Optimizer Evaluation

(Version 0.06)

Preston Briggs
preston@cs.rice.edu

This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Preface

The program embedded in this document is intended to help determine the state of the art of optimization in production C compilers. I hope that the document itself will be useful to compiler writers, industrial and academic, and to programmers who are interested in learning what they can expect from a compiler.

This program should not be used to compare compilers. When comparison shopping for compilers, the following factors should be considered:

correctness The compiler shouldn't crash. Ought to generate correct code.

completeness cover the whole language? include extensions you want?

speed For most development, compilation speed is quite important.

space If it requires too much working space while compiling, you won't be able to run it on large routines.

error reporting

debugging support For correctness, but also for performance

cost A consideration for many individuals.

Certainly the engineers responsible for each compiler balance all these factors and make their design accordingly. It isn't fair to compare a quick development compiler (*e.g.*, lcc) with a slower optimizing compiler (*e.g.*, gcc) on the basis of optimization. They have different goals and strengths.

Of course, many people *are* interested in code quality – that is, the speed of the code generated for real applications (especially *their* applications). The examples in this program are *not* real; they are tiny, artificial cases, useful only because they help expose certain picky details of optimizer design. On the other hand, it can be argued that attention to the details tested here will pay off in terms of more complete optimization of real programs.

Contents

Chapter 1

Introduction

[1]

Many people invent their own approaches to problems as they arise. While I applaud invention, reinvention is less interesting. I am particularly distressed by inferior approaches to well-understood problems.

[5, 8, 6, 7]

get all the details right.

"Best" versus "best, simple" [19].

[12] want optimizer that doesn't degrade in the face of inlining

1.1 Goals

collect and record examples and counter-examples

interested in finding out which algorithms are used in industry

does anyone do better than published work?

educates by showing what is (and is not) possible, weaknesses and strengths of different algorithms

1.2 Related Work

[25]

[20]

less detailed, different goals; they're interested in how much optimization affects different programs. I want details about whether particular optimizations are done well.

1.3 Approach

discuss state of the art, and perhaps some common alternatives (and their weaknesses)

a series of examples (in C for now), test code, result code, time

when creating test cases, verify behaviour by examining object code

try to show real-world examples

1.3.1 Optimizations Covered

dead code elimination Not the same as elimination of unreachable code

constant propagation

value numbering elimination of redundant computations

code motion primarily moving invariant code out of loops

strength reduction

Well known, widely applicable, I understand them

1.3.2 Other Possibilities

I haven't tried to cover every possible optimization. Important but neglected areas include:

- instruction scheduling
- register allocation
- instruction selection
- vectorization
- cache management
- pointer/structure/array/alias analysis

Nor have I tried to test every desirable feature of a compiler, or even the optimizer.

- correctness
- efficiency (time and space) – I wish these were reported along with SPECmarks.

I haven't even tried to cover any language beyond C. Certainly a similar test would be practical, and I think desirable, for Fortran compilers.

1.4 Acknowledgements

Ken Kennedy, Keith Cooper, Linda Torczon, Cliff Click, Rob Shillner, Steve Carr, Brian Koblenz, Michael Lewis, John Elder, Robert Metzger

Chapter 2

Experiments

The plan is to conduct a large number of experiments to test the abilities of an optimizer.

2.1 Program Organization

3a "global.h"≡

```
    ⟨Include files 3d, ...⟩
    ⟨Types 3e⟩
    ◇
```

3b "main.c"≡

```
    #include "global.h"
    ⟨Constants 7a, ...⟩
    ⟨Variables 9b⟩
    ⟨Prototypes 4, ...⟩
    ◇
```

See *3c, 6b, 8cd, 9e, 10.*

3c "main.c"≡

```
    int main()
    {
      ⟨Initialize 9c⟩
      puts("Begin tests, version 0.06");
      ⟨Evaluation drivers 11a, ...⟩
      puts("Tests completed");
      exit(0);
    }
    ◇
```

See *3b, 6b, 8cd, 9e, 10.*

3d ⟨Include files⟩ ≡

```
    #include <stdlib.h>
    #include <stdio.h>
    ◇
```

See 3a, *6a, 9a.*

2.2 Controlling the Experiments

Each experiment will be controlled by `test`, a routine taking two function parameters, `base_case` and `test_case`, each of type `FunPar`.

3e ⟨Types⟩ ≡

```
    typedef void (*FunPar)(int *);
    ◇
```

See 3a.

The `test` routine will repeatedly measure the time required to run each of the parameter functions. If the average times required for the two functions do not differ significantly, `test` will return `1`, indicating that the compiler was able to optimize the test case to match the (hand optimized) base case. If the time required for the base case is significantly less than the time required for the test case, `test` will return `0`, indicating that the compiler did not perform the optimization in question.

Of course, there's a third possibility: The test case might require significantly less time than the base case. In this event, `test` will complain and halt, printing the `id` of the bungled test. This shouldn't happen during an actual run, though it might happen during development of the test cases.

4 ⟨Prototypes⟩ ≡
```
    int test(int ok,
             FunPar base_case,
             FunPar test_case,
             int *data,
             int n,
             char *id);
```
◇

See 3b, *7b, 8e, 17, 25, 32b, 39, 45c.*

The `data` parameter allows us to specify some arbitrarily-structured data to be passed as an argument to each of the routines.

2.2.1 Theory

The question of whether the times required for the two functions differs *significantly* is hard. On most modern machines, there are at least three components contributing to measurement uncertainty:

1. timer quantization error,
2. jitter caused by context switches, and
3. cache pollution caused by competing processes.

In the next three sections, I'll outline approaches to each of these problems.

Timer Quantization Errors

The timers available on some common workstations are relatively coarse, typically 60 or 100Hz. Since a sample run may start anywhere within the interval of a single timer tick, the measured length of a run may vary from its true value by up to one tick. If our timing runs are too short, this error will dominate our measurements. The way around the problem is to ensure that tests run for many ticks. For example, if a single test runs for more than 100 ticks, we would expect that the quantization error would be less than 1%.

Since the tests will typically be quite short (perhaps a dozen instructions each), I'll wrap a loop around each one. By setting the number of iterations for each loop high enough, I can ensure that each test runs long enough (say, at least 100 ticks) to minimize quantization errors.

Context Switches

Context switches under a multitasking operating system exacerbate the quantization error described above. A process may be interrupted anywhere in a tick and control returned at some other point. Furthermore, a potentially unbounded number of interuptions may occur during a given run. The effect of each interruptions is introduce a new error of up to one tick, either or longer or shorter. While we cannot predict the number of interruptions that might occur during a single run, we can expect that the cummulative errors introduced would be *normally distributed.*

To control this source of error, we'll measure each routine several times and compare the results using statistical analysis based on Student's t distributions [23, Section 11–3]. This is the desired approach given the following assumptions:

- the size of the sample sets is small,

- they are *normally distributed,* and
- they have the same *variance.*

Minimizing the size of the sample sets (the number of times we have to measure each routine) is desirable, since we'd like the complete set of tests to finish is a reasonable time. The assumption of identical variance turns out not to matter; that is, under the conditions of our experiment (two sample sets of equal size), the test described below is equivalent to a variance test.

We begin by collecting a sample set for each routine x and y, where the number of samples in each set is identical and will be called n. (A reasonable number of samples might be 10.) Compute arithmetic means $\bar{x}$ and $\bar{y}$ for the sample sets, where

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Compute the variance σ_x and σ_y for each sample set, where

$$\sigma_x = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}$$

Compute an estimate of the pooled variance $\hat{\sigma}$, where

$$\hat{\sigma} = \frac{\sigma_x + \sigma_y}{2}$$

Finally, compute the statistic T, where

$$T = \frac{\bar{x} - \bar{y}}{\sqrt{2\hat{\sigma}/n}}$$

Large absolute values of T suggest that it is unlikely that $\bar{x} = \bar{y}$. Thus, if $|T| > t(2n-2, p)$, we say that $\bar{x}$ differs from $\bar{y}$ at the p level (where t is a table of magic values copied from a statistics book [23, Table A–8]). For example, if $n = 10$ and $T = 2.61$, we can examine the row given by $t(18, *)$ in Table 2.1 and we see that $2.878 < T < 2.552$, leading us to reject the hypothesis that $\bar{x} = \bar{y}$, with less than 1% chance of error.

Cache Pollution

We use the term "cache pollution" to encompass an entire set of difficulties introduced by the memory hierarchy. In the worst case, we might see effects from the data cache, the instruction cache, a second-level cache, and the TLB. There seem to be two sources of cache pollution:

self interference In the absence of context switches, we would expect that the possibility of self interference would be minimized since each test is quite small, certainly fitting into the I cache, and only references

Degrees of	Probability						
freedom	0.005	0.01	0.025	0.05	0.1	0.15	...
⋮							
16	2.921	2.583	2.120	1.746	1.363	1.071	
17	2.898	2.567	2.110	1.740	1.333	1.069	
18	**2.878**	**2.552**	**2.101**	**1.743**	**1.330**	**1.067**	
19	2.861	2.539	2.093	1.729	1.328	1.066	
20	2.845	2.528	2.086	1.725	1.325	1.064	
⋮							

Table 2.1 Student's t Distributions

a tiny amount of data, which will easily fit into the D cache. A single, preliminary run of a test should provoke any initial cache misses, effectively warming them up.

Note that a machine with a combined cache (versus separate I and D caches) may provoke conflicts between instructions and data, particularly in a direct-mapped cache. Since the conflict may (unpredicably) hurt one case without harming another, it won't really be possible to ensure fair comparisons on such machines.

interference from other processes Besides the quantization errors discussed above, a context switch can also leave the cache in a non-deterministic state, typically provoking a string of cache misses.

Again, the tactic of using many runs comes to mind. Unfortunately, the errors introduced in this case are *not* normally distributed. Cache misses always add to the run time; they never decrease it. This suggests using the minimum time from several runs rather than computing the mean. Unfortunately, this notion conflicts with our earlier approach to minimizing the effects of quantization errors due to context switches.

I don't have a compelling solution to the problems raised here. It may be a topic for research. I'll have to talk to some statistics people about it. Things that come to mind:

- Obviously, try to get the machine in single-user mode, or at least lightly loaded.
- If the standard deviations seem "too high" (more than 2 clock ticks, maybe?), complain that the data is too noisy and ignore the results.

Is there any chance of bounding, statistically, the chance for error? I'm already unsure about the combination of running for a long time and running many times. Sure, the more/longer we run the thing, the more confidence we can have in our conclusion; but what is the real chance for error?

2.2.2 Practice

When collecting samples, run once, ignoring the result, just to get the Instruction cache stable. Collect 10 times from one, then 10 timings from the other.

Work on tests so that the differences are significant.

6a ⟨Include files⟩ ≡

```
#include <math.h>
◇
```

See 3a, *3d, 9a*.

6b "main.c"≡

```
int test(int ok,
         FunPar base_case,
         FunPar test_case,
         int *data,
         int n,
         char *id)
{
  int ret_val = 0;
  printf("    %d) %s - ", n, id);
  if (1 /* always test, for now */ || ok) {
    double base_times[SAMPLES], test_times[SAMPLES];
    long int iterations = determine_iterations(base_case, data);
    collect_timings(iterations, test_case, data, test_times);
    collect_timings(iterations, base_case, data, base_times);
    ⟨Analyze timing results 7c⟩
  }
  else puts("skipping");
  return ret_val;
}
◇
```

See *3bc, 8cd, 9e, 10*.

7a ⟨Constants⟩ ≡
```
#define SAMPLES 10
/* #define CLOCKS_PER_SEC 1000000       for SPARC */
◇
```
See 3b, *7d*.

7b ⟨Prototypes⟩ ≡
```
long int determine_iterations(FunPar base_case,
                              int *data);

void collect_timings(long int iterations,
                     FunPar routine,
                     int *data,
                     double *times);
◇
```
See 3b, *4, 8e, 17, 25, 32b, 39, 45c*.

Note that we've done a little algebraic manipulation, effectively rewriting the computation of T as

$$T = (\bar{x} - \bar{y})\sqrt{\frac{n}{\sigma_x + \sigma_y}}$$

which looks a little uglier, but should save some time and precision. Additionally, we take care to avoid division by 0. This is a real possibility, especially with relatively slow clocks. It doesn't imply any sort of problem; it only means that all the times measured for each were identical.

7c ⟨Analyze timing results⟩ ≡
```
{
  double base_mean = mean(base_times);
  double test_mean = mean(test_times);
  double result = test_mean - base_mean;
  double base_variance = variance(base_mean, base_times);
  double test_variance = variance(test_mean, test_times);
  double sum = base_variance + test_variance;
  if (sum == 0.0) {
    if (result == 0.0)     ret_val = 1;
    else if (result < 0.0) ret_val = -1;
  }
  else {
    result = result * sqrt(SAMPLES / sum);
    if (fabs(result) <= CUTOFF) ret_val = 1;
    else if (result < 0.0)      ret_val = -1;
  }
  if (ret_val > 0)       puts("yes");
  else if (ret_val == 0) puts("no");
  else                   puts("bad test");
  ⟨Check for exceptionally noisy data 8a⟩
}◇
```
See 6b.

We'll define `CUTOFF` to limit ourselves to a 1% chance of mistakenly claiming an optimization was not performed.

7d ⟨Constants⟩ ≡
```
#define CUTOFF 2.552
◇
```
See 3b, *7a*.

8a ⟨Check for exceptionally noisy data⟩ ≡

```
{
  int n = noisy(base_variance) || noisy(test_variance);
  if (n)
    puts("\t(warning -- the timings were not very consistent)");
  if (1 /* always print, for now */ || n || ret_val == -1)
    ⟨Print times and statistical measures 8b⟩
}◇
```

See 7c.

I've broken this out as a separate scrap so I can use it conveniently during development of the tests.

8b ⟨Print times and statistical measures⟩ ≡

```
{
  int i;
  puts("\nTimings\n\tbase-case\ttest-case");
  for (i=0; i<SAMPLES; i++)
    printf("\t%-9.3f\t%-9.3f\n", base_times[i], test_times[i]);
  printf("means\t%-9.3f\t%-9.3f\n", base_mean, test_mean);
  printf("vars\t%-9.6f\t%-9.6f\n", base_variance, test_variance);
  printf("sds\t%-9.6f\t%-9.6f\n", sqrt(base_variance), sqrt(test_variance));
  printf("result\t%-9.6f\n\n", result);
}◇
```

See 8a.

We'll assume the data was noisy if the standard deviation is more than 2 ticks.

8c "main.c"≡

```
int noisy(double variance)
{
  double std_dev = sqrt(variance);
  return std_dev > 2.0 / ticks_per_sec;
}
◇
```

See *3bc, 6b, 8d, 9e, 10.*

8d "main.c"≡

```
double mean(double *times)
{
  int i;
  double sum = 0.0;
  for (i=0; i<SAMPLES; i++)
    sum += times[i];
  return sum / SAMPLES;
}
◇
```

```
double variance(double mean, double *times)
{
  int i;
  double sum = 0.0;
  for (i=0; i<SAMPLES; i++) {
    double diff = mean - times[i];
    sum += diff * diff;
  }
  return sum / (SAMPLES - 1);
}
◇
```

See *3bc, 6b, 8c, 9e, 10.*

8e ⟨Prototypes⟩ ≡

```
double mean(double *times);
double variance(double mean, double *times);
◇
```

See 3b, *4, 7b, 17, 25, 32b, 39, 45c.*

2.2.3 Timing Routines

In the bad old days, timing routines were fairly machine dependent and programs like this had to be customized for each machine. Fortunately, there's now some standardization in the area and we are able to use routines from ANSI C's standard runtime library to avoid some of this porting effort.

9a ⟨Include files⟩ ≡
```
#include <time.h>
◇
```
See 3a, *3d, 6a*.

One might think that the value of `CLOCKS_PER_SEC` would indicate the number of timer ticks per second, allowing convenient determination of the timer resolution. Unfortunately, this isn't the case. Some systems define `CLOCKS_PER_SEC` to be a very large value, despite a fairly coarse timer.

9b ⟨Variables⟩ ≡
```
clock_t ticks_per_sec = 0;
◇
```
See 3b.

9c ⟨Initialize⟩ ≡
```
{
  clock_t start, diff;
  ⟨Make sure clock() works on this system 9d⟩
  start = clock();
  diff = clock() - start;
  while (diff == 0)
    diff = clock() - start;
  ticks_per_sec =  CLOCKS_PER_SEC / diff;
  printf("Timer seems to run at %d ticks/second\n", ticks_per_sec);
}◇
```
See 3c.

Though `clock` is a standard routine, systems are not required to support it; as an alternative, they may return `-1`. On such systems, an alternative approach will be required (*i.e.*, we're back to customizing the code for every machine).

9d ⟨Make sure `clock()` works on this system⟩ ≡
```
{
  if (clock() == -1) {
    fputs("\nThe routine 'clock' is not supported\n", stderr);
    exit(-1);
  }
}◇
```
See 9c.

9e "main.c"≡
```
long int determine_iterations(FunPar base_case,
                              int *data)
{
  long int iterations = 1;
  base_case(data);  /* an initial, untimed invocation */
  while (1) {
    long int i;
    clock_t start = clock();
    for (i=0; i<iterations; i++)
      base_case(data);
    if (((double) clock() - start) / CLOCKS_PER_SEC > 100.0 / ticks_per_sec)
      return iterations;
    iterations += iterations;
  }
}
◇
```
See *3bc, 6b, 8cd, 10*.

10 "main.c"≡

```
void collect_timings(long int iterations,
                     FunPar routine,
                     int *data,
                     double *times)
{
  int i;
  routine(data);  /* an initial, untimed invocation */
  for (i=0; i<SAMPLES; i++) {
    long int j;
    clock_t start = clock();
    for (j=0; j<iterations; j++)
      routine(data);
    times[i] = ((double) (clock() - start)) / CLOCKS_PER_SEC;
  }
}
◇
```

See *3bc, 6b, 8cd, 9e.*

Chapter 3

Dead Code Elimination

best approach is based on use-def chains [15, 16, 17]
 using liveness analysis is inferior [9]
 ref-counting is inferior
 SSA-based version of use-def chains is cheaper [13, Section 7.1]
 Allows removal of useless control flow, and therefore more useless expressions
 partially dead code
 within block, across blocks, simple loop, complex loop case, dead control flow (if and loop)

11a ⟨Evaluation drivers⟩ ≡
```
    {
      int ok;
      puts("Dead code elimination");
      ⟨Dead code elimination 11c, ...⟩
    }◇
```
See 3c, *18a, 26a, 33a, 40a.*

3.1 Local

3.1.1 The Easiest Case

This illustrates one of the easiest possible cases. The initialization of `j` is useless, so the add and the first two references to `data` may be removed, as shown in the result case.

11b "dead.c"≡
```
    void dead_test1(int *data)                    void dead_result1(int *data)
    {                                             {
      int j = data[0] + data[1];                    int j;
      j = data[2];                                  j = data[2];
      data[j] = 2;                                  data[j] = 2;
    }                                             }
    ◇                                             ◇
```
See *12ac, 13ac, 14ac, 15ac, 16ac.*

This bit of code is typical of the way we'll invoke `test`. We initialize a small array of data, then call `test`, passing in the test data and the two routines to be compared. The first parameter is simply a flag indicating the test is to be performed (in other cases, we'll pass in a variable controlling whether the comparison is to be performed or skipped). The last two parameters are used to identify the test.

11c ⟨Dead code elimination⟩ ≡
```
    {
      int data[3] = { 0, 1, 2 };
      ok = test(1, dead_result1, dead_test1, data, 1, "basic block");
    }◇
```
See 11a, *12bd, 13bd, 14bd, 15bd, 16bd.*

3.1.2 Slightly More Complex

12a "dead.c"≡

```
void dead_test2(int *data)
{
  int j = data[0] + data[1];
  int k = j + data[2];
  int m = k + data[3];
  int n = m + data[4];
  j = data[2];
  data[j] = 2;
}
◇
```

```
void dead_result2(int *data)
{
  int j;
  j = data[2];
  data[j] = 2;
}
◇
```

See *11b, 12c, 13ac, 14ac, 15ac, 16ac.*

12b ⟨Dead code elimination⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3, 4 };
  ok = test(ok, dead_result2, dead_test2, data, 2, "basic block");
}◇
```

See 11a, *11c, 12d, 13bd, 14bd, 15bd, 16bd.*

3.2 Global

3.2.1 Across Basic Blocks

Easy

12c "dead.c"≡

```
void dead_test3(int *data)
{
  int k = 0;
  int j = data[1];
  if (j) {
    k++;
    j = k + data[0] * j;
  }
  else
    k--;
  j = data[2];
  data[j] = 2;
  data[3] = k;
}
◇
```

```
void dead_result3(int *data)
{
  int k = 0;
  int j = data[1];
  if (j)
    k++;
  else
    k--;
  j = data[2];
  data[j] = 2;
  data[3] = k;
}
◇
```

See *11b, 12a, 13ac, 14ac, 15ac, 16ac.*

12d ⟨Dead code elimination⟩ ≡

```
{
  int data[4] = { 0, 1, 2 };
  ok = test(ok, dead_result3, dead_test3, data, 3, "across basic blocks");
}◇
```

See 11a, *11c, 12b, 13bd, 14bd, 15bd, 16bd.*

More Interesting

13a "dead.c"≡

```
void dead_test4(int *data)
{
  int k = 0;
  int j = data[1];
  if (j) {
    k++;
    j = k + j * data[0];
  }
  else
    k--;
  if (data[4] & 1)
    k++;
  else {
    k--;
    j++;
  }
  j = data[2];
  data[j] = 2;
  data[3] = k;
}
◇
```

```
void dead_result4(int *data)
{
  int k = 0;
  int j = data[1];
  if (j)
    k++;
  else
    k--;
  if (data[4] & 1)
    k++;
  else
    k--;
  j = data[2];
  data[j] = 2;
  data[3] = k;
}
◇
```

See *11b, 12ac, 13c, 14ac, 15ac, 16ac.*

13b ⟨Dead code elimination⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3, 4 };
  ok = test(ok, dead_result4, dead_test4, data, 4, "across basic blocks");
}◇
```

See 11a, *11c, 12bd, 13d, 14bd, 15bd, 16bd.*

3.2.2 Around Loops

Simple

13c "dead.c"≡

```
void dead_test5(int *data)
{
  int i, j;
  int k = 0;
  int stop = data[0];
  for (i=0; i<stop; i++) {
    k = k * data[1];
    j = data[2];
    data[j] = 2;
  }
}
◇
```

```
void dead_result5(int *data)
{
  int i, j;
  int stop = data[0];
  for (i=0; i<stop; i++) {
    j = data[2];
    data[j] = 2;
  }
}
◇
```

See *11b, 12ac, 13a, 14ac, 15ac, 16ac.*

13d ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 2, 1, 2 };
  ok = test(ok, dead_result5, dead_test5, data, 5, "around loops");
}◇
```

See 11a, *11c, 12bd, 13b, 14bd, 15bd, 16bd.*

Interesting

14a "dead.c"≡

```
void dead_test6(int *data)
{
  int i;
  int k = 0;
  int m = 0;
  int n = 0;
  int stop = data[0];
  for (i=0; i<stop; i++) {
    int j = data[1];
    if (j) {
      n = j * stop + m;
      k++;
    }
    else {
      m = n * stop + k;
      k--;
    }
    j = data[2];
    data[j] = 2;
  }
  data[3] = k;
}
◇
```

```
void dead_result6(int *data)
{
  int i;
  int k = 0;
  int m = 0;
  int n = 0;
  int stop = data[0];
  for (i=0; i<stop; i++) {
    int j = data[1];
    if (j)
      k++;
    else
      k--;
    j = data[2];
    data[j] = 2;
  }
  data[3] = k;
}
◇
```

See *11b, 12ac, 13ac, 14c, 15ac, 16ac.*

14b ⟨Dead code elimination⟩ ≡

```
{
  int data[4] = { 2, 1, 2 };
  ok = test(ok, dead_result6, dead_test6, data, 6, "around loops");
}◇
```

See 11a, *11c, 12bd, 13bd, 14d, 15bd, 16bd.*

3.3 Dead Control Flow

3.3.1 Conditionals

Simple

14c "dead.c"≡

```
void dead_test7(int *data)
{
  int k = 0;
  int j = data[2];
  data[j] = 2;
  if (j) k++;
  else k--;
}
◇
```

```
void dead_result7(int *data)
{
  int j = data[2];
  data[j] = 2;
}
◇
```

See *11b, 12ac, 13ac, 14a, 15ac, 16ac.*

14d ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 0, 1, 2 };
  ok = test(1, dead_result7, dead_test7, data, 7, "useless conditionals");
}◇
```

See 11a, *11c, 12bd, 13bd, 14b, 15bd, 16bd.*

Interesting

15a "dead.c"≡

```
void dead_test8(int *data)
{
  int i;
  int k = 0;
  int stop = data[0];
  for (i=0; i<stop; i++) {
    int j = data[1] + k;
    if (j) k++;
    else k--;
    j = data[2];
    data[j] = 2;
  }
}
◇
```

```
void dead_result8(int *data)
{
  int i;

  int stop = data[0];
  for (i=0; i<stop; i++) {
    int j;
    j = data[2];
    data[j] = 2;
  }
}
◇
```

See *11b, 12ac, 13ac, 14ac, 15c, 16ac.*

15b ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 2, 1, 2 };
  ok = test(ok, dead_result8, dead_test8, data, 8, "useless conditionals");
}◇
```

See 11a, *11c, 12bd, 13bd, 14bd, 15d, 16bd.*

3.3.2 Loops

Conservative

15c "dead.c"≡

```
void dead_test9(int *data)
{
  int k = 0;
  int j;
  for (j=0; j<5; j++) k++;
  j = data[2];
  data[j] = 2;
}
◇
```

```
void dead_result9(int *data)
{
  int j;
  j = data[2];
  data[j] = 2;
}
◇
```

See *11b, 12ac, 13ac, 14ac, 15a, 16ac.*

15d ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 0, 1, 2 };
  ok = test(1, dead_result9, dead_test9, data, 9, "useless loops (conservative)");
}◇
```

See 11a, *11c, 12bd, 13bd, 14bd, 15b, 16bd.*

Aggressive

Removing a loop with unknown bounds (a potentially infinite loop) may change the behaviour of a non-terminating program.

16a "dead.c"≡

```
void dead_test10(int *data)                 void dead_result10(int *data)
{                                           {
  int k = 0;                                  int j;
  int j;                                      j = data[2];
  for (j=0; j<data[0]; j++) k++;              data[j] = 2;
  j = data[2];                              }
  data[j] = 2;                              ◇
}
◇
```

See *11b, 12ac, 13ac, 14ac, 15ac, 16c.*

16b ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 2, 1, 2 };
  (void) test(ok, dead_result10, dead_test10, data, 10, "useless loops (aggressive)");
}◇
```

See 11a, *11c, 12bd, 13bd, 14bd, 15bd, 16d.*

3.4 Partially Dead Code

Removal of partially-dead code is actually an unrelated optimization; it's closer in spirit to partial redundancy elimination. 2 papers in PLDI '94. This case can be done by a simple forward propagation.

16c "dead.c"≡

```
void dead_test11(int *data)                 void dead_result11(int *data)
{                                           {
  int i = data[0];                            int i = data[0];
  int j = data[1];                            int j = data[1];
  int k = i * j;                              if (i & j)
  if (i & j)                                    data[0] = i * j;
    data[0] = k;                              j = data[2];
  j = data[2];                                data[j] = 2;
  data[j] = 2;                              }
}                                           ◇
◇
```

See *11b, 12ac, 13ac, 14ac, 15ac, 16a.*

16d ⟨Dead code elimination⟩ ≡

```
{
  int data[3] = { 0, 1, 2 };
  (void) test(1, dead_result11, dead_test11, data, 11, "partially dead");
}◇
```

See 11a, *11c, 12bd, 13bd, 14bd, 15bd, 16b.*

3.5 Prototypes

17 ⟨Prototypes⟩ ≡

```
extern void dead_test1(int *data);          extern void dead_result1(int *data);
extern void dead_test2(int *data);          extern void dead_result2(int *data);
extern void dead_test3(int *data);          extern void dead_result3(int *data);
extern void dead_test4(int *data);          extern void dead_result4(int *data);
extern void dead_test5(int *data);          extern void dead_result5(int *data);
extern void dead_test6(int *data);          extern void dead_result6(int *data);
extern void dead_test7(int *data);          extern void dead_result7(int *data);
extern void dead_test8(int *data);          extern void dead_result8(int *data);
extern void dead_test9(int *data);          extern void dead_result9(int *data);
extern void dead_test10(int *data);         extern void dead_result10(int *data);
extern void dead_test11(int *data);         extern void dead_result11(int *data);
◇                                           ◇
```

See 3b, *4, 7b, 8e, 25, 32b, 39, 45c.*

Chapter 4

Constant Propagation

Best is [21, 22]
combines constant prop with elimination of unreachable code
ideal time to convert multiplies into shifts, adds, and subtracts [7]
can take further advantage of conditionals (assertion stuff)
basic block, extended basic block, dominance, global (pessimistic), global (optimistic), conditional constant, reassociation, through floats, intrinsics (but this is C)
maybe check for overflows

18a ⟨Evaluation drivers⟩ ≡

```
{
  int ok;
  puts("Constant propagation");
  ⟨Constant propagation 18c, ...⟩
}◇
```

See 3c, *11a, 26a, 33a, 40a.*

4.1 Local

4.1.1 Constant Folding

18b "cprop.c"≡

```
void cprop_test1(int *data)                          void cprop_result1(int *data)
{                                                    {
  int j = 1 + 2 * 4;                                   int j = 9;
  data[0] = j;                                         data[0] = j;
}                                                    }
◇                                                    ◇
```

See *19ace, 20bd, 21ac, 22ace, 23bd, 24a.*

18c ⟨Constant propagation⟩ ≡

```
{
  int data[1];
  ok = test(1, cprop_result1, cprop_test1, data, 1, "folding");
}◇
```

See 18a, *19bd, 20ace, 21bd, 22bd, 23ace, 24b.*

4.1.2 Propagating through Expressions

19a "cprop.c"≡

```
     void cprop_test2(int *data)                                void cprop_result2(int *data)
     {                                                          {
       int j = 1;                                                 int j;
       int k = 2;                                                 j = 9;
       int m = 4;                                                 data[0] = j;
       int n = k * m;                                           }
       j = n + j;                                               ◇
       data[0] = j;
     }
     ◇
```

See *18b, 19ce, 20bd, 21ac, 22ace, 23bd, 24a.*

19b ⟨Constant propagation⟩ ≡

```
     {
       int data[1];
       ok = test(ok, cprop_result2, cprop_test2, data, 2, "basic block");
     }◇
```

See 18a, *18c, 19d, 20ace, 21bd, 22bd, 23ace, 24b.*

4.2 Global

4.2.1 Extended Basic Blocks

19c "cprop.c"≡

```
     void cprop_test3(int *data)                                void cprop_result3(int *data)
     {                                                          {
       int j = 12345;                                             if (data[0])
       if (data[0])                                                 data[1] = 11112;
         data[1] = 1 + j - 1234;                                  else
       else                                                         data[2] = 12478;
         data[2] = 123 + j + 10;                                }
     }                                                          ◇
     ◇
```

See *18b, 19ae, 20bd, 21ac, 22ace, 23bd, 24a.*

19d ⟨Constant propagation⟩ ≡

```
     {
       int data[3] = { 1 };
       ok = test(ok, cprop_result3, cprop_test3, data, 3, "extended basic blocks");
     }◇
```

See 18a, *18c, 19b, 20ace, 21bd, 22bd, 23ace, 24b.*

This one exposes a problem I stumbled across in gcc (ok with `-O2`).

19e "cprop.c"≡

```
     void cprop_test4(int *data)                                void cprop_result4(int *data)
     {                                                          {
       int j = 12345;
       if (data[0])                                               if (data[0])
         data[1] = 1 + j - 1234;                                    data[1] = 11112;
       else                                                       else
         data[2] = 123 - j + 10;                                    data[2] = -12212;
     }                                                          }
     ◇                                                          ◇
```

See *18b, 19ac, 20bd, 21ac, 22ace, 23bd, 24a.*

20a ⟨Constant propagation⟩ ≡
```
{
  int data[3] = { 1 };
  (void) test(ok, cprop_result4, cprop_test4, data, 4, "extended basic blocks");
}◇
```
See 18a, *18c, 19bd, 20ce, 21bd, 22bd, 23ace, 24b.*

4.2.2 Dominators

20b "cprop.c"≡
```
void cprop_test5(int *data)
{
  int j = 5;
  if (data[0])
    data[1] = 10;
  else
    data[2] = 15;
  data[3] = j + 21;
}
◇
```
```
void cprop_result5(int *data)
{
  if (data[0])
    data[1] = 10;
  else
    data[2] = 15;
  data[3] = 26;
}
◇
```
See *18b, 19ace, 20d, 21ac, 22ace, 23bd, 24a.*

20c ⟨Constant propagation⟩ ≡
```
{
  int data[4] = { 1 };
  ok = test(ok, cprop_result5, cprop_test5, data, 5, "dominators");
}◇
```
See 18a, *18c, 19bd, 20ae, 21bd, 22bd, 23ace, 24b.*

Still more odd problems with gcc. The multiplies don't seem to provoke the problem; it also appears with simple adds.

20d "cprop.c"≡
```
void cprop_test6(int *data)
{
  int j = 5;
  if (data[0])
    data[1] = j * 10;
  else
    data[2] = j * 15;
  data[3] = j * 21;
}
◇
```
```
void cprop_result6(int *data)
{
  if (data[0])
    data[1] = 50;
  else
    data[2] = 75;
  data[3] = 105;
}
◇
```
See *18b, 19ace, 20b, 21ac, 22ace, 23bd, 24a.*

20e ⟨Constant propagation⟩ ≡
```
{
  int data[4] = { 1 };
  (void) test(ok, cprop_result6, cprop_test6, data, 6, "dominators");
}◇
```
See 18a, *18c, 19bd, 20ac, 21bd, 22bd, 23ace, 24b.*

4.2.3 DAGs

21a "cprop.c"≡

```
void cprop_test7(int *data)                       void cprop_result7(int *data)
{                                                 {
  int j;                                            if (data[0])
  if (data[0]) {                                      data[1] = 10;
    j = 5;                                          else
    data[1] = 10;                                     data[2] = 15;
  }                                                 data[3] = 26;
  else {                                          }
    data[2] = 15;                                 ◇
    j = 5;
  }
  data[3] = j + 21;
}
◇
```

See *18b, 19ace, 20bd, 21c, 22ace, 23bd, 24a.*

21b ⟨Constant propagation⟩ ≡

```
{
  int data[4] = { 1 };
  ok = test(ok, cprop_result7, cprop_test7, data, 7, "dags");
}◇
```

See 18a, *18c, 19bd, 20ace, 21d, 22bd, 23ace, 24b.*

Here's a similar looking case that I don't expect anyone to get, though it could be handled by first cloning the block containing the assignment to `data[3]`.

21c "cprop.c"≡

```
void cprop_test8(int *data)                       void cprop_result8(int *data)
{                                                 {
  int j, k;                                         if (data[0])
  if (data[0]) {                                      data[1] = 4;
    j = 4;                                          else
    k = 6;                                            data[2] = 3;
    data[1] = j;                                    data[3] = 210;
  }                                               }
  else {                                          ◇
    j = 7;
    k = 3;
    data[2] = k;
  }
  data[3] = (j + k) * 21;
}
◇
```

See *18b, 19ace, 20bd, 21a, 22ace, 23bd, 24a.*

21d ⟨Constant propagation⟩ ≡

```
{
  int data[4] = { 1 };
  (void) test(ok, cprop_result8, cprop_test8, data, 8, "dags (hard)");
}◇
```

See 18a, *18c, 19bd, 20ace, 21b, 22bd, 23ace, 24b.*

4.2.4 Loops

22a "cprop.c"≡

```
void cprop_test9(int *data)
{
  int i;
  int stop = data[0];
  int j = 21;
  for (i=1; i<stop; i++)
    j = (j - 20) * 21;
  data[1] = j;
  data[2] = i;
}
◇
```

```
void cprop_result9(int *data)
{
  int i;
  int stop = data[0];
  for (i=1; i<stop; i++)
    ;
  data[1] = 21;
  data[2] = i;
}
◇
```

See *18b, 19ace, 20bd, 21ac, 22ce, 23bd, 24a.*

22b ⟨Constant propagation⟩ ≡

```
{
  int data[3] = { 2 };
  ok = test(ok, cprop_result9, cprop_test9, data, 9, "loops");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22d, 23ace, 24b.*

4.3 Conditional Constants

22c "cprop.c"≡

```
void cprop_test10(int *data)
{
  int j = 1;
  if (j) j = 10;
  else   j = data[0];
  data[0] = j * 21 + data[1];
}
◇
```

```
void cprop_result10(int *data)
{
  data[0] = 210 + data[1];
}
◇
```

See *18b, 19ace, 20bd, 21ac, 22ae, 23bd, 24a.*

22d ⟨Constant propagation⟩ ≡

```
{
  int data[2] = { 1, 2 };
  ok = test(1, cprop_result10, cprop_test10, data, 10, "conditional constants");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22b, 23ace, 24b.*

The previous test might have been passed successfully by running a relatively simple constant propagator repeatedly if unreachable code is removed between passes. The next case can never be handled by such an approach.

22e "cprop.c"≡

```
void cprop_test11(int *data)
{
  int i;
  int stop = data[0];
  int j = 1;
  for (i=1; i<stop; i++)
    if (!j) j = i;
  data[1] = j;
  data[2] = i;
}
◇
```

```
void cprop_result11(int *data)
{
  int i;
  int stop = data[0];
  for (i=1; i<stop; i++)
    ;
  data[1] = 1;
  data[2] = i;
}
◇
```

See *18b, 19ace, 20bd, 21ac, 22ac, 23bd, 24a.*

23a ⟨Constant propagation⟩ ≡

```
{
  int data[3] = { 2 };
  (void) test(ok, cprop_result11, cprop_test11, data, 11, "conditional constants");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22bd, 23ce, 24b.*

4.4 Conditional-Based Assertions

We need something expensive like the division; otherwise, the multiply can be hidden by scheduling on the RS/6000.

23b "cprop.c"≡

```
void cprop_test12(int *data)                           void cprop_result12(int *data)
{                                                      {
  int j = data[0];                                       int j = data[0];
  if (j == 5)                                            if (j == 5)
    j = j * 21 + 25 / j;                                   j = 110;
  data[1] = j;                                           data[1] = j;
}                                                      }
◇                                                      ◇
```

See *18b, 19ace, 20bd, 21ac, 22ace, 23d, 24a.*

23c ⟨Constant propagation⟩ ≡

```
{
  int data[2] = { 5 };
  ok = test(1, cprop_result12, cprop_test12, data, 12, "conditional-based assertions");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22bd, 23ae, 24b.*

23d "cprop.c"≡

```
void cprop_test13(int *data)                           void cprop_result13(int *data)
{                                                      {
  int j = data[1];                                       int j = data[1];
  int k = data[0];                                       int k = data[0];
  if (j == 5 && k == 10)                                 if (j == 5 && k == 10)
    j = j * 21 + 100 / k;                                  j = 115;
  data[2] = j;                                           data[2] = j;
}                                                      }
◇                                                      ◇
```

See *18b, 19ace, 20bd, 21ac, 22ace, 23b, 24a.*

23e ⟨Constant propagation⟩ ≡

```
{
  int data[3] = { 10, 5 };
  (void) test(ok, cprop_result13, cprop_test13, data, 13, "conditional-based assertions");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22bd, 23ac, 24b.*

4.5 Reassociation

24a "cprop.c"≡

```
void cprop_test14(int *data)                    void cprop_result14(int *data)
{                                               {
  int i = 10;                                     int j = data[0];
  int j = data[0];                                int m = data[1];
  int k = 20;                                     if (data[2])
  int m = data[1];                                  data[3] = 60 + j + m;
  int n = 30;                                     else
  if (data[2])                                      data[0] = 10 * (j - 20 * (m - 30));
    data[3] = i + j + k + m + n;                }
  else                                          ◇
    data[0] = i * (j - k * (m - n));
}
◇
```

See *18b, 19ace, 20bd, 21ac, 22ace, 23bd.*

24b ⟨Constant propagation⟩ ≡

```
{
  int data[4] = { 0, 1, 2 };
  (void) test(ok, cprop_result14, cprop_test14, data, 14, "reassociation");
}◇
```

See 18a, *18c, 19bd, 20ace, 21bd, 22bd, 23ace.*

4.6 Algebraic Simplifications

$$\begin{array}{rcl}
x + 0 & \Rightarrow & x \\
x - 0 & \Rightarrow & x \\
x \times 0 & \Rightarrow & 0 \\
x \times 1 & \Rightarrow & x \\
x/1 & \Rightarrow & x \\
x \cap \text{false} & \Rightarrow & \text{false} \\
x \cap \text{true} & \Rightarrow & x \\
x \cup \text{false} & \Rightarrow & x \\
x \cup \text{true} & \Rightarrow & \text{true} \\
x \ll 0 & \Rightarrow & x \\
0 \ll x & \Rightarrow & 0 \\
\text{mod}(x, 1) & \Rightarrow & 0
\end{array}$$

4.7 Floating-Point Operations

Sometimes propagation of floating-point constants at compile time can lead to loss of precision. In other cases, it's safe and accurate. Furthermore, sometimes users don't consider such losses important.

We should at least check to see how many of the safe cases are handled, and perhaps if they're doing some of the shakier cases (not so much to pass judgement as to show who does what).

conversions, rounding, truncation

4.8 Prototypes

25 ⟨Prototypes⟩ ≡

```
extern void cprop_test1(int *data);
extern void cprop_test2(int *data);
extern void cprop_test3(int *data);
extern void cprop_test35(int *data);
extern void cprop_test4(int *data);
extern void cprop_test5(int *data);
extern void cprop_test6(int *data);
extern void cprop_test7(int *data);
extern void cprop_test8(int *data);
extern void cprop_test9(int *data);
extern void cprop_test10(int *data);
extern void cprop_test11(int *data);
extern void cprop_test12(int *data);
extern void cprop_test13(int *data);
extern void cprop_test14(int *data);
◇
extern void cprop_result1(int *data);
extern void cprop_result2(int *data);
extern void cprop_result3(int *data);
extern void cprop_result4(int *data);
extern void cprop_result5(int *data);
extern void cprop_result6(int *data);
extern void cprop_result7(int *data);
extern void cprop_result8(int *data);
extern void cprop_result9(int *data);
extern void cprop_result10(int *data);
extern void cprop_result11(int *data);
extern void cprop_result12(int *data);
extern void cprop_result13(int *data);
extern void cprop_result14(int *data);
◇
```

See 3b, *4, 7b, 8e, 17, 32b, 39, 45c.*

Chapter 5

Value Numbering

[11]
  [4]
  reassociation
  basic blocks, extended basic blocks, dominance regions, global, gated SSA
  combined with constant prop, unreachable code due to cprop, unreachable code due to value numbering
  improvements possible via conditionals (assertions)

26a ⟨Evaluation drivers⟩ ≡

```
     {
       int ok;
       puts("Value numbering");
       ⟨Value numbering 26c, ...⟩
     }◇
```

See 3c, *11a, 18a, 33a, 40a.*

5.1 Local

[11]

5.1.1 Expressions

26b "valnum.c"≡

```
     void vnum_test1(int *data)                                    void vnum_result1(int *data)
     {                                                             {
       data[0] = data[1] * data[3] - data[1] * data[3];              data[0] = 0;
     }                                                             }
     ◇                                                             ◇
```

See *26d, 27bd, 28ac, 29ac, 30bd, 31ac.*

26c ⟨Value numbering⟩ ≡

```
     {
       int data[4] = {0, 1, 2, 3 };
       ok = test(1,  vnum_result1, vnum_test1, data, 1, "expressions");
     }◇
```

See 26a, *27ace, 28bd, 29b, 30ace, 31b, 32a.*

26d "valnum.c"≡

```
     void vnum_test2(int *data)                                    void vnum_result2(int *data)
     {                                                             {
       data[0] = data[1] * data[3] - data[3] * data[1];              data[0] = 0;
     }                                                             }
     ◇                                                             ◇
```

See *26b, 27bd, 28ac, 29ac, 30bd, 31ac.*

27a ⟨Value numbering⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 3 };
  (void)  test(ok, vnum_result2, vnum_test2, data, 2, "expressions");
}◇
```

See 26a, *26c, 27ce, 28bd, 29b, 30ace, 31b, 32a.*

5.1.2 Basic Blocks

27b "valnum.c"≡

```
void vnum_test3(int *data)                              void vnum_result3(int *data)
{                                                       {
  int n;                                                  int k = data[2];
  int j = data[1] * data[3];                              data[k] = 2;
  int i = data[3];                                        data[0] = 0;
  int m = data[1];                                      }
  int k = data[2];                                      ◇
  data[k] = 2;
  n = m * i;
  data[0] = n - j;
}
◇
```

See *26bd, 27d, 28ac, 29ac, 30bd, 31ac.*

27c ⟨Value numbering⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 3 };
  ok = test(ok, vnum_result3, vnum_test3, data, 3, "basic block");
}◇
```

See 26a, *26c, 27ae, 28bd, 29b, 30ace, 31b, 32a.*

Check commutativity since some compilers seem to miss it.

27d "valnum.c"≡

```
void vnum_test4(int *data)                              void vnum_result4(int *data)
{                                                       {
  int n;                                                  int k = data[2];
  int j = data[1] * data[3];                              data[k] = 2;
  int i = data[3];                                        data[0] = 0;
  int m = data[1];                                      }
  int k = data[2];                                      ◇
  data[k] = 2;
  n = i * m;
  data[0] = n - j;
}
◇
```

See *26bd, 27b, 28ac, 29ac, 30bd, 31ac.*

27e ⟨Value numbering⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 3 };
  (void) test(ok, vnum_result4, vnum_test4, data, 4, "basic block");
}◇
```

See 26a, *26c, 27ac, 28bd, 29b, 30ace, 31b, 32a.*

5.1.3 Extended Basic Blocks

28a "valnum.c"≡

```
void vnum_test5(int *data)            void vnum_result5(int *data)
{                                      {
  int j = data[1] * data[3];             int j = data[1] * data[3];
  if (data[3] == 3) {                    if (data[3] == 3) {
    int n;                                 int k = data[2];
    int i = data[3];                       data[k] = 2;
    int m = data[1];                       data[0] = 0;
    int k = data[2];                     }
    data[k] = 2;                         else if (data[0] & 1) {
    n = m * i;                             j = 3 + data[2] - j;
    data[0] = n - j;                       data[j] = 2;
  }                                      }
  else if (data[0] & 1) {              }
    j = 3 + data[2] - j;               ◇
    data[j] = 2;
  }
}
◇
```

See *26bd, 27bd, 28c, 29ac, 30bd, 31ac.*

28b ⟨Value numbering⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 3 };
  ok = test(ok, vnum_result5, vnum_test5, data, 5, "extended basic block");
}◇
```

See 26a, *26c, 27ace, 28d, 29b, 30ace, 31b, 32a.*

5.2 Dominators

Can be handled pretty easily using SSA. Not published – due to Jonathan Brezin.

28c "valnum.c"≡

```
void vnum_test6(int *data)            void vnum_result6(int *data)
{                                      {
  int n;                                 int j = data[1] * data[3];
  int j = data[1] * data[3];             if (data[0])
  int m = data[1];                         j = j + 3;
  int k = j;                             else
  if (data[0])                             j = j - 3;
    j = j + 3;                           j = data[2] + j;
  else                                   data[j] = 2;
    j = j - 3;                           data[4] = 0;
  n = data[3];                         }
  j = data[2] + j;                     ◇
  data[j] = 2;
  data[4] = k - m * n;
}
◇
```

See *26bd, 27bd, 28a, 29ac, 30bd, 31ac.*

28d ⟨Value numbering⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3, 4 };
  ok = test(ok, vnum_result6, vnum_test6, data, 6, "dominators");
}◇
```

See 26a, *26c, 27ace, 28b, 29b, 30ace, 31b, 32a.*

5.3 Global

[4]

5.3.1 DAGs

29a "valnum.c"≡

```
void vnum_test7(int *data)
{
  int i, j, k;
  int m = data[1];
  int n = data[3];
  if (data[0]) {
    j = m * n;
    i = data[2];
    data[i] = 2;
    k = m * n;
  }
  else {
    j = 5;
    k = 5;
  }
  data[0] = k - j;
}
◇
```

```
void vnum_result7(int *data)
{
  int i;
  if (data[0]) {
    i = data[2];
    data[i] = 2;
  }
  data[0] = 0;
}
◇
```

See *26bd, 27bd, 28ac, 29c, 30bd, 31ac.*

29b ⟨Value numbering⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3, 4 };
  ok = test(1,  vnum_result7, vnum_test7, data, 7, "global DAGs");
}◇
```

See 26a, *26c, 27ace, 28bd, 30ace, 31b, 32a.*

5.3.2 Loops

29c "valnum.c"≡

```
void vnum_test8(int *data)
{
  int i;
  int stop = data[3];
  int m = data[4];
  int n = m;
  for (i=0; i<stop; i++) {
    int k = data[2];
    data[k] = 2;
    data[0] = m - n;
    k = data[1];
    m = m + k;
    n = n + k;
  }
}
◇
```

```
void vnum_result8(int *data)
{
  int i;
  int stop = data[3];
  for (i=0; i<stop; i++) {
    int k = data[2];
    data[k] = 2;
    data[0] = 0;
  }
}
◇
```

See *26bd, 27bd, 28ac, 29a, 30bd, 31ac.*

30a ⟨Value numbering⟩ ≡
```
{
  int data[5] = { 0, 1, 2, 3, 4 };
  ok = test(ok, vnum_result8, vnum_test8, data, 8, "global loops");
}◇
```
See 26a, *26c, 27ace, 28bd, 29b, 30ce, 31b, 32a.*

5.4 Conditional-Based Assertions

5.4.1 Simple

30b "valnum.c"≡
```
void vnum_test9(int *data)                void vnum_result9(int *data)
{                                         {
  int i = data[0];                          int i = data[0];
  int j = data[1];                          int j = data[1];
  if (i == j)                               if (i == j)
    data[2] = (i - j) * 21;                   data[2] = 0;
  else                                      else
    data[2] = i + j;                          data[2] = i + j;
}                                         }
◇                                         ◇
```
See *26bd, 27bd, 28ac, 29ac, 30d, 31ac.*

30c ⟨Value numbering⟩ ≡
```
{
  int data[3] = { 0, 0 };
  ok = test(1, vnum_result9, vnum_test9, data, 9, "conditional-based assertions");
}◇
```
See 26a, *26c, 27ace, 28bd, 29b, 30ae, 31b, 32a.*

5.4.2 Derived

This one is quite hard.

30d "valnum.c"≡
```
void vnum_test10(int *data)               void vnum_result10(int *data)
{                                         {
  int i = data[0];                          int i = data[0];
  int m = i + 1;                            int m = i + 1;
  int j = data[1];                          int j = data[1];
  int n = j + 1;                            int n = j + 1;
  data[2] = m + n;                          data[2] = m + n;
  if (i == j)                               if (i == j)
    data[3] = (m - n) * 21;                   data[3] = 0;
}                                         }
◇                                         ◇
```
See *26bd, 27bd, 28ac, 29ac, 30b, 31ac.*

30e ⟨Value numbering⟩ ≡
```
{
  int data[4] = { 0, 0 };
  ok = test(ok, vnum_result10, vnum_test10, data, 10, "conditional-based assertions");
}◇
```
See 26a, *26c, 27ace, 28bd, 29b, 30ac, 31b, 32a.*

5.5 Conditional Value Numbers

Analogous to the conditional constants found by Wegman and Zadecks's constant propagation algorithm [22], combines value numbering and removal of unreachable code to find more equalities than could be found by running the passes separately. This example is due to Cliff Click, who has an algorithm to handle these cases.

31a "valnum.c"≡

```
void vnum_test11(int *data)
{
  int n;
  int stop = data[3];
  int j = data[1];
  int k = j;
  int i = 1;
  for (n=0; n<stop; n++) {
    if (j != k) i = 2;
    if (i != 1) k = 2;
    data[data[2]] = 2;
  }
  data[1] = i;
}
◇
```

```
void vnum_result11(int *data)
{
  int n;
  int stop = data[3];
  for (n=0; n<stop; n++)
    data[data[2]] = 2;
  data[1] = 1;
}
◇
```

See *26bd, 27bd, 28ac, 29ac, 30bd, 31c.*

31b ⟨Value numbering⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 3 };
  ok = test(1, vnum_result11, vnum_test11, data, 11, "conditional value numbers");
}◇
```

See 26a, *26c, 27ace, 28bd, 29b, 30ace, 32a.*

5.6 Combining Value Numbering and Constant Propagation

This one is quite hard. Combines global value numbering, constant propagation, and elimination of unreachable code. Again, Click's approach should handle this sort of case.

31c "valnum.c"≡

```
void vnum_test12(int *data)
{
  int n;
  int stop = data[3];
  int j = data[1];
  int k = j;
  int i = 1;
  for (n=0; n<stop; n++) {
    if (j != k) i = 2;
    i = 2 - i;
    if (i != 1) k = 2;
    data[data[2]] = 2;
  }
  data[1] = i;
}
◇
```

```
void vnum_result12(int *data)
{
  int n;
  int stop = data[3];
  for (n=0; n<stop; n++)
    data[data[2]] = 2;
  data[1] = 1;
}
◇
```

See *26bd, 27bd, 28ac, 29ac, 30bd, 31a.*

32a ⟨Value numbering⟩ ≡
```
{
  int data[4] = { 0, 1, 2, 3 };
  ok = test(ok, vnum_result12, vnum_test12, data, 12, "cprop + vnum");
}◇
```
See 26a, *26c, 27ace, 28bd, 29b, 30ace, 31b.*

5.7 Reassociation

5.7.1 Expressions

5.7.2 Local

5.7.3 Global

5.8 Algebraic Simplifications

$$\begin{aligned}
x = x &\Rightarrow \text{true}\\
x \neq x &\Rightarrow \text{false}\\
x > x &\Rightarrow \text{false}\\
x \geq x &\Rightarrow \text{true}\\
x < x &\Rightarrow \text{false}\\
x \leq x &\Rightarrow \text{true}\\
x \cap x &\Rightarrow x\\
x \cup x &\Rightarrow x\\
x - x &\Rightarrow 0
\end{aligned}$$

5.9 Prototypes

32b ⟨Prototypes⟩ ≡
```
extern void vnum_test1(int *);
extern void vnum_test2(int *);
extern void vnum_test3(int *);
extern void vnum_test4(int *);
extern void vnum_test5(int *);
extern void vnum_test6(int *);
extern void vnum_test7(int *);
extern void vnum_test8(int *);
extern void vnum_test9(int *);
extern void vnum_test10(int *);
extern void vnum_test11(int *);
extern void vnum_test12(int *);
extern void vnum_result1(int *);
extern void vnum_result2(int *);
extern void vnum_result3(int *);
extern void vnum_result4(int *);
extern void vnum_result5(int *);
extern void vnum_result6(int *);
extern void vnum_result7(int *);
extern void vnum_result8(int *);
extern void vnum_result9(int *);
extern void vnum_result10(int *);
extern void vnum_result11(int *);
extern void vnum_result12(int *);
◇
```
See 3b, *4, 7b, 8e, 17, 25, 39, 45c.*

Chapter 6

Code Motion

Redundancies that can't simply be removed. The examples I'll test for are partial redundancy elimination, loop unswitching, and the possibility of hoisting entire invariant control structures out of loops.

33a ⟨Evaluation drivers⟩ ≡
```
{
  int ok;
  puts("Code motion");
  ⟨Code motion 33c, ...⟩
}◇
```
See 3c, *11a, 18a, 26a, 40a.*

6.1 Invariant Expressions

[18]

6.1.1 DAGs

both of these examples could be done via cloning followed by value numbering (or some other form of simple redundancy elimination)

The Prototypical Case

33b "motion.c"≡
```
void motion_test1(int *data)
{
  int i;
  if (data[1])
    i = data[0] + data[3];
  else {
    data[data[2]] = 2;
    i = 5;
  }
  data[3] = data[0] + data[3];
  data[4] = i;
}
◇
```
```
void motion_result1(int *data)
{
  int i;
  int j;
  if (data[1]) {
    j = data[0] + data[3];
    i = j;
  }
  else {
    data[data[2]] = 2;
    i = 5;
    j = data[0] + data[3];
  }
  data[4] = j;
  data[5] = i;
}
◇
```
See *34ac, 35ac, 36ac, 37ac, 38ac.*

33c ⟨Code motion⟩ ≡
```
{
  int data[5] = { 0, 1, 2, 3 };
  ok = test(1, motion_result1, motion_test1, data, 1, "DAGs");
}◇
```
See 33a, *34bd, 35bd, 36bd, 37bd, 38bd.*

Edge Placement

34a "motion.c"≡

```
void motion_test2(int *data)
{
  int j;
  int i = 1;
  if (data[1]) {
    data[data[2]] = 2;
    j = data[0] + data[3];
    i = i + j;
  }
  data[4] = data[0] + data[3];
  data[5] = i;
}
◇
```

```
void motion_result2(int *data)
{
  int j;
  int i = 1;
  if (data[1]) {
    data[data[2]] = 2;
    j = data[0] + data[3];
    i = i + j;
  }
  else
    j = data[0] + data[3];
  data[4] = j;
  data[5] = i;
}
◇
```

See *33b, 34c, 35ac, 36ac, 37ac, 38ac.*

34b ⟨Code motion⟩ ≡

```
{
  int data[6] = { 0, 1, 2, 3 };
  ok = test(ok, motion_result2, motion_test2, data, 2, "DAGs (edge placement)");
}◇
```

See 33a, *33c, 34d, 35bd, 36bd, 37bd, 38bd.*

6.1.2 Loops

34c "motion.c"≡

```
void motion_test3(int *data)
{
  int i = 0;
  int k = data[2];
  int j = data[0];
  do
    i = 21 * j + i + 1;
  while (i < k);
  data[4] = i;
}
◇
```

```
void motion_result3(int *data)
{
  int i = 0;
  int k = data[2];
  int j = 21 * data[0];
  do
    i = j + i + 1;
  while (i < k);
  data[4] = i;
}
◇
```

See *33b, 34a, 35ac, 36ac, 37ac, 38ac.*

34d ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3 };
  ok = test(1, motion_result3, motion_test3, data, 3, "loops");
}◇
```

See 33a, *33c, 34b, 35bd, 36bd, 37bd, 38bd.*

See if they're smart enough to hoist a divide (in this case, a safe move).

35a "motion.c"≡

```
void motion_test4(int *data)
{
  int i = 0;
  int j = data[2];
  do {
    data[i] = 2 / j + i - 1;
    i++;
  } while (i < data[2]);
}
◇
```

```
void motion_result4(int *data)
{
  int i = 0;
  int j = 2 / data[2];
  do {
    data[i] = j + i - 1;
    i++;
  } while (i < data[2]);
}
◇
```

See *33b, 34ac, 35c, 36ac, 37ac, 38ac.*

35b ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3 };
  (void) test(ok, motion_result4, motion_test4, data, 4, "loops (hoisting divide)");
}◇
```

See 33a, *33c, 34bd, 35d, 36bd, 37bd, 38bd.*

Irreducible Loops

35c "motion.c"≡

```
void motion_test5(int *data)
{
  int i = 0;
  int j = data[0];
  if (data[1])
    goto here;
  j = data[3] + j;
  do {
    i++;
here:
    data[i] = 21 * j + i;
  } while (i < data[2]);
}
◇
```

```
void motion_result5(int *data)
{
  int i = 0;
  int j = data[0];
  if (data[1]) {
    j = j * 21;
    goto here;
  }
  j = data[3] + j;
  j = j * 21;
  do {
    i++;
here:
    data[i] = j + i;
  } while (i < data[2]);
}
◇
```

See *33b, 34ac, 35a, 36ac, 37ac, 38ac.*

35d ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3 };
  (void) test(ok, motion_result5, motion_test5, data, 5, "irreducible loops");
}◇
```

See 33a, *33c, 34bd, 35b, 36bd, 37bd, 38bd.*

6.1.3 Reassociation

36a "motion.c"≡

```
void motion_test6(int *data)
{
  int j = data[1];
  int k = data[2];
  int i = data[0];
  int n = data[3];
  do
    i = j + i + k;
  while (i < n);
  data[4] = i;
}
◇
```

```
void motion_result6(int *data)
{
  int j = data[1];
  int k = data[2];
  int i = data[0];
  int n = data[3];
  int m = j + k;
  do
    i = m + i;
  while (i < n);
  data[4] = i;
}
◇
```

See *33b, 34ac, 35ac, 36c, 37ac, 38ac.*

36b ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 0, 3 };
  (void) test(ok, motion_result6, motion_test6, data, 6, "reassociation");
}◇
```

See 33a, *33c, 34bd, 35bd, 36d, 37bd, 38bd.*

Aggressive Code Motion

moving an expression onto a path where it may not have been executed, possibly slowing the code.

36c "motion.c"≡

```
void motion_test7(int *data)
{
  int i = data[1];
  int j = data[0];
  do {
    if (i & 1) data[data[2]] = 21 * j + i;
    i++;
  } while (i < data[3]);
}
◇
```

```
void motion_result7(int *data)
{
  int i = data[1];
  int j = data[0] * 21;
  do {
    if (i & 1) data[data[2]] = j + i;
    i++;
  } while (i < data[3]);
}
◇
```

See *33b, 34ac, 35ac, 36a, 37ac, 38ac.*

36d ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 4, 10 };
  ok = !test(1, motion_result7, motion_test7, data, 7, "aggressive");
}◇
```

See 33a, *33c, 34bd, 35bd, 36b, 37bd, 38bd.*

Loop Rotation

37a "motion.c"≡

```
void motion_test8(int *data)                      void motion_result8(int *data)
{                                                 {
  int i = 0;                                        int i = 0;
  int j = data[0];                                  int j = data[0];
  while (i < data[2]) {                             if (i < data[2]) {
    data[i] = 21 * j + i;                             j = j * 21;
    i++;                                              do {
  }                                                     data[i] = j + i;
}                                                       i++;
◇                                                     } while (i < data[2]);
                                                    }
                                                  }
                                                  ◇
```

See *33b, 34ac, 35ac, 36ac, 37c, 38ac.*

If they're doing aggressive code motion, then we won't be able to distinguish the conservative approach of using loop rotation.

37b ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3 };
  (void) test(ok, motion_result8, motion_test8, data, 8, "loop rotation");
}◇
```

See 33a, *33c, 34bd, 35bd, 36bd, 37d, 38bd.*

The test above might have been handled by the front end simply translating the `while` loop into an `if` containing a `do-while` (which I think of as the correct approach). Here we try the same test, but using a loop built with `goto` statements.

37c "motion.c"≡

```
void motion_test9(int *data)                      void motion_result9(int *data)
{                                                 {
  int i = 0;                                        int i = 0;
  int j = data[0];                                  int j = data[0];
loop:                                               if (i >= data[2]) return;
  if (i >= data[2]) return;                         j = j * 21;
    data[i] = 21 * j + i;                         loop:
    i++;                                              data[i] = j + i;
    goto loop;                                        i++;
}                                                   if (i < data[2]) goto loop;
◇                                                 }
                                                  ◇
```

See *33b, 34ac, 35ac, 36ac, 37a, 38ac.*

37d ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 3 };
  (void) test(ok, motion_result9, motion_test9, data, 9, "loop rotation");
}◇
```

See 33a, *33c, 34bd, 35bd, 36bd, 37b, 38bd.*

6.2 Invariant Control Flow

6.2.1 Hoisting Invariant Control Flow

[14]. Hmmm. This one could also be done via loop unswitching.

38a "motion.c"≡

```
void motion_test10(int *data)
{
  int j;
  int p = data[1];
  int i = data[0];
  do {
    if (p)
      j = 1;
    else
      j = 2;
    i = i + j;
    data[data[2]] = 2;
  } while (i < data[3]);
}
◇
```

```
void motion_result10(int *data)
{
  int j;
  int p = data[1];
  int i = data[0];
  if (p)
    j = 1;
  else
    j = 2;
  do {
    i = i + j;
    data[data[2]] = 2;
  } while (i < data[3]);
}
◇
```

See *33b, 34ac, 35ac, 36ac, 37ac, 38c.*

38b ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 10 };
  (void) test(ok, motion_result10, motion_test10, data, 10, "invariant control structures");
}◇
```

See 33a, *33c, 34bd, 35bd, 36bd, 37bd, 38d.*

6.2.2 Loop Unswitching

[2]

opportunity arises when using nested DO loops in Fortran 77 and sometimes with nested `for` loops in C.

38c "motion.c"≡

```
void motion_test11(int *data)
{
  int p = data[1];
  int i = data[0];
  do {
    if (p)
      i = i + 1;
    else
      i = i + 2;
    data[data[2]] = 2;
  } while (i < data[3]);
}
◇
```

```
void motion_result11(int *data)
{
  int p = data[1];
  int i = data[0];
  if (p)
    do {
      i = i + 1;
      data[data[2]] = 2;
    } while (i < data[3]);
  else
    do {
      i = i + 2;
      data[data[2]] = 2;
    } while (i < data[3]);
}
◇
```

See *33b, 34ac, 35ac, 36ac, 37ac, 38a.*

38d ⟨Code motion⟩ ≡

```
{
  int data[5] = { 0, 1, 2, 10 };
  (void) test(ok, motion_result11, motion_test11, data, 11, "loop unswitching");
}◇
```

See 33a, *33c, 34bd, 35bd, 36bd, 37bd, 38b.*

6.3 Prototypes

39 ⟨Prototypes⟩ ≡

```
extern void motion_test1(int *);
extern void motion_test2(int *);
extern void motion_test3(int *);
extern void motion_test4(int *);
extern void motion_test5(int *);
extern void motion_test6(int *);
extern void motion_test7(int *);
extern void motion_test8(int *);
extern void motion_test9(int *);
extern void motion_test10(int *);
extern void motion_test11(int *);
```
◇

```
extern void motion_result1(int *);
extern void motion_result2(int *);
extern void motion_result3(int *);
extern void motion_result4(int *);
extern void motion_result5(int *);
extern void motion_result6(int *);
extern void motion_result7(int *);
extern void motion_result8(int *);
extern void motion_result9(int *);
extern void motion_result10(int *);
extern void motion_result11(int *);
```
◇

See 3b, *4, 7b, 8e, 17, 25, 32b, 45c.*

Chapter 7

Strength Reduction

[10, 3]

[9] and other versions based on partial redundancy elimination are weaker

linear function test replacement, odd induction variables, control flow in loop, goto loops, nested loops, irreducible loops, reduction of mod, div, etc.

straight-line code, loop invariants (vs. constants)

40a ⟨Evaluation drivers⟩ ≡

```
    {
      int ok;
      puts("Strength reduction");
      ⟨Strength reduction 40c, ...⟩
    }◇
```

See 3c, *11a, 18a, 26a, 33a.*

7.1 Induction Variable and Constant

Testing a `while` loop might also be interesting; however, the proper method of handling them would be to perform loop rotation first, which is already covered elsewhere.

40b "strength.c"≡

```
    void strength_test1(int *data)            void strength_result1(int *data)
    {                                         {
      int i = 0;                                int i = 0;
      do {                                      do {
        data[data[2]] = 2;                        data[data[2]] = 2;
        i = i + 1;                                i = i + 21;
      } while (i * 21 < data[1]);               } while (i < data[1]);
    }                                         }
    ◇                                         ◇
```

See *41ac, 42ac, 43ac, 44ac, 45a.*

40c ⟨Strength reduction⟩ ≡

```
    {
      int data[3] = { 0, 22, 2 };
      ok = test(1, strength_result1, strength_test1, data, 1, "iv * constant");
    }◇
```

See 40a, *41bd, 42bd, 43bd, 44bd, 45b.*

7.2 Induction Variable and Region Constant

41a "strength.c"≡

```
void strength_test2(int *data)              void strength_result2(int *data)
{                                           {
  int k = data[0];                            int k = data[0];
  int i = 0;                                  int i = 0;
  do {                                        do {
    data[data[2]] = 2;                          data[data[2]] = 2;
    i = i + 1;                                  i = i + k;
  } while (i * k < data[1]);                  } while (i < data[1]);
}                                           }
◇                                           ◇
```

See *40b, 41c, 42ac, 43ac, 44ac, 45a.*

41b ⟨Strength reduction⟩ ≡

```
{
  int data[3] = { 1, 3, 2 };
  (void) test(ok, strength_result2, strength_test2, data, 2, "iv * rc");
}◇
```

See 40a, *40c, 41d, 42bd, 43bd, 44bd, 45b.*

7.3 Multiplying Two Induction Variables

41c "strength.c"≡

```
void strength_test3(int *data)              void strength_result3(int *data)
{                                           {
  int i = data[0];                            int i = data[0];
  int j = data[1];                            int j = data[1];
  do {                                        int k = i * j;
    data[data[2]] = 2;                        do {
    i = i + 1;                                  data[data[2]] = 2;
    j = j + 1;                                  i = i + 1;
  } while (i * j < data[3]);                    k = k + j;
}                                               j = j + 1;
◇                                               k = k + i;
                                              } while (k < data[3]);
                                            }
                                            ◇
```

See *40b, 41a, 42ac, 43ac, 44ac, 45a.*

41d ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 100 };
  (void) test(ok, strength_result3, strength_test3, data, 3, "iv * iv");
}◇
```

See 40a, *40c, 41b, 42bd, 43bd, 44bd, 45b.*

7.4 Irreducible Loops

42a "strength.c"≡

```
void strength_test4(int *data)              void strength_result4(int *data)
{                                           {
  int i;                                      int i;
  if (data[1]) {                              if (data[1]) {
    i = 2;                                      i = 42;
    goto here;                                  goto here;
  }                                           }
  i = 0;                                      i = 0;
  do {                                        do {
    i = i + 1;                                  i = i + 21;
here:                                       here:
    data[data[2]] = 2;                          data[data[2]] = 2;
  } while (i * 21 < data[3]);                 } while (i < data[3]);
}                                           }
◇                                           ◇
```

See *40b, 41ac, 42c, 43ac, 44ac, 45a.*

42b ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 100 };
  (void) test(ok, strength_result4, strength_test4, data, 4, "irreducible loop");
}◇
```

See 40a, *40c, 41bd, 42d, 43bd, 44bd, 45b.*

7.5 Control Flow in the Loop

42c "strength.c"≡

```
void strength_test5(int *data)              void strength_result5(int *data)
{                                           {
  int i = 0;                                  int i = 0;
  while (1) {                                 while (1) {
    i = i + 1;                                  i = i + 21;
    if (data[1] && i * 21 > data[3])            if (data[1] && i > data[3])
      break;                                      break;
    data[data[2]] = 2;                          data[data[2]] = 2;
  }                                           }
}                                           }
◇                                           ◇
```

See *40b, 41ac, 42a, 43ac, 44ac, 45a.*

42d ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 100 };
  (void) test(ok, strength_result5, strength_test5, data, 5, "control flow in loop");
}◇
```

See 40a, *40c, 41bd, 42b, 43bd, 44bd, 45b.*

7.6 More Complex Induction Variables

There are many possibilities [24]. I'm only covering a few cases.

7.6.1 Increment by a Region Constant

43a "strength.c"≡

```
void strength_test6(int *data)
{
  int j = data[0];
  int i = data[1];
  do {
    data[data[2]] = 2;
    i = i + j;
  } while (i * 21 < data[3]);
}
◇
```

```
void strength_result6(int *data)
{
  int j = data[0] * 21;
  int i = data[1] * 21;
  do {
    data[data[2]] = 2;
    i = i + j;
  } while (i < data[3]);
}
◇
```

See *40b, 41ac, 42ac, 43c, 44ac, 45a.*

43b ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 1, 1, 2, 100 };
  (void) test(ok, strength_result6, strength_test6, data, 6, "inc by rc");
}◇
```

See 40a, *40c, 41bd, 42bd, 43d, 44bd, 45b.*

7.6.2 Monotonic Induction Variables

43c "strength.c"≡

```
void strength_test7(int *data)
{
  int i = 0;
  do {
    if (data[1])
      i = i + 1;
    else
      i = i + 2;
    data[data[2]] = 2;
  } while (i * 21 < data[3]);
}
◇
```

```
void strength_result7(int *data)
{
  int i = 0;
  do {
    if (data[1])
      i = i + 21;
    else
      i = i + 42;
    data[data[2]] = 2;
  } while (i < data[3]);
}
◇
```

See *40b, 41ac, 42ac, 43a, 44ac, 45a.*

43d ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 2, 100 };
  (void) test(ok, strength_result7, strength_test7, data, 7, "monotonic iv");
}◇
```

See 40a, *40c, 41bd, 42bd, 43b, 44bd, 45b.*

7.6.3 Mutual Induction Variables

44a "strength.c"≡

```
void strength_test8(int *data)
{
  int i = 0;
  do {
    int j = i + 1;
    data[data[2]] = j;
    i = j + 1;
  } while (i * 21 < data[3]);
}
◇
```

```
void strength_result8(int *data)
{
  int i = 0;
  int n = 0;
  do {
    int j = i + 1;
    n = n + 21;
    data[data[2]] = j;
    i = j + 1;
    n = n + 21;
  } while (n < data[3]);
}
◇
```

See *40b, 41ac, 42ac, 43ac, 44c, 45a.*

44b ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 1, 100 };
  (void) test(ok, strength_result8, strength_test8, data, 8, "mutual iv's");
}◇
```

See 40a, *40c, 41bd, 42bd, 43bd, 44d, 45b.*

7.7 Multiple Strides

44c "strength.c"≡

```
void strength_test9(int *data)
{
  int i = 0;
  do {
    data[data[2]] = i;
    i = i + 1;
  } while (i * 21 < data[3]);
}
◇
```

```
void strength_result9(int *data)
{
  int i = 0;
  int j = 0;
  do {
    data[data[2]] = i;
    i = i + 1;
    j = j + 21;
  } while (j < data[3]);
}
◇
```

See *40b, 41ac, 42ac, 43ac, 44a, 45a.*

44d ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 1, 100 };
  (void) test(ok, strength_result9, strength_test9, data, 9, "multiple strides");
}◇
```

See 40a, *40c, 41bd, 42bd, 43bd, 44b, 45b.*

7.8 Linear Function Test Replacement

45a "strength.c"≡

```
void strength_test10(int *data)
{
  int stop = data[3];
  int i = 0;
  do {
    data[data[2]] = 21 * i;
    i = i + 1;
  } while (i < stop);
}
◇
```

```
void strength_result10(int *data)
{
  int stop = data[3] * 21;
  int i = 0;
  do {
    data[data[2]] = i;
    i = i + 21;
  } while (i < stop);
}
◇
```

See *40b, 41ac, 42ac, 43ac, 44ac.*

45b ⟨Strength reduction⟩ ≡

```
{
  int data[4] = { 0, 1, 1, 20 };
  (void) test(ok, strength_result10, strength_test10, data, 10, "test replacement");
}◇
```

See 40a, *40c, 41bd, 42bd, 43bd, 44bd.*

7.9 Prototypes

45c ⟨Prototypes⟩ ≡

```
extern void strength_test1(int *);
extern void strength_test2(int *);
extern void strength_test3(int *);
extern void strength_test4(int *);
extern void strength_test5(int *);
extern void strength_test6(int *);
extern void strength_test7(int *);
extern void strength_test8(int *);
extern void strength_test9(int *);
extern void strength_test10(int *);
◇
```

```
extern void strength_result1(int *);
extern void strength_result2(int *);
extern void strength_result3(int *);
extern void strength_result4(int *);
extern void strength_result5(int *);
extern void strength_result6(int *);
extern void strength_result7(int *);
extern void strength_result8(int *);
extern void strength_result9(int *);
extern void strength_result10(int *);
◇
```

See 3b, *4, 7b, 8e, 17, 25, 32b, 39.*

Appendix A

Indices

A.1 Files

A.2 Macros

A.3 Identifiers

Appendix B

Results

Try to report dispassionately. Just give the facts, with some interpretation when helpful; but don't accuse of laziness or stupidity. Similarly, don't brag on anyone.

Need version number of compiler

can we get Cray Research, KSR, Paragon (Portland Group), other supers?

design a form for reporting results electronically

B.1 Apple

MPW

B.2 Borland

B.3 Convex

B.4 Cray Computer

B.5 Cray Research

B.6 DEC

B.6.1 Alpha

Rob

B.6.2 DECstation

MIPS, Rob

B.7 Gnu CC

need report by machine (since SPARC version seems fairly inferior)

B.8 Hewlett-Packard

Steve

B.9 IBM

B.9.1 RS/6000

B.10 Microsoft

B.11 Silicon Graphics

Rob

B.12 Sun

B.12.1 Solaris

B.13 Tera

Brian

B.14 Texas Instruments

Reid

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] Frances E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[4] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[5] Marc A. Auslander and Martin E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[6] Robert L. Bernstein. Producing good code for the case statement. *Software – Practice and Experience*, 15(10):1024–1024, October 1985.

[7] Robert L. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[9] Fred C. Chow. *A Portable Machine-Independent Global Optimizer – Design and Measurements.* PhD thesis, Stanford University, December 1983.

[10] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[11] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[12] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[14] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 1986.

[15] Ken Kennedy. Global dead computation elimination. SETL Newsletter 111, Courant Institute of Mathematical Sciences, New York University, August 1973.

[16] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3:163–179, 1978.

[17] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[18] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[19] Michael L. Powell. A portable optimizing compiler for Modula-2. *SIGPLAN Notices*, 19(6):310–318, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[20] Rafael H. Saavedra and Alan J. Smith. Performance characterization of optimizing compilers. Technical Report 525, Department of Computer Science, University of Southern California, August 1992.

[21] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, January 1985.

[22] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[23] Frank L. Wolf. *Elements of Probability and Statistics.* McGraw-Hill, second edition, 1974.

[24] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[25] Michael Wolfe and Tom Macke. Where are the optimizing compilers? *SIGPLAN Notices*, 20(11):64–68, November 1985.