Register Allocation with ILP:

An analysis of optimality

Sean Harrison
CSCI5525
University of Colorado Boulder
Boulder, Colorado, USA
sean.harrison@colorado.edu

Sean Donohoe
CSCI5525
University of Colorado Boulder
Boulder, Colorado, USA
sedo8743@colorado.edu

ABSTRACT

Register allocation is a key optimization stage during compilation. Unfortunately, the problem is also known to be NP-complete, making obtaining an optimal solution incredibly expensive. Approximation algorithms exist in which optimality is sacrificed for speed, and they work very well for the majority of applications. But how much optimality is sacrificed for speed? In this paper, a greedy approximation algorithm is compared to an ILP formulation for understanding loss of optimality. Several optimizations are also proposed for speeding up the ILP formulation, including a new hybrid algorithm which mixes the greedy approximation and the ILP to balance speed and optimality.

ACM Reference Format:
Sean Harrison and Sean Donohoe. 2017. Register Allocation with ILP:: An analysis of optimality. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Register allocation is well known to be NP-complete, making it an incredibly difficult problem to solve. The most well known register allocation algorithms use the variable liveness interference graph framed as a graph coloring problem to tackle the allocation. While algorithms to solve the complete allocation problem exist, they can take an exceptionally long time to find a solution. As a result, most compilers use an approximation algorithm to attempt to color the interference graph.

The greedy register allocation implemented in this course does not produce an optimal solution, but does compute a feasible solution quickly. For the majority of modern applications, losing some optimality in the coloring stage is not seen as detrimental to overall code performance. While this is good for the effectiveness of modern compilers, the sub-optimality of a greedy approximation begs the question "how much worse is greedy than optimal?"

One complete algorithm for register allocation is the integer linear programming formulation of graph coloring. Linear programming can be an effective method of solving some traditionally difficult problems, and it is well known that the graph coloring problem can be expressed as an integer linear program. Researchers in [1] have even determined a formulation for bitwise register allocation in an attempt to further reduce spilling variables. These researchers, much like many others, have found that despite the ability to find an optimal solution with ILP the time to find a solution can be excessive, even with industry-level solvers. While the continuous linear programming problem can be solved in polynomial time using interior point methods, integer linear programming must use the traditional simplex algorithm to find an exact solution. Simplex is an exponential algorithm, and integer linear programming introduces an extra layer of exponential time on top of simplex. Despite this, most problems formulated as integer linear programs run much faster than their worst case runtime, meaning there is potential to find a fast, complete algorithm through ILP. Is it worth even having a faster complete approach, though? If the greedy approximation is consistently close to the optimal solution, then an optimal algorithm may not even be worth exploring.

1.1 Motivation

Embedded electronics are extremely memory limited, so spilling of variables is not always feasible. Aside from the restriction on memory size in these systems, they also often have low-performance memory, making the cost of spilling variables much higher. Since embedded systems are built in large quantities, with small executables pre-loaded, they are well suited for pairing with optimal register allocation algorithms, despite the longer solution times.

1.2 Question

Given that integer linear programming can take a substantial amount of time to find a solution, a natural next question is whether an algorithm can be determined to balance optimality with speed. Speeding up an integer linear program typically involves looking at reducing the problem size and reducing the difficulty of finding a solution.This can be done by reducing the total number of constraints, reducing the total number of variables, or adding constraints which reduce the size of the problem space. The ILP formulation of coloring the interference graph will largely benefit from reducing the complexity of the graph. Should any colorings be determined before the graph coloring stage, they will also assist in making the problem easier to solve by reducing the problem space.

2 ILP FORMULATION

The formulation of the register allocation problem involves slight adjustments to the basic graph coloring problem:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\min \Sigma_i \, spillcost(i) * S_i, \forall i \in variables \quad (1)$$

$$(\Sigma_{colors} \, C_i) + S_i = 1, \forall i \in variables, \forall C \in colors \quad (2)$$

$$C_i + C_j <= 1, \forall (i,j) \in graph, \forall C \in colors \quad (3)$$

$$C_i = 1, \forall (C,i) \in precoloring \quad (4)$$

$$C_i \in 0,1, \forall i \in variables, \forall C \in colors \quad (5)$$

$$S_i \in 0,1, \forall i \in variables \quad (5)$$

where $spillcost(i)$ is the result of analyzing the potential memory cost should variable i be spilled, S_i is a variable indicating whether variable i should be spilled, and C_i is a variable indicating whether variable i should be colored color C.

The objective function (1) states that the goal of the problem is to minimize the number of potential memory accesses produced by the register allocation. S_i will only be set if it is optimal to spill variable i, meaning only the least harmful variables will be spilled.

(2) states that only one color from the set of colors may be chosen for variable i. Furthermore, if variable i is spilled, then i can't be colored with a register color.

(3) states that for every pair of adjacent nodes (i, j) in the interference graph, they cannot be assigned the same color C. The spill variables S_i are not included in this constraint, since it is valid for adjacent vertices to be spilled.

(4) states that for any variable i which has been assigned a coloring C before the formulation, set the corresponding color variable C_i to 1. This will help reduce the problem space, and potentially allow the ILP to determine a solution faster.

Finally, (5) simply enforces that this is a 0-1 ILP by stating that all variables can take the value of 0 or 1.

3 OPTIMIZATIONS

Our initial formulation for ILP was producing a comically high number of constraints, over 100,000 for fairly simple programs. This resulted in ILP taking far too long to produce a solution, making it overly impractical. In order to reduce the constraints and increase the speed of the ILP formulations we introduce following optimizations.

3.1 Double Edge Elimination

The first optimization made for the ILP was to reduce the complexity of the interference graph. The naive interference graph contains double edges between every pair of interfering nodes, meaning redundant constraints are added to the problem. While number of constraints doesn't necessarily correlate with problem complexity, it does increase solve time by the size of the system of equations, and the overall potential number of vertices which could be traversed in the polytopes.

By removing redundant edges between vertices, we can reduce the number of constraints produced by the interference graph by half, therefore potentially reducing the time to find a solution.

3.2 Pre-Coloring Analysis

The next optimization, included in the outlined ILP formulation, was to include pre-colored node assignments. This has the effect of reducing problem complexity by reducing the complexity of the polytopes, or reducing the problem space. In its base implementation, pre-coloring won't provide a huge reduction in problem complexity since nodes are only pre-colored when caller-save registers are included in the live set. If some nodes can be pre-colored outside of live set analysis, then there exists strong potential for this optimization to drastically reduce problem complexity.

3.3 Spill Cost Analysis

The final optimization made was to improve the optimality of the register allocation. In the basic graph coloring formulation, there isn't a need for an objective function since any solution found is valid. With the introduction of spill variables, the objective function becomes minimizing the number of spilled variables generated. While this does attempt to minimize the number of variables that go to memory, there can be multiple solutions which achieve this result. While all of these solutions are optimal with respect to the ILP formulation, they may be suboptimal in the sense that some produce more potential memory accesses. To address this sub-optimality in the base formulation, a new objective function is introduced which attempts to minimize both the number of variables spilled and the number of potential memory accesses resulting from a coloring.

To form this objective function the spill cost, or number of memory accesses resulting from spilling a variable, is calculated for each variable before coloring. The spill variables are then weighted by this cost and the ILP attempts to minimize the sum of the weighted variables.

While this new formulation should produce a more optimal solution, the solution time will likely increase since the problem becomes more difficult to solve.

3.4 Optimization Results

As can be seen in figures 1, 2, and 3 the ILP formulation takes substantially longer than the greedy coloring approach, even with our optimizations. It can also be seen that, as predicted, using static analysis in all but figure 3 minutely slows down the ILP formulation, but also helps to produce more optimal results. It can also be seen that the single best improvement was removing the redundant double edges in the constraints.

4 HYBRID ALGORITHM

It is obvious that using our solver with compiled python is not fast enough to be a reasonable algorithm for register allocation.

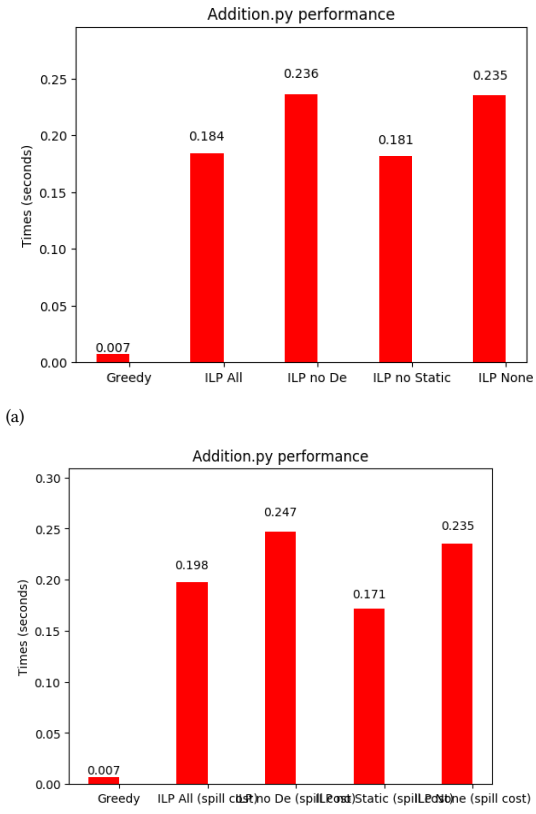


Figure 1: Compilation time for addition based code

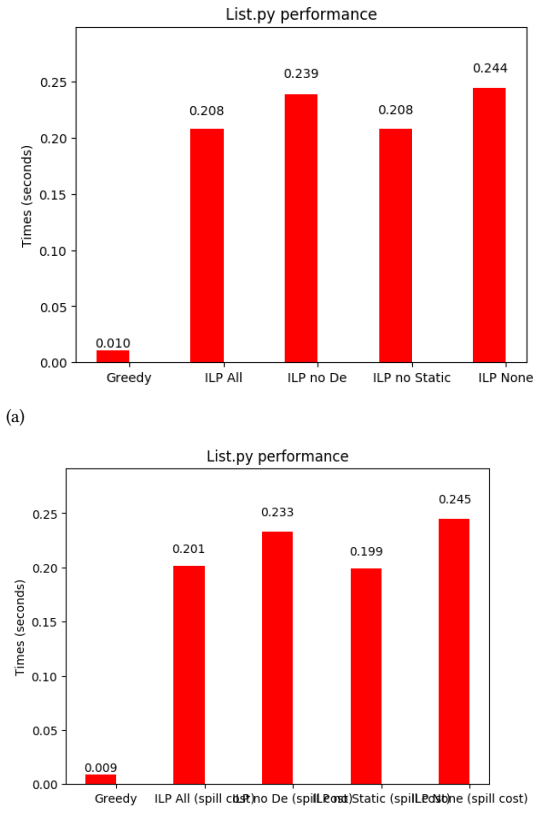


Figure 2: Compilation time for list based code

One potential optimization suggested in [1] was to use a partial solution from the greedy algorithm to suggest solutions to the ILP. What if a solution was obtained from the greedy algorithm and mixed with a solution from ILP? How much does the run time improve? Will optimality be drastically affected? We introduce a hybrid algorithm which attempts to balance solutions obtained by the greedy algorithm and ILP formulation in order to speed up the solution time.

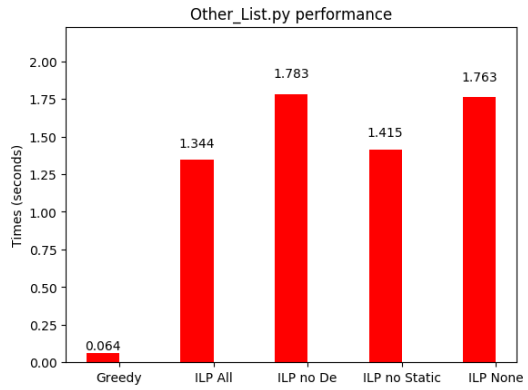
4.1 Implementation

The first stage of the hybrid algorithm is to run a single pass of the greedy algorithm and use the assignments as a guess as to what colorings might work for the program. The algorithm assigns these guesses a probability of being correct. If an assignment's probability is lower than some threshold, then the coloring is dropped, otherwise it is kept as a correct coloring. These pre-colorings are fed into the ILP formulation to help ILP find a solution faster. The use of probabilities obtained from the greedy approach does imply a potential for suboptimal solutions, but does speed up the calculation. In order to see what sort of trade off we were obtaining between optimality and speed we ran the hybrid approach with the probability weight applied to the greedy guess between 25%*and*75%

4.2 Results

4.2.1 Compile Speed. As can be seen in figures 4 and 5 utilizing the hybrid approach resulted in a pretty significant speed up of around 25% over the optimized ILP formulation. The hybrid is still substantially slower than the greedy approach however, as can be seen in the memory utilization subsection, on average it also achieves less memory accesses. Also, as we increased the weighted percent on the greedy guess, in general, we decreased the compilation time, but the more we weighted the greedy approach the more likely we were to drift from the optimal coloring.

4.2.2 Memory utilization. Figures 6, 7, and 8 show the memory accesses of the base ILP formulation, best case greedy, worst case greedy, average case greedy, and hybrid ILP ranging from 25%*to*75%. In both figures 6 and 7 all variants of ILP and hybrid ILP beat out the worst and average case of greedy, but in figure 8 hybrid-75 performed substantially worse than even the worst case of greedy. Also in figure 7 hybrid-75 may have beaten greedy, but it performed noticeably worse than every other variant of the hybrid and the base ILP formulation. In consideration of this poor memory performance of hybrid-75 compared to its minimal compilation time speed up compared to hybrid-50 we concluded that hybrid-50 was the best trade off between performance speed up and spilled memory optimality.



(a)

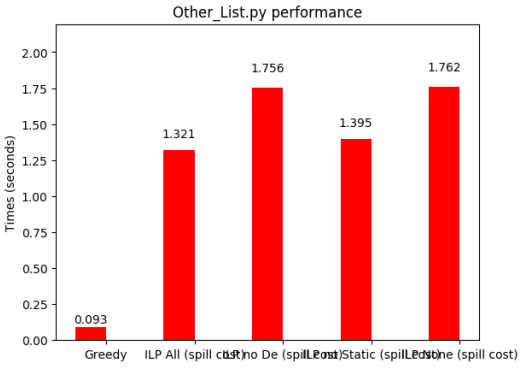


Figure 3: Compilation time for other list based code

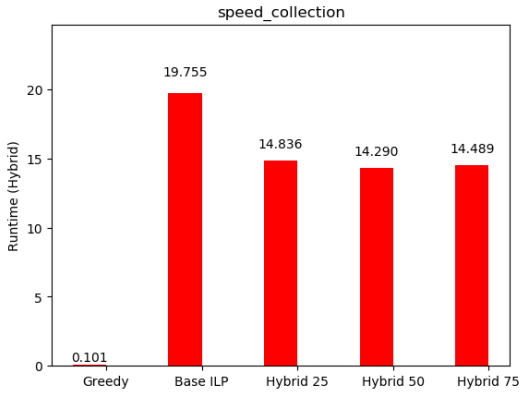


Figure 4: Compilation time for while loop with various weight percentages in the hybrid implementation

5 BOTTLENECKS

The biggest bottleneck encountered was a result of compiling python. Explication and flattening introduced so many temporary variables that the number number of constraints exploded in size. The researchers in **(author?)** [1] used ILP on languages with explicit typing such as C, meaning less temporary variables and less constraints. Other bottlenecks encountered include only using a simple ILP solver that can only run on a single thread. The researchers in [1] used a commercial grade solver that could run parallelized ILP iterations, making the run times much faster.

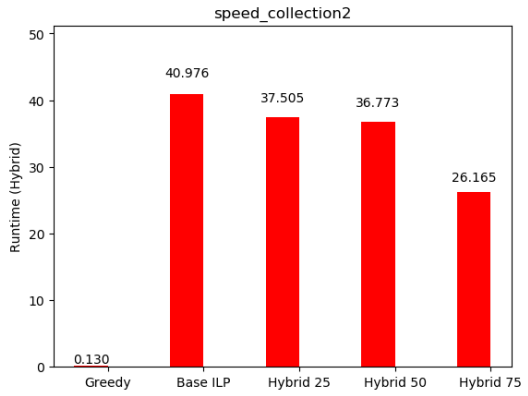


Figure 5: Compilation time for function code with various weight percentages in the hybrid implementation

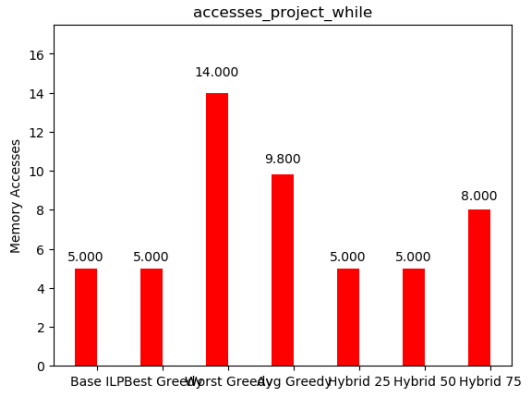


Figure 6: Average memory access over 5 runs for while loop code

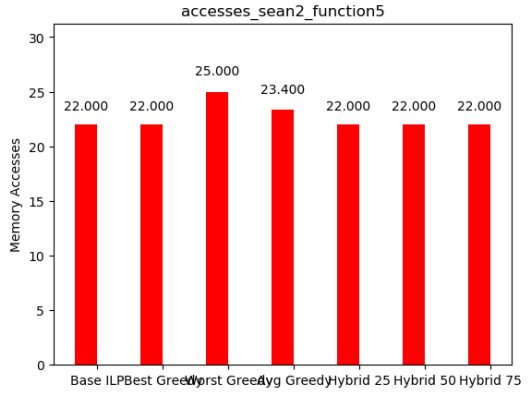


Figure 7: Average memory access over 5 runs for function code

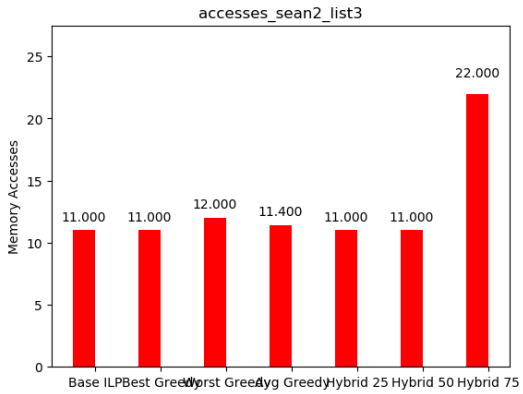


Figure 8: Average memory access over 5 runs for list code

6 FUTURE WORK

There are two avenues we would like to pursue if we were to take this project further.

6.1 Coalescence

One such avenue would be to do static analysis to perform variable coalescence. Variable coalescence is using the interference graph to determine what variables can be assigned to the same register without actually assigning the variables to a register. This allows for substantially less constraints to be passed into the ILP formulation. Variable coalescence also has the added benefit of identifying which variables will need to be spilled before performing the ILP computations, allowing ILP to be computed in a single pass, instead of the multiple we are doing now to allow variable spillage [2].

6.2 True Optimality

Another extension that could be made to this algorithm would be to include a complete, one-pass formulation. The algorithm as presented iterates through multiple stages of solutions: coloring, spilling, adjusting with temporaries, repeat until fixed. As a result, there is potential for the presented formulation to produce suboptimal results, as the variables chosen to be spilled each iteration might introduce more temporaries or memory accesses than needed.

Rather than running multiple iterations of the coloring, the possible temporary variables resulting from spilling could be added to the ILP formulation, allowing for the determination of an optimal coloring in one pass. Introducing potential temporary variables into the formulation involves building a worst-case interference graph, or one in which all variables are assumed to be spilled. All possible temporary variables are therefore included in the ILP formulation. The following constraints can then be added to the formulation to indicate whether a temporary variable is actually introduced in the final coloring:

$(S_i + S_j) - 1 <= T_ij, \forall (i,j)$ resulting in temporary T_ij when (i,j) spilled

Here T_ij is a variable indicating whether the temporary resulting from spilling variable i and variable j is introduced to the program. The objective function becomes

$$\min (\Sigma\ spillcost(i) * S_i) + (\Sigma\ T_ij)$$

The objective function therefore tries to both minimize the spill cost of the coloring, and the number of newly introduced temporary variables.

While this formulation is guaranteed to produce optimal results, the need to operate on the worst-case interference graph means the number of constraints and variables will explode. Given how long it takes to run the presented ILP, this formulation was not tested. However, should enough hardware and software resources exist, this formulation would provide a much better guarantee for minimizing memory usage.

7 CONCLUSION

While many modern computers have the resources available for suboptimal memory allocations during compilation, many embedded devices can benefit from an optimal register allocation. This can come at a significant computational cost, however, as optimal allocation is an NP-complete problem. The ILP formulation proposed provides an optimal allocation algorithm. The formulation speed was improved through some optimizations making the interference graph less complex, and by reducing the solution space through the use of greedy hints in a hybrid algorithm. While the solution times of the improved algorithms are still drastically outmatched by the greedy algorithm, the optimizations provide a fairly significant performance improvement over the base ILP formulation. This suggests that there are potentially more properties of interference graphs that could be exploited to make the solution time faster. Paired with our hybrid algorithm and complete formulation, further optimizations and better solvers could provide programmers with a fast, optimal register allocation algorithm.

REFERENCES

[1] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, Raghavendra Udupa. Optimal Bitwise Register Allocation using Integer Linear Programming. https://grothoff.org/christian/lcpc2006.pdf

[2] âĂIJAn Example of Coloring with Precolored Nodes.âĂİ Princeton University, The Trustees of Princeton University, www.cs.princeton.edu/ appel/modern/text/color1.html.