6.035 Phase 4 Documentation

Decaf Optimizing Compiler

An Bo Chen, Alex Dang, Youran (Yoland) Gao
{anbochen, alexdang, youran}@mit.edu

Design & Implementation

Common Subexpression Elimination (CSE)

Enable this optimization with flag: `--opt cse`

Common subexpression elimination is performed as a CFG pass by first running an available expressions (AE) fix point algorithm similar to the one seen in lecture and then visiting all basic blocks in the program. Note that expressions that contain global variables do not participate in common subexpression elimination due to the possibility of side effects and the expressions in the bit vector are only bin op expressions. We use AE to find the set of the available expressions at the beginning of each basic block, which would be `IN[n]` for each block `n`. In the `GEN[n]` procedure, we make sure to not generate available expressions that we also kill in the same block.

At the beginning of the CSE analysis, we have `exprToTemp`, a mapping of `CfgExpression` to their corresponding unique temp variable, `varToExprs`, a mapping of `CfgVariable` to a set of `CfgExpressions` that have the variable as one of its operands, and `IN[n]` for each block `n`. For each assign statement in block `n`, if the right hand expression is available via `IN[n]` or a previously seen expression in the block and has not been killed yet, we replace right hand expression with the corresponding temp variable using `exprToTemp`. If not, then we append a new assign statement afterwards with the corresponding temp variable = left hand variable of the current assign statement. Regardless of what the right hand expression is, we kill all available expressions and seen expressions corresponding to the left hand variable using `varToExprs`.

Below is an example of a simple Decaf program with common subexpression enabled and the associated assembly instruction counts by type. Note that the number of assembly instructions increases slightly because we add some new assign statements when we save variables to temps. This enables Copy Propagation to propagate the use in the copy statements and Dead Code Elimination to remove extraneous assign statements. You can run `./run.sh --target cfg --opt cse tests/codegen/input/_cse-<TEST NUMBER>.dcf` to check out the new CFG applied with CSE. Note that variables without initializers are initialized with a default value.

Decaf	**Unoptimized**	**Optimized**

void main(){
  int a, b, c, d;
 	
  c = a + b;
  d = a + b;
}
	main() => void {
	  _.main_1 @entry @exit
	  (a, b, c, d, x)
	  > a = 0
	  > b = 0
	  > c = 0
	  > d = 0
	  > _.t = a + b
	  > c = _.t
	  > _.t_2 = a + b
	  > d = _.t_2
	}
		main() => void {
		  _.main_1 @entry @exit
		  (a, b, c, d)
		  > a = 0
		  > b = 0
		  > c = 0
		  > d = 0
		  > _.t = a + b
		  > _.tcse = _.t
		  > c = _.t
		  > _.t_2 = _.tcse
		  > d = _.t_2
		}

Count of Assembly Instructions:	movq: 13 jmp: 2 add: 1 enter: 1 leave: 1 ret: 1	movq: 15 jmp: 2 add: 1 enter: 1 leave: 1 ret: 1

Here is an example Decaf program with conditional branching:

Decaf	**Unoptimized**	**Optimized**

void main(){
  int a, b, c, d, e, f;
  bool x;
  a = 1;
  b = 2;
  x = true;
  c = a + b;
  if (x) {
  a = 1;
  d = a + b;
  }
  else {
  e = a + b;
  }
}
	main() => void {
	  _.main_1 @entry
	  (a, b, c, d, e, f, x)
	  > a = 0 (default)
	  > b = 0 (default)
	  > c = 0
	  > d = 0
	  > e = 0
	  > f = 0
	  > x = 0
	  > a = 1
	  > b = 2
	  > x = 1
	  > _.t = a + b
		main() => void {
		  _.main_1 @entry
		  (a, b, c, d, e, f, x)
		  > a = 0
		  > b = 0
		  > c = 0
		  > d = 0
		  > e = 0
		  > f = 0
		  > x = 0
		  > a = 1
		  > b = 2
		  > x = 1
		  > _.t = a + b

	<pre> > c = _.t
	  x ?
	  T-> _.main_2
	  F-> _.main_3
	
	  _.main_2
	  > a = 1
	  > _.t_2 = a + b
	
	  > d = _.t_2
	  --> _.main_4
	
	  _.main_3
	  > _.t_3 = a + b
	  > e = _.t_3
	  --> _.main_4
	
	  _.main_4 @exit
	}</pre>
		<pre> **> _.tcse = _.t**
		  > c = _.t
		  x ?
		  T-> _.main_2
		  F-> _.main_3
		
		  _.main_2
		  > a = 1(killed a + b)
		  > _.t_2 = a + b
		  > _.tcse = _.t_2
		  > d = _.t_2
		  --> _.main_4
		
		  _.main_3
		  > _.t_3 = _.tcse
		  > e = _.t_3
		  --> _.main_4
		
		  _.main_4 @exit
		}</pre>

Count of Assembly Instructions:	movq: 25 jmp: 4 add: 3 cmpq: 1 enter: 1 je: 1 jne: 1 leave: 1 ret: 1	movq: 29 jmp: 4 add: 2 cmpq: 1 enter: 1 je: 1 jne: 1 leave: 1 ret: 1

Copy Propagation (CP)

Enable this optimization with flag: `--opt cp`

Copy propagation is performed as a CFG pass by visiting all basic blocks in the program. We then visit all the statements in each basic block. Firstly, on all assign statements, the left hand variable is killed, regardless of what the right hand side is. When there is an assign statement such that both the left and right sides are of type `CfgVariable`, it is possible to add this mapping to `tmpToVars`. Global variables do not participate in copy propagation due to the possibility of side effects.

Then, for most uses of variables, we check the `tmpToVars` map to see if any copies can be propagated. Function parameters do not participate in copy propagation due to the possibility of side effects.

Below is an example of a simple Decaf program with copy propagation enabled and the associated assembly instruction counts by type. Note that the number of assembly instructions

do not change because we are only changing the variable that is being copied. This enables Dead Code Elimination to remove extraneous assign statements, but CP itself does not affect the generated code size directly. You can run `./run.sh --target cfg --opt cp tests/codegen/input/_cp-<TEST NUMBER>.dcf` to check out the new CFG applied with CP.

Decaf	**Unoptimized**	**Optimized**

`void main(){`
`  int a, b, c, d, x;`
 	
`  a = x;`
`  b = a;`
`  c = b;`
`  d = b + c;`
`}`
	`main() => void {`
	`  _.main_1 @entry @exit`
	`  (a, b, c, d, x)`
	`  > a = 0`
	`  > b = 0`
	`  > c = 0`
	`  > d = 0`
	`  > x = 0`
	`  > a = x`
	`  > b = a`
	`  > c = b`
	`  > _.t = b + c`
	`  > d = _.t`
	`}`
		`main() => void {`
		`  _.main_1 @entry @exit`
		`  (a, b, c, d, x)`
		`  > a = 0`
		`  > b = 0`
		`  > c = 0`
		`  > d = 0`
		`  > x = 0`
		`  > a = x`
		`  > b = x`
		`  > c = x`
		`  > _.t = x + x`
		`  > d = _.t`
		`}`

Count of Assembly Instructions:	`movq: 16`
	`jmp: 2`
	`add: 1`
	`enter: 1`
	`leave: 1`
	`ret: 1`
		`movq: 16`
		`jmp: 2`
		`add: 1`
		`enter: 1`
		`leave: 1`
		`ret: 1`

Constant Propagation (CSP)

Enable this optimization with flag: `--opt csp`

Constant propagation is performed in the same file as copy propagation using almost exactly the same procedure, except with constants. If variable is assigned to a constant, it is added to the `tmpToVars` map, and when the left-hand variable is reassigned, it is removed from the map.

Then, for most `CfgExpression` types, we check the `tmpToVars` map to see if any constants can be propagated. Function parameters and global variables do not participate in constant propagation due to the possibility of side effects.

Below is an example of a simple Decaf program with constant propagation enabled and the associated assembly instruction counts by type. You can run `./run.sh --target cfg --opt csp tests/codegen/input/_cp-<TEST NUMBER>.dcf` to check out the new CFG applied with CSP.

Decaf	**Unoptimized**	**Optimized**

`void main(){`
`  int a, b, c, d;`
 	
`  a = 5;`
`  b = a;`
`  c = b;`
`  d = c + b;`
`}`
	`main() => void {`
	`  _.main_1 @entry @exit`
	`  (a, b, c, d)`
	`  > a = 0`
	`  > b = 0`
	`  > c = 0`
	`  > d = 0`
	`  > a = 5`
	`  > b = a`
	`  > c = b`
	`  > _.t = c + b`
	`  > d = _.t`
	`}`
		`main() => void {`
		`  _.main_1 @entry @exit`
		`  (a, b, c, d)`
		`  > a = 0`
		`  > b = 0`
		`  > c = 0`
		`  > d = 0`
		`  > a = 5`
		`  > b = 5`
		`  > c = 5`
		`  > _.t = 5 + 5`
		`  > d = _.t`
		`}`

Count of Assembly Instructions:	**movq: 14** jmp: 2 add: 1 enter: 1 leave: 1 ret: 1	**movq: 12** jmp: 2 add: 1 enter: 1 leave: 1 ret: 1

Algebraic Simplification

Enable this optimization with flag: `--opt asimple`

Algebraic simplification is implemented as a pass over the CFG. On each pass, it scans for `CfgExpression` that have `CfgValue` that are able to be simplified using simple algebraic rules. If the two sides are of type `CfgImmediate`, the value is computed and returned as another immediate. Otherwise, the following simplification rules are checked:

Algebraic Simplification rules:
`0 + x = x, a + 0 = x`
`0 - x = -x, a - 0 = x`
`x * 1 = x, 1 * x = x, x * 0 = 0, 0 * x = 0`
`x / 1 = x, 0 / x = 0, x / 0 = 0 (undef)`
`x % 1 = 0`
Boolean Simplification rules:
`x && true = x, true && x = x`
`x && false = false, false && x = false`
`x || true = true, true || x = true`
`x || false = x, false || x = false`

Below is an example of the algebraic simplification being done on a simple Decaf program and the associated assembly instruction counts by type.

Decaf	**Unoptimized**	**Optimized**

`void main(){`
`  int a, b;`
`  bool c;`
 	
`  a = 1 + 2;`
`  b = 2 * 5;`
`  c = 10 == b;`
`}`
	`main() => void {`
	`  _.main_1 @entry @exit`
	`  (a, b, c)`
	`  >  a = 0`
	`  >  b = 0`
	`  >  c = 0`
	`  >  _.t = 1 + 2`
	`  >  a = _.t`
	`  >  _.t_2 = 2 * 5`
	`  >  b = _.t_2`
	`  >  _.t_3 = 10 == b`
	`  >  c = _.t_3`
	`}`
		`main() => void {`
		`  _.main_1 @entry @exit`
		`  (a, b, c)`
		`  >  a = 0`
		`  >  b = 0`
		`  >  c = 0`
		`  >  _.t = 3`
		`  >  a = _.t`
		`  >  _.t_2 = 10`
		`  >  b = _.t_2`
		`  >  _.t_3 = 10 == b`
		`  >  c = _.t_3`
		`}`

Count of Assembly Instructions:	**movq: 18** jmp: 2 ~~**add: 1**~~ cmove: 1 cmpq: 1 enter: 1 ~~**imul: 1**~~ leave: 1 ret: 1	**movq: 12** jmp: 2 cmove: 1 cmpq: 1 enter: 1 leave: 1 ret: 1

Dead Code Elimination (DCE)

Enable this optimization with flag: `--opt dce`

Dead Code elimination is implemented on the function scope level. Within this scope, it first passes over all blocks and computes the variables that are assigned to, and (upwards) used within each block. Then, it traverses the control flow graph to compute liveness as in class (though it differs slightly from the slides in that it does account for global variables). Then, it passes over all blocks and eliminates the dead code using the liveness information. There is a special caveat that dead code where variables are assigned to function calls are not eliminated, but just converted to the function call without the assignment since the function may have side effects.

Finally, it repeats the above steps once again until the code stops changing, since it is possible that eliminating some dead code exposes more dead code. There is an optimization possible here in terms of performance- I had an idea where the liveness computation could be modified so as to focus only on the “essential” statements i.e. returns and function calls- but this was not done due to time constraints.

Perhaps the next step to further improve the DCE step would be to implement a heuristic which determines whether a Decaf function is pure or not- currently DCE assumes all function calls

may have side effects and thus does not eliminate some dead code containing pure function calls.

Below is an example of DCE working on a simple Decaf program:

Decaf	**Unoptimized**	**Optimized**

`import printf;`
`int get_int ( int x ) {`
`  return x;`
`}`
`void main ( ) {`
`  int a, b, c, d, e;`
`  a = get_int ( 2 );`
`  b = get_int ( 3 );`
`  c = a + b;`
`  d = c + b;`
`  e = a * b;`
`  printf ( "%d\n", e );`
`}`
	`main() => void {`
	`  _.main_1 @entry @exit`
	`  (a, b, c, d, e)`
	`  > a = 0`
	`  > b = 0`
	`  > c = 0`
	`  > d = 0`
	`  > e = 0`
	`  > _.t = get_int(2)`
	`  > a = _.t`
	`  > _.t_2 = get_int(3)`
	`  > b = _.t_2`
	`  > _.t_3 = a + b`
	`  > c = _.t_3`
	`  > _.t_4 = c + b`
	`  > d = _.t_4`
	`  > _.t_5 = a * b`
	`  > e = _.t_5`
	`  > printf(_.str_1, e)`
	`}`
		`main() => void {`
		`  _.main_1 @entry @exit`
		`  (a, b, c, d, e)`
		
		`  > _.t = get_int(2)`
		`  > a = _.t`
		`  > _.t_2 = get_int(3)`
		`  > b = _.t_2`
		
		`  > _.t_5 = a * b`
		`  > e = _.t_5`
		`  > printf(_.str_1, e)`
		`}`

Count of Assembly Instructions:	**movq: 30** jmp: 4 call: 3 ~~**add: 2**~~ enter: 2 leave: 2 ret: 2 imul: 1 leaq: 1	**movq: 17** jmp: 4 call: 3 enter: 2 leave: 2 ret: 2 imul: 1 leaq: 1

Difficulties with Phase 4

Known Problems

- Global copy and constant propagation do not work properly at merge points. The temporary fix is to disable copy/constant propagation between blocks. We ran out of time to fix it as it involves implementing splitting.

Phase 3 Fixes

Phase 3 Private Tests Fixes

1. We failed to consider that some declarations are re-assigned on every iteration. For example, declarations inside of `while` and `for` loops.

   We fixed this by separating declarations from assignments, so that they are reassigned on each iteration in a CfgAssignStatement

2. There was an assembly bug with division and modulo with the instructions generated.

   We needed to use the cqto assembly instruction because we needed to sign extend %rax to the octoword %rdx:%rax

Highlight for Feedback

- *Are there any good resources for helping us implement global copy propagation?*

Contributions

In Phase 4, An Bo wrote most of the Common Subexpression Elimination (CSE) optimization and Yoland wrote most of Copy Propagation (CP) and Algebraic Simplification optimizations. An Bo and Yoland both worked together to debug both CP and CSE. Alex wrote most of the Dead Code Elimination code. We collaborated in writing the documentation.