to help us extract quantifiable results, nesting is not essential for code generation to be affected significantly.

2. **Loop unrolling**. A common compiler optimization is unrolling loops into multiple instances of the loop body. Loop unrolling is atypical in that it leverages *increasing* code size (due to multiple copies of the loop body) to reduce the number of branch conditions that need to be evaluated and thus the number of instructions to be executed. Importantly, if a loop is unrolled into `n` copies, special consideration will need to be taken up-front within the generated code to handle cases where the number of iterations will not be a multiple of `n`. Such optimizations will often involve the compiler generating the moral equivalent of Duff's Device — a `switch` statement and an associated additional, unconventional branch into the code block associated with the `switch`. For example, consider a function, `loop`, taking as its argument an integer number of iterations, `n`:

```
void loop(int n)
    // Do something `n` times. The behavior is undefined unless `n > 0`.
{
    for (int i = 0; i < n; ++i) {
        doSomething();
   }
}
```

A compiler might choose to translate `loop` to an equivalent function, `loopUnrolled4`, in which the loop body is unrolled four times:

```
void loopUnrolled4(int n)
    // Do something `n` times. The behavior is undefined unless `n > 0`.
{
    int i = 0;
    switch (n % 4) {
        do {
            case 0: doSomething();
            case 3: doSomething();
            case 2: doSomething();
            case 1: doSomething();
            ++i;
        } while (i < (n+3)/4);
    }
}
```

Compilers will attempt to choose the optimal number of times to unroll a loop based on (1) their ability to predict the results of the loop termination condition (and perhaps the cost of evaluating it), (2) the size of the loop body, and (3) the optimal use of space in the instruction cache for the target CPU. Supplying an appropriate compiler-accessible assumption construct can, however, significantly reduce the size of transformed code.

Let's now consider a function, `loopAssumed`, whose contract advertises that the number of iterations *must* be an integer multiple of 4 or else the behavior is undefined:

```
void loopAssumed4(int n)
    // Do something `n` times. The behavior is undefined unless `n >= 0`
    // and `n` is an integer multiple of 4.
{
    ASSUME(n % 4 == 0);
    for (int i = 0; i < n; ++i) {
        doSomething();
   }
}
```

With the explicit `ASSUME(n % 4 == 0)` in place, the compiler is permitted to translate the function to an equivalent one, such as `loopAssumedUnrolled4`:

```
void loopAssumedUnrolled4(int n)
{
    for (int i = 0; i < n; i += 4) {
        doSomething();
        doSomething();
        doSomething();
        doSomething();
    }
}
```

Note that as long as the assumption on the number of iterations, `n`, is an integer multiple of how many times the compiler chooses to unroll the loop body, this seemingly improved transformation can be readily achieved.

3. **Vectorization.** When loops are unrolled, another common transformation that can be applied to the unrolled loop is automatic (compiler-initiated) vectorization. This transformation replaces many instructions with a single CPU-specific instruction capable of doing all those operations simultaneously. Consider, for example, a function, `sum`, that sums a given number of double-precision floating-point values:

```
double sum(double* data, int count)
    // Sum the specified `count` numbers in the specified `data` array.  The
    // behavior is undefined unless the first `count` members of `data` are
    // valid (non-NaN) objects of type `double`, `count` is a multiple of 8,
    // and `data` is aligned on a 64-byte boundary.
{
    ASSUME(count % 8 == 0);                                    // #1
```

```
    ASSUME(reinterpret_cast<std::uintptr_t>(data) % 64 == 0);  // #2
    double output = 0;
    for (int i = 0; i < count; ++i) {
        output += data[i];
    }
    return output;
}
```

Using just the first compiler-accessible assumption construct (`ASSUME #1`), the compiler may choose to unroll the loop and then transform the code:

```
double sumVectorized(double* data, int count)
{
    double output = 0;
    for (int i = 0; i <= count; i += 8) {
        __m512d v = _mm512_loadu_pd(data + i);  // Read 8 doubles.
        output += _mm512_reduce_add_pd(v);      // vectorized addition
    }
    return output;
}
```

Access to the second assumption (`ASSUME #2`) allows for use of an aligned load, which would replace the `__mm512_load`**`u`**`_pd` CPU instruction with `__mm512_load_pd` a more specialized instruction that might show significant improvements in execution time. Even without these assumptions, many similar scenarios will still see vectorization at the compiler's discretion, but substantial extra code, including *runtime* branch statements, will need to be emitted to handle cases where the data is (1) not a multiple of the number of elements that can be operated on in a single SIMD instruction or (2) not sufficiently aligned.

4. **Memory Aliasing.** Absent specific knowledge, a compiler must generate code that works properly for *all* valid input. In particular, a compiler can rarely safely assume that two distinct pointers, independently passed to a function, do not point to overlapping objects. For example, any write-through to an object referenced by one pointer could potentially invalidate the compiler's knowledge about the value referenced by some other pointer. Consider, for example, a function, `foo`, that takes two integer pointers, `a` and `b`, sets the value to which `a` points to 17, and returns twice the value to which `b` points. The behavior is undefined if the storage (typically 4 bytes) to which `a` points overlaps that of `b`.

```
int foo(int *a, int *b)
    // Store 17 in `*a` and return `2 * *b`; the behavior is undefined
    // if `a == b`. Note that, on platforms that do not require *natural*
    // *alignment*, this precondition might not be sufficient.
{
    int output = 0;
    output += *b;  // reads `b`
    *a = 17;
    output += *b;  // must read `b` again
    return output;
}
```

With larger objects and user-defined types or both, the task of guarding against aliasing becomes even more complex. The C language provides a specific keyword, `restrict`, which informs the compiler that a pointer does not refer to the same address as any other pointer:

```
int fooInC(int *a, int * restrict b)
{
    int output = 0;
    output += *b;  // reads `b`
    *a = 17;
    output += *b;  // `b` does not need to be read again.
    return output;
}
```

C++ does not provide `restrict`, but an assumption should be able to provide the same (or at least sufficiently similar) information:

```
int fooInCpp(int *a, int *b)
{
    ASSUME(a != b);
    int output = 0;
    output += *b;  // reads `b;`
    *a = 17;
    output += *b;  // `b` does not need to be read again.
    return output;
}
```

Note that this is just a simple case where C's `restrict` keyword is effective and improves code generation. Using our home-grown, compiler-accessible `ASSUME` macro in a way that more fully approximates the power of `restrict` could, however, be challenging.

5. **Floating-point arithmetic.** The floating-point data types provided in C++ — `float`, `double`, and `long double` — have the sometimes-frustrating property of not being *regular*. These types reserve a series of representations, each of which is not a number (NaN) and has the interesting property of returning `false` from all comparison operations, including comparison with itself! Because the compiler is required to deal with any valid floating-point object, even one that doesn't represent a number, seemingly simple source code might require an unexpectedly complex object-code generation to accommodate pathological cases that are designed to never occur in most typical applications. Consider two functions, `f` and `g`, each taking a single argument of type `double`, each of which returns its argument, provided that the argument is not a NaN:

```
double f(double argument)
    // Return the specified `argument`.
{
    if (argument * argument < 0) { return 0; }        // always false
    return argument;
}

double g(double argument)
    // Return the specified `argument`. The behavior is undefined if
    // `argument` is a NaN.
```

```
{
    if (argument * argument >= 0) { return argument; } // true except for a NaN
    return 0;
}
```

On most platforms, the first function, `f`, compiles to a single return statement (`return val;`), while `g` must have a branch to handle returning `0` when passed a NaN. As one might surmise, the implementer of `g` had little concern for what `g`'s behavior would be when passed a NaN. In such cases, one could employ the structure of `g` and achieve the generated-object-code compactness of `f` by supplying an appropriate compiler-accessible assumption construct explicitly:

```
#include <cmath>  // std::isnan(double)

double g2(double argument)
{
    ASSUME(!std::isnan(argument));  // or maybe `ASSUME(argument == argument);`
    if (argument * argument >= 0) { return argument; }  // always true
    return 0;
}
```

By informing the compiler explicitly that it may assume no NaN will ever be passed into `g2`, object code need not be generated to handle a NaN's noisome idiosyncrasies.

6. **Signed arithmetic.** Overflowing a *signed* integral type in both C and C++ is explicitly UB, whereas overflow on an *unsigned* integral type is defined and required to wrap. Hence, one can encounter expressions where the compiler can make optimizations for a signed expression that it could not make for a structurally similar unsigned expression. For example, consider two functions, `f1` and `f2`, trafficking in unsigned and signed integers, respectively[9]:

```
unsigned int f1(unsigned int i)
    // Return something (wide contract).
{
    unsigned int j, k = 0;
    for (j = i; j < i + 10; ++j)
        ++k;
    return k;
}
```

```
int f2(int i)
    // Return 10 (narrow contract).
{   ASSUME(i >= 0);
    int j, k = 0;
    for (j = i; j < i + 10; ++j)
        ++k;
    return k;
}
```

Notice that `f1` can overflow; hence, object code must be generated to accommodate that possibility. On the other hand, the compiler can assume that no overflow will occur for `f2`; hence, the body of `f2` can safely be optimized to simply return `10`, provided the caller respects that the behavior is undefined unless `INT_MIN <= i && i < INT_MAX - 10`. This example shows how just, by using a *signed* type, we introduce an *implicit* compiler-accessible assumption that the computation cannot overflow, and thus choice of *signed*-ness can substantially affect code generation.

[9] https://www.airs.com/blog/archives/120

Further indicating to the compiler that the argument cannot be negative, however, requires an *explicit* one as the original function was implicitly undefined for negative values.

Each of the six use cases above provides realistic opportunities for meaningfully affecting generated code. This paper, however, will focus on just the first one, *eliding conditional branches*.

PLAN

Compile-time assumptions might benefit code generation — now or in the future — in innumerable ways. Given that `[[assume]]` is currently part of the C++ working draft, slated for release as part of C++23, we wanted to provide dispositive proof that compiler-accessible assumptions *can* measurably reduce run times in C++ code.

Since our goal is to provide an *existence proof*, we opted to pick just one optimization theory — *eliding conditional branches*, which is the first item in the previous section's list of benefits — and to write a microbenchmark to explore that space (see the following section, *Apparatus*, for more details):

1. Create a trivially simple, portable, and ideally reusable custom framework that can be run standalone on any conforming platform (i.e., hardware/compiler combination).
2. Making use of pseudo-recursive macros and conditional compilation, devise a test function having two compile-time integer parameters suitable for creating a family of functions that can be used to exhibit and quantify the effects of nested `if-else` statements versus, as a control, some other, inert way of similarly increasing code size.
3. Design, for each `if-else` branch, a (perhaps distinct) conditional expression tied to a single runtime parameter of the function such that, for a particular chosen *magic* value of the runtime parameter, the expression will (1) always evaluate to `true` and (2) minimize the likelihood that a compiler, absent an explicit assumption construct, will be able to identify that particular magic value when optimizing.
4. Insert an optionally-enabled (controlled via conditional compilation) compiler-accessible assumption that the runtime function parameter is, in fact, our chosen magic number. The mechanism used to deliver the assumption depends on the host platform: Use a home-grown construct based on assignment through a null pointer if that works; otherwise, use a compiler-specific intrinsic (assuming one exists and provides consistent results).
5. Run each member of the 2D family of inputs *with* and *without* compiler-accessible assumptions enabled and graph the ratios of compile times,

generated code sizes, and run times as surfaces in 3D and, for exposition purposes, as a series of 2D heat maps.

6. Determine if and to what extent compiler-accessible assumptions affect our code generation and runtime behavior.
7. Contrast the effect of the compiler-accessible assumption constructs on the test function's single runtime argument for increasingly nested *if-else depth* (first dimension) juxtaposed with similarly increasing *body size* (second dimension), given that the increasing the value in either dimension by 1 effectively doubles the size of the compiler input in `testfunc.cpp`.
8. Where meaningful, assess whether the net effect of compiler-accessible assumptions on code generation improves or degrades compile times, code sizes, and especially run times.

Importantly, this experiment models one function having a *narrow contract*; hence, not all syntactically valid inputs are considered defined behaviors by the function's author. If the compiler is unaware of the preconditions in a function's English contract, it will be obliged to generate object code that has no purpose in any correct program. Conversely, by dint of an explicit assumption construct, the compiler may instead assume that it needs to handle only a single input value (which could be generalized to any sufficiently reduced range of inputs) and, hence, much of the supererogatory object code that might otherwise be generated may now be elided.

To bind this proposed, highly customized, artificial microbenchmark to the real world, let's consider just one more example motivating this particular theoretical optimization opportunity. Suppose some popular library provides an `inline` function, `algo`, such that the body of `algo` is always visible to each of its callers. Suppose further that this function sports a *wide* contract, meaning that it is prepared to handle every valid combination of inputs:

```
inline double algo(int i, double, const char *cp)
// This inline function has a wide contract.
{
    // ... (Body is visible to caller's compiler.)
}
```

One can reasonably suspect that not every client will require the full, wide contract supported by the robust `algo` library function.

Let's now consider three tiny but not necessarily inline client functions — `f`, `g`, and `h` — in turn. The first, `f`, takes a single argument (via parameter `i`), which is required to be non-negative:

```
double f(int i)
    // The behavior is undefined unless `i >= 0`.
{
    ASSUME(i >= 0); // E.g., `if (i >= 0)` may also be elided in `algo`.
```

```
    return algo(i, 1.0, nullptr ); // Much of `algo` may be elided automatically!
}
```

Because the compiler for `f` above can see two literal arguments coming into `algo`, the generated code can already be heavily elided. Providing the additional compiler-accessible assumption in the body of `f` affords additional information at compile time that might lead to further object-code improvement. Note that the compiler would be within its rights to choose not to inline `algo` in `f`, in which case none of these optimizations would take effect. (Naughty compiler!)

Let's now consider a client function, `g`, that requires a (non-null) null-terminated string as input:

```
#include <limits> // std::numeric_limits<double>::quiet_NaN();

double g(int x, char* string)
    // The behavior is undefined unless `string` refers to an NTBS.
{
    ASSUME(string); // sufficient to avoid conditional check for nullness
    return algo(x, std::numeric_limits<double>::quiet_NaN(), string);
}
```

In `g` above, the compiler might be able to do something with the knowledge that this instantiation of `algo` will always get a quiet NaN as its second argument, but that's entirely beyond our control. On the other hand, by stating explicitly that `string` will not be null, we enable the client compiler (i.e., of `g`) to elide all such redundant checks and associated actions from the generated object of `algo` inlined for `g`.

Finally, we look at one more client function, `h`, that specifies a highly restricted range for `algo`'s middle argument:

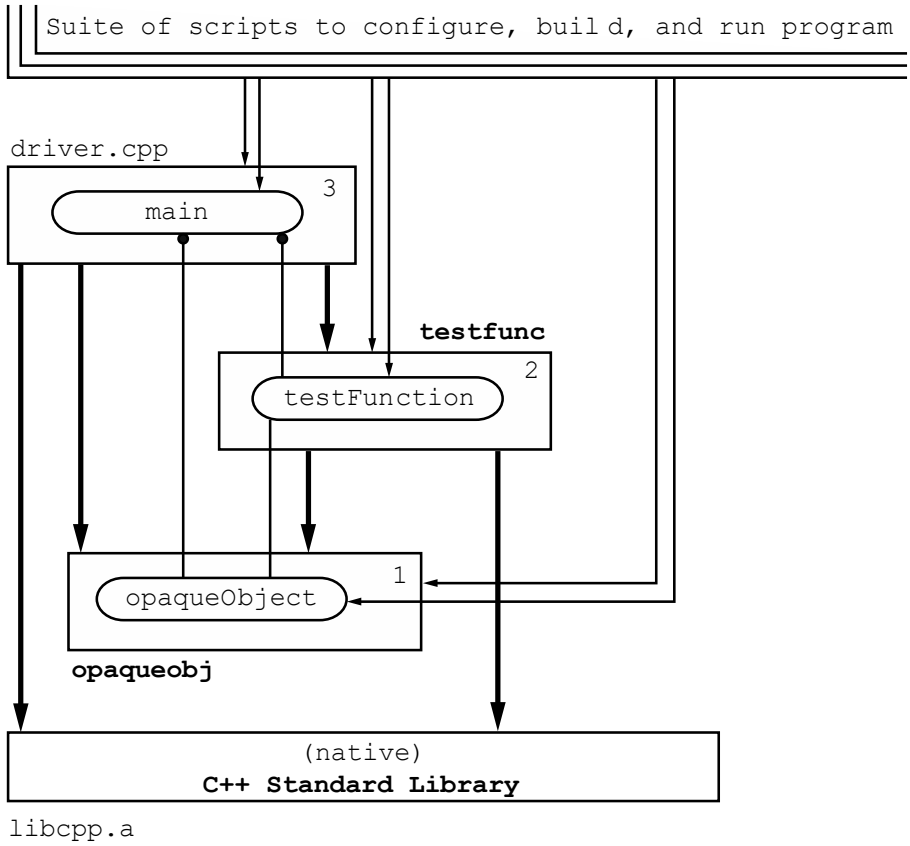
```
double h(int i, double d = 1.0)
    // The behavior is undefined unless `0.5 <= d <= 2.0`.
{
    ASSUME(0.5 <= d); ASSUME(d <= 2.0);
    return algo(i, 1.0/d, nullptr); // `cp` is 0 and `1.0/d` will always be finite.
}
```

In this third client function, `h`, the compiler automatically knows that any test for a `null` value of `cp` within `algo` will be `true` and thus can elide that test along with all code that handled a non-null `cp`. When the compiler is explicitly provided with an accessible assumption indicating the range to which `d` is bounded, it may then determine the range of the *results* for operations on `d`, potentially exclude infinite and NaN *results*, and thus reduce the amount of generated object code by — in particular — eliding nested conditional branches.

APPARATUS

Our deliberately simple, portable microbenchmark framework consists of various scripts that build and drive a C++ program comprising three local

translation units (TUs) that, in turn, depend on *only* the platform's native Standard Libraries.



Each of these three local TUs serves a distinct purpose in this microbenchmarking framework:

> **`driver.cpp`** — a reusable `main` driver file that, once compiled and linked, reads command-line configuration arguments to control a `for` loop used to invoke a suitably customized test function repeatedly.
>
> **`testfunc.cpp`** — a component that defines an *insulated* function, `testFunction`, which is parameterized via conditional compilation to yield the precise function whose *physical* characteristic (e.g., compile time, object-code size, and run time) we wish to analyze.
>
> **`opaqueobj.cpp`** — a tiny component that defines an insulated `volatile` object that can be leveraged in the other TUs to prevent some compiler optimizations.

In the remainder of this section, we'll take a more in-depth look at each part of our framework.

At the first level (labeled 1) of our local physical hierarchy, we have a component, **`opaqueobj`**, that contains the *opaque* definition of an *insulated* `volatile unsigned int` object, `opaqueObject`, whose file-scope, statically initialized state is the value `1u`. The compiler has no means of determining that this value will not change, so it must not make any assumptions about what actual values `opaqueObject` might or might not contain:

```
// opaqueobj.cpp
volatile unsigned int opaqueObject = 1;
```

The express purpose of this tiny component is to enable us to both (1) remove optimizations that would otherwise cause a microbenchmark to become irrelevant and thereby (2) mimic the effects that might be seen in real code as it scales. Importantly, by using the value of an insulated `volatile` object, the compiler will be unable to make any assumptions about the state of this object when compiling our other TUs. Although modern compilers having link-time optimization enabled might be able to make such assumptions anyway, we deliberately avoid building any of our test executables with link-time optimization enabled to preserve the efficacy of the benchmark.

At the third level (labeled 3) of our component hierarchy, we have the `driver.cpp` file, which defines `main`. This small TU is responsible for processing any runtime program arguments from the command line, such as the number of times to call `testFunction` from within a tight loop:

```
// driver.cpp
#include <testfunc.h>  // `unsigned int testFunction(unsigned int);`
#include <opaqueobj.h> // `volatile unsigned int opaqueObject;`
#include <iostream>    // `std::cout`

int main(int argc, const char *argv[])
{
    int k = argc > 1
        ? std::atoi(argv[1])      // Parse the first argument if provided.
        : 100'000'000;            // 100 million is the default.
    std::cout << k << std::endl;  // Print # loop iterations.

    int log2_i = 0;       // log base 2 of first trial `(i = 1)`, is 0.
    long long iNext = 1;  // when to print next `log2_i`

    for (long long i = 1; i <= k; ++i) { // `i` is the `i`th trial
        if (i >= iNext) {
            std::cout << ' ' << log2_i++ << std::flush;
            iNext <<= 1;
        }
        opaqueObject = testFunction(opaqueObject);
    }
    return 0;
}
```

Before starting the loop, the program prints, to `stdout`, the runtime-specified loop-iteration count. The body of the loop is deliberately kept minimal but, to enable the human user to monitor progress during longer benchmarks, has an extra Boolean test that is used to successively print, with exponentially

decreasing frequency, $\log_2$ of the current loop index to `stdout`. Finally, the loop invokes the (insulated) `testFunction` on the opaque statically initialized `volatile unsigned int opaqueObject` and then assigns it the resulting inscrutable value upon its return.

Importantly, on each iteration of the loop, the value `opaqueObject` is repeatedly filtered through the benchmark's test function. Without the knowledge that this value will stay `1`, which we have carefully prevented the compiler from concluding within any single TU of our microbenchmark program, the compiler is forced to generate object code for `testFunction` to account for a wide range of `opaqueObject` values. In practice, however, when the value is initialized to `1`, it will stay `1` indefinitely, and our assumption for this benchmark will be that the value is always `1`. Because we have secretly structured this microbenchmark program such that `testFunction` is always invoked with the same `opaqueObject` value, `1`, we are able to introduce that valid assumption (or not) into `testFunction` (see below) and thereby observe whatever physical benefits enabling such a compiler-accessible assumption might bring us.

Sandwiched between the `main` driver and the component defining `opaqueObject` at the second level (labeled 2), is the **`testfunc`** component, which defines our highly compile-time-customizable, insulated `testFunction`, taking a single `unsigned int` value and returning some inscrutable value of the same type:

```
// testfunc.h
#ifndef INCLUDED_TESTFUNC
#define INCLUDED_TESTFUNC
unsigned int testFunction(unsigned int v);
#endif
```

We need to pass in some unknown argument so that the compiler of this TU cannot automatically elide code based on what it already knows about the caller. We need the test function to return some unknowable value to prevent the client compiler from optimizing away the call or discovering other artificial optimizations that would tend not to mimic real-world code as it scales up. Finally, we need to make sure that `testFunction`, when passed our *magic* value of `1u`, always returns that same *magic* value of `1u`, thereby ensuring that the value of our volatile `opaqueObject` will remain stable through each iteration of our test-driver loop and we can introduce an assumption that our function parameter, `v`, always has a value of `1u` as well.

What makes this component especially unusual is the extent to which it can be aggressively customized at compile time to create a 2D family of functions having wildly varying object sizes, parameterized by two non-negative integers, *M* and *N*, in which *M* describes the *depth* of a symmetric `if-else` tree and *N* indicates *size* and is proportional to the logarithm of the number of additional

sequential instructions in each `if-else` code block (*body*). One effective way of achieving such flexibility is to create a sequence of pseudo-recursive macros and then use conditional compilation to select the appropriate recursion depth using `-D` on the compilation command line:

```
// testfunc.cpp
#include <testfunc.h>   // declaration of `testFunction`
#include <opaqueobj.h>  // declaration of `opaqueObject`

#define X opaqueObject
#define BODY0 r += X ; r *= X; r -= X; r *= X; // If `X` is 1, `r` is unchanged.
#define BODY1 BODY0 BODY0
#define BODY2 BODY1 BODY1
#define BODY3 BODY2 BODY2
// : : :
#define BODYJ BODYI BODYI

#ifndef BODY // `BODY` must be defined using `-D` to one of the above.
#error BODY is not defined
#endif

#define IFELSE0 { BODY } // `BODY` is the customization point for body size.
#define IFELSE1 if (EXPR1) { IFELSE0 r &= X; } else { IFELSE0 r |= X; }
#define IFELSE2 if (EXPR2) { IFELSE1 r &= X; } else { IFELSE1 r |= X; }
#define IFELSE3 if (EXPR3) { IFELSE2 r &= X; } else { IFELSE2 r |= X; }
// : : : : : : : : : : : : :
#define IFELSEJ if (EXPRJ) { IFELSEI r &= X; } else { IFELSEI r |= X;

#ifndef IFELSE // `IFELSE` must be defined using `-D` to one of the above.
#error IFELSE is not defined
#endif

unsigned testFunction(unsigned int v)
{
    #ifdef ASSUME // compiler-switch to enable home-grown assumption construct
    if (v != 1) *(int *)0 = 0; // If `v` is not `1`, we have undefined behavior.
    #endif

    unsigned int r = v; // Start the macro ball rolling.

    IFELSE; // customization point for if-else depth.

    return r; // designed to return `1u` provided `v` was `1u`
}
```

Also configurable at compile-time are whether an explicit compiler-accessible assumption construct operating on the input parameter is enabled and, if so, whether to use native compiler intrinsics (not shown) or some other home-grown language construct (e.g., assigning to a dereferenced null pointer) to introduce UB if the assumption would ever evaluate to `false`.

Additionally, not all branch conditions (see *EXPR1*, *EXPR2*, . . . , above) benefit equally from compiler-accessible assumptions, especially when the compiler can infer during its normal optimization procedures, without an assumption, that certain values will lead to predictable branching for the given expressions. For many common nested expressions, the compiler might be able to glean

useful optimization information even without explicit compiler-accessible assumptions.

Initially, one of the early expressions was `1 == v`. On the branch where the conditional expression was true (i.e., the branch our framework would always take), all remaining expressions in terms of `v` were deducible at compile time, meaning that the nonassumed build of our test function performed only a single comparison and branch at run time; hence, we saw no significant benefit in the assumed build. Instead, we chose to make the comparison expressions distinctly independent so that the result of one expression could not be used to determine the results of the remaining expressions. The benchmark data shown in this paper is the result of using a series of expressions where the *m*th expression is `true` only if a particular bit (bit *m*) is not set in `v` (note that there is no expression when *m* is 0):

```
#define IFELSE1 if (0 == v ^ (v << (v + 0))) { IFELSE0 } else { IFELSE0 }
#define IFELSE2 if (0 == v ^ (v << (v + 1))) { IFELSE1 } else { IFELSE1 }
#define IFELSE3 if (0 == v ^ (v << (v + 2))) { IFELSE2 } else { IFELSE2 }
   :       :     :  :    :    :     :   :          :       :        :
```

We also made sure that the sequence of linear instructions was not amenable to optimization due to concurrence. Moreover, we chose our arithmetic calculations such that, provided the `opaqueObject` had an initial value of `1u`, after each repeated sequence of four arithmetic operations, `opaqueObject` would again have that same value but — by explicit design — the compiler would have no way to know that. Thus `opaqueObject == 1u` remains a stable invariant throughout the repeated calls to `testFunction`.

We developed several scripts of varying degrees of portability, each of which assist in gathering data needed to understand how our benchmark performs. For each set of values of the control variables (*if-else depth*, *body size*, and *assumption enablement*), we perform six steps:

1. Compile separately the `testfunc.cpp` TU with the appropriate compile-time parameters and build options.
2. Measure the time taken by the compilation process.
3. Measure the size of the resulting object file, `testfunc.o`.
4. Link the object files from `testfunc.o`, `driver.o`, and `opaqueobj.o` into an executable.
5. Run the resulting executable with a sufficiently large input size (e.g., 10^8) to ensure execution times are substantial enough (e.g., > 1 second) to get meaningful results.
6. Measure the time taken for the executable to run to completion.

The *if-else depth, M,* is controlled by defining (typically using `-D`) the value of the macro `IFELSE` to one of the numbered or lettered `IFELSE<N>` macros

available in `testfunc.cpp`. The *body size*, *N*, for each of the `if` and `else` blocks is similarly controlled by defining the value of `BODY`. Both of these macros must be defined for `testfunc.cpp` to compile. Assumption is disabled by default and can be enabled if the macro `ASSUME` is defined.

As a purely illustrative example, consider two independent build-link-run sequences for this benchmark program, targeting a specific compiler and presupposing that the other TUs have already been compiled. To understand what the script is doing as it executes, before each sequence, we output (`echo -n`) the *if-else depth*, *body size,* and *assumption enablement* — `_` for *disabled* and `@` for *enabled*:

```
# simple (pedagogical) script file to run two independent tests
echo -n "(3 1 _)"
    time g++ -O3 -I. -c -DIFELSE=IFELSE3 -DBODY=BODY1 testfunc.cpp
    stat -c%s testfunc.o
    g++ driver.o testfunc.o opaqueobj.o -o test_3_1
    time ./test_3_1 10000000
echo -n "(0 6 @)"
    time g++ -O3 -I. -c -DIFELSE=IFELSE0 -DBODY=BODY6 -DASSUME testfunc.cpp
    stat -c%s testfunc.o
    g++ driver.o testfunc.o opaqueobj.o -o test_0_6_a
    time ./test_0_6_a 100000000
```

Sourcing this file produces several lines of output[10]:

```
(3 1 _)
real 0m0.328s
user 0m0.171s
sys 0m0.123s
2680
10000000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
real 0m0.166s
user 0m0.078s
sys 0m0.030s
(0 6 @)
real 0m0.397s
user 0m0.203s
sys 0m0.139s
2480
100000000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
real 0m14.967s
user 0m14.890s
sys 0m0.000s
```

Notice that, along with various compile time, object size, and run time data, information sufficient to characterize each program run is embedded within the output:

```
(1, 6, _) ... 10000000 ; N=1; M=6; assume disabled; called 10MM (10^7) times.
(3, 0, @) ... 100000000 ; N=3; M=0; assume enabled; called 100MM (10^8) times.
```

[10] ThinkPad 480s running GCC 11.2.0 (c. 2021)

Running a postprocessing script to collect and render the raw output might yield something a bit more user friendly:

```
Config.  <-- Compile Time -->   .o    Num   <----- Run Time ----->
(N M A)   real   user  system  Size  Calls  real user system
-------  ------ ------ ------  ----  -----  ------- ------- ------
(1 3 _)  0.328s 0.171s 0.123s  2680   10MM   0.166s  0.078s 0.030s
(6 0 @)  0.397s 0.203s 0.139s  2480  100MM  14.967s 14.890s 0.000s
```

We now have all the tools we need to start exploring the space of the theoretical advantage we're focusing on: *eliding conditional branches*.

EXPERIMENT

Given the apparatus discussed in the previous section, we are now ready to describe how we proceeded to collect the data for this paper. Recall that our framework supports compile-time parameters that include (*M*) the *depth* of the `if-else` tree, (*N*) the $\log_2$ of the *size* of the body of code expanded within each `if-else` block, and (*A*) whether some compiler-accessible *assumption* is enabled. The number of invocations of the test function can also be passed to the generated executable, as a command-line argument, at run time.

The raw data we collect from each run includes (1) the time required to compile `testfunc.cpp`, which defines just our `testFunction`, (2) the size in bytes of the object file, i.e., `testfunc.obj` for MSVC and `testfunc.o` on all other platforms, and (3) the time needed to run the linked overall benchmark program. Although we collect wall, user, and system times in these experiments, we typically run on dedicated hardware, so we have opted to use *wall* time for all time (duration) measurements. For further analysis, we also run the linked benchmark program on Linux with the `perf` command to gather CPU performance counters such as instruction count, branch count, and L1 data cache loads and stores.

Running the benchmark program for a particular configuration — i.e., compile-time value of *if-else depth (M)*, *body size* (*N)*, and whether compiler-accessible assumptions are enabled *(A)* — produces raw data that, by itself, is not especially informative. But by running the same *if-else depth* and *body size partial* configuration twice, first with assumptions *(A) disabled* (`` `_` ``) and then *enabled* (`` `@` ``), we get two distinct raw data points that we can use to assess the effect of providing the explicit assumption on the baseline code for whatever data is of interest.

For example, we could use a simple script to compile, link, and test two such related configurations:

```
# simple (pedagogical) script file to run two independent tests
echo -n "(5 2 _)" #disabled
    time g++ -O3 -I. -c -DIFELSE=IFELSE5 -DBODY=BODY2 testfunc.cpp
    stat -c%s testfunc.o
    g++ driver.o testfunc.o opaqueobj.o -o test_5_2
    time ./test_5_2 100000000
```

```
echo -n "(5 2 @)" #enabled
    time g++ -O3 -I. -c -DIFELSE=IFELSE5 -DBODY=BODY2 -DASSUME testfunc.cpp
    stat -c%s testfunc.o
    g++ driver.o testfunc.o opaqueobj.o -o test_5_2_a
    time ./test_5_2_a 100000000
```

After executing these commands and collecting the output, we get a meaningful pair of data points that we can use to quantify the effects of adding compiler-accessible assumptions for this particular (*if-else depth* = 5, *body size* = 2) partial configuration of `testFunction`:

```
Config.  <-- Compile Time -->   .o    Num   <---- Run Time ---->
(M N A)   real   user  system  Size  Calls   real   user  system
-------  ------ ------ ------  ----  -----  ------ ------ ------
(5 2 _)  1.321s 0.640s 0.514s  7340  100MM  1.401s 1.311s 0.000s
(5 2 @)  0.493s 0.265s 0.093s  1408  100MM  1.267s 1.234s 0.000s
```

For example, suppose the baseline compile time *before* any changes, *B*, is 1.321s and the compile-time *after* assumptions are enabled, *A*, is 0.493s:

Relative Compile Time

$$\frac{\boldsymbol{after}}{\boldsymbol{before}} = \frac{\boldsymbol{assumptions}}{\boldsymbol{baseline}} \frac{\boldsymbol{A}}{\boldsymbol{B}} = \frac{\mathbf{0.493}\boldsymbol{s}}{\mathbf{1.321}\boldsymbol{s}} = \mathbf{0.3732}$$

As the calculation above shows, compile-time for this partial configuration with compiler-accessible assumptions *enabled* is approximately 37% of what it would be when *not* enabled; in other words, compiler-accessible assumptions improved compilation speed by nearly 300% in this instance.

Applying the same A/B ratio for (a) code size (in bytes) and (b) run time (wall seconds) for this particular configuration also yields improvements:

(a) Relative Code Size

$$\frac{\boldsymbol{A}}{\boldsymbol{B}} = \frac{\mathbf{1401}\ \boldsymbol{bytes}}{\mathbf{7340}\ \boldsymbol{bytes}} = \mathbf{0.1909}$$

(b) Relative Run Time

$$\frac{\boldsymbol{A}}{\boldsymbol{B}} = \frac{\mathbf{1.267}\boldsymbol{s}}{\mathbf{1.401}\boldsymbol{s}} = \mathbf{0.9044}$$

Thus, we see that in this build mode (`-O3` on GCC), the size of the generated `testFunc.o` file is more than five times as large *without* compiler-accessible assumptions as *with* them. Moreover, we also observe a decrease in the overall relative run time of nearly 10%.

Throughout the remainder of this paper, we will use this A/B ratio on linear measures, such as time or size, to exhibit the relative effects of compiler-accessible assumptions. We deliberately avoid reciprocal measures, such as speed or frequency, which can be needlessly confusing. What's important about this ratio is that it (1) is unitless and (2) uniformly describes the data obtained from a closely related pair of runs of the benchmark *with* compiler-accessible assumptions represented as a multiplicative factor of the data obtained *without* them:

Now that we fully understand the meaning of a single *relative* data point, we can organize how we collect and present the data for a 2D array of compile-time values for *if-else depth* and *body size*. The dimensions of the input array are governed primarily by what is feasible to build on our available platforms. Increasing *if-else depth* or *body size* by just one essentially doubles the amount of executable source code in the implementation of `testFunction`. Hence, values of the sum of these parameters that exceed a fairly low threshold (on the order of 20 to 30) would often lead to compilations that failed in various ways. Large compilations that did not fail could take hours or even days to complete and hence were deemed infeasible.

For these practical reasons, we found that a maximum *if-else depth* of 13 and *body size* of 7 mostly succeeded for the platforms on which we ran our experiments while still providing us with a sufficient breadth of gathered data. The few cases where we could not build a particular configuration of our benchmark occurred only when using MSVC and only when *both* parameters approached their respective maximum values; in such cases, we have left the corresponding relative result blank.

We organized our script so that each relative pair — *nonassumed* and *assumed* for the same *if-else depth* and *body size* partial configuration — was output consecutively. Next, we grouped all eight partial configuration pairs with *if-else depth* = 0 first. We followed that by a second block of eight pairs having *if-else depth* = 1, and so on:

```
if-else depth = 0, body size = 0, A = _
if-else depth = 0, body size = 0, A = @
if-else depth = 0, body size = 1, A = _
if-else depth = 0, body size = 1, A = @
: : : : : : : :
if-else depth = 0, body size = 7, A = _
if-else depth = 0, body size = 7, A = @
if-else depth = 1, body size = 0, A = _
if-else depth = 1, body size = 0, A = @
: : : : : : : :
if-else depth = 13, body size = 7, A = _
if-else depth = 13, body size = 7, A = @
```

Running a script to gather this data and invoking the program 14 × 8 × 2 = 224 times on a given platform for a given set of build-optimization levels is what we mean by running an experiment.

Several important reasons for creating a 2D array of relative data points bear mention:

1. A full experiment contains a great amount of data (typically multiple runs of 224 distinct configurations of the benchmark program).
2. Visualization helps us observe trends and relationships as the two compile-time parameters subtly interact.

3. Because both the *M* and *N* parameters we have chosen for this benchmark (*if-else depth* and *body size*, respectively) affect the source-code size of the function being compiled similarly for this benchmark, we can assess how they trade off at a given overall input size.
4. Because there's a family of similar functions, we introduce random compiled-object-quality noise with each incremental change in configuration (e.g., associated with page and cache-line alignments), which will tend to average out and cannot be addressed by simply rerunning the same executable multiple times alone.

Once we have a script capable of running an experiment on a 2D family of functions (with and without assumptions enabled), we can improve our measurements by repeating each experiment on the current platform at a given build level some number (e.g., 10) of times. Looking at each grid point individually, we might choose to drop, say, the top and bottom two values and average the remaining six. Such filtering and smoothing of the raw data points will remove much of the noise introduced by running programs on real multitasking machines and provide a better feel for the intrinsic variations among each data point we seek.

Platforms

We collected benchmark results on several platforms based largely on what we had readily accessible. To avoid biasing our results to a single platform, we inspected data on each of three operating system and CPU-architecture combinations:

1. Windows on x86, including a personal laptop (Intel Kaby Lake), a desktop machine (Intel Skylake), and a hosted virtual machine (Intel Haswell E/EP).
2. Linux on x86, including a desktop machine (Intel Gulftown EP) and a hosted machine dedicated to benchmarking (Intel Cascade Lake)
3. macOS on AArch64, including several Apple MacBook Pro personal laptops (Apple M1 Pro and Apple M1 Max)

Our most extensive analysis was performed on the x86 machine running Linux, which was dedicated solely to this benchmarking effort. Where available, we attempted our experiment with multiple different compiler toolchains (in the order in which we tried each particular compiler suite for the first time):

1. The GNU Compiler Collection (GCC), versions 10.3.0 and 12.2.0
2. Clang, version 15.0.1
3. Microsoft Visual C++ (MSVC), 2019 and 2022

4. Intel ICX 2022.2.0.20220730

As we gathered data, we focused on obtaining complete results, including sufficiently large numbers of runs to attenuate unrelated noise (as might occur in multiple runs of *the identical* executable). In particular, we chose to use the median value of multiple runs to represent a normalized raw value as input to the analysis performed for a small number of platforms, which we present at the end of this paper.

Finally, we discovered early on that some compilers would treat compiler-accessible assumption constructs that rely on explicit undefined behavior, specifically assigning through a null pointer, differently to those that employ native compiler intrinsics (e.g., `__builtin_assume`). Some compilers would aggressively optimize our benchmark using either, while others would not optimize at all based on explicit UB.

Where we saw it have some effect, we used assignment through a null pointer for our benchmarks, and in other situations we used a platform-specific compiler intrinsic, which ultimately led to the preprocessor logic used to implement our compiler-accessible assumption construct in `testFunction`:

```
#if !defined(ASSUME_INTRINSIC)
    if (1 != v)
        *(int*) 0 = 0; // if v is not 1 we have UB
#else // defined(ASSUME_INTRINSIC)
    #if defined(__clang__)
        __builtin_assume(1 == v);
    #elif defined(__INTEL_COMPILER) || defined(_MSC_VER)
        __assume(1 == v);
    #elif defined(__GNUC__)
    if (1 != v)
        __builtin_unreachable();
    #endif
#endif
```

For GCC and Clang, we did not define the `ASSUME_INTRINSIC` macro; for MSVC and Intel, however, we did, as neither of these two later compilers showed any difference in generated code when using just assignment through a null pointer.

Results and Analysis

This section explores the results of executing our plan on several platforms. In all experiments, each data point will represent the ratio of the average values for a benchmark compiled with assumptions enabled versus disabled.

First, we will show the effects of compiler-accessible assumptions observed on compile times and object sizes — two metrics indicative of the assumption's impact but necessarily correlating to the most compelling metric, i.e., overall run times. Then, we will illustrate the runtime performance impact we observed. Finally, we'll explore in detail some low-level CPU performance metrics that we gathered for one particular platform, discussing how those

metrics might reflect the behavior of our benchmark function and how they might have contributed to achieving the runtime changes we measured.

In general, we ran experiments using four compilers — GCC, Clang, MSVC, and the Intel compiler — on a mix of Linux (x86), Windows (x86), and macOS (AArch64: Apple M1 Pro) machines. All builds used the most aggressive optimization level available from the toolchain, `-O3` on most platforms and `/O2` with MSVC. For each configuration (*if-else depth*, *body size*, *assumption enablement*), enough runs were performed to achieve confidence in the precision of the answer (generally more so with the shorter runtimes), and we show their respective median values.

Throughout this section, we will present a representative sample of platforms. Our results for all platforms not explicitly mentioned here were generally similar in form to those we discuss.

Compile Times

We can posit that assumptions will impact compilation speed in a few different ways:

- Providing additional assumptions, gives the compiler more code to process and analyze, requiring more work and thereby plausibly increasing compilation times.

- By allowing the compiler to elide branches, sometimes early during the compilation process, we reduce the amount of code that needs to be more aggressively optimized during later phases, perhaps decreasing compilation time.

- Compilers may treat very large functions differently based on bounds compiler vendors determine, which can result in a large function not being optimized; on some compilers, compiler-accessible assumptions might even affect which side of such boundaries our benchmark function would reside.

We begin our investigation by looking at the ratios of compile times using GCC — with compiler-accessible assumptions versus without— depicted as the 3D surface shown in Figure 1.

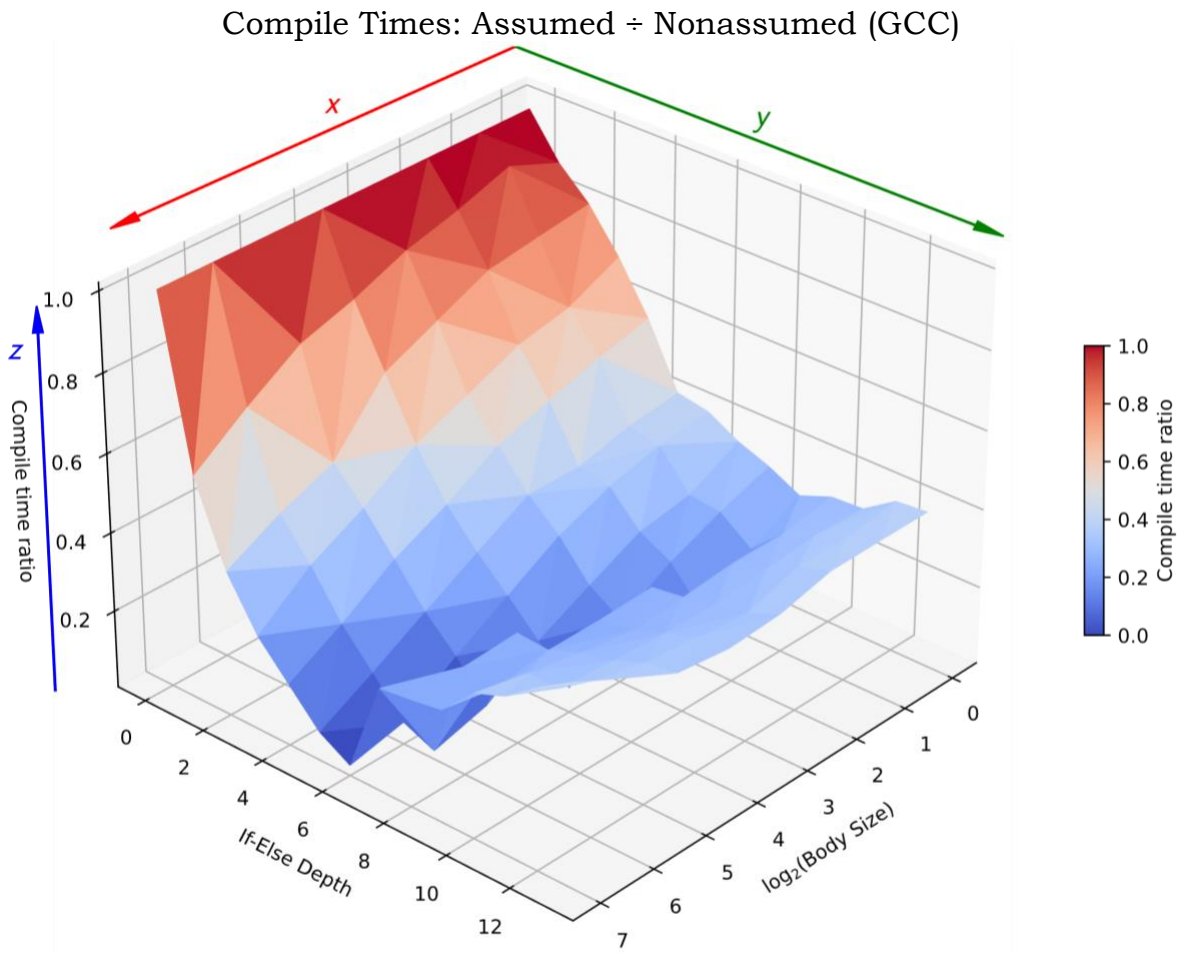


Figure 1: This surface represents ratios of compile duration: assumed versus nonassumed. Both height and color indicate compile time ratio (Cascade Lake, GCC 12.2.0, -O3).

Here we have graphed the ratio of compilation times as we varied the *if-else depth* from 0–12 (*y*-axis) and the *body size* from 2^0 to 2^7 lines (*x*-axis). Note that the graph is oriented so that the origin is located in the back, behind the surface, to better display the shape of this graph; hence, the corner in the lower front represents the maximum displayable values for *x* and *y*.

As the *body size* increased, a corresponding significant decrease occurred in relative compilation time with assumptions enabled. Given that our benchmark demonstrates that adding a simple assumption results in the elimination of substantial portions of the resulting generated code (all but one of the $2^{(\text{if-else depth})}$ branches), obviating having to optimize further and generate those superfluous branches appears to dominate compile time.

While this 3D view is visually striking and easy to understand, we found it limiting when attempting to interpret many of the other results we had measured. Therefore, we found visualizing our results as heat maps more serviceable, and the heat maps more readably encapsulate the details of our

results. The relationship between our 3D surface and our heat maps is illustrated in Figure 2 by successively rotating the data in Figure 1.

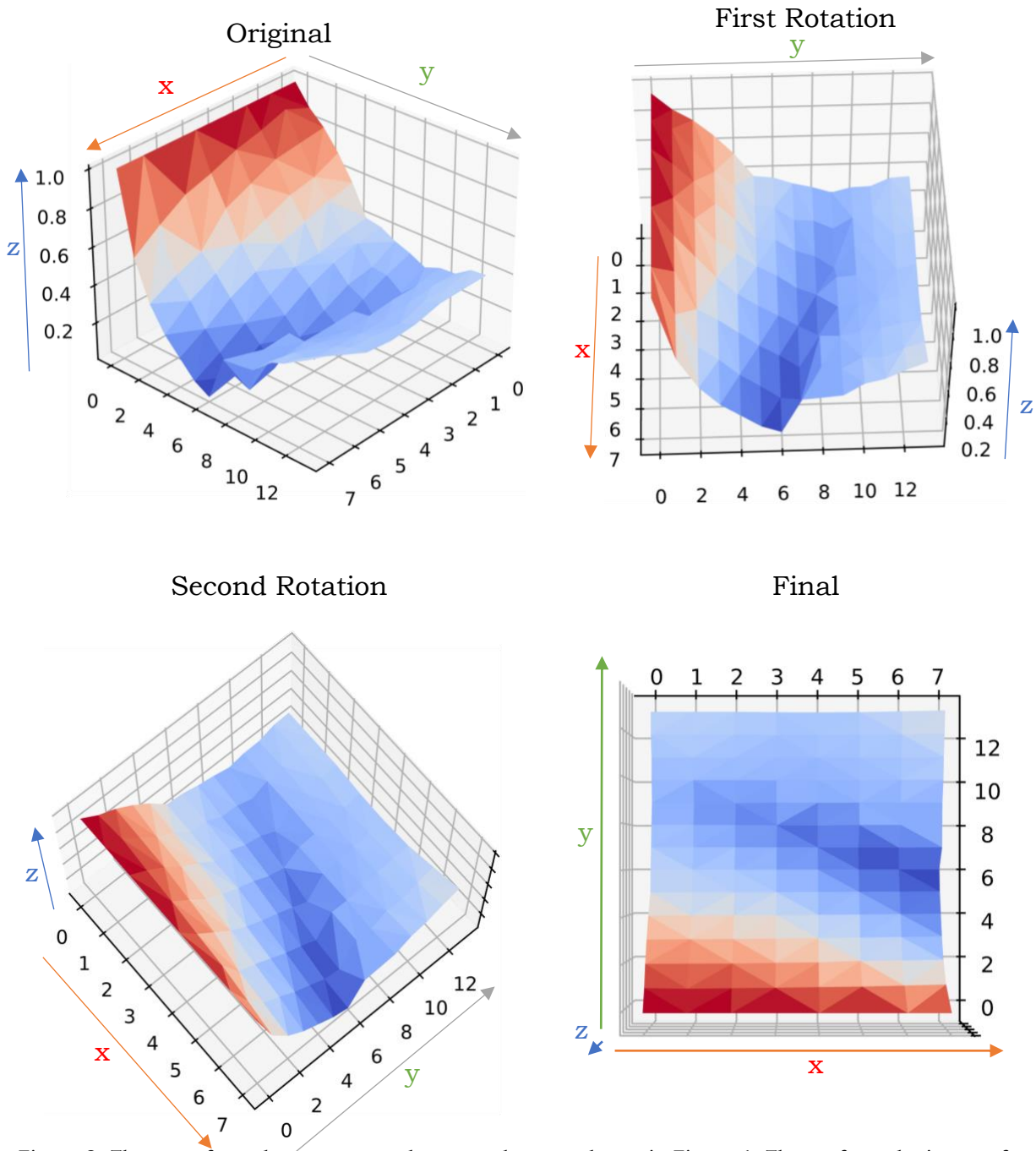


Figure 2: These surface plots represent the same data as shown in Figure 1. The surface plot is transformed into a heat map via three successive rotations as a visual aid (Cascade Lake, GCC 12.2.0, -O3).

As we can see, the 2D heat map shown in Figure 3 represents the same data as the 3D graphs shown in Figures 1 and 2, but now we can read the individual result values explicitly and more readily observe small quantitative differences across surrounding data points.

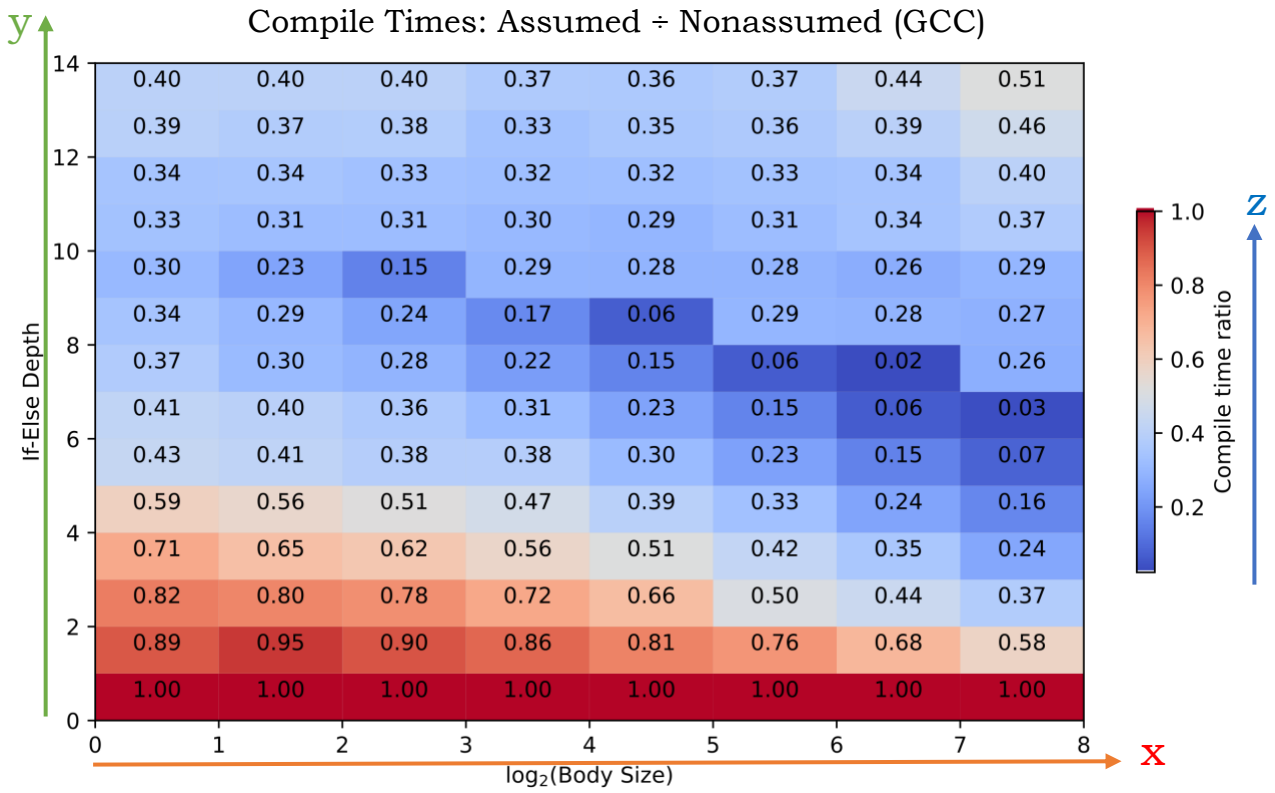


Figure 3: This heat map represents ratios of compile duration: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

Exploring different platforms, we observed similar results when compiling using Clang, as can be seen in Figure 4.

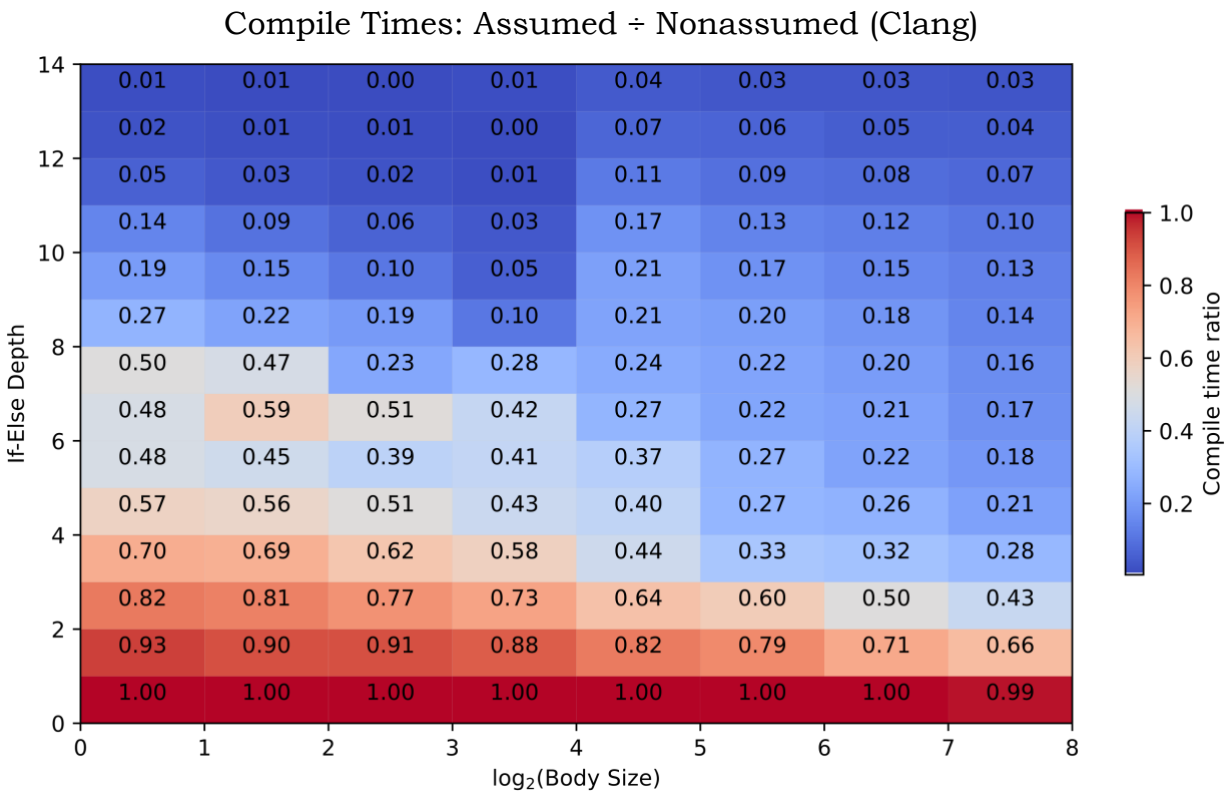


Figure 4: This heat map represents ratios of compile duration: assumed versus nonassumed (Cascade Lake, Clang 15.0.1, -O3).

Interestingly, when using MSVC, we observed results of a different shape, as shown in Figure 5.

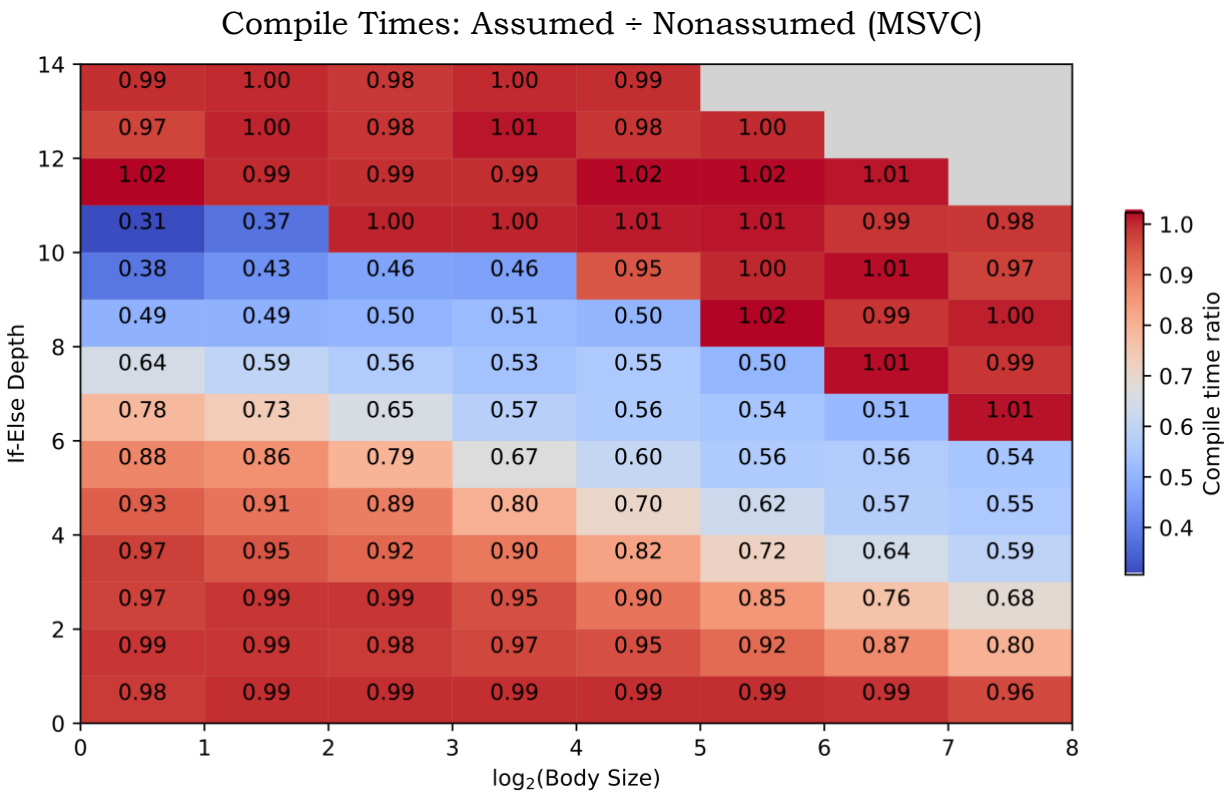


Figure 5: This heat map represents ratios of compile times: assumed versus nonassumed. Gray cells indicate unavailable data (Shadow Cloud VM, Haswell E/EP, MSVC 19.33.31630, /O2).

Past a certain downward-sloping size frontier[11], enabling compile-time assumptions results in minimal differences in compile times, and, given the relatively large compilation times (on the order of minutes or more), the small variance observed is not unexpected. In these cases, it seems as though the compiler has considered our assumption and, based on decision-making metrics of its own, decided to do very little differently for code generation than it did without the assumptions enabled. This size-frontier pattern will resurface again in this compiler for both code sizes and run times.

Object Sizes

The generated code in our benchmark function primarily consists of two kinds of programmatic constructs:

1. Comparisons and branch instructions that implement the `if-else` statements
2. Arithmetic operations that make up the body on each of the $2^{(\text{if-else depth})}$ branches

[11] Note that the aspect ratio of this heat map is skewed to make the *x* axis look disproportionately larger than the *y* axis, which belies the intent that increasing by one along either axis essentially doubles (asymptotically) the number of source lines of input to the compiler.

The object size, which we can easily measure, also includes a fair bit of metadata on the contents of our translation unit, but that metadata is fixed and independent of the variables in our experiment. Measuring the size of just the generated function would exclude this fixed overhead but is not as implementable reliably in a portable fashion. We should expect that, when assumptions are enabled and when a compiler properly infers all branch results based on our benchmark's assumption, exactly 0 comparisons and branches and only a single instance of the $2^{(\textit{if-else depth})}$ bodies are generated. The relative code size should, therefore, approximate (asymptotically) a $1/2^{(\textit{if-else depth})}$ ratio of object sizes (enabled/disabled) when the compiler fully exploits compiler-accessible assumptions.

Figure 6: This heat map represents ratios of object sizes: assumed vs nonassumed (GCC, 12.2.0, -O3).

The shown GCC (and unshown Clang and ICX) results in Figure 6 clearly illustrate this expected result. As either *body size* and *if-else depth* grow, it soon dominates the object file format's fixed-size overhead, and we observe a general trend in our results matching our expected, exponentially decreasing ratio of object sizes to *if-else depth*.

Next, consider the heat map representation for relative object-code sizes when using MSVC, as shown in Figure 7.

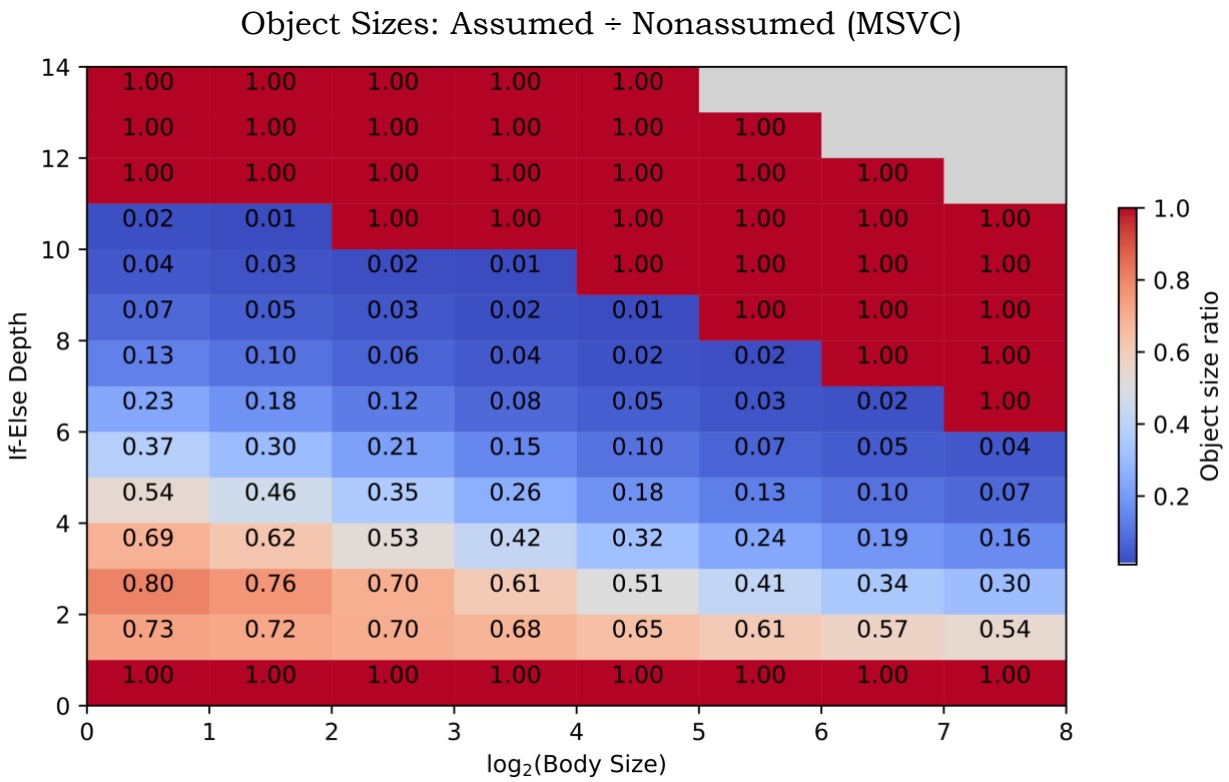


Figure 7: This heat map represents ratios of object sizes: assumed versus nonassumed. Gray cells indicate unavailable data (MSVC 19.33.31630, /O2).

Again, MSVC shows similar results to the other platforms in the cases where the assumption impacted object size. For those *if-else depth* and *body size* configurations where we saw no change in the generated object-file size, we observed, at most, a negligible (±3%) difference in corresponding compile times.

Further inspection of the generated objects showed that they are identical; the assumed and nonassumed object files had no differences in the cases above the clear threshold, which indicates that, as we pass some threshold of function size or complexity or both, this platform no longer attempts to reason about the impact of our assumption, and thus no branches are elided.

As expected, measuring the linked task size showed the same trends we saw with the object file size since the other object files in our framework remained unchanged across builds.

Run Times

With compiled executables in hand, we ran each configuration several times and measured the amount of time each process took to run to completion. The median value was used to show the runtime ratios.

Let's begin with the results for relative run times obtained using GCC, as depicted in Figure 8.

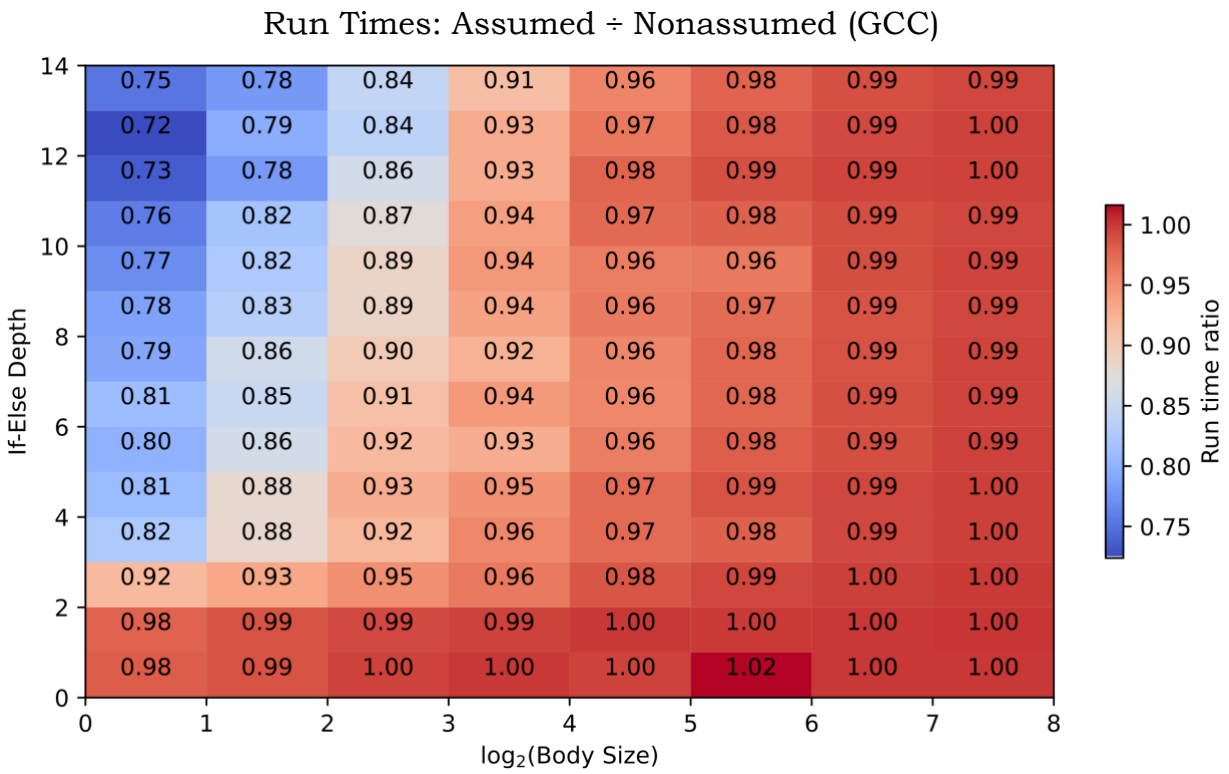


Figure 8: This heat map represents ratios of run times: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

Similar results for relative run times obtained using Clang are depicted in Figure 9.

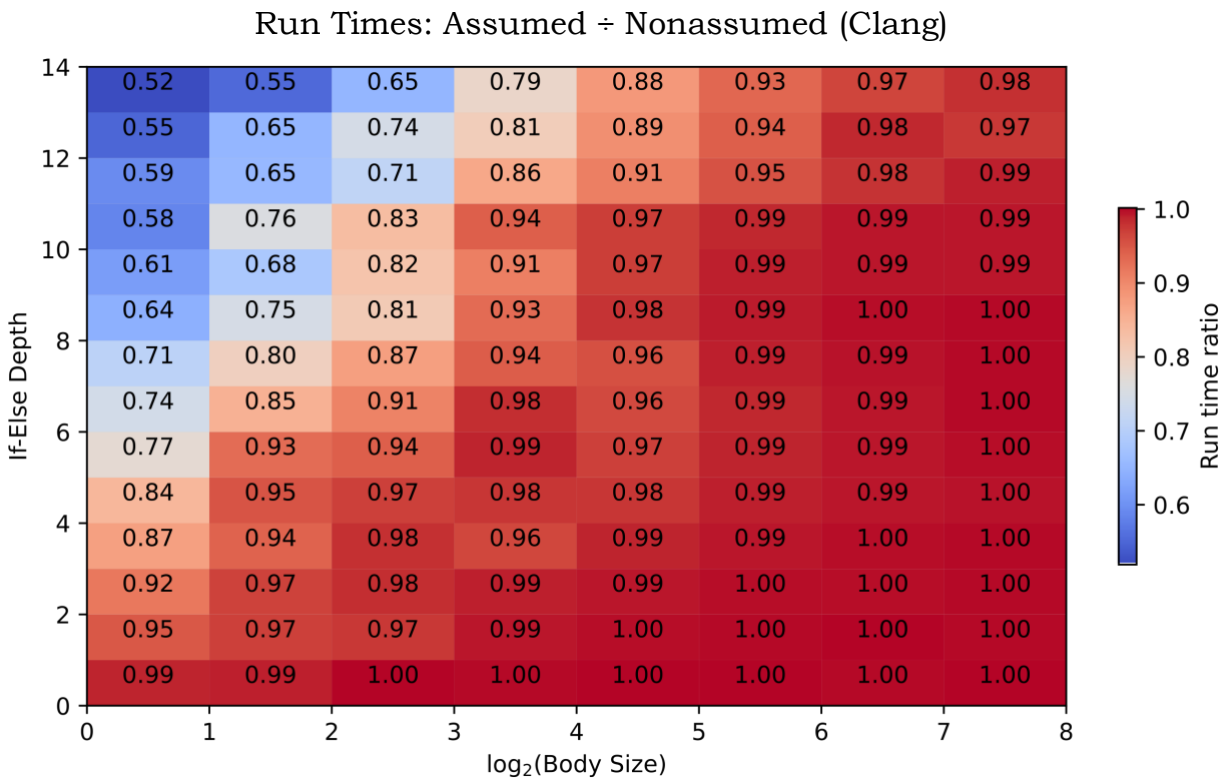


Figure 9 This heat map represents ratios of run times: assumed versus nonassumed (Cascade Lake, Clang 15.0.1, -O3).

The first important takeaway is that in no case did we see a performance degradation on either of these platforms due to enabling our compiler-accessible assumptions. Overall, as the *if-else depth* increased, we saw a downward trend in relative run times. As expected, an increase in *body size* diminished the improvement in run times because a more significant portion of run time was spent in the body (which was deliberately chosen to be immune to assumption-based optimization) and not in the comparisons and branches, which were being elided by our assumption.

When comparing the relative run times for Clang and GCC on the same hardware, Clang showed a reduction of 48%, a marked improvement due to compiler-accessible assumptions compared to GCC, which peaked at only a 28% reduction. Upon inspection, however, this greater relative benefit seemed to result from GCC's more aggressive optimization of the unassumed build than Clang's, and thus we saw less of a relative bump in the assumed build using GCC.

Next, we focused on the relative run times on MSVC, depicted in Figure 10.

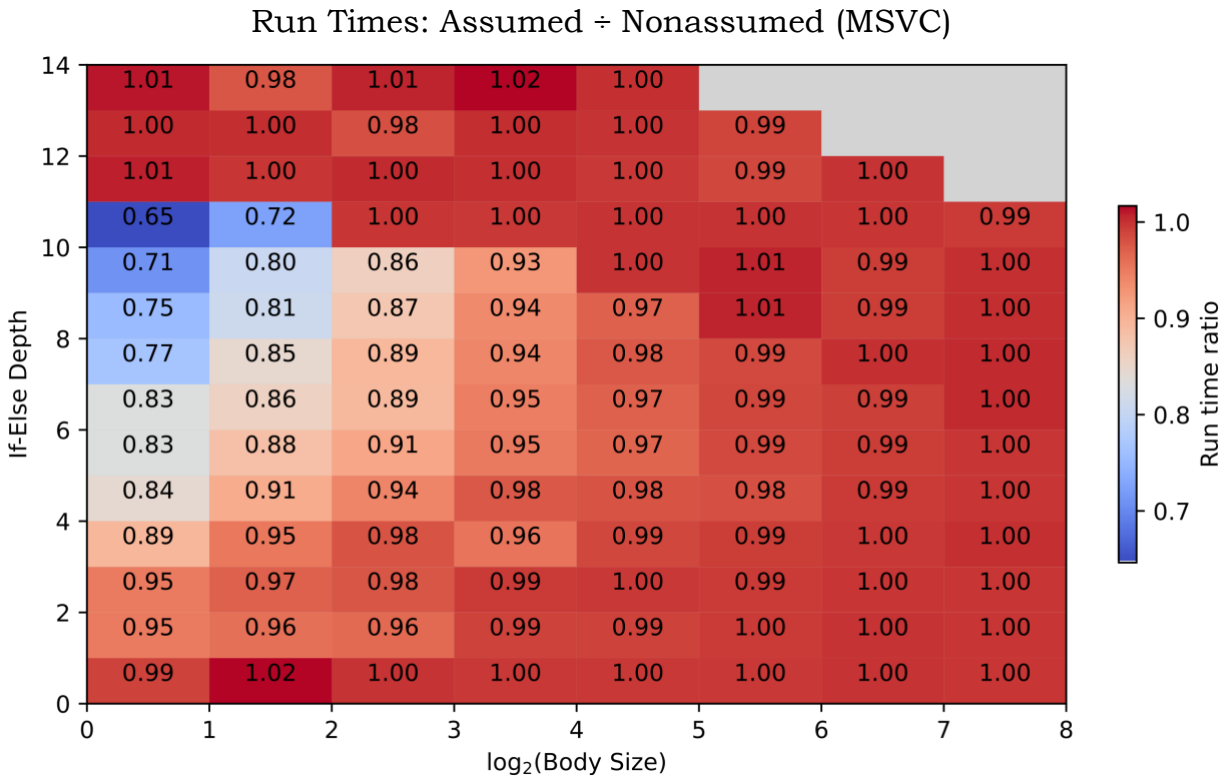


Figure 10: This heat map represents ratios of run times: assumed versus nonassumed. Gray cells indicate unavailable data (Shadow Cloud VM, Haswell E/EP, MSVC 19.33.31630, /O2).

MSVC showed the same trend where the assumed build with all branches elided took relatively less run time (i.e., ran faster) than the nonassumed build as the *if-else depth* increased for smaller-sized programs. But, yet again, past a certain input-size threshold (based on the combined *if-else depth* and *body size*), we observed that compiler-accessible assumptions had no meaningful difference in runtimes.

We know, from having inspected the pairs of executables, that they were byte-for-byte identical; hence, we would not expect any significant difference in relative runtimes, and any difference that we did observe would be attributable solely to unrelated noise while running the benchmark. Still, meaningful information can be gleaned from this data.

If we look at each of the 34 entries in question, we will find that they group quite *normally* around 1.00: **2** (0.98), **5** (0.99), **21** (1.00), **5** (1.01), and **1** (1.02). The mean, median, and mode are 0.9994, 1.00, and 1.00, respectively. In fact, this normal distribution is remarkably close to perfect, reflecting an absence of bias.

Now suppose hypothetically that the executables were not identical and that there was some minuscule and arguably insignificant drift downward (say -0.1%) associated with an enabled assumption. Taken over these 34 entries, we

expect the cumulative (negative) drift to be a measurable -0.034. Given our observed random cumulative drift of -0.02 (where the expected drift is zero), even a mere average drift of -0.01% is something we should probably be able to observe.

Now consider that just one *experiment* involves not 34, but 8 × 14 = 112 separate pairs of trials. In that case, we could expect a cumulative drift of .112, whereas we would expect the typical random cumulative to *decrease* in distance from 0.00. Hence, even an average drift of just -0.1% would have been *well* above the noise level for this particular experiment.

Looking across the four major compilers surveyed in this paper, GCC, Clang, and Intel compilers exhibit increasing relative runtime performance with increasing *if-else depth*, but that relative performance advantage diminishes with increasing *body size*. MSVC also follows this trend for smaller inputs; however, we observed zero change in the object code size produced above a certain size threshold and, correspondingly, no meaningful effect on runtime performance.

Hardware Performance Counter Measurements

By comparing the trends of object size and runtime ratios, we can observe that most of the decrease in object sizes does not translate to faster run times, which matches our expectations since most of the code being eliminated from generated objects is dead code. As *body size* increased, an ever-smaller proportion of the code eliminated was (otherwise executed) comparisons and branches.

To better understand the runtime results we observed, we reran our benchmarks using the `perf` tool on Linux to collect various CPU performance data (counters). These results tell an interesting story of how improved code generation can and cannot realize runtime performance gains. We will present this level of detail for GCC on Linux only, though we obtained similar results for Clang and ICX on the same hardware.

Branch Instructions

Our benchmark primarily alters generated code by removing branches and the resulting dead code. When executing a nonassumed build, we should expect to see each `if-else` construct (up to the depth we have configured) execute a test-and-branch operation. On the other hand, the assumed build should have all of these branches elided. Overall, according to our measurements, this theory generally holds.

For example, the relative branches executed during a run of our benchmark with and without compile-time assumptions enabled is depicted in Figure 11.

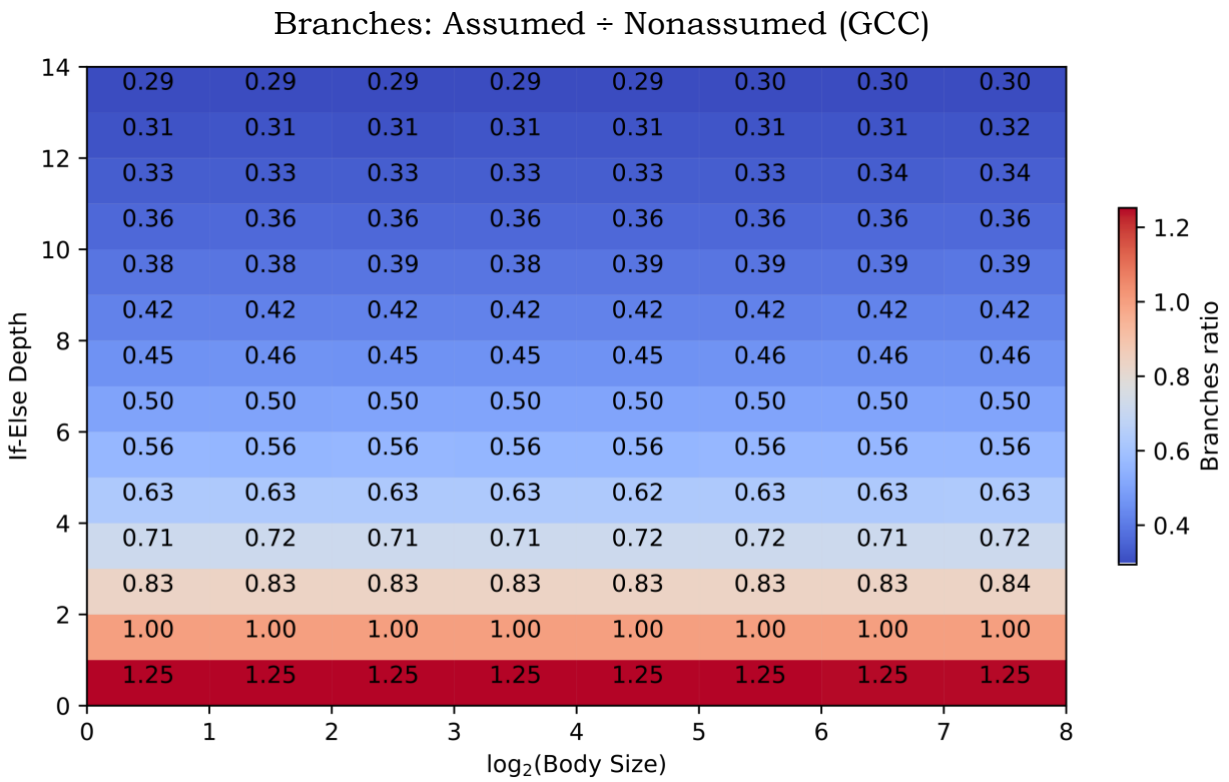


Figure 11: This heat map represents ratios of branches executed at run time: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

An interesting observation, the number of branches at an *if-else depth* of 0 was greater for the assumed build than for the unassumed one. When we inspected the generated code, we observed that our makeshift assumption contraption in GCC, implemented with an `if` statement followed by assignment through a null pointer, actually generated the comparison and branch followed by a `UD2` instruction, which indicates to the CPU that an illegal state has been reached. This extra branch, resulting from translating our makeshift compiler-accessible assumption construct to a runtime check, did not appear when we used `__builtin_unreachable()` on GCC or with the other compilers on Linux whether we employed a native compiler-intrinsic or our makeshift compiler-accessible assumption construct to elide branches.

Furthermore, these results confirm that the number of branches is independent of the *body size*, which we expected as each `if-else` body in our benchmarks is deliberately composed exclusively of sequential arithmetic operations. The reduction in branches correlated with the reduction in run time, but not completely and directly. More specifically, even with a significant reduction in branches, the reduction in run time was often negligible for the smaller cases.

One might reasonably posit that this lack of relative difference is due to the CPU's branch predictor correctly guessing all of these branches in the unassumed build (thereby making their complete removal have minimal effect) until a sufficient *if-else depth* is reached.

Instructions

C++ programmers commonly treat the number of instructions executed as a primitive proxy for run time. Unsurprisingly, this assumption has limited utility and does not always hold, as evidenced by our benchmarks. For example, consider the relative measured number of instructions executed in our benchmark runs on GCC, as depicted in Figure 12.

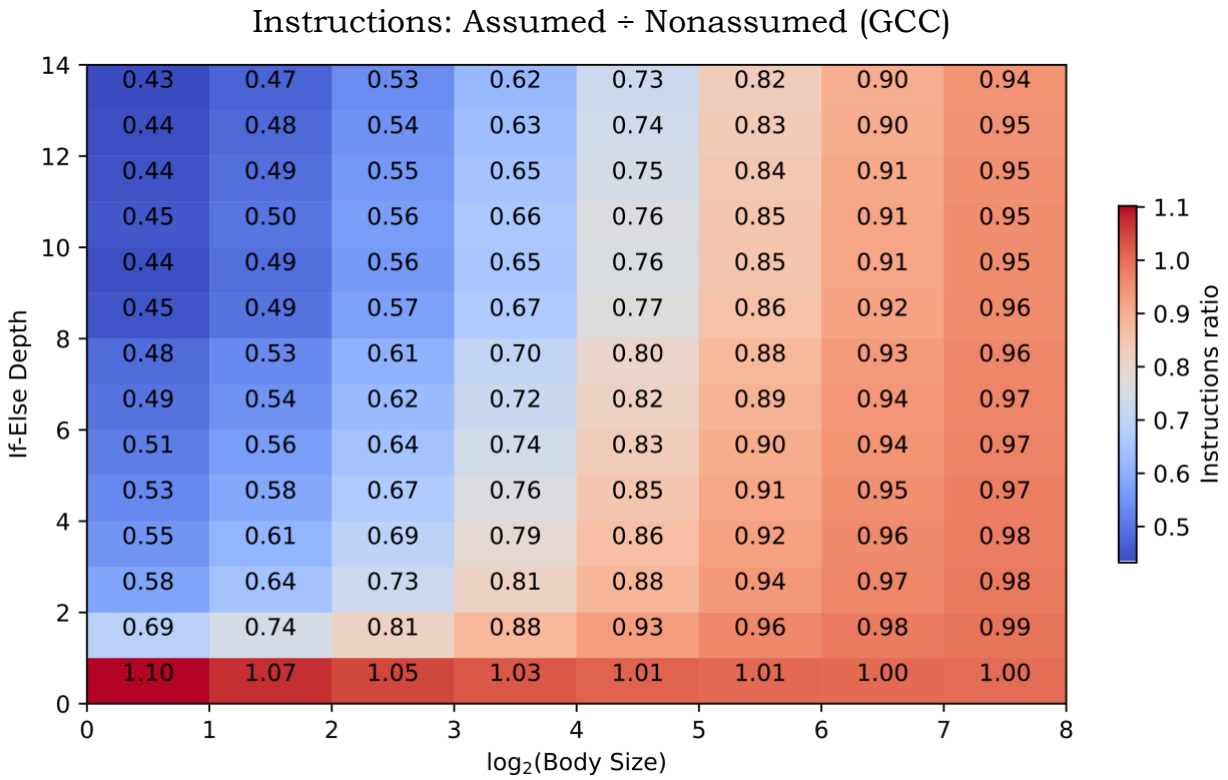


Figure 12: This heat map represents ratios of executed instructions: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

The reduced number of instructions executed roughly correlates to the number of comparisons and branches elided by our assumption. The reduction increases as the *if-else depth* increases, and the magnitude of that reduction decreases as the size of the body increases.

Again, the removed instructions do not impact runtime in proportion to the number of instructions in the body that remain in the assumed build because the instructions are either branch instructions that are reliably predicted or

comparisons of values already in registers, both of which contribute minimally to run time. The relative values for executed instructions are similar to those for run times, suggesting that these branches and comparisons, however cheap, are not free.

For the smallest cases, the same effect we saw earlier when measuring branches exhibits itself here when counting instructions, resulting in a small increase in instruction counts due to the branches and comparisons generated by our home-grown compiler-accessible assumption construct on GCC. These extra instructions were dominated by the decrease in instructions for any non-*if-else depth*. We saw no increases in the number of instructions with other compilers.

Level 1 Data Cache Loads

Interpreting memory access measurements is challenging but can often account for a significant portion of run time in modern CPUs. Within our benchmark, each operation in the body requires memory accesses to load or store the volatile object. For example, we measured level 1 data cache loads to see how many of those instructions led to actual loads from memory by the CPU when compiled under GCC, as depicted in Figure 13.

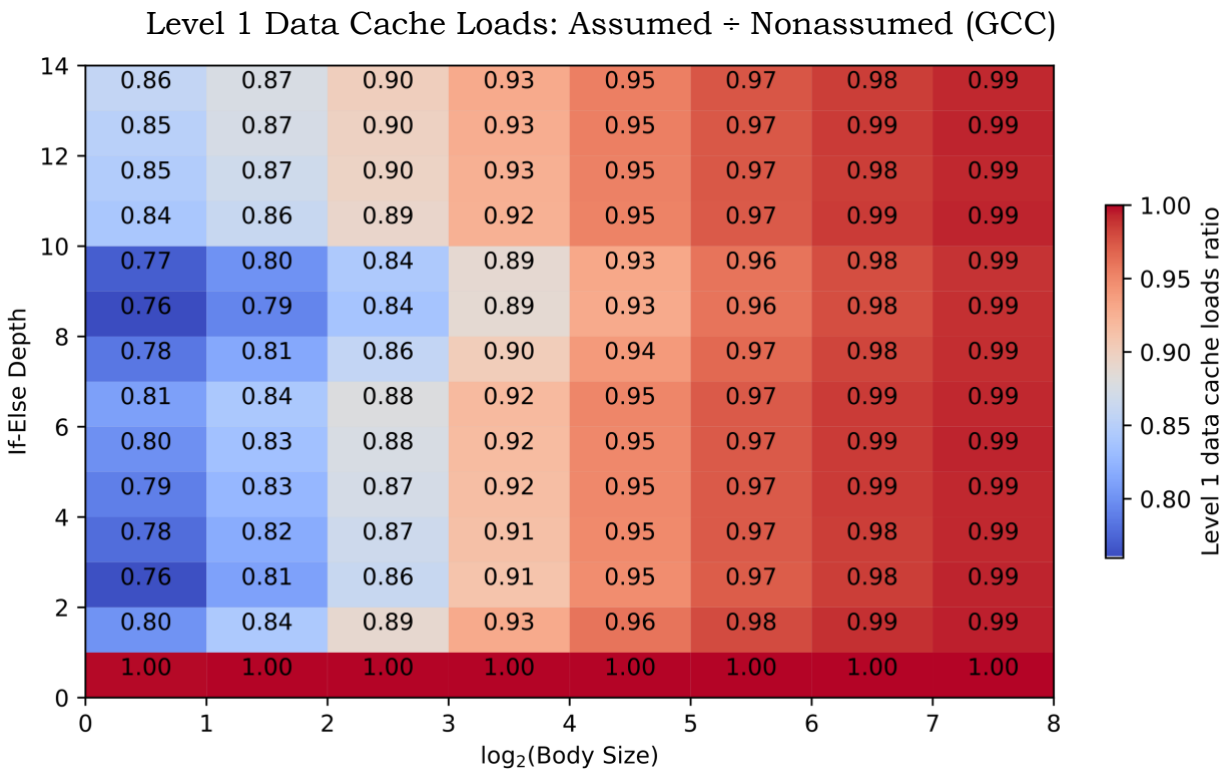


Figure 13: This heat map represents ratios of level 1 data cache loads: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

Here we see a trend of decreased relative loads when our assumption elides at least one branch (i.e., *if-else depth* $\geqslant 1$). One might reasonably posit that this result is due to the compiler's enhanced ability to reorder instructions leading to the CPUs' enhanced ability to pipeline consecutive function calls without stopping between each iteration to evaluate a series of comparisons and branch instructions. As we would expect, the longer the series of arithmetic instructions per body (increasing *body size*), the less relative effect reordering the ends of a single sequential arithmetic sequence will have. The cases where *if-else depth* is 0 further support this theory by demonstrating that there is no corresponding impact on memory-usage behavior when no branches are elided.

Level 1 Data Cache Stores

Level 1 data cache stores, which correspond to writes to memory, show a different result. For example, consider the relative cache stores for our benchmark program running on GCC, as depicted in Figure 14.

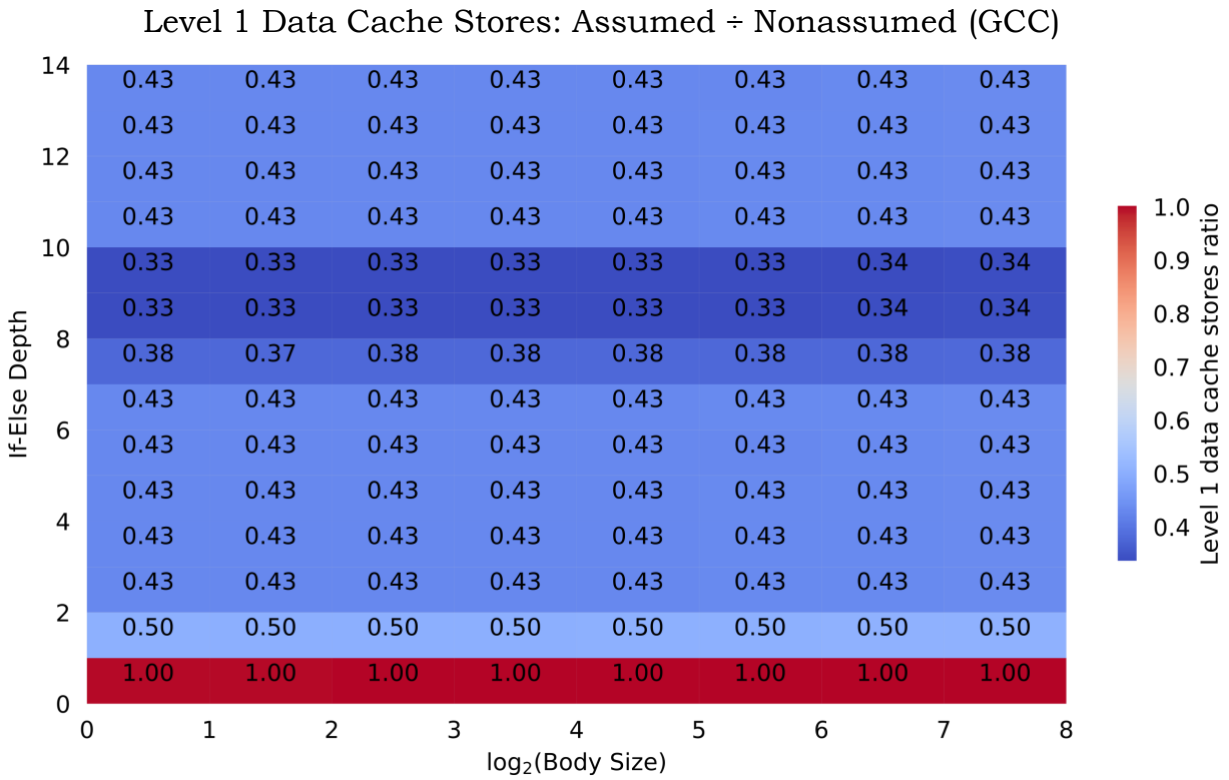


Figure 14: This heat map represents ratios of level 1 data cache stores: assumed versus nonassumed (Cascade Lake, GCC 12.2.0, -O3).

Inspecting the actual measurements, the cases with no branches (i.e., those with an *if-else depth* of 0 or enabled assumption) showed a very stable number of data cache stores, around 3.2 per iteration. The cases with no branches roughly doubled the number of stores per iteration. Clang and ICX builds exhibited this same behavior. The presence of these branches, all of which involved comparisons on the nonvolatile function parameter `v`, added additional data cache loads was undetermined from the generated object code.

Summary

Overall, we observed several measurable attributes of our benchmark improve when we enabled assumptions. The general trends followed our expectations in two ways:

1. Increasing the *if-else depth* showed greater improvement in most cases where there was any meaningful effect.
2. Increasing *body size* diluted the benefits since a larger fraction of the implementation of our `testFunction` consisted of sequential code specifically designed not to be impacted by enabling the compiler-accessible assumption.

More importantly, we can take away a few significant lessons from these results: Enabling compiler assumptions often led to easily observable improvements in generated code, such as code size or branches taken, but had a muted impact on run times. As we scaled higher, improvements that showed little to no effect in run time for smaller values of *if-else depth* quickly began to exhibit meaningful benefits, indicating that although modern hardware has made some operations, such as branches, very inexpensive, those operations are not entirely free and that practical limits apply to what compilers and hardware can achieve unassisted.

CONCLUSION

Demonstrating that compiler-accessible assumptions, such as `__builtin_assume`, reduce run time is difficult because (1) code for which such assumptions make no difference is common and (2) even when substantially different object code is generated, run times are often not meaningfully affected. Hence, absent objective data to the contrary, one can easily albeit mistakenly conclude that — due to modern CPU architectures, branch predictors, and ever-maturing compiler technology — no meaningful runtime performance benefits remain to be had from employing explicit assumption constructs. The justifications of these conjectures are also misguided.

In this paper, we began by presenting many alternate cogent theories for *why* making explicit assumptions available to the compiler *might* reduce run time; for example, unsigned arithmetic is required to wrap, whereas signed arithmetic, in conjunction with explicit knowledge that the (signed) result will never be negative, enables the compiler to optimize, assuming the expression will not overflow.

We settled for demonstrating improved runtime performance for just one pattern: nested `if-else` statements depending on predicates whose values are informed by explicit compiler-accessible assumption constructs. We then described a simple, portable, custom benchmarking framework comprising a general-purpose driver that calls a particular compile-time-parameterized family of related test functions.

To observe whether compiler-accessible assumptions *can* have a meaningful effect on run times, we organized our family of functions on a single integer, `x`, around two compile-time integer parameters: (M) is the depth of a balanced tree of `if-else` statements whose conditions involved the value of `x`, and (N) is the power-of-two number of consecutive nonbranching statements in an unrelated sequential computation. A third, Boolean parameter controls whether the compiler is aware of the assumption that $x = 1$.

An *experiment* consisted of building each of 112 configurations ($M \in [0..13] \times N \in [0..7]$) with and without the compiler-accessible assumption enabled. Running the experiment allowed us to obtain the respective compile times, generated object-code sizes, and program run times for each combination of values for M and N — with and without compiler-visible assumptions enabled. We then graphed the ratios of each of these three respective metrics — with-to-without assumptions enabled — as a surface in a three-dimensional heat map, thereby allowing us to readily and visually observe the detailed interactions between *M*, *N*, and the Boolean flag. We repeated this experiment across several platforms (e.g., Windows, Linux, and macOS) and compilers (e.g., Clang, GCC, MSVC), running on various hardware.

The totality of our results for experiments across multiple compilers and computer architectures involving sufficiently optimized build modes revealed that increasing the depth of the `if-else` tree (*M*) consistently demonstrated meaningfully reduced run times when compiler-accessible assumptions were enabled. This relative reduction in run times was most pronounced (up to 48%) for smaller numbers (*N*) of consecutive nonbranching statements. Larger values of (*N*) produced proportionally smaller generated object code with assumptions enabled but offered minimal effect on run times.

Importantly, throughout all of the benchmark tests we performed, there were no cases where we observed any meaningful increase in runtime as the result of enabling compiler-accessible assumptions. In those few cases (MSVC only) where we did observe a (spurious) increase (1-2%), it was due purely to random noise as the object code run in either case — with or without assumptions — was byte-for-byte *identical*.

Now that an explicit-assumption construct — the new `[[assume]]` attribute — is part of the language, we can expect to see compilers providing these same benefits to code that is entirely standard-conforming and hence portable. As this now-portable, compiler-accessible assumption construct becomes more widely used, we expect to see commensurate improvements in compiler technology to leverage explicit-assumption information and thus continue to provide ever-increasing *runtime* performance.