15611 Lab 5 Report

Ethan Cheong & Wu Meng Hui

May 7, 2024

1 Introduction

Our L5 compiler can be found at https://github.com/15-411-S24/KiyosiIto/tree/lab5, with commit hash 6b49ef2fdcbae7faa14e65b281a300be1e55d3e4. The original structure of our compiler backend after L4 is shown in Figure 1. Compiler phases are located in their corresponding named directories in `\lab5\lib\`. Intermediate representations are located in the `\lab5\lib\data` directory; IR corresponds to `tree.mli` and 2AS corresponds to `assem.mli`.

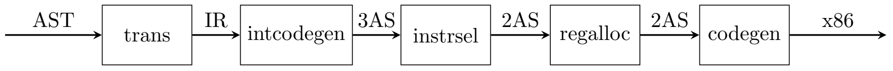

Figure 1: Original compiler backend.

For L5, we implemented Tail-Call Optimization (TCO), Function Inlining, Aggressive Deadcode Elimination (ADCE), Simple Copy Propagation (SCP), Strength Reduction (SR), Register Coalescing in register allocation, and Peephole Optimizations. ADCE, SCP and SR are implemented on Static Single-Assignment Form (SSA); we implement passes to convert 3-Address Assembly to and from SSA so these optimizations can be run. Figure 2 shows the backend after including these passes. We also implemented Sparse Conditional Constant Propagation (SCCP) as an alternative to SCP, and compare the two in a later section.

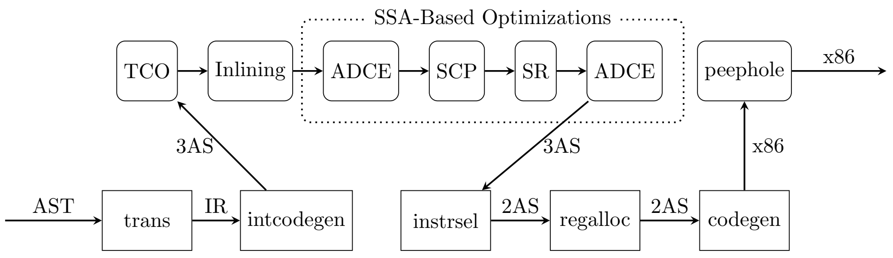

Figure 2: Compiler backend with optimization passes.

To allow easy benchmarking of optimizations, boolean flags for whether to activate the optimization are located at the start of `top.ml`. The `-O1` flag runs the compiler with all currently active optimizations.

1.1 Unsafe Mode

The `--unsafe` flag runs our compiler in *unsafe mode*, which eliminates safety checks from generated code. This reduces the size of generated code in unsafe mode and opens up room for certain optimizations.

1.1.1 Eliminating Memory Safety Checks

In translation (`\trans\trans.ml`), any checks for unsafe memory accesses in expressions and statements are eliminated. The relevant expressions and statements are `Deref`, `Alloc_array`, `Array_access`, `Struct_access`, `Assign` with memory-type lvalues, `Asnop_pure_mem` and `Asnop_impure_mem`. For example, pseudocode for the translation of a pointer dereference `Deref(e)` is (following notational convention from [Hof24b]):

```
ins(Deref(e)) = [
    ins(e)
    t1 <- res(e)
    if t1 == 0 then mem_exn else deref
mem_exn:
    exception(mem)
deref:
    t2 <- Mem_read(t1)
]
res(Deref(e)) = t2
```

but in unsafe mode, this becomes

```
ins(Deref(e)) = [
    ins(e)
    t1 <- res(e)
    t2 <- Mem_read(t1)
]
res(Deref(e)) = t2
```

1.1.2 Eliminating Shift Checks

As per the L2 specification, shift instructions throw an arithmetic exception if the shift quantity k is not in the range $[0, 32)$. For example, a 3-address assembly shift instruction `dest = lhs << rhs` is converted to the following instructions during instruction selection (`\instrsel\instrsel.ml`), which throws an arithmetic exception by dividing by zero:

```
    movl rhs %ecx
    test %ecx %ecx
    jl shift_exn
    cmp 31 %ecx
    jg shift_exn
    movl lhs, %r10d
    sal %ecx, %r10d
    movl %r10d, dest
    jmp shift_end
shift_exn:
    movl $0, %r10d
    idivl %r10d
shift_end:
```

However, in unsafe mode the following assembly is emitted instead:

```
    mov lhs, %r10d
    mov rhs %ecx
    sal %ecx %r10d
    mov %r10d dest
```

2 Optimization Passes

Cycle counts and executable sizes for all optimization passes discussed in the report are located in appendix. Experiments were run on a 2020 M1 MacBook Air (M1), with 16GB Memory and running on macOS Ventura 13.6.1. `timecompiler` was run on a Docker container running the `cmu411` autograder image with Linux AMD64 emulation.

All graphs are located in the appendix.

2.1 L4 Cleanup

While not technically an optimization pass, we added a flag `l4_cleanup` to `top.ml`. When this flag is set to `true`, translation of array accesses changes to take advantage of x86 *disp(base, index, scale)* memory addressing.

For example, in unsafe mode, an array access of the form `arr_exp[index_exp]` with type τ is translated as follows:

```
ins(arr_exp[index_exp]) = [
    ins(arr_exp);
    a <- res(arr_exp);
    ins(index_exp);
    i <- res(index_exp);
    t2 <- Mem_read(-8(a))
    i3 <-zero-ext- i
    a4 <- i3 * sizeof(tau)
    a5 <- a + a4
    t6 <- Mem_read(a5)
]
res(arr_exp[index_exp]) = t6
```

But when cleanup is enabled, the translation is instead

```
ins(arr_exp[index_exp]) = [
    ins(arr_exp);
    a <- res(arr_exp);
    ins (index_exp);
    i <- res(index_exp);
    t2 <- Mem_read(-8(a))
    i4 <-zero-ext- i
    t3 <- Mem_read(0(a, i, sizeof(tau))
]
res(arr_exp[index_exp]) = t3
```

provided `sizeof(tau)` results in a valid scale factor (1, 2, 4 or 8).

When comparing the generated x86, using the cleanup flag provides improvements over the baseline in terms of speedup compared to GCC -O1 for all the test cases, in both safe and unsafe mode (Figure 3).

In addition, the size of the generated executable is strictly smaller on all the bench test cases, except for `jen.l4`, which remains the same size (Figure 4). As such, we always leave this flag on when testing subsequent optimizations.

2.2 Tail-Call Optimization

`\tailcall\tailcall.ml` performs basic TCO without accumulation. We refer to [Hof24a]; however, since our representation of 3-Address Assembly does not contain specially named argument registers, we instead introduce new temps to play the role of argument registers. Functions are marked as tail-recursive during the translation phase, and this information is propagated down all the way to the 3-Address Assembly level. Consider the following function:

```
int powacc (int b, int e, int a) {
  if (e == 0) return a;
  else
    return powacc(b, e-1, a*b);
}

int main() {
  return powacc(2, 3, 0);
}
```

Without tail-call optimization, the generated 3-Address Abstract Assembly looks like

```
powacc(t3, t4, t5):
if(t4==0) then if_true_f_powacc_1 else if_false_f_powacc_1
if_true_f_powacc_1:
ret t5
if_false_f_powacc_1:
t8 <-- t3
t9 <-- t4 - 1
t10 <-- t5 * t3
t6 <-- powacc(t8, t9, t10)
ret t6

main():
t11 <-- 2
t12 <-- 3
t13 <-- 0
t7 <-- powacc(t11, t12, t13)
ret t7
```

Our basic TCO pass introduces extra temps to serve the role of argument registers:

```
powacc_prologue(t3, t4, t5):
t14 <-- t3
t15 <-- t4
t16 <-- t5
powacc:
if (t15==0) then if_true_f_powacc_1 else if_false_f_powacc_1
if_true_f_powacc_1:
ret t16
if_false_f_powacc_1:
t8 <-- t14
t9 <-- t15 - $1d
t10 <-- t16 * t14
t16 <-- t10
t15 <-- t9
t14 <-- t8
jmp powacc

_c0_main():
t11 <-- 2
t12 <-- 3
t13 <-- 0
t7 <-- powacc_prologue(t11, t12, t13)
ret t7
```

This ensures the generated 3-Address Assembly remains compatible with our code to set up function calls, and also eliminates the need to set up the stack for the tail-recursive call.

When run on the benchmark test cases, we found that running TCO had nearly no effect on the test cases (as reflected in Figure 5), because very few of the test cases actually contain tail recursive functions; instead, they require basic accumulation.

We transform eligible functions into tail-recursive form with accumulation in `\tailcall\maketailrec.ml`; the code is run if the `_basic_accumulation` flag is set to true. We do detection and transformation of eligible functions at the AST level, after parsing and elaboration. This is because our compiler currently detects tail-recursive functions during the translation phase, so transforming eligible functions into tail-recursive form before translation allows them to then undergo TCO. For example, the AST for a program like

```
int div2(int n) {
    if (n == 1 || n == 0) {
        return 0;
    }
    return 1 + div2(n / 2);
}

int main() {
    return div2(10);
}
```

(which was adapted from `mist.l4`) is transformed into the AST for

```
int div2(int n, int acc) {
    if (n == 1 || n == 0) {
        return acc;
    }
    return div2(n / 2, acc+1);
}

int main() {
    return div2(10, 0);
}
```

which allows `div2` to subsequently undergo TCO.

We found that combining TCO with basic accumulation was only effective on `mist.l4`, simply because it was the only test case which presented such an opportunity for optimization. Nonetheless, when combined with other optimizations, this resulted in a significant speedup for `mist.l4`, as we will show in section 3.

2.3 Function Inlining

The code for function inlining is in `\inlining\inlining.ml`. Again, we base our implementation on [Hof24a]. We use the number of temps defined in a function as an inlining heuristic, since we already keep track of these for register allocation purposes in each function header. When the `_function_inlining` flag is set to true, any non-recursive functions defined in the program with fewer than `_inline_threshold` temps will be inlined into their callers. We set `_inline_threshold` to 35; this was tuned by choosing the minimum number of temps so that the appropriate "small" functions in the benchmark test cases were inlined.

As observed in Figure 6, Function Inlining provides significant speedups over our baseline compiler for certain test cases - in particular, albert, danny, fannkuch, frank, georgy, janos and yyb all see significant improvements, with yyb going from a 2.5x speedup over GCC -O1 to a nearly 8x speedup. These same test cases also see a significant increase in executable size (Figure 7), reflecting the inherent trade-off between speed and executable size in cases where inlining is possible. Interestingly, even programs that did not see speedups from inlining (such as daisy) saw significant size increases of up to 15%. If desired, a possible way to deal with this would be to choose an inlining heuristic that is more close correlated with executable size than the number of temps - for example, the number of instructions in 3-Address Assembly.

2.4 SSA-Based Optimizations

2.4.1 Conversion to SSA

We use [App98] as a reference for conversion to SSA. While not an optimization, conversion into and out of SSA is necessary to do the subsequent SSA-based optimizations in this section. Setting the `_ssa` flag to true in `top.ml` runs the code in `\ssa\abstrtossa.ml`.

To build the Control Flow Graph in `\ssa\buildcfg.ml`, our implementation requires that all incoming 3-Address Assembly is in basic block structure; each basic block starts with a label and ends with a return/conditional jump/unconditional jump/memory exception. Any instructions after one of these and before the next label are marked as junk and ignored.

2.4.2 Aggressive Deadcode Elimination

ADCE is located in `\ssa\ssa_opts\deadcode_elim_ml`; we refer to [Tor11] for implementation details. ADCE can be toggled with the `_deadcode_elim` flag in `top.ml`. Deadcode elimination is able to identify unnecessary blocks and lines of code that are unexecuted to reduce the file size, and is also helpful for removing unnecessary Phi functions before constant propagation and conversion back into 3-Address Assembly.

However, we notice that ADCE + cleanup provides negligible benefits over cleanup; in fact, most test programs become slightly slower and larger due to the extra phi functions introduced when converting in and out of SSA (Figures 8, 9). The exception is `jen.l4`, which sees a 13% reduction in executable size and `ncik.l4`, which sees a slight performance improvement. The advantages of ADCE mainly come from combining ADCE with other optimizations.

2.4.3 Simple Constant Propagation

We use [App98] as a reference for SCP. Their algorithm has the advantage of allowing us to perform several optimizations at once, including

- Constant Propagation
- Copy Propagation
- Constant Folding (which was listed as a Peephole Optimization in the 15-411 lecture notes)
- Replacing conditional jumps with constant conditions with unconditional jumps

Our implementation of SCP is in `\ssa\ssa_opts\constant_prop.ml`, and can be toggled with the `_constant_prop` flag. Notably, we need to run ADCE before SCP; we translate while loops into a do-while structure, which results in an empty basic block. This block needs to be removed with ADCE before we run SCP on the program. We also run ADCE after SCP to eliminate any redundant moves.

Examining Figure 10, almost all programs see a slight reduction in executable size after running SCP and ADCE. Some of the programs see a slight increase in executable size when run in unsafe mode; notably, the executable size of `jen.l4` is nearly halved.

In terms of runtime, there isn't a significant increase due to constant propagation for many programs - `arrays_and_loops.l4` and `ncik.l4` see the biggest improvement (Figure 11).

2.4.4 Sparse Conditional Constant Propagation

We also implement SCCP in `\ssa\ssa_opts\sccp.ml`, using [App98] as a reference. One supposed benefit of conditional constant propagation is that unreachable code would not affect the constant propagation. For instance, in SCCP, if some temporary values are never accessed or defined, they would not affect the evaluation of some instructions, such as phi functions. In these cases, the phi functions may be evaluated and constant-propagated.

However, when run on the benchmark test cases, the effectiveness of SCCP and SCP seem to be comparable, with slight variance across the test cases (Figure 12). The results suggest that either SCP or SCCP can be implemented to achieve comparable speedups, with SCCP giving a slight edge in terms of average speedup. We use SCP in our final compiler because it does not result in a significant drop in performance on `georgy.l4`, and since our compiler already performs well enough on `yyb.l4` and `jack.l4` with SCP.

We also note that SCCP results in a reduction in size in almost all of the benchmark test cases, which aligns with our understanding of how SCCP works (Figure 13). In a compiler that was optimizing for size of executable instead, using SCCP might be preferable.

SCCP and SCP generally provide speedups across all cases. They also improve/worsen performance in the same direction across the test cases over the baseline. However, there are situations where constant propagation worsen performance, as in `daisy.l4`. This slowdown occurs due to the limited number of constant operations in the function calls, since there are many loops and memory accesses instead of arithmetic operations. Since there are few constants to propagate, the program ends up slower due to the additional phi functions that result from conversion into SSA format.

2.4.5 Strength Reduction

We adapted a simplified version of strength reduction from [War12]; it is activated by setting the `_strength_reduction` flag in `top.ml` to true. The implementation is located in `\ssa\ssa_opts\strength.ml`. Strength reduction was originally implemented as a standalone peephole optimization on 3-Address Assembly; however, because our translation phase assigns all constants in code to temps, we need to run constant propagation to replace temps with their constants before strength reduction can take effect. As such, we implement it on SSA, after our SCP pass; activating strength reduction without SCP leads to no observable speedup.

Multiplication and division by powers of 2 are converted into the equivalent left and right shift instructions. Multiplications were also translated into load effective address instructions if one of the operands is 3, 5, or 9.

Figure 14 shows the comparison of the 3 SSA optimizations on the test cases. To better see the impact of the effect of strength reduction, Figure 15 shows the speedup of the test cases when implementing all the SSA-based optimizations over just L4 cleanup.

There are some instances where strength reduction does not improve the speedup, such as in `janos.ml`, due to the additional instructions adding complexity, such as LEA instructions that are introduced. While LEA instructions can improve calculation speeds, the translation of multiplications into LEA instructions may result in more intermediate values being spilled to stack, which is the case of `frank.l4`.

2.4.6 De-SSA

The conversion from SSA back to 3-Address Assembly was again adapted from chapter 19 of [App98]. Our implementation is located in `\ssa\ssatoabstr.ml`.

2.5 Improving Register Allocation

2.5.1 Maximum Cardinality Search

Our L1-L4 compilers contain a working implementation of Maximum Cardinality Search (MCS) for finding a simplicial elimination ordering for the greedy graph coloring algorithm in `\regalloc\maxcardsearch.ml`. However, we instead always provided a fixed ordering to the greedy graph coloring algorithm (which was simply the allocatable registers in alphabetical order) by setting the `_skip_mcs` flag to 0 - this would skip MCS for programs with greater than 0 nodes in the interference graph. This was done in order to reduce compilation time and avoid compiler timeouts. For L5, we compare the effect of never doing MCS and always doing MCS (by setting `_skip_mcs = Int.max_value`) on the benchmark programs:

Using MCS almost always results in both a speedup (Figure 16) and reduction in executable size (Figure 17). This is because using the optimal ordering for greedy colouring results in fewer temps remaining uncoloured (at least for programs with chordal interference graphs) and limits spilling to the stack, reducing stack accesses. As such, we always leave MCS on when testing other optimizations.

2.5.2 Register Coalescing

We implemented register coalescing as per [Hof24c] in `\regalloc\greedycolouring.ml`. The lecture notes color registers with a number, and then assign available registers to numbers at the end. However, our implementation allocates registers eagerly by checking the set of available registers and allocating the highest priority one available. To minimize spilling, when we compare two register-allocated variables and their neighbors, if there are no registers that are available and non-interfering, the registers are not coalesced.

As expected, register coalescing results in a decrease in executable size for all programs, since unnecessary moves are eliminated (Figure 19). Somewhat unexpectedly, it also results in increased cycle counts for almost all programs (Figure 18). When examining the generated x86 assembly, this happens because it results in more temps being spilled to the stack; this is particularly expensive when this happens in instructions of the form `disp(base, index, scale)`. Since the combination of SCP and ADCE performs a similar role to register coalescing, we compare which performs better in a subsequent section.

Since most programs contain many variables, the registers are fully used. There many not much sufficient registers to be allocated for coalesced variables, resulting in spills to the stack. Although moves are eliminated, the performance degrades due to memory accesses, which add more to the cycle count than the removal of moves. Thus, register coalescing should only be used when there are few variables, or when there are few overlapping live ranges.

2.6 Peephole Optimizations

As previously mentioned, common peephole optimizations such as Constant Folding and Strength Reduction were implemented as SSA-Based Optimizations. As such, the only peephole optimizations we implemented separately (located in `\peephole\peephole.ml`) are

- Letting jumps fall-through
- Changing the order of conditional jumps

Our compiler's conversion to SSA requires that code is in basic block structure. This results in the following pattern often appearing in generated assembly code:

```
    ...
    goto label
    label:
    ...
```

When we identify this, we simply remove the `goto` instruction.

Changing the ordering of conditional jumps is also useful due to the way our `If` statements are translated. Because of the way our compiler translates boolean expressions, the following code pattern often appears in our generated assembly code:

```
...
cmp A, B
je true_label
jmp false_label
true_label:
...
```

In this case, it is possible to change the conditional jump by "inverting" the condition and reordering the instructions to produce the following code:

```
...
cmp A, B
jne false_label
true_label:
...
```

Letting jumps fallthrough is always useful since there is no benefit to keep a jump to the next line in assembly. However, while the reordering of the conditional jumps may be useful, its performance is highly dependent on the frequency of each jump. Assuming that this comparison is encountered frequently, if the condition evaluates to false more often, then reordering the basic blocks for the `false_label` to be where the fallthrough occurs can be more useful. Otherwise, the current implementation will produce more speedups.

For instance,

```
int main() {
    int i = 0;
    int b = 0;
    while (i < 10000000) {
        if (i == 100) {
            b++;
        } else {
            b += 0;
        }
        i++;
    }
    return b;
}
```

In the above code, since `i != 100` occurs more frequently than `i == 100`, if the comparisons were reordered for the false label to be the one falling through, there may be more benefit from letting the instruction fall through. Deciding on which branch to take (branch prediction) is a possible extension for our compiler; for L5, we simply use the heuristic that the true label is taken more often.

When we do all peephole optimizations, we actually get significant slowdowns in many test cases, as seen in Figure 21. Intuitively, test cases should not be significantly slower, since there are fewer instructions that are executed when the jumps are reordered. A possible explanation could be due to how jumps occur in assembly. According to [Int], there are differences between the types of jumps. There may be differences in performance when using jumps and conditional jumps to a label, and reordering the jumps may have worsened performance as the more inefficient jump is used more often.

3 Putting it all together...

When we activate all optimizations at the same time, we get speedups shown in Figure 23, as well as an average decrease in executable size (Figure 25). We note that certain test cases (jack unsafe, jen, mat unsafe, monica and ncik) show unsatisfactory speedup, despite seeing speedups in them when optimizations are tested individually. As such, we have to consider the effect of interactions between our optimizations. [1]

3.1 Should I coalesce?

Recall that register coalescing actually resulted in slowdowns in many of our test cases. When we turn on all optimizations except for register coalescing, we get the results in Figure 26, 27. We can tell from the graphs that register coalescing is now in fact necessary for speedups; the combined effect of our optimizations has creates opportunities for register coalescing to actually have a positive effect on the code.

[1]Somehow, no matter what we do, our numbers for mist and julia don't change on gradescope, even though our generated assembly looks good and we record speedups when testing locally.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[Hof24a] Jan Hoffmann. *Lecture Notes on Function Optimizations*. Mar. 2024.

[Hof24b] Jan Hoffmann. *Lecture Notes on Intermediate Representation*. Feb. 2024.

[Hof24c] Jan Hoffmann. *Lecture Notes on Optimizing Register Allocation*. Mar. 2024.

[Int] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals*.

[Tor11] Keith D. Cooper Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2011.

[War12] Henry Warren. *Hacker's Delight*. Addison-Wesley Professional, 2012.

A Cycle Counts and Executable Sizes

B Graphs

We use the following shorthand for labelling active optimization passes in the figures:

Optimization Pass	Shorthand
L4 Cleanup	4
Tail-Call Optimization	t
Tail-Call Optimization with Basic Accumulation	T
Function Inlining	i
Aggressive Deadcode Elimination	d
Simple Constant Propagation	c
Sparse Conditional Constant Propagation	Sc
Strength Reduction	s
Register Coalescing	r
Peephole Optimizations (all)	p
Peephole Optimizations (only fall-through)	P

Test case	gcc -O0	gcc -O1	none	4	4t	4i	4ti	4d	4dc
albert.l4	3.122148e+10	2.708830e+10	1.555929e+10	timeout	timeout	1.225886e+10	timeout	timeout	timeout
albert.l4 –unsafe	5.679428e+09	1.765403e+09	8.973391e+09	8.437995e+09	8.434832e+09	5.285815e+09	5.326507e+09	8.020612e+09	7.830348e+09
arrays_and_loops.l4	1.406911e+10	1.234260e+10	9.644835e+09	9.319724e+09	timeout	timeout	timeout	timeout	8.689372e+09
arrays_and_loops.l4 –unsafe	4.337953e+09	1.292763e+09	6.597395e+09	6.187519e+09	6.243729e+09	6.362293e+09	6.413551e+09	6.632489e+09	5.494401e+09
daisy.l4	3.333612e+10	3.211596e+10	4.458215e+09	4.280299e+09	4.354652e+09	4.302595e+09	4.314327e+09	4.305088e+09	4.317734e+09
daisy.l4 –unsafe	2.371756e+09	6.074478e+08	2.445188e+09	2.290122e+09	2.318349e+09	2.311169e+09	2.303621e+09	2.345076e+09	2.369431e+09
danny.l4	9.254590e+09	6.467804e+09	3.539589e+09	3.376518e+09	3.403819e+09	1.782560e+09	1.785711e+09	3.385738e+09	3.398481e+09
danny.l4 –unsafe	1.616402e+09	4.339315e+08	2.947079e+09	2.725637e+09	2.757491e+09	9.892093e+08	9.830302e+08	2.777456e+09	2.799256e+09
fannkuch.l4	1.171305e+11	1.151513e+11	timeout	timeout	timeout	timeout	timeout	timeout	timeout
fannkuch.l4 –unsafe	1.296214e+10	7.042870e+09	1.584330e+10	1.367026e+10	timeout	7.648881e+09	7.676947e+09	1.299143e+10	1.303448e+10
frank.l4	1.356448e+09	1.197720e+09	5.720691e+08	5.580510e+08	5.622670e+08	4.485585e+08	4.508758e+08	5.578417e+08	5.642931e+08
frank.l4 –unsafe	3.168332e+08	8.549986e+07	4.421353e+08	4.309262e+08	4.355625e+08	2.701593e+08	2.710471e+08	4.369262e+08	4.360535e+08
georgy.l4	3.155523e+09	2.911545e+09	1.129494e+09	1.127066e+09	1.129355e+09	7.857140e+08	7.945124e+08	1.125614e+09	1.130377e+09
georgy.l4 –unsafe	9.943405e+08	3.901752e+08	1.354100e+09	1.348090e+09	1.379184e+09	8.586936e+08	8.595699e+08	1.352561e+09	1.372618e+09
jack.l4	3.142895e+09	3.066881e+09	4.758038e+08	4.182068e+08	4.275075e+08	4.204850e+08	4.218708e+08	4.247020e+08	4.266766e+08
jack.l4 –unsafe	1.768423e+08	1.014782e+08	6.051195e+08	1.629097e+08	1.697784e+08	1.641727e+08	1.661779e+08	1.726714e+08	1.748806e+08
janos.l4	5.026915e+08	4.659817e+08	2.242401e+08	2.091353e+08	2.325499e+08	1.314936e+08	1.337610e+08	2.121784e+08	2.390843e+08
janos.l4 –unsafe	1.648442e+08	1.121873e+08	1.385583e+08	1.214985e+08	1.402870e+08	1.126254e+08	1.173667e+08	1.267134e+08	1.405112e+08
jen.l4	6.517745e+09	1.339421e+09	4.144002e+09	3.306009e+09	3.311192e+09	3.303604e+09	3.307461e+09	3.899393e+09	3.464135e+09
jen.l4 –unsafe	6.524487e+09	1.339934e+09	4.150467e+09	3.321546e+09	3.322742e+09	3.304155e+09	3.318026e+09	3.890881e+09	3.464975e+09
julia.l4	7.314819e+09	5.242276e+09	timeout	1.274579e+10	1.292916e+10	1.241352e+10	timeout	1.521919e+10	timeout
julia.l4 –unsafe	4.788097e+09	2.031016e+09	timeout	1.250516e+10	timeout	1.249023e+10	1.436064e+10	timeout	1.553700e+10
leonardo.l4	6.643761e+09	6.346855e+09	9.735365e+09	9.636180e+09	timeout	9.669463e+09	9.641083e+09	9.670278e+09	9.652023e+09
leonardo.l4 –unsafe	7.067363e+09	6.332045e+09	9.687426e+09	9.648433e+09	timeout	timeout	timeout	9.765632e+09	timeout
loooops.l4	1.899316e+10	1.640177e+10	1.550857e+10	1.301711e+10	timeout	timeout	timeout	1.499313e+10	timeout
loooops.l4 –unsafe	7.492324e+09	3.816876e+09	1.449863e+10	1.363237e+10	timeout	timeout	timeout	1.420299e+10	1.410839e+10
mat.l4	1.175714e+10	1.137805e+10	3.551660e+09	2.975916e+09	3.038606e+09	3.019971e+09	3.015592e+09	3.024349e+09	3.100600e+09
mat.l4 –unsafe	1.181949e+09	3.809198e+08	1.293642e+09	8.112257e+08	8.288424e+08	8.305651e+08	8.290373e+08	8.369711e+08	8.512473e+08
mist.l4	7.014198e+09	1.046598e+09	1.333605e+10	1.109063e+10	timeout	1.108021e+10	1.111667e+10	1.307455e+10	timeout
mist.l4 –unsafe	6.729085e+09	7.738156e+08	1.313028e+10	1.100422e+10	1.120328e+10	1.097331e+10	timeout	timeout	timeout
monica.l4	1.979763e+09	1.188159e+09	1.177542e+09	1.154546e+09	1.180818e+09	1.181819e+09	1.184302e+09	1.175179e+09	1.183460e+09
monica.l4 –unsafe	1.759948e+09	1.243902e+09	1.174744e+09	1.156362e+09	1.187701e+09	1.173963e+09	1.173615e+09	1.173535e+09	1.175937e+09
ncik.l4	1.604166e+10	1.522487e+10	6.178446e+09	5.746811e+09	5.811934e+09	5.844880e+09	5.810503e+09	5.671773e+09	5.379431e+09
ncik.l4 –unsafe	2.946298e+09	1.118591e+09	3.372369e+09	3.112697e+09	3.210482e+09	3.101118e+09	3.088282e+09	3.344969e+09	3.260050e+09
ronald.l4	4.628220e+09	4.405764e+09	3.329196e+09	3.251788e+09	3.307597e+09	2.697738e+09	2.701838e+09	3.322239e+09	3.320785e+09
ronald.l4 –unsafe	8.464510e+08	4.335986e+08	1.632215e+09	1.503094e+09	1.533270e+09	1.465585e+09	1.456322e+09	2.179814e+09	1.545415e+09
yyb.l4	3.425380e+10	2.368886e+10	9.751809e+09	9.490457e+09	timeout	2.938434e+09	2.959083e+09	timeout	9.595713e+09
yyb.l4 –unsafe	8.426217e+09	5.570512e+08	8.402379e+09	8.010305e+09	timeout	timeout	timeout	8.112323e+09	timeout

Table 1: Recorded runtimes on our machine

4dcs	dSc	dc	4 no mcs	4r	4p	4Tidcspr	4Tidcsp
timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
7.830889e+09	8.326728e+09	8.054686e+09	8.204236e+09	8.473987e+09	8422521436.0	4.464691e+09	4564730606.0
8.784857e+09	8.641645e+09	8.786189e+09	timeout	timeout	9588967754.0	8.727097e+09	8672802014.0
5.501593e+09	5.619199e+09	5.579844e+09	6.799246e+09	6.234389e+09	6317353909.0	5.635726e+09	5543075898.0
4.563167e+09	4.102713e+09	4.172202e+09	4.479797e+09	4.227378e+09	4264303501.0	3.846695e+09	3533960157.0
1.974547e+09	2.414776e+09	2.397007e+09	2.346294e+09	2.255994e+09	2277211933.0	1.784351e+09	1796416068.0
3.381191e+09	3.327751e+09	3.377839e+09	3.448836e+09	3.271724e+09	3360804064.0	1.414428e+09	1682474073.0
2.799487e+09	2.750599e+09	2.759992e+09	2.726185e+09	2.753318e+09	2766838511.0	9.390722e+08	1004676766.0
timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
timeout	1.518973e+10	1.777957e+10	timeout	timeout	timeout	6.591355e+09	7066412862.0
6.570150e+08	5.706002e+08	5.740842e+08	5.837351e+08	5.753258e+08	570813818.0	4.615065e+08	463191547.0
4.198299e+08	4.416665e+08	4.429554e+08	4.344454e+08	4.392513e+08	435392114.0	2.436862e+08	248129776.0
1.124723e+09	1.655451e+09	1.124115e+09	1.125701e+09	1.120182e+09	1121630911.0	6.325140e+08	664017427.0
1.355395e+09	1.074201e+09	1.349461e+09	1.358711e+09	1.354164e+09	1352700625.0	5.170323e+08	537186099.0
4.305229e+08	4.395281e+08	4.590083e+08	4.428509e+08	4.226339e+08	421344609.0	3.786730e+08	401552760.0
1.760496e+08	5.862125e+08	5.808875e+08	1.696822e+08	1.675133e+08	165258588.0	1.669035e+08	169796245.0
2.121041e+08	2.070391e+08	2.081592e+08	2.164148e+08	2.161704e+08	210894588.0	1.150725e+08	123251000.0
1.319022e+08	1.313177e+08	1.326997e+08	1.316471e+08	1.297257e+08	124799197.0	1.123610e+08	115604437.0
3.502330e+09	3.648022e+09	3.460417e+09	3.281908e+09	3.471517e+09	3524460762.0	3.339580e+09	3465431600.0
3.502279e+09	3.635978e+09	3.454095e+09	3.280220e+09	3.503029e+09	3516846371.0	3.323918e+09	3507227329.0
timeout	1.548043e+10	1.546543e+10	1.287835e+10	timeout	timeout	timeout	timeout
1.009677e+10	1.548043e+10	1.550258e+10	timeout	1.277567e+10	timeout	9.700285e+09	timeout
9.371674e+09	9.694936e+09	9.687230e+09	9.715571e+09	timeout	timeout	9.228908e+09	9208673811.0
9.232541e+09	9.744589e+09	9.705831e+09	9.723473e+09	9.822931e+09	timeout	9.099558e+09	9210955337.0
timeout	1.467474e+10	1.459952e+10	timeout	timeout	timeout	timeout	timeout
1.406724e+10	1.401866e+10	1.399454e+10	timeout	timeout	timeout	1.363859e+10	timeout
3.052661e+09	3.387084e+09	3.559918e+09	2.974970e+09	2.993097e+09	2893283272.0	2.725712e+09	4047539970.0
8.425061e+08	1.236984e+09	1.235165e+09	8.383573e+08	8.297518e+08	810629443.0	8.498739e+08	842705370.0
timeout	1.297468e+10	1.361708e+10	1.125273e+10	1.125722e+10	timeout	timeout	timeout
3.327773e+09	1.270980e+10	timeout	timeout	1.116321e+10	timeout	3.039226e+09	3146349105.0
1.177269e+09	1.175482e+09	1.172418e+09	1.176529e+09	1.180164e+09	1181034370.0	1.207401e+09	1175488823.0
1.175405e+09	1.184821e+09	1.209755e+09	1.175574e+09	1.179468e+09	1171389547.0	1.195015e+09	1165434613.0
5.387217e+09	5.548476e+09	5.674791e+09	6.008809e+09	5.755509e+09	5715740414.0	5.438869e+09	5746516095.0
3.239644e+09	3.520840e+09	3.522197e+09	3.234383e+09	3.138672e+09	3166793756.0	3.169743e+09	3164506543.0
3.282617e+09	3.334285e+09	3.354083e+09	3.411933e+09	3.310809e+09	3259109220.0	2.491872e+09	2627906089.0
1.531995e+09	1.632516e+09	1.622388e+09	1.603437e+09	1.572946e+09	1513794542.0	8.516865e+08	894465963.0
timeout	9.590732e+09	1.192122e+10	9.571107e+09	timeout	timeout	2.232145e+09	2342646348.0
timeout	8.224332e+09	8.216495e+09	timeout	timeout	timeout	8.143198e+09	timeout

Table 2: Recorded runtimes on our machine (continued)

Test case	gcc -O0	none	4	4t	4i	4ti	4d	4dc	4dcs
albert.l4	3.122148e+10	timeout	timeout	timeout	24008.0	timeout	timeout	timeout	timeout
albert.l4 –unsafe	5.679428e+09	13768.0	11208.0	timeout	12536.0	12536.0	11720.0	11336.0	11368.0
arrays_and_loops.l4	1.406911e+10	timeout	8584.0	timeout	timeout	timeout	timeout	8328.0	8312.0
arrays_and_loops.l4 –unsafe	4.337953e+09	7800.0	7576.0	timeout	7640.0	7640.0	7640.0	7448.0	7432.0
daisy.l4	3.333612e+10	53400.0	50664.0	timeout	55944.0	55944.0	50840.0	49256.0	49736.0
daisy.l4 –unsafe	2.371756e+09	22104.0	19352.0	timeout	22280.0	22280.0	19496.0	19432.0	19288.0
danny.l4	9.254590e+09	17912.0	17832.0	timeout	20888.0	20888.0	17960.0	17576.0	17579.0
danny.l4 –unsafe	1.616402e+09	10808.0	10728.0	timeout	11432.0	11432.0	10984.0	10888.0	10888.0
fannkuch.l4	1.171305e+11	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
fannkuch.l4 –unsafe	1.296214e+10	8456.0	7912.0	timeout	7976.0	7976.0	7896.0	7816.0	timeout
frank.l4	1.356448e+09	15808.0	15408.0	timeout	20016.0	20016.0	15472.0	15136.0	15952.0
frank.l4 –unsafe	3.168332e+08	11360.0	10944.0	timeout	12016.0	12016.0	10912.0	10896.0	10944.0
georgy.l4	3.155523e+09	22416.0	21984.0	timeout	25792.0	25792.0	21872.0	21440.0	21584.0
georgy.l4 –unsafe	9.943405e+08	13888.0	13440.0	timeout	17616.0	17616.0	13360.0	13312.0	13312.0
jack.l4	3.142895e+09	24056.0	22600.0	timeout	24808.0	24808.0	22744.0	22200.0	22232.0
jack.l4 –unsafe	1.768423e+08	12760.0	11272.0	timeout	12680.0	12680.0	11368.0	11336.0	11336.0
janos.l4	5.026915e+08	16800.0	16272.0	timeout	18144.0	18144.0	16288.0	15824.0	15840.0
janos.l4 –unsafe	1.648442e+08	10912.0	10368.0	timeout	11376.0	11376.0	10384.0	10288.0	10240.0
jen.l4	6.517745e+09	43448.0	43448.0	timeout	43464.0	43464.0	37784.0	23816.0	23816.0
jen.l4 –unsafe	6.524487e+09	43288.0	43288.0	timeout	43304.0	43304.0	37608.0	23656.0	23656.0
julia.l4	7.314819e+09	13192.0	12984.0	timeout	13720.0	timeout	12968.0	timeout	timeout
julia.l4 –unsafe	4.788097e+09	9656.0	9432.0	timeout	10488.0	10440.0	timeout	9384.0	9368.0
leonardo.l4	6.643761e+09	9288.0	9048.0	timeout	9224.0	9224.0	9064.0	8968.0	9080.0
leonardo.l4 –unsafe	7.067363e+09	7736.0	7496.0	timeout	timeout	timeout	7496.0	timeout	7512.0
loooops.l4	1.899316e+10	timeout	9224.0	timeout	timeout	timeout	9288.0	timeout	timeout
loooops.l4 –unsafe	7.492324e+09	timeout	8264.0	timeout	timeout	timeout	8312.0	8136.0	8120.0
mat.l4	1.175714e+10	12264.0	11640.0	timeout	12600.0	12600.0	11640.0	11496.0	11560.0
mat.l4 –unsafe	1.181949e+09	8792.0	8168.0	timeout	8568.0	8568.0	8184.0	8136.0	8152.0
mist.l4	7.014198e+09	timeout	8936.0	timeout	10008.0	10008.0	8952.0	timeout	timeout
mist.l4 –unsafe	6.729085e+09	8072.0	7784.0	timeout	8808.0	timeout	timeout	timeout	7784.0
monica.l4	1.979763e+09	9544.0	9176.0	timeout	9624.0	9624.0	9176.0	9112.0	9128.0
monica.l4 –unsafe	1.759948e+09	7672.0	7304.0	timeout	7480.0	7480.0	7304.0	7304.0	7288.0
ncik.l4	1.604166e+10	20456.0	19352.0	timeout	20040.0	20040.0	19432.0	18616.0	18648.0
ncik.l4 –unsafe	2.946298e+09	12296.0	11144.0	timeout	11656.0	11656.0	11240.0	11160.0	11160.0
ronald.l4	4.628220e+09	22816.0	20688.0	timeout	21984.0	21984.0	20736.0	19456.0	19632.0
ronald.l4 –unsafe	8.464510e+08	12416.0	10272.0	timeout	timeout	10736.0	10400.0	10304.0	10320.0
yyb.l4	3.425380e+10	19472.0	19104.0	timeout	timeout	22336.0	timeout	18512.0	timeout
yyb.l4 –unsafe	8.426217e+09	14224.0	13840.0	timeout	timeout	timeout	13904.0	timeout	timeout

Table 3: Recorded executable sizes on our machine

dSc	dc	4 no mcs	4r	4p	4Tidcspr	4Tidcsp
timeout	timeout	timeout	timeout	timeout	timeout	timeout
12952.0	13896.0	11576.0	11144.0	11160.0	11992.0	11928.0
8424.0	8536.0	timeout	timeout	timeout	8216.0	8344.0
7592.0	7672.0	7528.0	7560.0	7528.0	7368.0	7416.0
49576.0	51976.0	51496.0	49144.0	48408.0	47400.0	51544.0
21736.0	22184.0	19576.0	18760.0	19000.0	20008.0	21208.0
16536.0	17720.0	17912.0	17592.0	17336.0	17528.0	19416.0
10616.0	10968.0	10728.0	10568.0	10568.0	10216.0	11016.0
timeout	timeout	timeout	timeout	timeout	timeout	timeout
8264.0	8376.0	timeout	timeout	7832.0	7672.0	7816.0
15184.0	15536.0	15504.0	15248.0	14944.0	18208.0	19168.0
11200.0	11296.0	10960.0	10848.0	10736.0	11088.0	11424.0
21184.0	21888.0	22032.0	21584.0	21232.0	22048.0	23616.0
13584.0	13760.0	13472.0	13184.0	13152.0	15488.0	16416.0
22728.0	23672.0	22776.0	22152.0	21672.0	21944.0	23320.0
12552.0	12824.0	11352.0	11016.0	11128.0	12008.0	12488.0
15792.0	16352.0	16336.0	16000.0	15808.0	15568.0	16544.0
10624.0	10832.0	10416.0	10192.0	10240.0	10272.0	10832.0
24200.0	23816.0	43528.0	43416.0	41336.0	21688.0	21800.0
24056.0	23656.0	43368.0	43272.0	41176.0	21560.0	21656.0
12696.0	12968.0	12968.0	timeout	timeout	timeout	timeout
timeout	9592.0	timeout	9352.0	timeout	9960.0	timeout
9048.0	9208.0	9064.0	timeout	timeout	8536.0	8776.0
7672.0	7752.0	7496.0	7464.0	timeout	7352.0	7448.0
8952.0	9048.0	timeout	timeout	timeout	timeout	timeout
8104.0	8136.0	timeout	timeout	timeout	7992.0	timeout
11848.0	12104.0	11672.0	11480.0	11320.0	11656.0	12072.0
8680.0	8776.0	8184.0	8056.0	8104.0	8280.0	8424.0
9048.0	9144.0	8952.0	8904.0	timeout	timeout	timeout
8008.0	timeout	timeout	7768.0	timeout	8440.0	timeout
9320.0	9480.0	9176.0	9096.0	9000.0	8888.0	9128.0
7608.0	7672.0	7304.0	7256.0	7272.0	7208.0	timeout
18792.0	19720.0	19512.0	18952.0	18648.0	17320.0	timeout
11816.0	12312.0	11144.0	10952.0	11032.0	10936.0	timeout
20192.0	21600.0	20816.0	20224.0	19760.0	18720.0	19600.0
11584.0	12432.0	10464.0	10224.0	10240.0	10608.0	10560.0
18464.0	18880.0	19264.0	timeout	timeout	16288.0	timeout
13808.0	13920.0	timeout	timeout	13552.0	11376.0	timeout

Table 4: Recorded executable sizes on our machine (ctd)

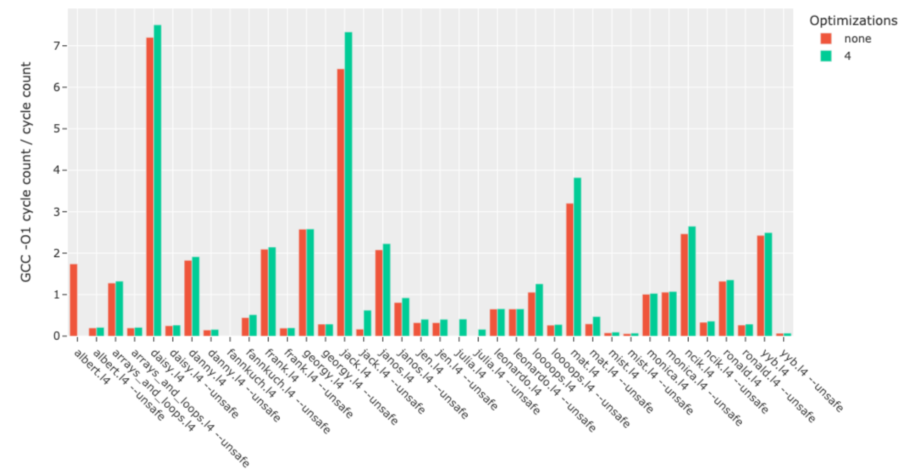

Figure 3: Improvement over GCC -O1 cycle count for L4 Cleanup versus baseline. Test programs that timed out are not included.

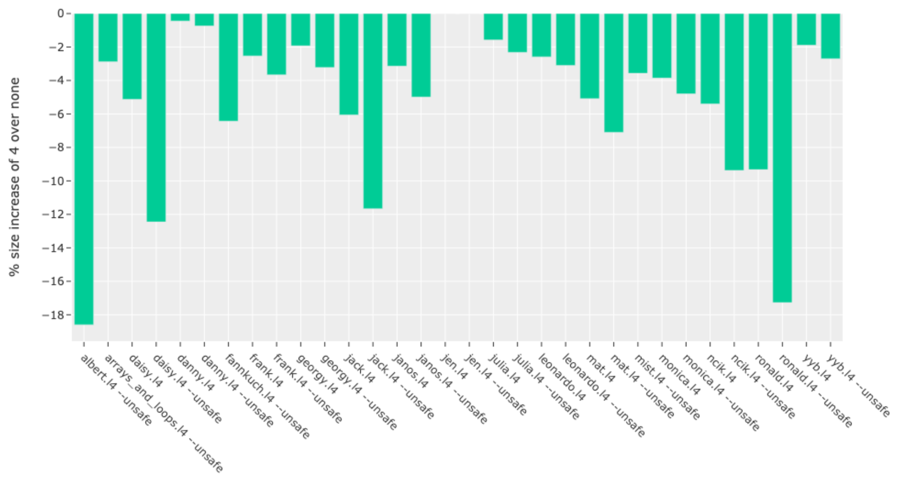

Figure 4: Executable size increase of L4 Cleanup over baseline. Test programs that timed out are not included.

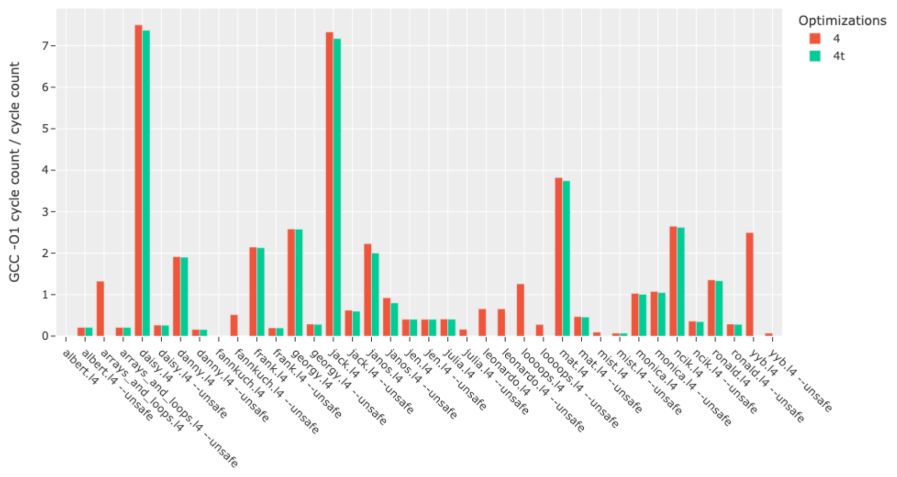

Figure 5: Improvement over GCC -O1 cycle count for L4 Cleanup + Basic TCO versus L4 Cleanup. Test programs that timed out are not included.

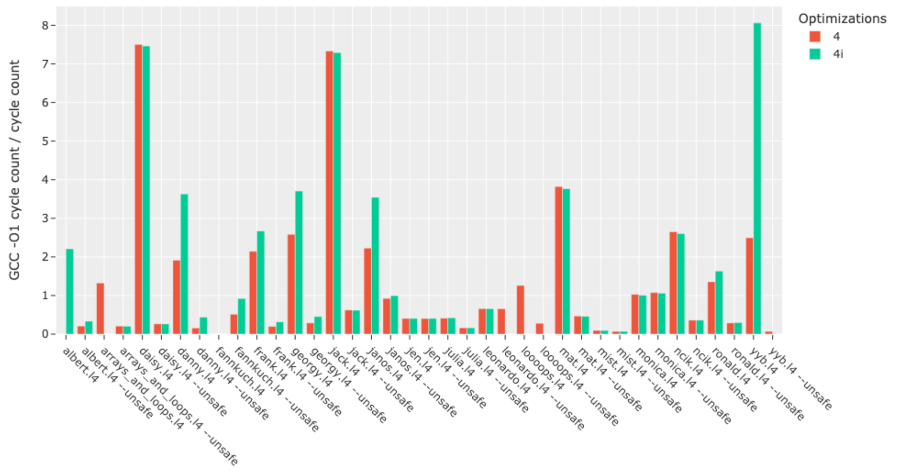

Figure 6: Improvement over GCC -O1 cycle count for L4 Cleanup + Function Inlining versus L4 Cleanup. Test programs that timed out are not included.

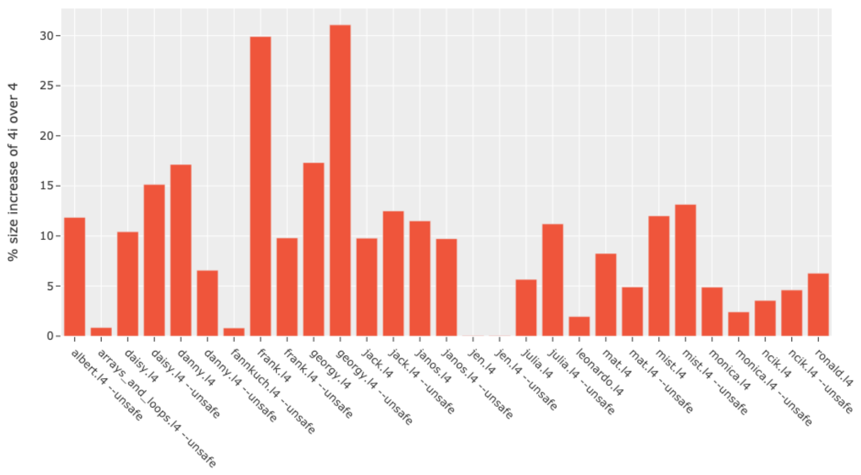
Figure 7: Executable size increase of L4 Cleanup + Inlining over L4 Cleanup. Test programs that timed out are not included.

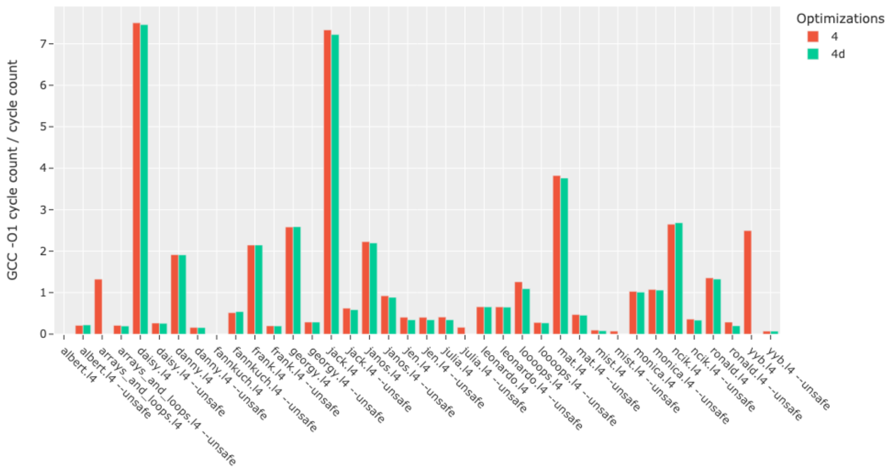
Figure 8: Improvement over GCC -O1 cycle count for L4 Cleanup + ADCE versus L4 Cleanup. Test programs that timed out are not included.

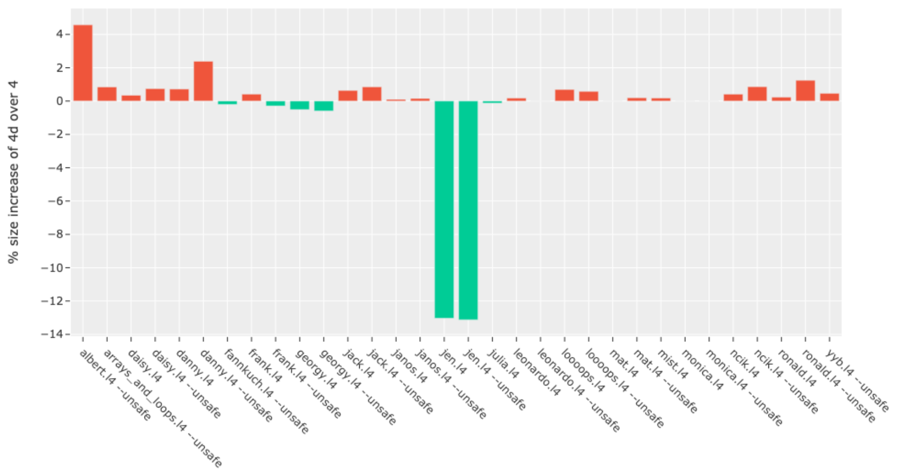

Figure 9: Executable size increase for L4 Cleanup + ADCE over L4 Cleanup. Test programs that timed out are not included.

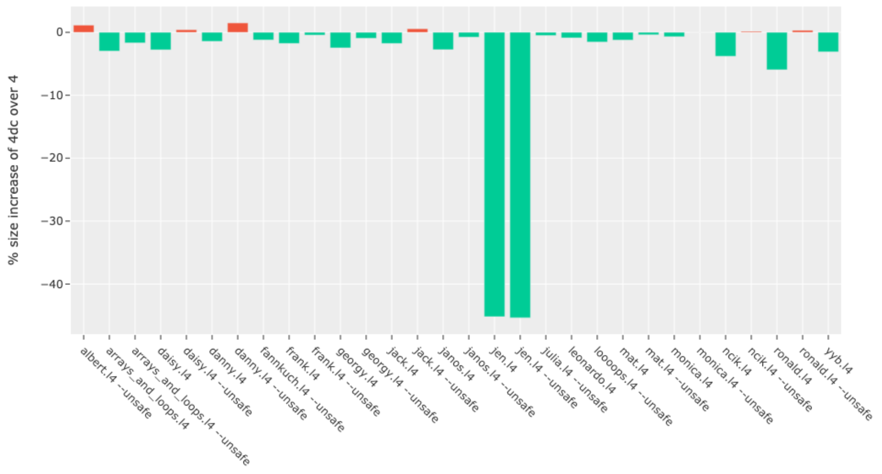

Figure 10: Executable size increase for L4 Cleanup + ADCE + SCP over L4 Cleanup. Test programs that timed out are not included.

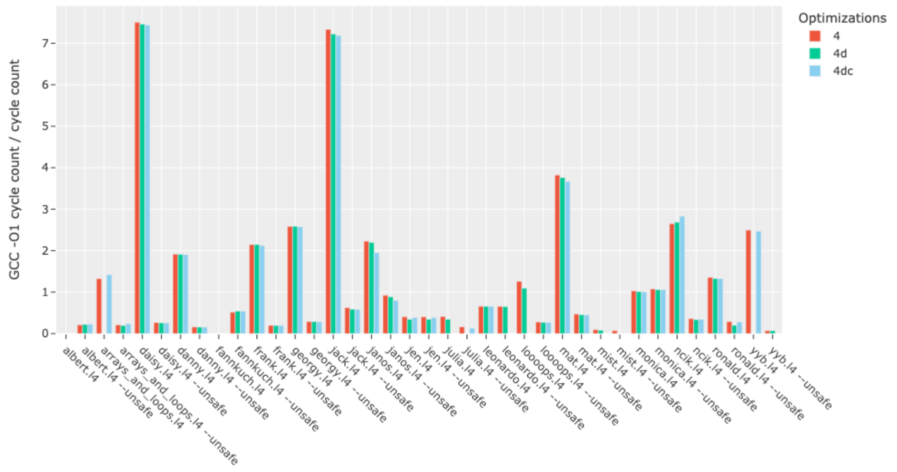

Figure 11: Improvement over GCC -O1 cycle count for L4 Cleanup versus L4 Cleanup + ADCE versus L4 Cleanup + ADCE + SCP. Test programs that timed out are not included.

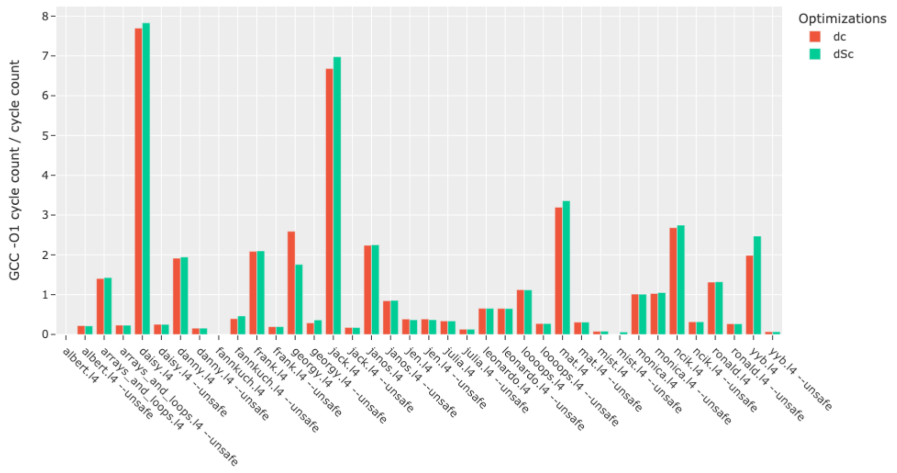

Figure 12: Improvement over GCC -O1 cycle count for ADCE + SCP versus ADCE + SCCP. Test programs that timed out are not included.

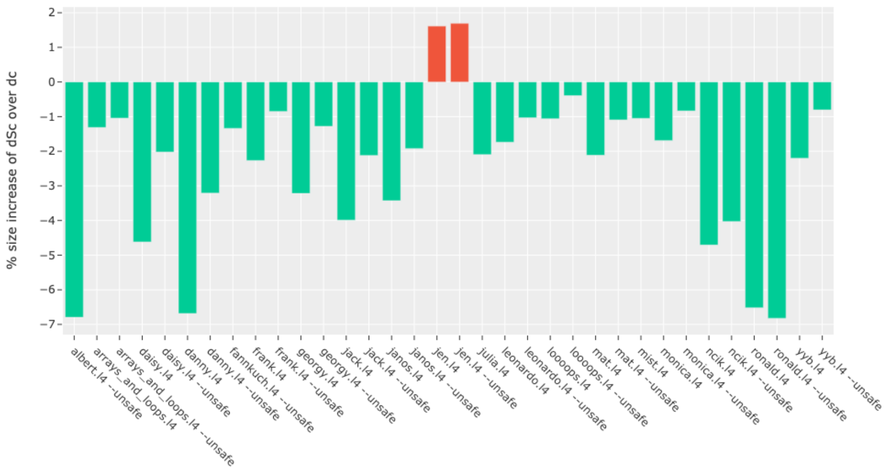

Figure 13: Executable size increase of ADCE + SCCP over ADCE + SCP. Test programs that timed out are not included.

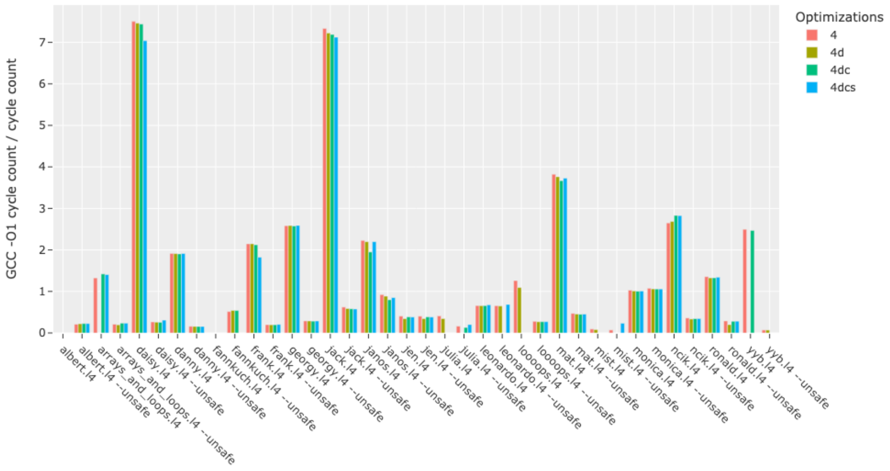

Figure 14: Improvement over GCC -O1 cycle count for L4 Cleanup versus L4 Cleanup + ADCE versus L4 Cleanup + ADCE + SCP versus L4 Cleanup + ADCE + SCP + SR. Test programs that timed out are not included.

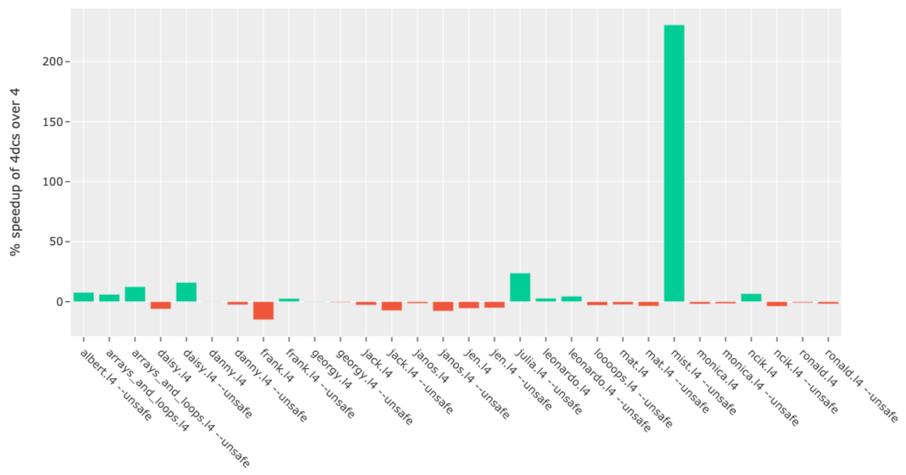

Figure 15: Speedup for L4 Cleanup + ADCE + SCP + SR over L4 Cleanup. Test programs that timed out are not included.

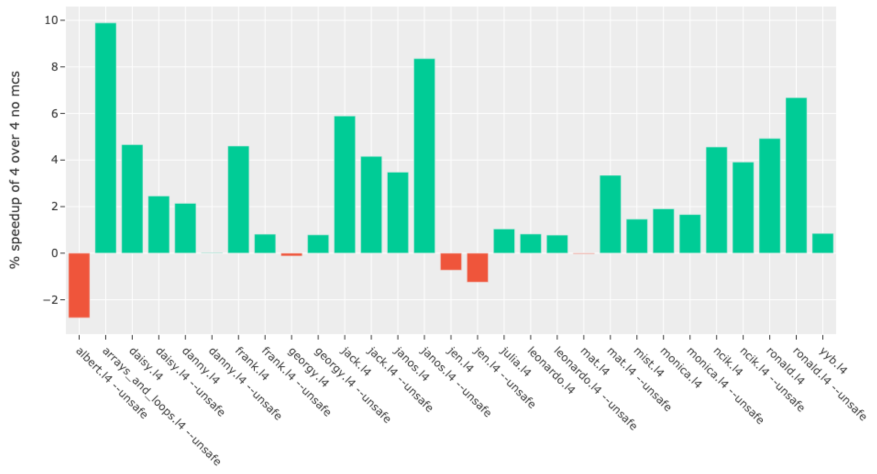

Figure 16: Speedup for L4 Cleanup over L4 Cleanup without MCS. Test programs that timed out are not included.

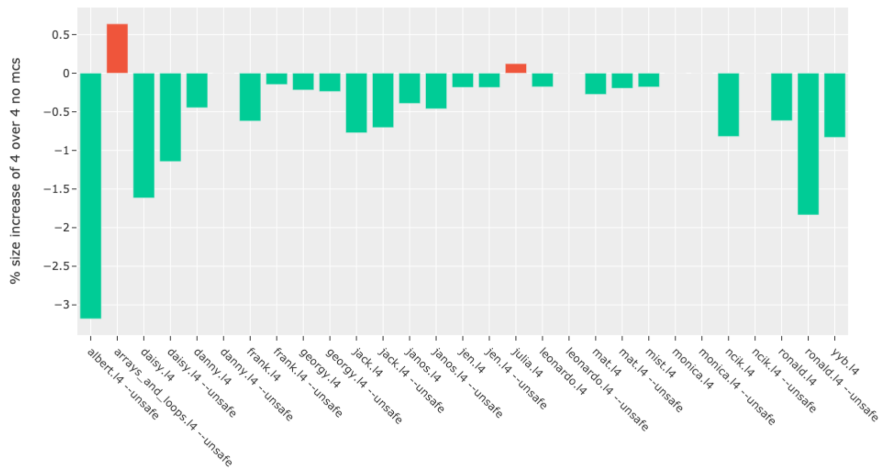

Figure 17: Increase in Executable Size for L4 Cleanup over L4 Cleanup without MCS. Test programs that timed out are not included.

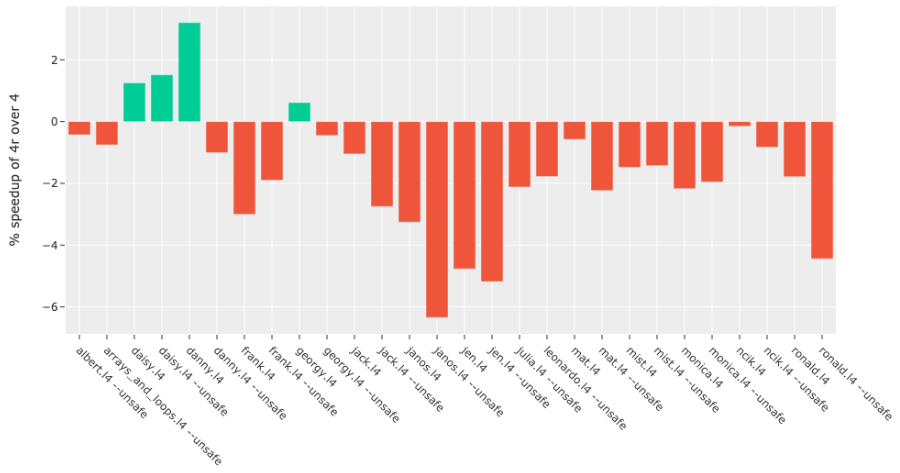

Figure 18: Speedup for L4 Cleanup + Register Coalescing over L4 Cleanup. Test programs that timed out are not included.

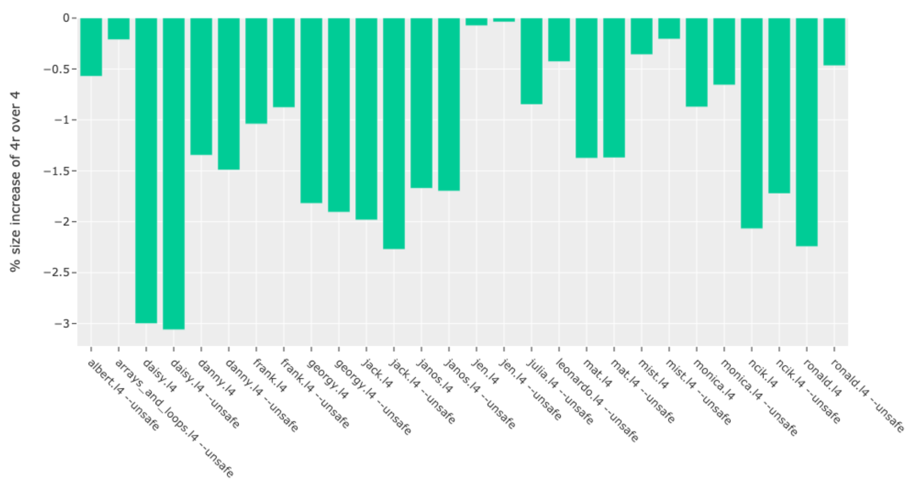

Figure 19: Increase in Executable Size for L4 Cleanup + Register Coalescing over L4 Cleanup. Test programs that timed out are not included.

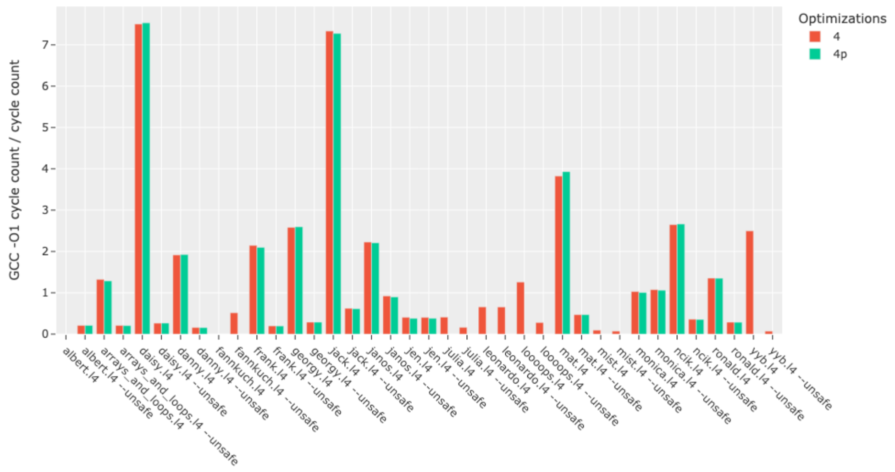

Figure 20: Improvement over GCC -O1 cycle count for L4 Cleanup + Peephole versus L4 Cleanup. Test programs that timed out are not included.

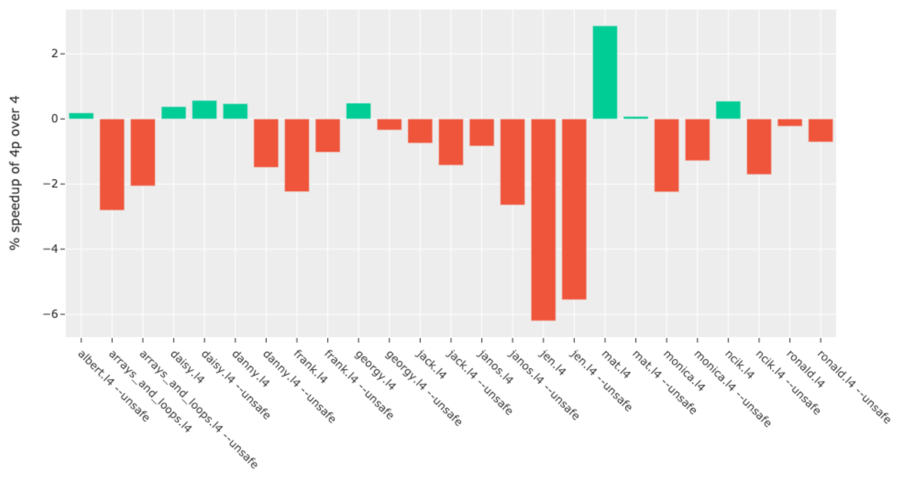

Figure 21: Speedup for L4 Cleanup + Peephole over L4 cleanup. Test programs that timed out are not included.

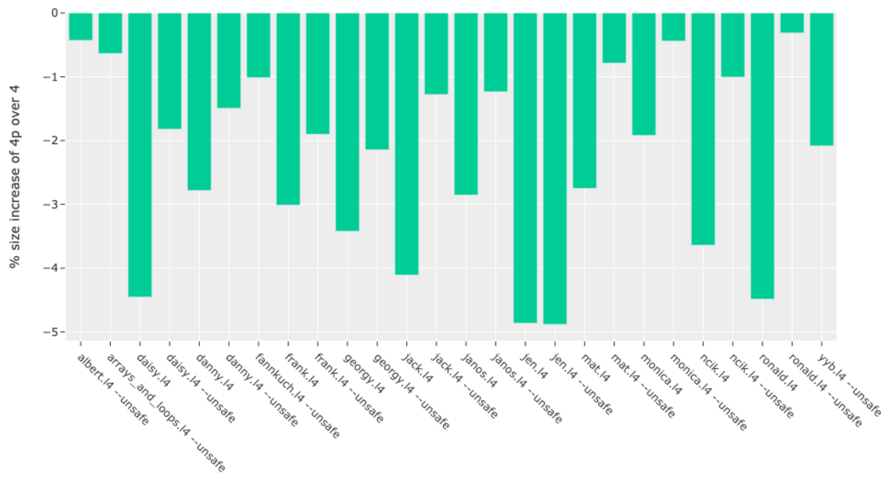

Figure 22: Increase in Executable Size for L4 Cleanup + Peephole over L4 Cleanup. Test programs that timed out are not included.

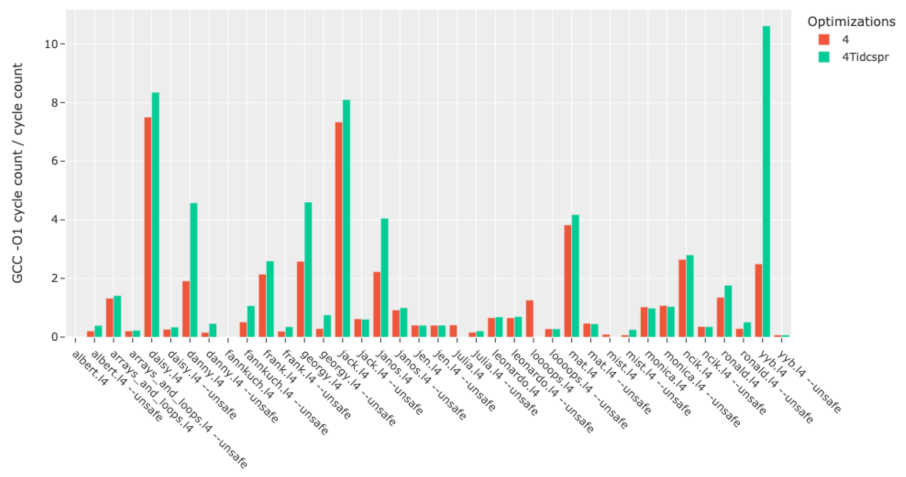

Figure 23: Improvement over GCC -O1 cycle count with all optimizations active versus L4 Cleanup. Test programs that timed out are not included.

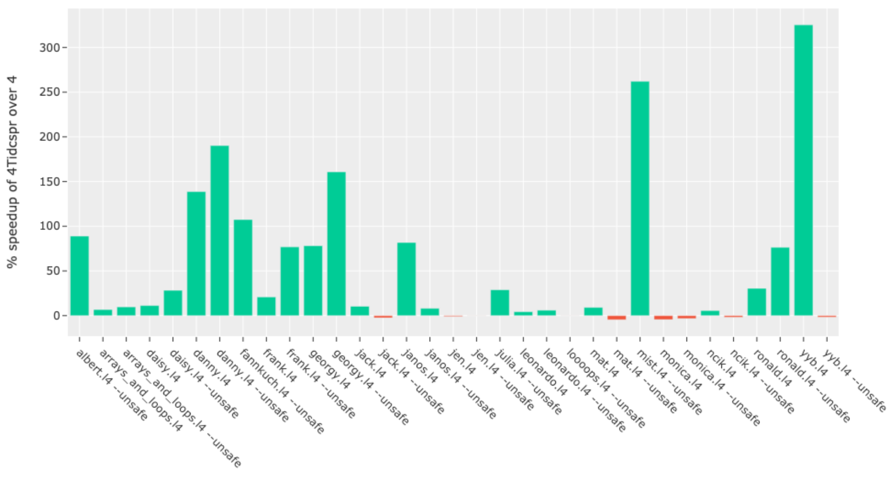

Figure 24: Speedup for all optimizations active over L4 cleanup. Test programs that timed out are not included.

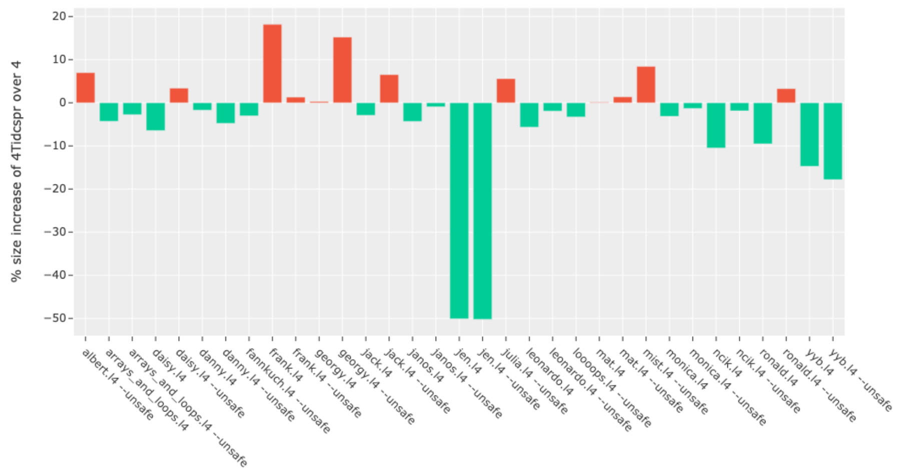

Figure 25: Increase in Executable Size for all optimizations active over L4 Cleanup without MCS. Test programs that timed out are not included.

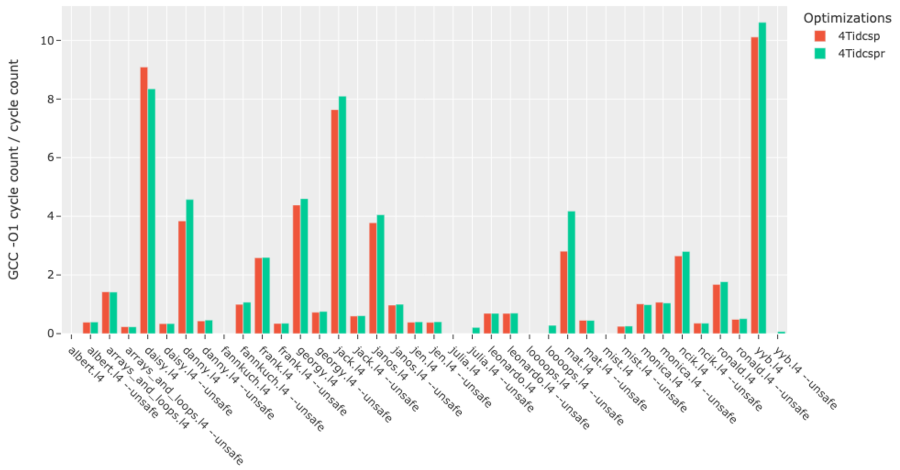

Figure 26: Improvement over GCC -O1 cycle count with all optimizations active versus leaving out only register coalescing. Test programs that timed out are not included.

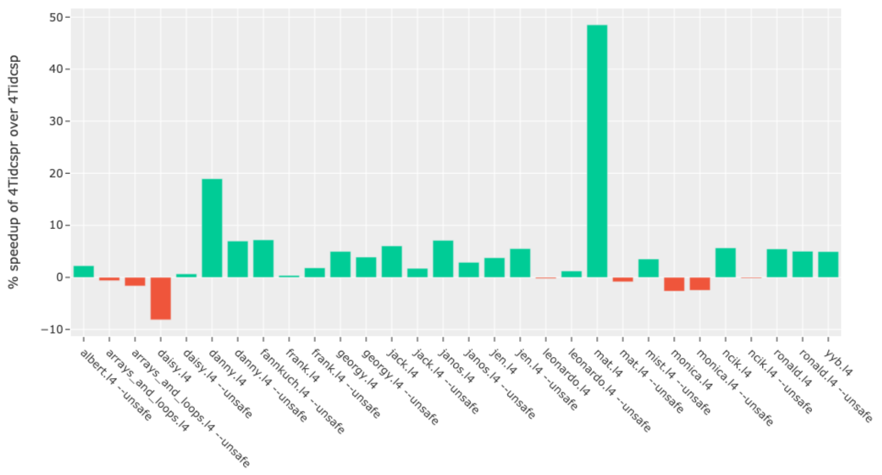

Figure 27: Speedup for all optimizations active over leaving out only register coalescing. Test programs that timed out are not included.

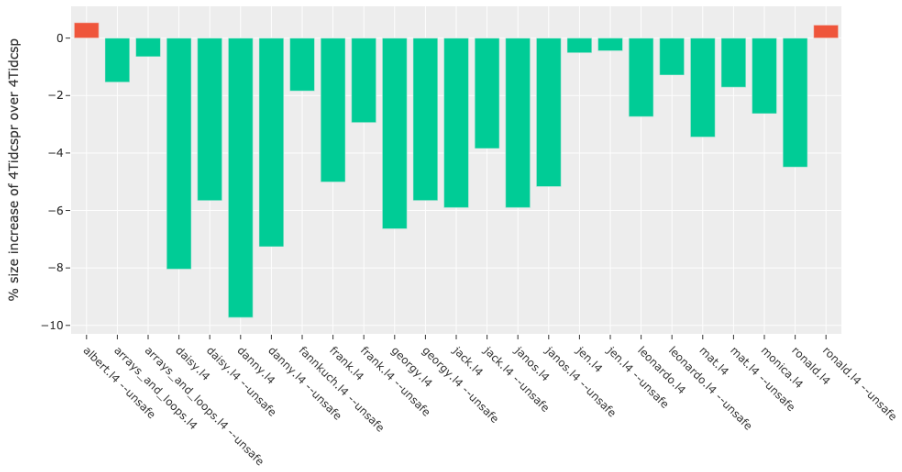

Figure 28: Increase in Executable Size for all optimizations active over leaving out only register coalescing. Test programs that timed out are not included.