Case Study A: DCE and LVN

P1

```
root@b409df863aff:/tmp/bril# bril2json < benchmarks/core/fizz-buzz.bril | brili -p 5
1
2
-2
4
total_dyn_inst: 148
root@b409df863aff:/tmp/bril# bril2json < benchmarks/core/fizz-buzz.bril | python3 examples/tdce.py | brili -p 5
1
2
-2
4
total_dyn_inst: 144
```

P2

- command:

```
brench example.toml > results.csv
```

- `results.csv` (also available here):

```
benchmark,run,result
primes-between,baseline,574100
primes-between,tdce,574100
primes-between,lvn,571439
hanoi,baseline,99
hanoi,tdce,99
hanoi,lvn,99
lcm,baseline,2326
lcm,tdce,2326
lcm,lvn,2326
pascals-row,baseline,146
pascals-row,tdce,139
pascals-row,lvn,68
up-arrow,baseline,252
up-arrow,tdce,252
up-arrow,lvn,252
euclid,baseline,563
euclid,tdce,562
euclid,lvn,271
binary-fmt,baseline,100
binary-fmt,tdce,100
```

```
binary-fmt,lvn,100
fact,baseline,229
fact,tdce,228
fact,lvn,167
ackermann,baseline,1464231
ackermann,tdce,1464231
ackermann,lvn,1464231
is-decreasing,baseline,127
is-decreasing,tdce,127
is-decreasing,lvn,123
factors,baseline,72
factors,tdce,72
factors,lvn,72
bitshift,baseline,167
bitshift,tdce,167
bitshift,lvn,98
check-primes,baseline,8468
check-primes,tdce,8419
check-primes,lvn,4189
palindrome,baseline,298
palindrome,tdce,298
palindrome,lvn,298
pythagorean_triple,baseline,61518
pythagorean_triple,tdce,61518
pythagorean_triple,lvn,61518
orders,baseline,5352
orders,tdce,5352
orders,lvn,5352
sum-bits,baseline,73
sum-bits,tdce,73
sum-bits,lvn,73
sum-divisors,baseline,159
sum-divisors,tdce,159
sum-divisors,lvn,159
totient,baseline,253
totient,tdce,253
totient,lvn,253
sum-sq-diff,baseline,3038
sum-sq-diff,tdce,3036
sum-sq-diff,lvn,1715
bitwise-ops,baseline,1690
bitwise-ops,tdce,1689
bitwise-ops,lvn,1689
perfect,baseline,232
perfect,tdce,232
perfect,lvn,231
reverse,baseline,46
reverse,tdce,46
reverse,lvn,38
recfact,baseline,104
recfact,tdce,103
recfact,lvn,63
```

```
armstrong,baseline,133
armstrong,tdce,130
armstrong,lvn,130
mod_inv,baseline,558
mod_inv,tdce,555
mod_inv,lvn,304
digital-root,baseline,247
digital-root,tdce,247
digital-root,lvn,247
loopfact,baseline,116
loopfact,tdce,115
loopfact,lvn,78
rectangles-area-difference,baseline,14
rectangles-area-difference,tdce,14
rectangles-area-difference,lvn,14
sum-check,baseline,5018
sum-check,tdce,5018
sum-check,lvn,5018
birthday,baseline,484
birthday,tdce,483
birthday,lvn,277
gcd,baseline,46
gcd,tdce,46
gcd,lvn,46
fitsinside,baseline,10
fitsinside,tdce,10
fitsinside,lvn,10
quadratic,baseline,785
quadratic,tdce,783
quadratic,lvn,500
relative-primes,baseline,1923
relative-primes,tdce,1914
relative-primes,lvn,1097
catalan,baseline,659378
catalan,tdce,659378
catalan,lvn,659378
collatz,baseline,169
collatz,tdce,169
collatz,lvn,169
fizz-buzz,baseline,3652
fizz-buzz,tdce,3552
fizz-buzz,lvn,2103
```

P3

- CSE:
  LVN assign numbers (indices) to every value (which might be of form `(op, #i, #j)`) in a basic block. If a variable, say, `x` is reassigned (i.e., the previously assigned value is killed), then all occurrences of `x` having the previously assigned value (including the `x`

in the previous definition) are replaced with a new name, say, `x'`. By this method, for every definition of a variable (say `x`) in a basic block, the definition is replaced with `x = id x'`, where `x'` is the associated canonical variable of the assigned value of `x`. That is, the value of each definition is copied from the first variable that is also assigned the same value to.

- DCE:
  By the mechanism of LVN explained above, the dead code problem in each basic block can be reduced to the basic dead code problem, which can be solved by the iterative, two-pass algorithm given in slide 9 of Prof. Liao's slide deck for 10/11. Recall that this algorithm eliminates the definitions of unused variables in the entire function (procedure).
- Copy propagation:
  Basic LVN cannot achieve copy propagation without the knowledge the semantics of `id` instruction, which copies the value of the argument and produces the same value. By translating the value `id #i` into `#i` for any instruction `y = id x`, where `#i` is the index of the value of `x`, it can replace the occurrences of target variables with values they have copied from other variables.
- Constant propagation:
  Similar to copy propagation, to propagate constants, the LVN framework has to be aware of the semantics of `id` and `const` instructions. The difference is that whenever LVN sees an instruction `id x` and knows that variable `x` has value `const n`, it replaces `id x` not just with `id x'`, where `x'` is the canonical variable of value `const n`, but with `const n`.