Contents

Project Phase 3 Code Quality Analysis

Abhijit Singh asingh811@gatech.edu

July 2021

1 About this project

This report analyses the overall performance of various allocation methods described in the Design Internals of this project.

2 The basics

We implement two allocation algorithms: block and naive. Naive simply loads and stores all vars into and from registers per instruction On the other hand, Block tries to load in as many vars in advance as possible per block Unfortunately, due to time constraints, the the Briggs coloring algorithm was not implemented in time.

3 Expected behaviors

3.1 Naive algorithm

We expect this to on average produce 7 times the number of variables per line of IR code. This number should remain consistent regardless of how large code blocks, functions are. Generally speaking, it would do the following

1)Load Variables : On average 3 vars per instruction of the form a = b + c (the most common type of instruction)

2) Instruction itself: Instructions like addition have a one to one mapping to the IR code. However, certain branch instructions require multiple extra instructions to be generated

3) Store Variables: store all the variables previously loaded (3 on average)

Therefore, expected code size = 3 loads + 1 instruction + 3 stores = 7 x num instructions.

3.2 Block based algorithm

We expect this algorithm to produce far less code that the Naive algorithm in the general case

Say we have N instructions in a block B

Each on average uses 3 variables however, on average, it re-uses R variables already loaded, and 3-R variables is new/unique to the given line

In such a case for block B, we would generate (3-R) times N loads and N stores per block.
However, if we run out of registers, we would need to do individual loads and stores every time per instruction
Say we have X registers, and we reserve P registers for individual loads and stores mentioned above
The total number of expected instructions = N + (min(X-P,((3-R) * N)) + max(0,N-min(X-P,((3-R) * N)))) * 2

Explanation
N is the number of instructions in total
The expression min(X-P,((3-R) * N)) computes number of registers used, upto a maximum of X-P
max(0, N-P- the above expression) gives us the remaining instructions that will have to be manually loaded and stored each time, assuming the "R" words are already loaded into other, permanent block registers

The above expression can be minimised by R= 3 (i.e. when the same 3 vars are used over and over.) Example:

a = b + c
c = b - a
b = b**a
And this can be maximised, if all of the vars are different (R=0)

a = b + c
d = e + f
g = h + i
In the above worst case, we will end up with as many load and stores as the Naive algorithm

Say if we have B blocks with N instructions on average, total program instruction = B(N + (min(X-P,((3-R) * N)) + max(0,N-min(X-P,((3-R) * N)))) * 2)

3.3 A word on the Briggs algorithm

While we did not implement this algorithm, we would expect improvements across block boundaries. Since it uses inter-block register coloring, redundant registers across branches would not have to be stored and loaded at the beginning and the end of the block every time.

In the best case scenario, when there is low live range overlap, we would eliminate the multiple "B" from the basic intra block algorithm, and it would simply become
(N + (min(X-P,((3-R) * N)) + max(0,N-min(X-P,((3-R) * N)))) * 2)

4 Benchmarks

Here we test the above mentioned hypothesis for the algorithm using the -n and -b flags respectively for the two algorithms. We used the SPIM-keepstats utility to count instructions, and used a simple sed call to count the generated code size (in number of lines)

All tested files are included in the appendix

Note that benchmarks such as function_call2 may be skewed since we load and store registers on the stack frequently upon calling and returning from functions

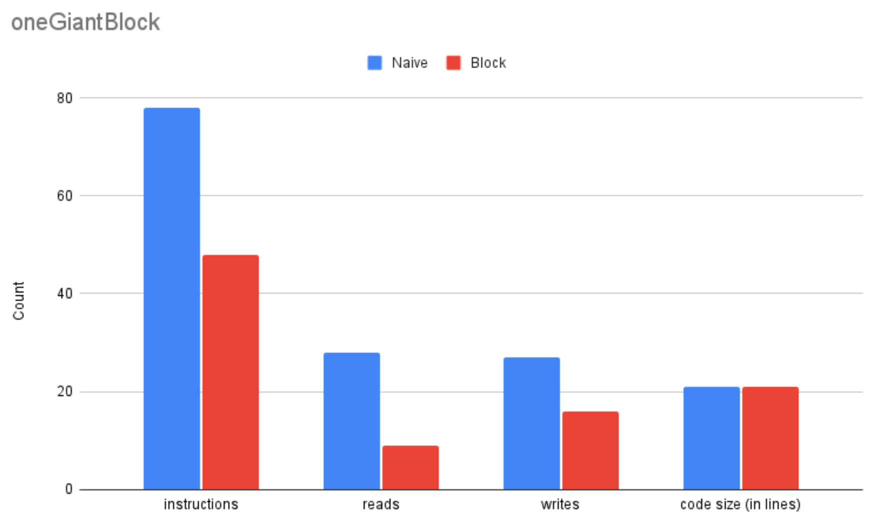

Figure 1: oneGiantBlock.ir: This is for a simple program with one large main procedure (i.e. only one block)

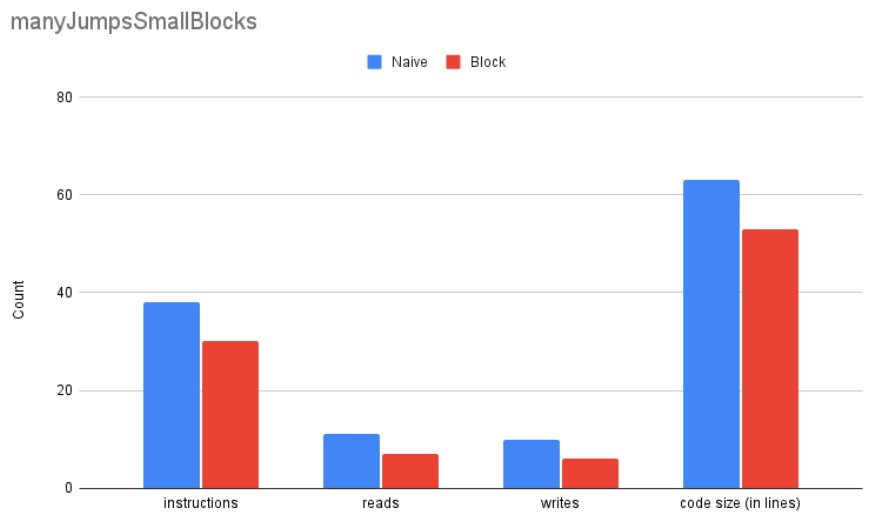

Figure 2: manyJumpsSmallBlocks.ir: The program contains several small blocks

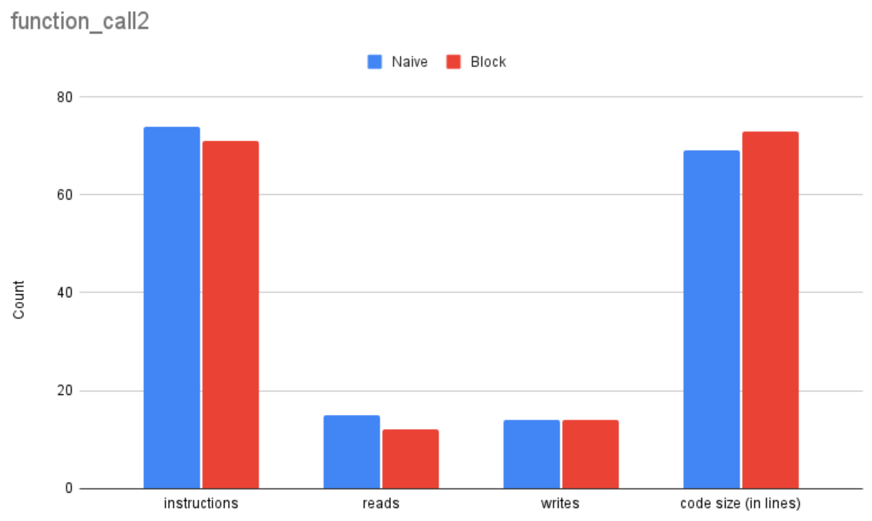

Figure 3: function_call2.ir: Taken from the provided benchmark programs, the program is heavily focused on function calls, and minimal assignment/other instructions

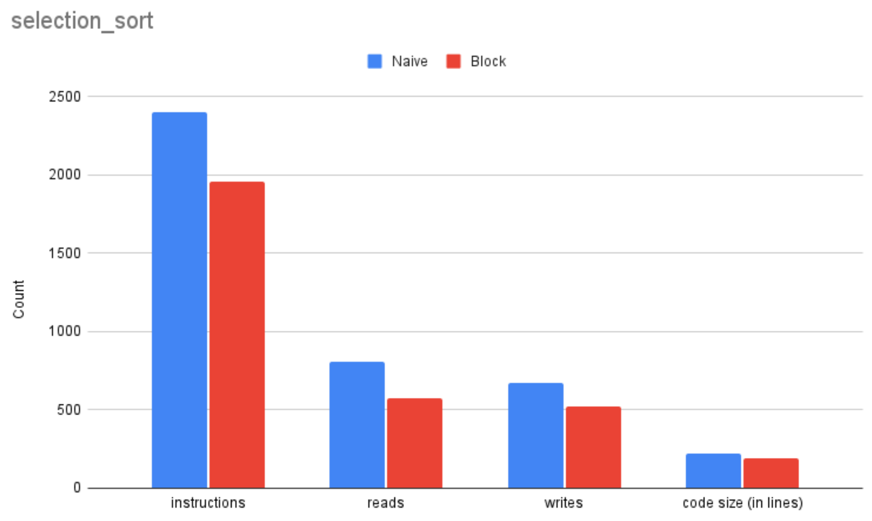

Figure 4: selectionsort.ir : A block heavy program with medium sized blocks

5 Conclusion

Large, few blocks heavily favor the intra block algorithm. OneGiantBlock.ir is a primary example of this fact. The number of executed instructions is halved, and reads/writes are almost a third of naive algorithm

. This would provide a huge boost in performance as reads/writes tend to be far heavier than standard ops

Smaller blocks see little benefit from the intra block allocation. In the pgoram manyJumpsSmallBlocks and function_call2, we see minimal performance benefit from using intra block.

Finally, while code size has less relevance in the 21st century, having a small binary is nonetheless beneficial especially when programming microcontrollers. Large blocks lead to smaller binaries with the block algorithm.

6 Further work

If we were permitted more time, we would implement the briggs algorithm. We can imagine that it would improve performance in a jump-heavy program significantly over the block algorithm.

7 Appendix: Code samples

We use the following sample IR files to illustrate the differences in code generation for the given code generation strategies implemented as part of this project

1) oneGiantBlock.ir

```
start_program oneGiantBlock
static-int-list:

start_function main
int main()
int-list: a, b, c, d, e, f, g, h
   main:
       assign, a, 10
       assign, b, 25
       add, a, b, g
       add, a, g, g
       add, a, g, g
       assign, c, 30
       assign, d, 45
       add, c, d, h
       assign, e, 60
       assign, f, 69
       add, a, b, g
       add, c, d, h
       add, c, c, h
       return ,,,
end_function main

end_program oneGiantBlock
```

Listing 1: oneGiantBlock.ir

2) manyJumpsSmallBlocks.ir

```
start_program manyJumpsSmallBlocks
static-int-list:

start_function main
int main()
int-list: a, b, c
    main:
       assign, a, 10
       assign, b, 20
       add, a, b, c
    _label1:
       assign, b, 10
       assign, a, 20
       add, c, b, a
       goto, _label4,,
    _label2:
       assign, b, 10
       assign, a, 20
       add, c, b, a
       goto, _label1,,
    _label3:
       assign, b, 10
       assign, a, 20
       add, c, b, a
       goto, _label2,,
     _label4:
       return ,,,
end_function main

end_program manyJumpsSmallBlocks
```

Listing 2: manyJumpsSmallBlocks.ir

3) function_call2.ir

```
start_program function_call2
static-int-list:

start_function square
int square(int _2_x)
int-list: _2_x, _t1
   square:
       mult, _2_x, _2_x, _t1
       return, _t1, ,
end_function square

start_function doubleSquare
int doubleSquare(int _3_y)
int-list: _3_y, _t2, _t3
   doubleSquare:
       mult, _3_y, 2, _t2
       assign, _3_y, _t2,
       callr, _t3, square, _3_y
       assign, _3_y, _t3,
       return, _3_y, ,
end_function doubleSquare

start_function main
int main()
int-list: _t0, result
   main:
       callr, _t0, doubleSquare, 10
       assign, result, _t0,
       call, printi, result
       return, 0, ,
end_function main

end_program function_call2
```

Listing 3: function_call2.ir

4) selection_sort.ir

```
start_program selection_sort
static-int-list:

start_function main
  int main()
```

```
int-list: cats[8], index, lowestIndex, subIndex, temp, size, x,
     smallest, first, _t1, _t2, _t3, _t4, _t5, _t6, _t7, _t8,
    _t9, _t10
  main:
      array_store, cats, 0, 7
      array_store, cats, 1, 2
      array_store, cats, 2, 10
      array_store, cats, 3, 20
      array_store, cats, 4, 5
      array_store, cats, 5, 6
      array_store, cats, 6, 44
      array_store, cats, 7, 33
      assign, size, 8,
      assign, index, 0,
  _loop_label_0:
      sub, size, 1, _t1
      brgt, index, _t1, _loop_label_1
      assign, lowestIndex, index,
      add, index, 1, _t2
      assign, subIndex, _t2,
  _loop_label_2:
      sub, size, 1, _t3
      brgt, subIndex, _t3, _loop_label_3
      array_load, _t5, cats, subIndex
      array_load, _t6, cats, lowestIndex
      assign, _t4, 0,
      brgeq, _t5, _t6, _if_label_5
      assign, _t4, 1,
  _if_label_5:
      brneq, _t4, 1, _if_label_4
      assign, lowestIndex, subIndex,
  _if_label_4:
      add, subIndex, 1, subIndex
      goto, _loop_label_2, ,
  _loop_label_3:
      array_load, _t7, cats, index
      assign, first, _t7,
      array_load, _t8, cats, lowestIndex
      assign, smallest, _t8,
      array_store, cats, index, smallest
      array_store, cats, lowestIndex, first
      add, index, 1, index
      goto, _loop_label_0, ,
  _loop_label_1:
      assign, index, 0,
  _loop_label_6:
       sub, size, 1, _t9
       brgt, index, _t9, _loop_label_7
       array_load, _t10, cats, index
       call, printi, _t10
       add, index, 1, index
       goto, _loop_label_6, ,
   _loop_label_7:
       return, 0, ,
end_function main

end_program selection_sort
```

Listing 4: selection_sort.ir