Compiler Design Document

Dataflow Analysis and Optimization

Team: Human suffering (@`ajyliew`, @`emshen`, @`skoppula`, @`srobin`)

This design document covers the implementation details of our Decaf compiler's new features of data flow analysis on the CFG and implementation of our optimizations. In our checkpoint document, we described our CFG generation, data flow, and CSE optimizations. We add onto that document here, describing four main categories of additional optimizations that we've made: dead code elimination, primitive register allocation, assembly usage optimizations, and poor man's copy propagation. Our four main high-level sections are organized accordingly. Throughout the document, we also highlight the structure of our codebase, and when relevant, specific parts of our compiler's source, `Graphviz` visualizations of our CFGs, and `x86_64` assembly.

As a note to the reader, all CFG visualizations can be reproduced by running our compiler with the `--dot` flag. For example,

```
./run.sh --debug --target=assembly --dot --opt=cse,dce -o expr.dot 02-expr.dcf
```

The set of optimization flags that are available (and subsequently describe) are:

```
{cse, dce, copy, cfg, regalloc}
```

Dead Code Elimination and Liveness (`dce`)

The notion of the *`in`*-map and and *`out`*-map set for every basic block is similar to what is described in our documentation of our CSE analysis, but now our analogue to the *`in`* `and` *`out`* bitvector is a set of live temporary variables[1]. We chose liveness to implement after CSE, because a large number of optimizations we were interested in required implementing def-use chains, or similar concepts.[2]

Since liveness is performed bottom-up, when we processed a basic block we initialize a basic block's *`out`* set of live variables with its parent's *`in`* set of live variables, and then iterated through the list of TACs backwards: we first determined the definition set (defined by the set of variable defines based on the specific TACs) and then took the set difference:

```
live variables - definition set
```

[1] By variable, we implicitly refer to the tuple of variable name and scope, represented as the hash of the containing symbol table object.

[2] For example, we were keen on implementing register allocation, and register allocation requires the creation of webs that track variable definitions and usages across TACs. We were also interested in copy propagation, and this required knowing whether a copy could be deleted by a certain point in time. These two optimization could provide potential memory-access savings that we hoped to realize.

to determine the set of live variables at the start of the basic block. Then, we determined the final *in* live variable set for the block by computing the blocks' set of variable uses and then taking the set union:

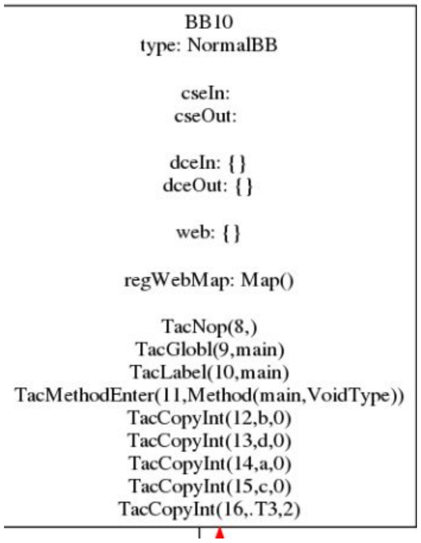
`live variables (union) use set`

to determine the set of live variables. These two relations more generally define the *gen and kill,* of our analysis and is analogous to what was presented in class for liveness. A fixed point algorithm is used to determine the *in* set and *out* set of live variables for every basic block.

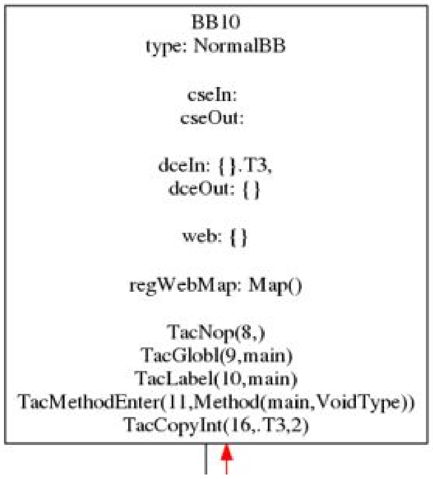
While we are performing this liveness analysis, if we find that a TAC defines a variable that is dead, then we do not process either the definition or use set of the TAC. This is equivalent to treating the TAC as though we have deleted it.

One important point is that we take a conservative approach to liveness checking with regards global variables. We assume that any method call that is not a callout will make all global variables live again, so we do not over-aggressively delete TACs that should not be deleted. We considered trying to determine whether the effect a method call would have on liveness but that significantly complicates the analysis (consider the case of recursive calls or two methods calling each other, for example - determining the set of globals that are modified would require at least one extra fixed point implementation).

To extend liveness to dead code elimination, we simply checked if the variable is live or not for a particular TAC that defines the variable. If not, we delete the TAC. When we are doing this, we update the set of live variables one TAC at a time, as before, to ensure that we are performing intra-block dead code elimination. The resulting set of live variables ends up being the same as the basic block's *in* set, calculated from the fix point across multiple basic blocks.



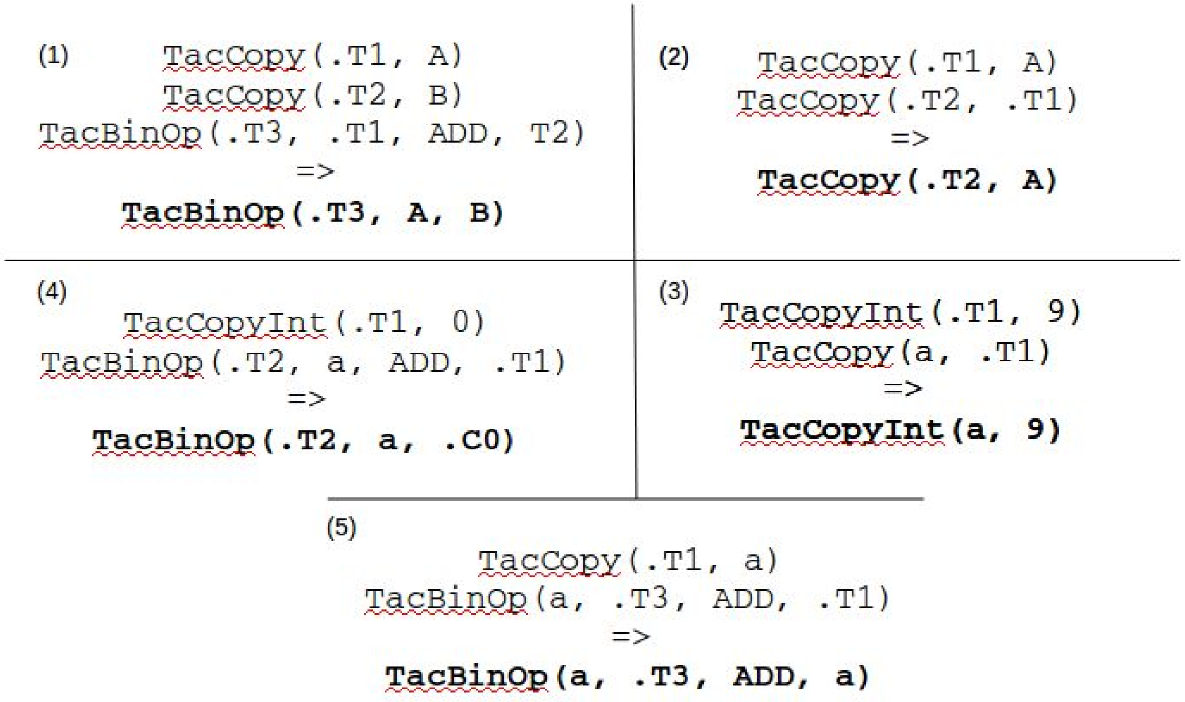
```
BB10
type: NormalBB

cseIn:
cseOut:

dceIn: { }
dceOut: { }

web: { }

regWebMap: Map()

TacNop(8,)
TacGlobl(9,main)
TacLabel(10,main)
TacMethodEnter(11,Method(main,VoidType))
TacCopyInt(12,b,0)
TacCopyInt(13,d,0)
TacCopyInt(14,a,0)
TacCopyInt(15,c,0)
TacCopyInt(16,.T3,2)
```



```
BB10
type: NormalBB

cseIn:
cseOut:

dceIn: { }.T3,
dceOut: { }

web: { }

regWebMap: Map()

TacNop(8,)
TacGlobl(9,main)
TacLabel(10,main)
TacMethodEnter(11,Method(main,VoidType))
TacCopyInt(16,.T3,2)
```

In the CFG output above, we see part of the effects of applying dead code elimination. This example is run on `tests/dataflow/input/cse-01.dcf`; the CFG to the left is without DCE optimization, and to the CFG to the right with the optimization. Every local field initialization requires initialization to zero as per the Decaf; however with DCE, many of these initializations are never used, and thus eliminated

Micro-Optimizations (Poor Man's Copy Propagation) (`copy`)
Taking time to view our CFG in Graphviz allowed us to notice that in many large programs and benchmarks, there were glaring inefficiencies in our data access patterns. Most notably, excess temporary variables were being created for single variable location accesses.



(1)
```
TacCopy(.T1, A)
TacCopy(.T2, B)
TacBinOp(.T3, .T1, ADD, T2)
=>
TacBinOp(.T3, A, B)
```

(2)
```
TacCopy(.T1, A)
TacCopy(.T2, .T1)
=>
TacCopy(.T2, A)
```

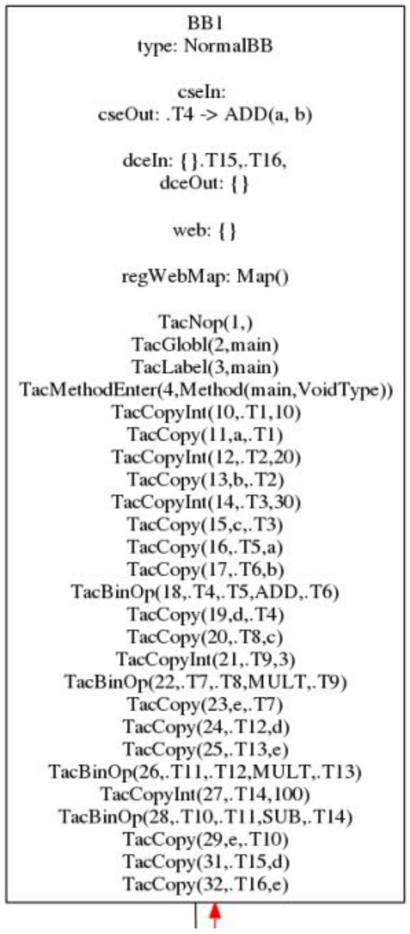
(4)
```
TacCopyInt(.T1, 0)
TacBinOp(.T2, a, ADD, .T1)
=>
TacBinOp(.T2, a, .C0)
```

(3)
```
TacCopyInt(.T1, 9)
TacCopy(a, .T1)
=>
TacCopyInt(a, 9)
```

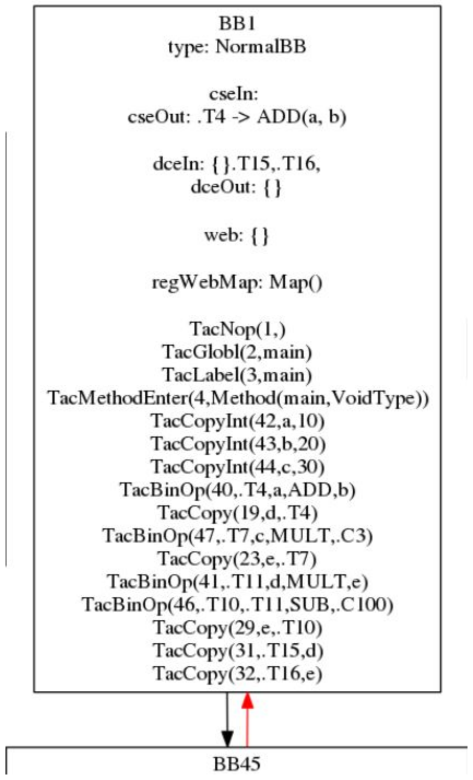
(5)
```
TacCopy(.T1, a)
TacBinOp(a, .T3, ADD, .T1)
=>
TacBinOp(a, .T3, ADD, a)
```

In the five examples above, the temporary used to hold the constant or symbolic variable is never actually needed, and we are able to reduce one memory operations. These five cases predominated the excessive temporary and copy usage that we noticed in our program. These four cases were simple to correct by pattern matching on the TACs in the CFG, and better time investment to implement than general copy propagation, because of the effectiveness of this simple heuristic in practice and in the benchmarks.

An example of these set of five optimizations in practice are given in the following CFGs output by our program running the CSE example `tests/codegen/input/02-expr.dcf`:



```
BB1
type: NormalBB

cseIn:
cseOut: .T4 -> ADD(a, b)

dceIn: {}.T15,.T16,
dceOut: {}

web: {}

regWebMap: Map()

TacNop(1,)
TacGlobl(2,main)
TacLabel(3,main)
TacMethodEnter(4,Method(main,VoidType))
TacCopyInt(10,.T1,10)
TacCopy(11,a,.T1)
TacCopyInt(12,.T2,20)
TacCopy(13,b,.T2)
TacCopyInt(14,.T3,30)
TacCopy(15,c,.T3)
TacCopy(16,.T5,a)
TacCopy(17,.T6,b)
TacBinOp(18,.T4,.T5,ADD,.T6)
TacCopy(19,d,.T4)
TacCopy(20,.T8,c)
TacCopyInt(21,.T9,3)
TacBinOp(22,.T7,.T8,MULT,.T9)
TacCopy(23,e,.T7)
TacCopy(24,.T12,d)
TacCopy(25,.T13,e)
TacBinOp(26,.T11,.T12,MULT,.T13)
TacCopyInt(27,.T14,100)
TacBinOp(28,.T10,.T11,SUB,.T14)
TacCopy(29,e,.T10)
TacCopy(31,.T15,d)
TacCopy(32,.T16,e)
```



```
BB1
type: NormalBB

cseIn:
cseOut: .T4 -> ADD(a, b)

dceIn: {}.T15,.T16,
dceOut: {}

web: {}

regWebMap: Map()

TacNop(1,)
TacGlobl(2,main)
TacLabel(3,main)
TacMethodEnter(4,Method(main,VoidType))
TacCopyInt(42,a,10)
TacCopyInt(43,b,20)
TacCopyInt(44,c,30)
TacBinOp(40,.T4,a,ADD,b)
TacCopy(19,d,.T4)
TacBinOp(47,.T7,c,MULT,.C3)
TacCopy(23,e,.T7)
TacBinOp(41,.T11,d,MULT,e)
TacBinOp(46,.T10,.T11,SUB,.C100)
TacCopy(29,e,.T10)
TacCopy(31,.T15,d)
TacCopy(32,.T16,e)

BB45
```

To the left are the instructions without Poor Man's Copy Propagation. To the right, notice the elimination of many temporaries before each `TacBinOp`, buying us significant runtime savings

Assembly Level Optimizations (`cfg`)[3]

In our unoptimized code generation, for a given ALU operation we would get addresses to temporary variables and then load the values into registers, perform the ALU operation, then store the result into the resulting temporary variable. To provide support for other optimizations and as a result of performing micro-optimizations, the Tac to Assembly generation methods were modified to support register and constant arguments instead of only temporary variable addresses. The benefits of such a change are described in the following assembly level optimizations.

Relational Operation Calculations

[3] Most of the assembly level operations affect both our unoptimized and optimized assembly generation, so these optimizations are not reflected in our constant factor speed-up. The `cfg` flag enable fast array bound checking.

Previously for relational operations, we would perform two conditional moves to determine the result and store it into a temporary variable as either “1” or “0” to represent true and false respectively. Instead, after performing a cmp instruction now we use

```
setcc, %al
```

To conditionally set the `%al` register to 1 or 0 based on whether or not the condition cc is true. We then sign extend using

```
movzx %al %r11
```

To obtain the equivalent result of the relational operation. We now only have one instruction whose result is conditional on the condition flags, and that instruction performs on small 8-bit data. The other instruction is sign extension to a full quadword, which is also not as time consuming as the previous two conditional moves that both operated on quadword data.

Array Bounds Checking
Previously we would put the constants 0 and the size of a array into separate temporary variables on the stack, and then these values would get loaded into registers from the stack to perform the relational operation that checks our index is within bounds. Instead of all this mess, we can instead omit creating these temporary variables, and then just directly perform the relational operation with the constant to perform the desired check. Since we are doing comparisons with constants, there is clearly no violation of correctness. We also now cut down on unnecessary memory accesses.
For example:

```
movq  $0, -16(%rbp) // temporary that holds zero

movq  -16(%rbp), %r10
movq  -104(%rbp), %r11 // index variable
cmpq  %r11, %r10
setl  %al
movzx   %al, %r11
movq  %r11, -112(%rbp)
```

now becomes

```
movq  -104(%rbp), %r11 // index variable
cmpq  %r11, $0
setl  %al
movzx   %al, %r11
movq  %r11, -112(%rbp)
```

Constant/Register Arguments Simplification
Aside from providing support for constant and register arguments in our TACs, we can also perform some assembly level optimizations on them. Normally we spend time loading values

from the stack into registers, performing an operation, and then storing that result into a temporary variable that lives on the stack. Now after register allocation, we may have the scenario where our destination argument is a register, and so we can simply perform the operations using the destination register and just leave the result inside the destination register. For example,

```
TacBinOp(%rax, %rax, ADD, 5)
```

would originally have become:

```
movq %rax. %r10
movq $5, %r11
addq %r10, %r11
movq %r11, %rax
```

It now instead becomes:

```
addq  $5, %rax
```

Register Allocation (`regalloc`)

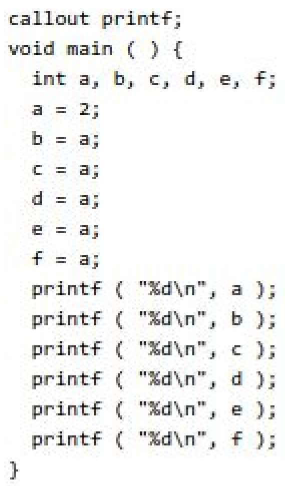
Our compiler makes use of free registers by using a standard register allocation algorithm. Due to time constraints, this was implemented relative to only TACs within the same basic blocks, since this did not require us to follow a fixed point algorithm. Instead, we can simply iterate through each basic block in our CFG, and begin by creating a series of def-use chains which contain start and end indexes of the associated variables. These start and end indexes map to the list of TACs contained within the associated basic block. Def-use chains are then condensed into webs so that we can allocate additional registers.

For register allocation, we are using a greedy algorithm that prioritizes webs based on their usage density. Usage density is calculated by dividing the number of TACs contained in the web by the number of total uses. We thus assign free registers to webs with the highest usage density first, while also reassigning the same register to webs that do not conflict with previous assignments of the same register. If there are no registers that can be assigned to a web without conflict, we resort to spilling onto the stack. Furthermore, because of the necessary overhead of copying into a register, and restoring a register after the web is finished, we first filter the webs based on number of total usages so that this overhead of storing in a register is offset by the speed gains by using register operations.

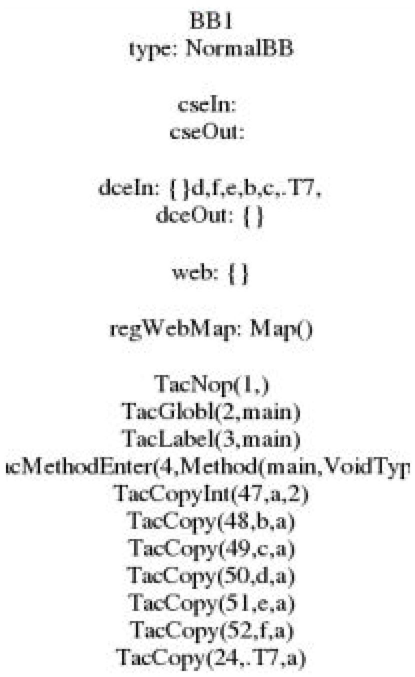
For the webs that do have free registers assigned, we must regenerate our TAC list to represent use of these free registers. We apply a simple replacement strategy, iterating over the TACs to replace instances of the memory address with the assigned register. There are a few stumbling blocks we had to overcome: a few include dealing with shifting def-use positions as we add TacCopy instructions to copy into and save from registers at the ends of def-use chains.

Another complexity we wrestled with was preserving these registers across method calls - we had passed all the code-gen and dataflow hidden tests without returning these registers to their original value, and passed all test cases. We realized we were using callee-saved registers only after failing a few of the optimization tests.

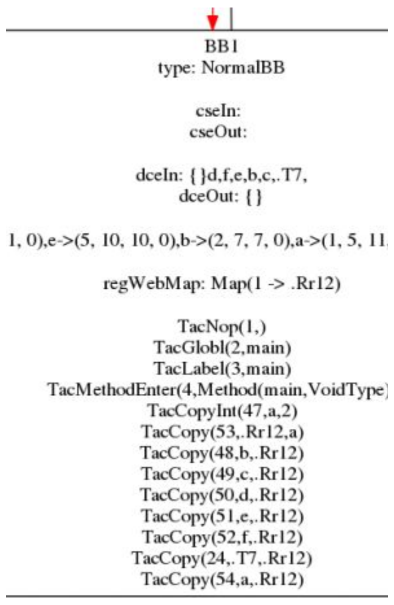
Here is a hopefully lucid demonstration of the effect register allocation has on speeding up instruction execution; costly memory read operations are replaced with register lookups. We can see this in the changing Three Address Codes in the program's CFG graph when running a sample program (`examples/regalloc.dcf`, pasted on the right) to illustrate the register allocation.



```
callout printf;
void main ( ) {
  int a, b, c, d, e, f;
  a = 2;
  b = a;
  c = a;
  d = a;
  e = a;
  f = a;
  printf ( "%d\n", a );
  printf ( "%d\n", b );
  printf ( "%d\n", c );
  printf ( "%d\n", d );
  printf ( "%d\n", e );
  printf ( "%d\n", f );
}
```



```
BB1
type: NormalBB

cseIn:
cseOut:

dceIn: { }d,f,e,b,c,.T7,
dceOut: { }

web: { }

regWebMap: Map()

TacNop(1,)
TacGlobl(2,main)
TacLabel(3,main)
cMethodEnter(4,Method(main,VoidTyp
TacCopyInt(47,a,2)
TacCopy(48,b,a)
TacCopy(49,c,a)
TacCopy(50,d,a)
TacCopy(51,e,a)
TacCopy(52,f,a)
TacCopy(24,.T7,a)
```



```
BB1
type: NormalBB

cseIn:
cseOut:

dceIn: { }d,f,e,b,c,.T7,
dceOut: { }

1, 0),e->(5, 10, 10, 0),b->(2, 7, 7, 0),a->(1, 5, 11

regWebMap: Map(1 -> .Rr12)

TacNop(1,)
TacGlobl(2,main)
TacLabel(3,main)
TacMethodEnter(4,Method(main,VoidType)
TacCopyInt(47,a,2)
TacCopy(53,.Rr12,a)
TacCopy(48,b,.Rr12)
TacCopy(49,c,.Rr12)
TacCopy(50,d,.Rr12)
TacCopy(51,e,.Rr12)
TacCopy(52,f,.Rr12)
TacCopy(24,.T7,.Rr12)
TacCopy(54,a,.Rr12)
```

To the left, we see copies occuring from a location in memory to a location (`a`) another location in memory (`b,c,d,e,f`). Adding register allocation (on the left) decides that given the high usage of a in the block, it may be worth it to save it in a register throughout its lifetime, and read and write to the assigned register (.R12) instead of the location in memory. Notice as well the additional overhead introduced by the TacCopy to load and restore the register.

Interestingly, we had to straighten out a few non-trivial overhauls to our unoptimized framework in order to support register allocation and a few other optimizations that used immediates (for example, our assembly optimizations). In addition to passing around memory addresses to symbolic and temporary variables, our assembly generation now had to support direct manipulation (in our TACs) of registers (denoted with convention “`.RXX`” in our compiler) and immediates (denoted with convention “`.CXX`” in our compiler).

Optimization Order

You can see our post-CFG generation optimizations applied in order in `Compiler.scala` from lines 136 to 233. We perform the following analysis/optimizations in the order: availability analysis, common subexpression elimination, liveness analysis, dead code elimination, poor man’s copy propagation, register allocation, and finally, assembly generation optimizations. Other optimization orders are possible, and in theory, it is possible to repeat procedures (e.g. performing dead code elimination multiple times), but we found that this did not significantly help our program runtimes.

More importantly, because we implemented availability and liveness analysis first, many of our subsequent optimizations depend on the existence of fixed point data from these procedures. For example, to check whether we can eliminate a TacCopy in poor man’s copy propagation, we refer to the liveness state, and functions we wrote to determine def-and-use, to determine whether removing a copy would affect correctness. These sort of dependencies limit the order in which we can feasibly perform optimizations.

Other Optimizations?

There were other optimizations we explored with implementing after reviewing the optimizer test cases. Specifically, we noticed that nearly all derby challenges ran iterated for-loops, that had few, if any, loop dependencies. We were interested in implementing loop hoisting and parallelization if we had more time; these would likely be very beneficial for our benchmark. The day before the derby program was due, we created a rough outline on how to integrate very basic multi-threaded code parallelization with the given library, but did not have time to implement this, unfortunately. Given that memory accesses are the largest part of our program latency, adapting our register algorithm to be more than intra-block would also likely result in benchmark improvements. Other optimizations, such as peephole, may also provide improvements, but due to the large number of saved memory accesses in our chosen optimizations, these seemed to us to be the most complexity and time efficient to implement.