lac.dcc.ufmg.br

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO
UFMG

New Optimization Sequences for Code-Size Reduction in LLVM

Fernando Magno Quintão Pereira, UFMG, Brazil - `fernando@dcc.ufmg.br`

Anderson Faustino da Silva, UEM, Brazil - `anderson@din.uem.br`

```
@techreport{Pereira20,
  title = {New Optimization Sequences for Code-Size Reduction
    in LLVM},
  author = {Fernando Magno Quint\~{a}o Pereira and
    Anderson Faustino da Silva},
  year = {2020},
  institution = {Universidade Federal de Minas Gerais},
  number = {02-2020}
}
```

New Optimization Sequences for Code-Size Reduction in LLVM

Fernando Magno Quintão Pereira and Anderson Faustino da Silva

November 14, 2020

Abstract

This report describes 22 new optimization sequences for LLVM. They contain between 5 to 19 optimizations, with an average of 11 optimizations. Given any C program, at least one of them is likely to outperform (or tie with) `opt -Os` and `opt -Oz`. The report describes the methodology to find these sequences, and points out links to scripts and repositories that can be used to reproduce the experiments.

1 Methodology

1.1 Benchmarks

Experiments in this report use benchmarks from the ANGHA. This is a collection of One Million compilable C programs [1][1] mined from open-source repositories. To find candidate sequences (see Section 1.2) we have selected 1,500 representative programs from this collection, using the following methodology:

1. Filter the 15K largest programs from the ANGHA collection. Size is measured as the number of LLVM instructions produced at the `-O0` optimization level.
2. Cluster these programs into 1,500 groups, using K-Means [3]. Clustering is applied on the feature vectors proposed by Namolaru *et al*[2]. These vectors consist of 56 program characteristics, such as number of CFG edges and number of basic blocks, for instance.
3. For each cluster, choose the program classified as it centroid to be part of the final benchmark suite.

[1]Available at `http://cuda.dcc.ufmg.br/angha`

1.2 Candidate Sequences

We shall derive a small set of optimization sequences from a larger collection of *candidate sequences*. To find these promising sequences of optimization, we proceed according to the following methodology:

1. For each one of the 1,500 representative programs, use GenS[2] to find 5,000 candidate sequences for these programs.
2. Select the best candidate sequence for each program.
3. Simplify the winning sequence for each program using a sequence reduction procedure [4]. Given an optimization sequence S and a program P, this algorithm removes elements from S while the resulting sequence yields the same benefits as S.
4. Out of this universe of 1,500 programs, we select the winning sequence for each of them. Call this set $S_{candidate}$.
5. Try every sequence in $S_{candidate}$ on every program, and collect the new winners. Call this new set S_{win}. we have that $|S_{win}| = 1,290$.

1.3 Covering Set

A *covering set* is a subset of the candidate sequences that beats the default optimization levels of LLVM (`clang -Os`, `clang -Oz`, `opt -Os` and `opt -Oz`) for every program in a given baseline. To find a covering set, we consider the candidate sequences from Section 1.2, the benchmarks from Section 1.1 and, as baseline, `clang -Oz`. We find a covering set by modifying a heuristic due to Purini and Jain [4]. We call our adaptation of their heuristic "Best-K". It works as follows:

1. Create a $1,500 \times 1,290$ matrix M_{opt}, with one row for each benchmark, and one column for each candidate sequence. Each cell $M_{opt}[i, j]$ contains the number of LLVM instructions that the i^{th} sequence yields for the j^{th} benchmark.
2. For every number K in the interval $[5 \ldots 100]$ do:
   (a) use Best-K to find a set with K sequences that covers (i.e., beats `clang -Oz` for) most of the programs.

[2]GenS is a genetic algorithm implemented on `pygmo` (available at `https://esa.github.io/pygmo2`).

(b) If we cannot increase the number of programs in which the cover set wins against the baseline when going from i to $i + 1$, then stop.

This methodology converged for $K = 22$. These 22 Sequences are given in Figure 1. These sequences could not defeat `clang -Oz` for 22 programs in the benchmark collection.

S00	**38**	-sroa -instcombine -early-cse-memssa -simplifycfg -instcombine -licm -indvars -simplifycfg -early-cse-memssa
S01	**47**	-jump-threading -loop-rotate -mem2reg -licm -indvars -slp-vectorizer -gvn -instcombine -early-cse-memssa -simplifycfg
S02	**67**	-jump-threading -mem2reg -early-cse-memssa -simplifycfg -instcombine -gvn -loop-rotate -jump-threading -licm -loop-idiom -simplifycfg -early-cse-memssa -loop-unroll -instcombine -ipsccp -simplifycfg
S03	**41**	-gvn -loop-rotate -mem2reg -slp-vectorizer -instcombine -indvars -early-cse-memssa
S04	**40**	-early-cse-memssa -tailcallelim -gvn -sroa -aggressive-instcombine -jump-threading -simplifycfg -instsimplify -early-cse-memssa
S05	**41**	-sroa -early-cse-memssa -licm -inferattrs -simplifycfg -jump-threading -indvars -memcpyopt -early-cse-memssa -simplifycfg -aggressive-instcombine
S06	**71**	-sroa -loop-unroll -simplifycfg -correlated-propagation -indvars -correlated-propagation -simplifycfg -instcombine -gvn -simplifycfg -instcombine -early-cse-memssa -jump-threading -slp-vectorizer -gvn -bdce -simplifycfg
S07	**80**	-mem2reg -tailcallelim -loop-rotate -slp-vectorizer -instcombine -slp-vectorizer -gvn -indvars -reassociate -mldst-motion -simplifycfg -jump-threading -correlated-propagation -instcombine -mldst-motion -speculative-execution -licm -gvn -simplifycfg
S08	**35**	-mem2reg -ipsccp -jump-threading -licm -instcombine -gvn -jump-threading
S09	**34**	-gvn -gvn -simplifycfg -instcombine -jump-threading
S10	**69**	-mem2reg -loop-rotate -gvn -instcombine -indvars -simplifycfg -early-cse-memssa -jump-threading -gvn -loop-unswitch -ipsccp -instcombine -simplifycfg
S11	**51**	-instcombine -sroa -simplifycfg -indvars -early-cse-memssa -slp-vectorizer -mldst-motion -instcombine -licm -loop-rotate -correlated-propagation -early-cse-memssa -simplifycfg
S12	**31**	-early-cse-memssa -instcombine -mem2reg -slp-vectorizer -instcombine -sroa -simplifycfg -early-cse-memssa
S13	**58**	-gvn -instcombine -simplifycfg -speculative-execution -loop-rotate -instcombine -jump-threading -inferattrs -gvn -correlated-propagation -instsimplify -simplifycfg
S14	**53**	-sroa -loop-rotate -reassociate -licm -instcombine -indvars -early-cse-memssa -mldst-motion -correlated-propagation -jump-threading -simplifycfg -instcombine -early-cse-memssa
S15	**71**	-licm -loop-rotate -bdce -simplifycfg -gvn -instcombine -simplifycfg -indvars -licm -speculative-execution -simplifycfg -mem2reg -adce -gvn -instcombine -speculative-execution -sroa -simplifycfg
S16	**31**	-instcombine -early-cse-memssa -indvars -sroa -licm -early-cse-memssa -simplifycfg
S17	**32**	-mem2reg -mldst-motion -simplifycfg -instcombine -early-cse-memssa -ipsccp -simplifycfg -instcombine
S18	**58**	-early-cse-memssa -sroa -correlated-propagation -early-cse-memssa -instcombine -reassociate -licm -early-cse-memssa -simplifycfg -instcombine -jump-threading -dse -reassociate -instcombine -early-cse-memssa
S19	**37**	-instcombine -gvn -loop-rotate -simplifycfg -jump-threading -early-cse-memssa -mem2reg -simplifycfg
S20	**65**	-licm -early-cse-memssa -sroa -jump-threading -indvars -simplifycfg -instcombine -gvn -instcombine -memcpyopt -loop-rotate -simplifycfg -early-cse-memssa
S21	**48**	-mem2reg -simplifycfg -slp-vectorizer -gvn -instsimplify -simplifycfg -memcpyopt -gvn -dse -instsimplify

Figure 1: Sequences that form the cover set discussed in Section 1.3. The second column reports the number of LLVM passes executed by each sequence. For some perspective, these numbers are, for the default sequences: `O0` = 11; `O1` = 229; `O2` = 277; `O3` = 281; `Os` = 264; and `Oz` = 260 passes.

2 Validation

To validate the cover set seen in Figure 1, we have selected the 2^{13} (8,192) largest single-function programs from ANGHA(taken from `http://cuda.dcc.ufmg.br/angha/files/suites/angha_kernels_largest_10k.tar.gz`). In this section we analyze the following research questions:

RQ1 How often each sequence in Figure 1 beats `opt -Os` and `opt -Oz`.

RQ2 How often each sequence in Figure 1 is provides the smallest target program, i.e., is the winner.

RQ3 How often each sequence in Figure 1 is the *sole* winner.

2.1 RQ1 – Beats Os/Oz

Figure 2 shows how often each sequence in Figure 1 wins over the baseline code-size reduction levels of `opt`. To give the reader some perspective, we include in this experiment `opt -O0`, which does not apply any optimization on the target programs.

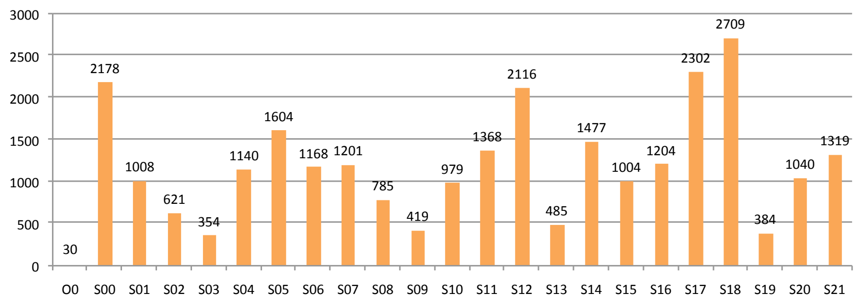


Figure 2: How often, in a universe of 8,192 programs, each sequence in Figure 1 improves (strictly) onto `opt -Os` and `opt -Oz`. The Y-axis shows number of programs.

2.2 RQ2 – Is one of the Winners

Figure 3 shows how often each sequence in Figure 1 leads to the smallest code. We compare all the 22 sequences, plus `opt -O0`, `opt -Os` and `opt -Oz`. We observe that `opt -Oz` is the most frequent winner. However, `opt -Os` is outperformed by `S18`, which is substantially shorter.

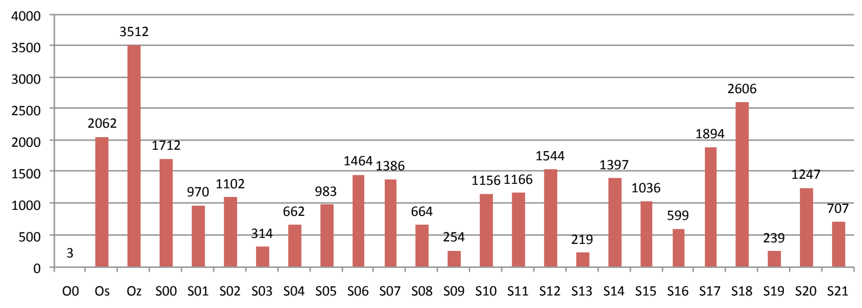


Figure 3: How often, in a universe of 8,192 programs, each sequence in Figure 1 yields the smallest code (possibly with ties) when compared to all the other sequences, plus `opt -O0`, `opt -Os` and `opt -Oz`. The Y-axis shows number of programs.

2.3 RQ3 – Is one of Sole Winner

Figure 4 shows how often each sequence in Figure 1 wins over all the other sequences, plus `opt -O0`, `opt -Os` and `opt -Oz`. Again, `opt -Oz` is the most frequent strict winner. However, `opt -Os` is outperformed by five sequences in Figure 1.

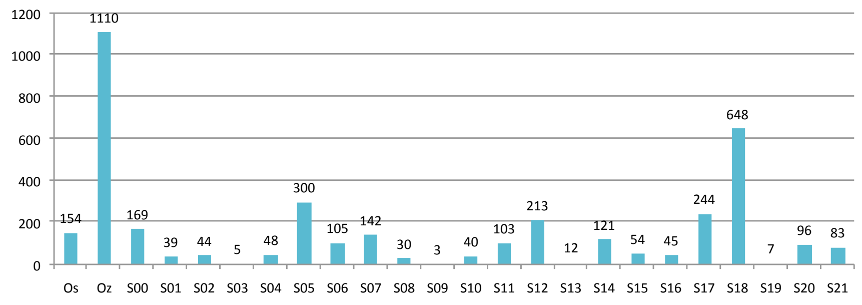


Figure 4: How often, in a universe of 8,192 programs, each sequence in Figure 1 yields the smallest code (without ties) when compared to all the other sequences, plus `opt -O0`, `opt -Os` and `opt -Oz`. The Y-axis shows number of programs.

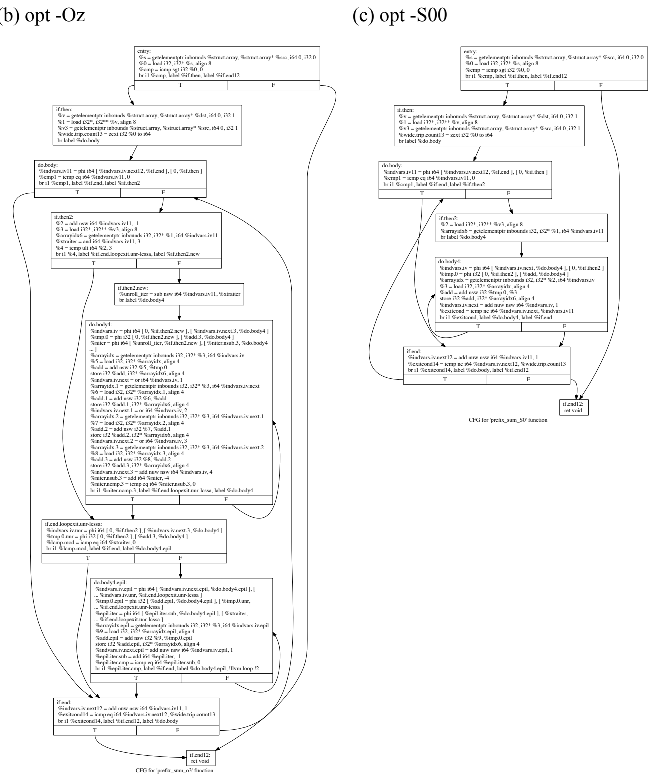
3 Discussion

The two default sequences used as baseline in Section 2, `opt -Os` and `opt -Oz` represent a tradeoff between size and speed. They usually yield shorter codes than the performance-oriented optimization levels (`-O1`, `-O2` and `-O3`). However, it is still relatively easy to find sequences of optimizations that produce shorter binaries. Figure 5 shows such a program. This program, taken from Angha, computes the sum of prefixes of a list of numbers. Nevertheless, its semantics is immaterial for this discussion.

(a) Original program

```
struct array {
  int s;
  int* v;
};

void prefixSum(
  struct array *src,
  struct array *dst
) {
  int i, j;
  int size = src->s;
  if (0 < size) {
    int* dst_v = dst->v;
    i = 0;
    do {
      int tmp = 0;
      j = 0;
      if (j < i) {
        int* src_v = src->v;
        do  {
          tmp += src_v[j];
          dst_v[i] = tmp;
          j++;
        } while (j < i);
      }
      i++;
    } while (i < size);
  }
}
```


(b) opt -Oz
(c) opt -S00
T
F
CFG for 'prefix_sum_o3' function
CFG for 'prefix_sum_S0' function

```
entry:
 %s = getelementptr inbounds %struct.array, %struct.array* %src, i64 0, i32 0
 %0 = load i32, i32* %s, align 8
 %cmp = icmp sgt i32 %0, 0
 br i1 %cmp, label %if.then, label %if.end12

if.then:
 %v = getelementptr inbounds %struct.array, %struct.array* %dst, i64 0, i32 1
 %1 = load i32*, i32** %v, align 8
 %v3 = getelementptr inbounds %struct.array, %struct.array* %src, i64 0, i32 1
 %wide.trip.count13 = zext i32 %0 to i64
 br label %do.body

do.body:
 %indvars.iv11 = phi i64 [ %indvars.iv.next12, %if.end ], [ 0, %if.then ]
 %cmp1 = icmp eq i64 %indvars.iv11, 0
 br i1 %cmp1, label %if.end, label %if.then2

if.then2:
 %2 = add nsw i64 %indvars.iv11, -1
 %3 = load i32*, i32** %v3, align 8
 %arrayidx6 = getelementptr inbounds i32, i32* %1, i64 %indvars.iv11
 %xtraiter = and i64 %indvars.iv11, 3
 %4 = icmp ult i64 %2, 3
 br i1 %4, label %if.end.loopexit.unr-lcssa, label %if.then2.new

if.then2.new:
 %unroll_iter = sub nsw i64 %indvars.iv11, %xtraiter
 br label %do.body4

do.body4:
 %indvars.iv = phi i64 [ 0, %if.then2.new ], [ %indvars.iv.next.3, %do.body4 ]
 %tmp.0 = phi i32 [ 0, %if.then2.new ], [ %add.3, %do.body4 ]
 %niter = phi i64 [ %unroll_iter, %if.then2.new ], [ %niter.nsub.3, %do.body4
... ]
 %arrayidx = getelementptr inbounds i32, i32* %3, i64 %indvars.iv
 %5 = load i32, i32* %arrayidx, align 4
 %add = add nsw i32 %5, %tmp.0
 store i32 %add, i32* %arrayidx6, align 4
 %indvars.iv.next = or i64 %indvars.iv, 1
 %arrayidx.1 = getelementptr inbounds i32, i32* %3, i64 %indvars.iv.next
 %6 = load i32, i32* %arrayidx.1, align 4
 %add.1 = add nsw i32 %6, %add
 store i32 %add.1, i32* %arrayidx6, align 4
 %indvars.iv.next.1 = or i64 %indvars.iv, 2
 %arrayidx.2 = getelementptr inbounds i32, i32* %3, i64 %indvars.iv.next.1
 %7 = load i32, i32* %arrayidx.2, align 4
 %add.2 = add nsw i32 %7, %add.1
 store i32 %add.2, i32* %arrayidx6, align 4
 %indvars.iv.next.2 = or i64 %indvars.iv, 3
 %arrayidx.3 = getelementptr inbounds i32, i32* %3, i64 %indvars.iv.next.2
 %8 = load i32, i32* %arrayidx.3, align 4
 %add.3 = add nsw i32 %8, %add.2
 store i32 %add.3, i32* %arrayidx6, align 4
 %indvars.iv.next.3 = add nuw nsw i64 %indvars.iv, 4
 %niter.nsub.3 = add i64 %niter, -4
 %niter.ncmp.3 = icmp eq i64 %niter.nsub.3, 0
 br i1 %niter.ncmp.3, label %if.end.loopexit.unr-lcssa, label %do.body4

if.end.loopexit.unr-lcssa:
 %indvars.iv.unr = phi i64 [ 0, %if.then2 ], [ %indvars.iv.next.3, %do.body4 ]
 %tmp.0.unr = phi i32 [ 0, %if.then2 ], [ %add.3, %do.body4 ]
 %lcmp.mod = icmp eq i64 %xtraiter, 0
 br i1 %lcmp.mod, label %if.end, label %do.body4.epil

do.body4.epil:
 %indvars.iv.epil = phi i64 [ %indvars.iv.next.epil, %do.body4.epil ], [
... %indvars.iv.unr, %if.end.loopexit.unr-lcssa ]
 %tmp.0.epil = phi i32 [ %add.epil, %do.body4.epil ], [ %tmp.0.unr,
... %if.end.loopexit.unr-lcssa ]
 %epil.iter = phi i64 [ %epil.iter.sub, %do.body4.epil ], [ %xtraiter,
... %if.end.loopexit.unr-lcssa ]
 %arrayidx.epil = getelementptr inbounds i32, i32* %3, i64 %indvars.iv.epil
 %9 = load i32, i32* %arrayidx.epil, align 4
 %add.epil = add nsw i32 %9, %tmp.0.epil
 store i32 %add.epil, i32* %arrayidx6, align 4
 %indvars.iv.next.epil = add nuw nsw i64 %indvars.iv.epil, 1
 %epil.iter.sub = add i64 %epil.iter, -1
 %epil.iter.cmp = icmp eq i64 %epil.iter.sub, 0
 br i1 %epil.iter.cmp, label %if.end, label %do.body4.epil, !llvm.loop !2

if.end:
 %indvars.iv.next12 = add nuw nsw i64 %indvars.iv11, 1
 %exitcond14 = icmp eq i64 %indvars.iv.next12, %wide.trip.count13
 br i1 %exitcond14, label %if.end12, label %do.body

if.end12:
 ret void
```

```
entry:
 %s = getelementptr inbounds %struct.array, %struct.array* %src, i64 0, i32 0
 %0 = load i32, i32* %s, align 8
 %cmp = icmp sgt i32 %0, 0
 br i1 %cmp, label %if.then, label %if.end12

if.then:
 %v = getelementptr inbounds %struct.array, %struct.array* %dst, i64 0, i32 1
 %1 = load i32*, i32** %v, align 8
 %v3 = getelementptr inbounds %struct.array, %struct.array* %src, i64 0, i32 1
 %wide.trip.count13 = zext i32 %0 to i64
 br label %do.body

do.body:
 %indvars.iv11 = phi i64 [ %indvars.iv.next12, %if.end ], [ 0, %if.then ]
 %cmp1 = icmp eq i64 %indvars.iv11, 0
 br i1 %cmp1, label %if.end, label %if.then2

if.then2:
 %2 = load i32*, i32** %v3, align 8
 %arrayidx6 = getelementptr inbounds i32, i32* %1, i64 %indvars.iv11
 br label %do.body4

do.body4:
 %indvars.iv = phi i64 [ %indvars.iv.next, %do.body4 ], [ 0, %if.then2 ]
 %tmp.0 = phi i32 [ 0, %if.then2 ], [ %add, %do.body4 ]
 %arrayidx = getelementptr inbounds i32, i32* %2, i64 %indvars.iv
 %3 = load i32, i32* %arrayidx, align 4
 %add = add nsw i32 %tmp.0, %3
 store i32 %add, i32* %arrayidx6, align 4
 %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
 %exitcond = icmp ne i64 %indvars.iv.next, %indvars.iv11
 br i1 %exitcond, label %do.body4, label %if.end

if.end:
 %indvars.iv.next12 = add nuw nsw i64 %indvars.iv11, 1
 %exitcond14 = icmp ne i64 %indvars.iv.next12, %wide.trip.count13
 br i1 %exitcond14, label %do.body, label %if.end12

if.end12:
 ret void
```

Figure 5: (a) Program that computes sum of prefixes of a list of numbers. (b) LLVM bytecodes produced by `opt -Oz`. (b) LLVM bytecodes produced by `opt -S00` (Sequence **S00** in Figure 1).

In this example, both, `opt -Os` and `opt -Oz` lead to programs with 68 LLVM instructions. There are nine sequences in Figure 1 that yield programs with 28 instructions. Furthermore, only one sequence, **S02**, generates a larger program, with 70 LLVM instructions. The default LLVM sequences generate large programs due to unrolling. Both these sequences unroll the outermost program loop twice.

Notice that this substantial code size reduction (68 vs 28 LLVM instructions) have been obtained with a much smaller number of passes. For instance, **S00** causes the execution of 38 LLVM passes, whereas `opt -Os` runs 264, and `opt -Oz` runs 260 passes.

References

[1] Anderson Faustino da Silva, Bruno Kind, José Wesley Magalh aes, Jerônimo Rocha, Breno Guimar aes, and Fernando Magno Quint ao Pereira. Anghabench: a synthetic collection of benchmarks mined from open-source repositories. Technical Report 01-2020, Universidade Federal de Minas Gerais, 2020.

[2] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *CASES*, pages 197–206, New York, NY, USA, 2010. ACM.

[3] Mahmoud Parsian. *k-Nearest Neighbors*, page 264–275. O'Reilly Media, Boston, USA, 2015.

[4] Suresh Purini and Lakshya Jain. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.