Register Allocation by Pseudo-Boolean Optimization

Register Allocation by Pseudo-Boolean Optimization

A thesis presented
by
Michael Robert Horton
to
The Department of Computer Science

in partial fulfillment of the requirements
for the degree of
Bachelor of Arts with Honors

Harvard University
Cambridge, Massachusetts
April 2020

Thesis advisor: Professor Stephen Chong Michael Robert Horton

Register Allocation by Pseudo-Boolean Optimization

Abstract

Register Allocation is a part of the compilation process in which virtual registers are mapped to a physical memory location. This problem is NP-hard yet also an integral part of the compilation process which has led to the development of various efficient heuristics for achieving near optimal allocations. These heuristics are used in modern compilers such as gcc and clang. In this thesis, we investigate register allocation via the solving a pseudo-Boolean optimization (PBO) problem.

We develop a suitable encoding of Register Allocation as a PBO Problem. This PBO Problem has constraints relating to virtual and physical register liveness and a cost model developed to find not just a correct register allocation, but the optimal register allocation with respect to number of spills and expected program execution. We allow the PBO Solver to run for a specified amount of time (the longest trial being 6 hours per function) before the most optimal solution to the PBO problem found is used to perform register allocation. We find that the heuristics are able to outperform the PBO Register Allocator. However, we find optimal solutions to 81% of functions in under 65 seconds and have runtimes within 3% of industry strength register allocation methods. Further work will add live-range splitting to the PBO Register Allocator which should improve performance against the heuristics.

Contents

Acknowledgments

I begin by thanking the Harvard Glee Club for spreading their glee, good humor, and joy even in these difficult times. I thank MD 309, PL @ Harvard, and the Chong Group for their continued support of my endeavors and for welcoming me into their lab. I thank David Parkes and Mike Smith for agreeing to read my thesis. I thank all of my friends who have supported me along the way. I thank Amy Gu for all the inspiration, support, and time she has given me in this and all my efforts. I am truly grateful for my advisor, Stephen Chong, who has advised me over the last year. Thank you for believing in me and pushing me to greater heights. Finally, I thank my mother and father who have encouraged and inspired me from the very beginning.

0 Introduction

The semi-serious Proebsting's Law states that compiler optimization is a field that makes marginal improvements to code execution [*]. The improvements to code that come from compiler optimization do not compare to the improvements made by fields such as processor optimization, parallelization, or even studies of productivity [17].

While not a compiler optimization, the problem of Register Allocation may fall under Proebsting's Law. Due to it being an NP-hard problem yet also necessary for every compiler, register allocation has been frequently studied and implemented using a variety of efficient heuristics, all in search of better code performance. Are these minimal improvements beneficial and are the heuristics close to an optimal register allocation? In this thesis, we explore this question by comparing current practices in register allo-

[*]Special thanks to Shrutarshi Basu for introducing me to Proebsting's Law

cation to a new method of solving by psuedo-Boolean optimization problems. While similar work has been done in the past, it focused on preserving efficient compile time. Additionally, the rise of the International SAT Competition has greatly improved the ability of satisfiability solvers to find solutions to complicated problems since optimal register allocation was last studied. This work attempts to determine if the optimal register allocation with respect to the cost model outperforms the current heuristics thus giving us data for making further improvements register allocation algorithms and areas where our current heuristics lag behind the optimal solution. This thesis investigates whether or not Register Allocation is a “solved problem” and whether compilers research should focus its attention elsewhere to improve generated code. The work in this thesis moves toward this goal by designing a suitable encoding of register allocation with a cost model representing optimality and comparing the results to current heuristics.

1
Background

1.1 Register Allocation

In the first stages of compilation of a program, the compiler assumes it has the use of an infinite number of virtual registers to store each of the program variables and any intermediate variables created during compilation. However, during the lowering of the program to a low-level language like MIPS or x86-64, the compiler must assign each use of these virtual registers to a physical memory location. Depending on the architecture and physical constraints of the machine, some of these virtual registers may need to be placed on the stack during execution and reloaded into a register just before they are used in the program. This process of assigning virtual registers to a physical register (or memory location) is register allocation.

The importance of register allocation stems from the locality of memory. Being able to place all virtual registers in physical registers allows for better locality of memory and limits the need to load data from memory. If this is not possible, then for each use of a virtual register a potentially expensive memory operation must be added to load the data to a physical register. This load may be from the L1 cache or in the worst case, main memory. This load will take more time than a register use as a register lookup takes 0 cycles, an L1 cache lookup takes 4 cycles, and a main memory lookup takes 200 cycles [2]. Thus, the difference between placing a virtual register in a physical register or in main memory can dramatically increase the execution time of a program. It is therefore crucial that register allocation can find an optimal allocation of virtual registers to physical registers to avoid expensive loads from memory.

1.2 Challenges of Register Allocation

The main challenge of register allocation is that it is an NP-hard problem (by reduction from graph coloring) [3]. Thus, any efficient compiler must rely on efficient heuristics to perform Register Allocation. As noted above, the difference in clock cycles between storing a virtual register in a physical register or in main memory can make a huge impact on the running time of the program. The following properties are considerations necessary for a correct register allocation and must be accounted for when constructing a problem representing the register allocation problem:

1.2.1 Liveness

A major concern of register allocation is dealing with liveness or live-ranges of the virtual registers. A live-range is the portions of a program during which a virtual register will be used. When a virtual (or physical) register is live, it must be in memory for use in instructions. When there are no more uses of a register, it is referred to as being dead. To preserve correctness of the program, registers that have overlapping live-ranges cannot be assigned the same physical register or the program would be incorrect. Consider the following example:

```
1: $x \leftarrow 2$
2: $y \leftarrow 3$
3: $z \leftarrow x + y$
```

In the above example, if x and y were assigned to the same physical register, then at the assignment of z, $x = y$ and thus z would take on the value 6 rather than the correct value of 5. Thus, liveness must be considered when determining a register allocation. Liveness is further complicated by the possibility of overlapping physical registers (called aliasing). For example, in x86-64, the 64-bit register **%rax** overlaps with the 32-bit register **%eax**. Thus, two virtual registers with overlapping live-ranges cannot be placed in **%rax** and **%eax** simultaneously. However, in MIPS there are no physically overlapping registers. Thus, register allocation is very dependent on the target register file and this target register file must be accounted for beyond simply checking the number of registers when performing register allocation.

1.2.2 Coalescing

One feature of Register Allocation that came out of the work by Chaitin et al. is the notion of coalescing [3]. This is the idea that there is a benefit for two virtual registers which are move-related to be assigned to the same physical register (assuming their live-ranges do not overlap). By move-related, we mean a pair of registers who are related by an instruction $x := y$. If virtual registers x and y were assigned to different physical registers, r_1 and r_2, then the instruction would become $r_2 := r_1$. However, if the two virtual registers are assigned to the same physical register, r_1, then this instruction would become $r_1 := r_1$. This is a no-op and can be removed from the program which saves an instruction during execution. An optimal register allocation should take into account any potential coalescing that can be performed as this reduces the code size.

1.2.3 Expected Executions

Another challenge with register allocation is predicting code execution. We cannot predict how code will run every time due to pseudo-randomness, user input, or other non-deterministic aspects of the code. For this reason, we cannot say with certainty that a certain instruction will be executed n number of times. This is a challenge for register allocation as certain tradeoffs being made may depend on the number of times an instruction is executed. For example, inserting spill code that is executed once may be fine for the overall performance, but inserting spill code that is inside a block of code that is executed n^5 times will significantly slow down the running time. Thus, branching code and the expected execution must be accounted for in some reasonable way when discussing register allocation. Predicting the number of executions of a block of code is why we refer to our register allocation as being optimal with respect to the cost model. If we had an oracle that could tell expected number of executions, we could have a truly optimal register allocator. This is not possible due to uncertainty arising from control-flow structures like graphs.

1.3 Previous Approaches to Register Allocation

A very simple approach to solving the register allocation problem is to assign virtual registers to physical registers as they are encountered and when no more physical registers are available, the remaining virtual registers are spilled.

A more sophisticated Register Allocation was developed by Chaitin in 1982 [3]. He utilized the NP-hard problem of graph coloring to solve register allocation by using conservative heuristics that approximated a solution to register allocation via the graph coloring problem. This conservative heuristic relies on the interference graph and determining virtual registers whose live ranges interfere with each other.

A further innovation in register allocation was the development of the greedy approach known as Linear Scan. The linear scan algorithm greedily assigns virtual registers to physical registers by analyzing live ranges [16]. It is similar to the basic approach, but with liveness information. Preference for physical registers is given to virtual registers with longer live-ranges. Any remaining virtual registers are either spilled or split into two new virtual registers with smaller live-ranges. With live-range splitting, the new virtual registers are attempted to be allocated to physical registers that have a virtual register assignment that may have conflicted with the original virtual register, but not with the smaller live-range. In this study, we forego these heuristics and solve the Optimal Register Allocation problem outright by encoding the Register Allocation problem for a function as a psuedo-Boolean Optimization problem.

1.4 Pseudo-Boolean Optimization

Pseudo-Boolean optimization (PBO) is a class of optimization problems. The problem has decision variables that may be assigned a Boolean value of 0 or 1. These decision variables may be combined using arithmetic in constraints. PBO is similar to Integer Programming in that it has decision variables and constraints to model a problem with an objective function or cost model which is to be minimized based on the assignment of the decision variables. However, while decision variables in an Integer

Program may be assigned any integer value, in PBO decision variables must be assigned to Boolean values 0 or 1. Constraints in both of these problems can make use of these decision variable, arithmetic, the relational operators, and negation of the decision variables. However, in PBO problems, decision variables can be combined in non-linear ways which distinguishes it from Integer Programming. The inclusion of non-linear terms in PBO problems allows for more expressive cost models which has been useful for producing a PBO problem for register allocation.

Consider the following example pseudo-Boolean optimization problem:

$$min : 2 \cdot x_0 \cdot x_1 - 3 \cdot x_2 \tag{1.1}$$

$$x_0 + x_1 + x_2 \leq 2 \tag{1.2}$$

$$x_1 + x_2 \leq 1 \tag{1.3}$$

$$x_0 = 0 \tag{1.4}$$

$$x_0, x_1, x_2 \in \{0, 1\} \tag{1.5}$$

Equation 1.1 is the cost model or objective function of the problem. It is the minimization of a non-linear term with two decision variables and a decision variable. Both terms are weighted by constant factors. Constraint 1.2 guarantees that we assign the value 1 to at most two of our decision variables while constraint 1.3 guarantees that exactly one of x_1 and x_2 is assigned the value 1. Finally, we constrain decision variable x_0 to be 0 with constraint 1.4. Thus, we have an optimal assignment of $x_0 \mapsto 0, x_1 \mapsto 0, x_2 \mapsto 1$ which has a cost of -3. We note that due to the constraints, there is another optimal solution with $x_1 \mapsto 1$. Constraint 1.5 is an implicit constraint of all PBO problems that guarantees assignments are to Boolean values.

1.5 LLVM

LLVM is a compiler framework developed by Chris Lattner in 2004[12]. The framework allows for optimizations on an intermediate representation before that representation is lowered to specific executable machine code. Programs are translated to an intermediate representation that is then optimized using a series of standard passes before being lowered to the target instruction set. A user can therefore write code in C++, optimize it for various purposes and then convert the intermediate code to MIPS, x86-64, or ARM depending on the target of the original C++ program. One feature of LLVM is the conversion to static single assignment (SSA) form. In this form, each virtual register has a single definition site. Thus, the form is useful for traversing uses of virtual registers. As each virtual register is only defined once, we do not have to be concerned with continually updating a stack slot with a new virtual register value whenever the definition is updated.

The main feature in LLVM is the pass. Passes can either provide some information at different levels of granularity (functions, modules, loops) which is an Analysis Pass or make changes to these same structures which are Transform Passes. LLVM provides analysis passes which provide information such as liveness, branch probabilities, and information about the targeted instruction and register sets. The Register Allocation pass is a MachineFunction pass that iterates over each machine function within a program to perform register allocation. LLVM is distributed with 4 register allocators: greedy (the default), pbqp, fast, and basic. The fast register allocator proceeds by basic block and is mainly used for debug code. LLVM does not allow the fast register allocator to be used as the register allocator for optimized code, so it is not used in our study. Greedy is the state of the art for LLVM which relies on a variation of the Linear Scan algorithm that also includes live-range splitting to improve better performance.

PBQP is a register allocation method that relies on modeling the Register Allocation as a partitioned-Boolean Quadratic Program which like pseudo-Boolean Optimization is an NP-hard problem. However, while we solve the problem outright, their solver relies on heuristics based on the work of Hames and Scholz to solve the program[7]. The basic register allocation allocates based on live-ranges and spill-weights and then proceeds to spill any remaining registers. While it is not a complicated algorithm, it provides a good baseline for register allocation.

1.5.1 clang

The clang and clang++ compilers are distributed as part of the LLVM framework. They are compilers that take in C and C++ code and can be connected to the LLVM backend to perform analysis and transform passes. Command line arguments allow for the specification of options such as register allocation method and specific optimizations to include or exclude during compilation. Using the clang compiler with different register allocation methods ensures that the compiler is not a variable when performing benchmarks of register allocation. Recent debate has been raised in the compiler community about the respective benefits of clang over gcc, which is the GNU C compiler. Benchmarking as recent as October of 2019 showed that clang was able to outperform gcc on certain benchmarks [11]. Thus, clang is a reasonable choice for the implementation of the PBO Register Allocator.

2
Design

In this chapter, we outline how the register allocation problem can be encoded as a pseudo-Boolean optimization problem by instantiating decision variables, formulating constraints, and developing a cost model. Constraints encode hard correctness requirements for the register allocation while the cost model ensures that the optimal register allocation is found. A formal specification of the constraints and cost model can be found in Appendix A.

2.1 Decision Variables

Decision variables are used according to the available physical registers for a program. For a virtual register, we will either assign it to a physical register for its lifetime or mark it as a spill. For virtual register n and physical register m, decision variable

$alloc_{n,m}$ is the decision variable representing allocating virtual register n to physical register m for the length of the lifetime of n. The decision variable $spill_n$ indicates that we will spill virtual register n. If a virtual register is spilled, then it must be stored in a stack slot and reloaded before each use of the physical register. For each of the n virtual registers, we create decision variables for each valid physical register which can be used for the virtual register and a special spill decision variable indicating if that virtual register is to be spilled. By valid we mean belonging to the correct register class. For example, in x86-64, certain virtual registers must be in the floating point register class and thus cannot be asssigned to integer physical registers. We choose to limit the initial creation of decision variables mapping a virtual register to a physical register which reduces the number of generated constraints and reduces generation of constraints of the form $x = 0$.

If a virtual register, n, is spilled (i.e. $spill_n = 1$), then we must indicate which physical register will be used to hold the value for each use of the physical register. For virtual register n, physical register m, and program point p, the decision variable $use^p_{n,m}$ indicates that virtual register n will be stored in physical register m at program point p.

2.2 Constraints

The constraints in the pseudo-Boolean optimization problem represents three basic principles of a correct register allocation: single assignment for use, live ranges, and correct usage.

2.2.1 Single Assignment at Use

This set of constraints ensure that each virtual register is assigned to exactly one physical register for every use. A constraint is created so that a virtual register is either spilled or assigned a physical register for the entire program but not both: for all virtual registers n and physical registers m:

$$spill_n + \sum_m alloc_{n,m} = 1 \tag{2.1}$$

In addition, a constraint ensures that if a virtual register is spilled, then the virtual register has a single physical register assigned to it for each use of the virtual register and if the virtual register is assigned a physical register for the duration of its lifetime, then it does not get assigned to an additional physical register for any of its specific uses. This constraint is that for all virtual registers n, physical registers m, and program points p, we have:

$$-spill_n + \sum_m use^p_{n,m} = 0 \tag{2.2}$$

2.2.2 Live Ranges

This set of constraints ensure the correctness of the register allocation with respect to live ranges. If two virtual registers are live at the same time, we must ensure they are not assigned to the same physical register (or aliased registers) for the duration of their lifetime. We construct a constraint to ensure that at most one of the virtual registers is assigned to a specific physical register for the duration of its lifetime. i.e. for all virtual registers n, n' such that the live ranges of n and n' overlap, and for all overlapping physical registers, m and m', we have:

$$alloc_{n,m} + alloc_{n',m'} \leq 1 \tag{2.3}$$

The second of the liveness constraints preserves the liveness of any physical registers. Certain physical registers (e.g. argument registers, return register, the instruction counter) are explicitly used in instructions prior to register allocation. Thus, constraints are added to ensure that if the live range of a virtual register overlaps with these explicitly used physical registers, then that virtual register is not assigned to that physical register (or any physical register that overlaps with that physical register). i.e. for all virtual registers n and physical registers m such that the live range of n overlaps with m, we have:

$$alloc_{n,m} = 0 \tag{2.4}$$

2.2.3 Correct Usage

This set of constraints ensure the correctness of the register allocation for each use of a virtual register. If two virtual registers have overlapping live ranges or are used in the same instruction, then we must guarantee that they are not assigned to the same physical register (or aliased physical registers). We do this by adding constraints similar to the live range constraints except we substitute the decision variable for a use rather than for an assignment for the duration of the lifetime: for all virtual registers with an overlapping live range, n and n', overlapping physical registers m and m', and program points p with uses of n, we have:

$$use^p_{n,m} + alloc_{n',m'} \leq 1 \tag{2.5}$$

$$use^p_{n,m} + use^p_{n',m'} \leq 1 \tag{2.6}$$

These constraints ensure that if virtual registers have overlapping live ranges or are used in the same instruction that they are not assigned to overlapping physical registers either for the duration of their live ranges or for the specific use at that program point.

Additionally, these constraints ensure that if a virtual register overlaps with a physical register live range or is used in an instruction with a physical register, m, that the virtual register is not assigned to any physical register, m' that overlaps with m at the use. For all virtual registers n, physical registers m such that the live ranges of n and m overlap, we have that:

$$use^{p}_{n,m} = 0 \tag{2.7}$$

2.3 Cost Model

2.3.1 Representation of Correct Register Allocations

While we do not provide a formal definition of a register allocation, a register allocation can be thought of as a mapping from every use of a virtual register to a phyiscal register. Additional mappings would be needed for spill/reload code, but for the purposes of this section we do not provide a formal specification. We now argue that any correct register allocation can be represented as a solution to the pseudo-Boolean Optimization problem described above.

To be a solution to the PBO problem, we must show that the hard constraints are satisfied: single assignment, live ranges, and correct usage. A correct register allocation will assign a virtual register to a physical register for the duration of its lifetime or spill the virtual register, so constraint 2.1 will be satisfied for every virtual register. If a correct register allocation spilled a virtual register but did not have a physical register for a usage at a certain program point, then the register allocation would not be a complete mapping which contradicts that it is a correct register allocation. Thus, we conclude that constraint 2.2 will be satisfied for every virtual register. A register allocation that allocates two virtual registers to the same (or aliased) physical register with overlapping live ranges would not maintain the correctness of the program, so it would not be a correct register allocation. Thus, we know that constraint 2.3 will hold. For similar reasons, owing to the overwriting of a live physical register, we know constraint 2.4 and constraint 2.7 hold. If constraint 2.5 does not hold, then we would be overwriting the value of a live virtual register which invalidates program correctness and thus cannot result from an invalid register allocation. If constraint 2.6 did not hold at some program point p, then that instruction which previous expected two distinct virtual registers would use the same virtual register twice. This is because violation of Constraint 2.6 would lead to two reloads into the same physical register immediately before use which does not maintain program correctness. (See the example at the beginning of Section 1.2.1)

Having shown that all the constraints hold for a correct register allocation, we conclude that any solution to the constructed PBO problem will be a correct register allocation.

Having shown that every correct register allocation is a solution to the PBO encoding we have created, we now attempt to find the optimal solution by introducing a cost model to the problem. As the pseudo-Boolean optimization problem we construct is a minimization problem, a bonus refers to a negative weight on the cost model term (satisfaction of the term will lower the cost function value) and a penalty is a positive weight on the cost model term (satisfaction of the term will raise the cost function value). The cost model takes liveness, coalescing information, and block frequency information to produce an optimal register allocation. The major components of the cost model are: coalescing bonuses and spill penalties. We refer to a fixed memory operation cost which is a weight relating to the relative penalty of making a memory operation. This value is kept in proportion to the coalescing bonus to balance the improvement to code provided by coalescing and the cost of accessing memory.

2.3.2 Coalescing Bonus

The cost model encourages the coalescing of virtual registers. Thus, a positive cost proportional to a single instruction is imposed when registers that are move related are not placed in the same register for the duration of their live ranges. This encourages assigning move related virtual registers to the same physical register which leads to the removal of a redundant move instruction during code generation and improve code performance. For example, if x and y are move related (e.g. $x := y$) then allocating with physical registers **%eax** and **%rdi** respectively leads to the instruction $mov\,\%eax,\ \%rdi$. However, if x and y are both allocated to $\%rdi$ then this instruction becomes $mov\,\%rdi,\ \%rdi$ which is a no-op and can be removed by a later compiler pass.

2.3.3 Spill Penalty

The cost model discourages the spilling of registers. A penalty is incurred when a register is spilled. The penalty of a spill is proportional to the frequency with which the virtual register is used. Thus, registers that are used more frequently are discouraged from being spilled (as this would lead to more reload and spill instructions which would hinder code performance).

2.3.4 Callee-Save Registers

Callee-Save registers are registers that must be preserved by the function being called according to the calling conventions of an instruction set. Thus, if a callee-save register is used in a function body, a push/pop instruction must be added to the function prologue and epilogue to preserve the values of these registers. Thus, an optimal register allocation would avoid the use of these registers where possible (as they add memory operations). However, in some cases the saving and restoring of a callee-save register is worth the benefit of an additional physical register to reduce the overall register pressure. Thus, as each use of a callee-save register would require an additional two memory operations (a spill in the function prologue and a reload in the function epilogue), our cost model discourages the use of callee-save registers by placing a penalty equal to two times the memory operation cost on a disjunction representing use of a callee-save register.

2.4 Size Analysis of the PBO Problem

Let **VR** denote the set of virtual registers and **PR** denote the set of physical registers. The total number of decision variables is $O(|\mathbf{VR}| \cdot |\mathbf{PR}|) + O(|\mathbf{VR}|) + O(|\mathbf{VR}| \cdot MaxU \cdot |\mathbf{PR}|)$ where $MaxU$ is the maximum number of uses by a single virtual register. The dominating term comes from the number of decision variables representing uses and so the number of decision variables is $O(|\mathbf{VR}| \cdot |\mathbf{PR}| \cdot MaxU)$.

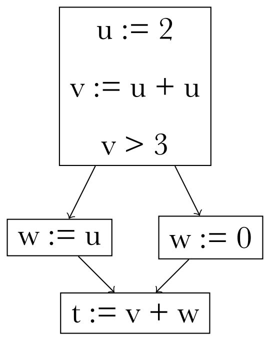
The number of constraints is $O(|\mathbf{VR}| + |\mathbf{CS}| + |\mathbf{VR}||\mathbf{PR}| + |\mathbf{VR}|^2 + |P|(|\mathbf{VR}|^2|\mathbf{PR}| + |\mathbf{PR}|))$ where **CS** is the set of callee-save registers (with $\mathbf{CS} \subseteq \mathbf{PR}$) and P is the set of all program points. This sum is dominated by the final term, so we conclude that the number of constraints is $O(|P||\mathbf{VR}|^2|\mathbf{PR}|)$.

2.5 Example Program and Constraints

Consider the following program:

```
1: $u \leftarrow 2$
2: $v \leftarrow u + u$
3: **if** $v > 3$ **then**
4:     $w \leftarrow u$
5: **else**
6:     $w \leftarrow 0$
7: **end if**
8: $t \leftarrow v + w$
```

We can construct a control-flow graph for the programming:



This control-flow graph shows the various conditional jumps the program may take as well as indicating which variables may have overlapping live ranges. For example, the CFG shows us that while u is not used in the true branch of the condition, it is used in the false branch, and so u and v have overlapping live ranges.

Assume that we have 3 physical registers (r_1,r_2,r_3) for allocation, none of the physical registers are aliased, and that there are no physical or virtual registers live-in to the program. Consider the following constraints which are generated for virtual register v:

Single Assignment:

$$alloc_{v,r_1} + alloc_{v,r_2} + alloc_{v,r_3} + spill_v = 1 \tag{2.8}$$

$$-spill_v + use^2_{v,r_1} + use^2_{v,r_2} + use^2_{v,r_3} = 0 \tag{2.9}$$

$$-spill_v + use^3_{v,r_1} + use^3_{v,r_2} + use^3_{v,r_3} = 0 \tag{2.10}$$

$$-spill_v + use^8_{v,r_1} + use^8_{v,r_2} + use^8_{v,r_3} = 0 \tag{2.11}$$

Constraint 2.8 guarantees that either v is assigned to one of the available physical registers for the duration of the life of the virtual register or it is spilled. We guarantee that the sum is exactly 1, so that exactly one of the decision variables must be selected. We want to ensure that each use of v has a designated physical register. Constraint 2.9 guarantees that if v is spilled then there will be a physical register assigned for use at point 2 and if it is not spilled, then all use decision variables will be 0 to avoid any ambiguity.

Live Range Correctness:

$$alloc_{v,r_1} + alloc_{u,r_1} \leq 1 \tag{2.12}$$

$$alloc_{v,r_2} + alloc_{u,r_2} \leq 1 \tag{2.13}$$

$$alloc_{v,r_3} + alloc_{u,r_3} \leq 1 \tag{2.14}$$

$$alloc_{v,r_1} + alloc_{w,r_1} \leq 1 \tag{2.15}$$

$$alloc_{v,r_2} + alloc_{w,r_2} \leq 1 \tag{2.16}$$

$$alloc_{v,r_3} + alloc_{w,r_3} \leq 1 \tag{2.17}$$

As v and u have overlapping live ranges constraint 2.12 guarantees that they are not both assigned to physical register r_1 for the entirety of their live ranges.

Usage Correctness:

$$use^2_{v,r_1} + alloc_{u,r_1} \leq 1 \tag{2.18}$$

$$use^2_{v,r_2} + alloc_{u,r_2} \leq 1 \tag{2.19}$$

$$use^2_{v,r_3} + alloc_{u,r_3} \leq 1 \tag{2.20}$$

$$use^3_{v,r_1} + alloc_{u,r_1} \leq 1 \tag{2.21}$$

$$use^3_{v,r_2} + alloc_{u,r_2} \leq 1 \tag{2.22}$$

$$use^3_{v,r_3} + alloc_{u,r_3} \leq 1 \tag{2.23}$$

$$use^4_{u,r_1} + alloc_{v,r_1} \leq 1 \tag{2.24}$$

$$use^4_{u,r_2} + alloc_{v,r_2} \leq 1 \tag{2.25}$$

$$use^4_{u,r_3} + alloc_{v,r_3} \leq 1 \tag{2.26}$$

$$use^4_{w,r_1} + alloc_{v,r_1} \leq 1 \tag{2.27}$$

$$use^4_{w,r_2} + alloc_{v,r_2} \leq 1 \tag{2.28}$$

$$use^4_{w,r_3} + alloc_{v,r_3} \leq 1 \tag{2.29}$$

$$use^8_{v,r_1} + alloc_{w,r_1} \leq 1 \tag{2.30}$$

$$use^8_{v,r_2} + alloc_{w,r_2} \leq 1 \tag{2.31}$$

$$use^8_{v,r_3} + alloc_{w,r_3} \leq 1 \tag{2.32}$$

$$use^8_{v,r_1} + use^8_{w,r_1} \leq 1 \tag{2.33}$$

$$use^8_{v,r_2} + use^8_{w,r_2} \leq 1 \tag{2.34}$$

$$use^8_{v,r_3} + use^8_{w,r_3} \leq 1 \tag{2.35}$$

Again because v and u have overlapping live ranges, we must ensure that they are not assigned to overlapping physical registers. Constraint 2.18 ensures that the use of v is not assigned to the same physical register that u has been assigned to for the duration of its live range.

As v is live at point 4 (but not used), we must constrain the uses of u and w to not be assigned the same physical register as v is assigned to for the duration of its live range. Constraint 2.24 ensures this for the use of u and physical register r_1.

Constraint 2.33 ensures that if v and w are both spilled, then they are not both assigned to physical register, r_1 for use at point 8. We note that a similar constraint is not generated for u and v at point 2. This is because this is the write use or definition of v and a read use or use of u. If both are marked for spill, the two could be assigned to the same physical register for use at this point and the correctness of the allocation would be preserved.

Consider the following two Register Allocations:

Register Allocation 1:

$$alloc_{u,r_1} \mapsto 1, alloc_{v,r_2} \mapsto 1, alloc_{w,r_1} \mapsto 1, alloc_{t,r_2} \mapsto 1,$$

$$\text{and all other decision variables are mapped to } 0$$

Register Allocation 2:

$$alloc_{u,r_1} \mapsto 1, spill_v \mapsto 1, use^2_{v,r_2} \mapsto 1, use^3_{v,r_2} \mapsto 1,$$

$$use^8_{v,r_1} \mapsto 1, alloc_{w,r_2} \mapsto 1, alloc_{t,r_2} \mapsto 1,$$

$$\text{and all other decision variables are } 0.$$

Both register allocations use two physical registers. In the first, no additional memory operations are needed. We can rewrite the program using the first register allocation:

```
1: $r_1 \leftarrow 2$
2: $r_2 \leftarrow r_1 + r_1$
3: **if** $r_2 > 3$ **then**
4:     $r_1 \leftarrow r_1$
5: **else**
6:     $r_1 \leftarrow 0$
7: **end if**
8: $r_2 \leftarrow r_1 + r_2$
```

This can easily be converted to x86-64 machine instructions using the mov, add, and gt instructions. We note that above at point 4, the operation $r_1 \leftarrow r_1$ is a move related pair of registers. Additionally, the register allocation used here has successfully coalesced the two virtual registers into the same physical register. As the instruction is a move from a physical register to itself, later passes will remove this no-op which reduces the code size and potentially improves performance.

The second register allocation requires additional memory operations and more memory. We use SS_1 to refer to a stack slot for a spilled virtual register:

```
1: $r_1 \leftarrow 2$
2: $r_2 \leftarrow r_1 + r_1$
3: $store \quad r_2, \quad SS_1$
4: $load \quad SS_1, \quad r_2$
5: **if** $r_2 > 3$ **then**
6:     $r_2 \leftarrow r_1$
7: **else**
8:     $r_2 \leftarrow 0$
9: **end if**
```

```
10: $load \quad SS_1, \quad r_1$
11: $r_2 \leftarrow r_1 + r_2$
```

Here, we consider an assignment that spills v. We must store the definition of v immediately after definition and reload it immediately before use into an appropriate physical register. After v is placed in physical register r_2 at its definition site, it is stored in stack slot SS_1. At its second use in the condition guard, we expect the value to be in physical register r_2. Thus, we load from SS_1 to physical register r_2. Note that the assignments in the if and else branches of the condition overwrite the current value of physical register r_2. Thus, we must load from SS_1 to a different register, r_1, for the third use of virtual register v. Thus, this register allocation leads to the insertion of three additional memory operations that could be eliminated by a better register allocation (like the first example). Another thing to note about this example is that the load of v into r_2 is redundant (i.e. the value of r_2 before point 5 is the correct value). Our cost model accounts for this fact and encourages allocations such as this that allow for the removal of redundant loads by later compiler passes.

2.6 Implementation

We now outline the implementation of the design choices discussed above. Implementation of the PBO Register Allocation Pass can be found here.

2.6.1 Creating PBO Problem

The design above was implemented in 1500 lines of C++ code within the LLVM framework (version 9.0.0svn). The PBO Register Allocator was written as an additional register allocation pass. The pass sets up decision variables, constraints, and the objective function for a suitable PBO problem. The constraints and objective function follow Pseudo-Boolean 2006/2007 input format in order to be passed to the PBO solver. This means decision variables are implemented using strings of the form xN where x is fixed and N is a running counter of the number of instantiated logical variables. Constraints were created by combining decision variable strings as needed for the type of constraint being constructed. To avoid limits on the size of strings, constraints were written to a separate "constraint file" before being written to the final file with correct header. The objective function remained a suitable length to be maintained in a single string.

2.6.2 File Pipeline

The constraints and objective function are written to a file of form "constraintsXN.opb" where X is the pid of the process and N comes from a running counter of how many functions have been allocated by the current PBO Register Allocator object. This allows for parallelization of the PBO Register Allocator. This file contains the PBO Problem representing Register Allocation for the function in question. This file begins with a metadata header which indicates the number of variables and constraints contained in the file. This final file is passed to the sat4j PBO solver[13] via forking of a separate process to invoke the command-line call. The PBO register allocation takes command line arguments for timeout, spill weight, and move weight parameters. The timeout argument determines the timeout for each function that is compiled. The PBO solver will attempt to solve and optimize for timeout seconds before returning the optimal solution found in that time (or UNKNOWN if no solution is found in that time). The spill weight and move weight parameters are for tuning of the cost model. The spill weight is the approximate cost of a single memory operation while move weight is the potential benefit of coalescing a move related pair of registers.

We use the Default solving method for the PBO problem although sat4j provides additional solving techniques which may be an area for future research. The output from the solver process is read back into the register allocation pass and decision variable values in the optimal register allocation are placed in a mapping from decision variables to their value (0 or 1).

2.6.3 Assignment

After reading in the results of the solver a pass over the program rewrites the virtual registers with the physical register they have been assigned and inserts spill/reload code as necessary for any use of the virtual register. The insertion of these instructions is the only change made to the program itself (besides assigning each virtual register use to a physical register). This differs from the method of the other clang register allocators which update a Virtual Register Mapping when performing register allocation. This virtual mapping is then used to rewrite the function with physical registers. Because the solution to the PBO problem gives a physical register for each use of a virtual register, we decided to rewrite the virtual registers with physical registers in place as all information is available to us after solving the PBO problem. After the register allocation pass concludes, clang continues through code generation and eventually produces the compiled code (whether that is assembly or an executable).

The PBO Register Allocator is passed as a command line argument to the clang compiler which is part of the LLVM infrastructure. By passing the register allocation procedure as a command line argument, we ensure that the only difference between register allocators is the register allocation phase. We have also allowed for tuning of parameters to the solver (i.e. the timeout, spill weight, and move bonus). However, we did not extend the gcc (and g++) compilers with a separate implementation of the PBO register allocator. As such, the comparison between the two may differ due to separate optimizations used. We include comparisons to gcc to provide another state-of-the-art compiler and for its prevalence in the compiler community.

2.6.4 Branch Probabilities

One feature of LLVM is the Block Frequency analysis pass. This pass gives an approximation of the expected execution of each basic block. We use this expected value to place a weight on the spilling of a virtual register. The weight is equal to the sum of block frequencies for each block that a virtual register is used in. We then use this sum as the penalty if the virtual register is spilled. Thus, virtual registers with a higher number of expected executions are prioritized to be assigned to a physical register for longer than those that have fewer expected executions.

3

Evaluation

The PBO Register Allocator was evaluated on an Amazon Web Service EC2 Ubuntu server with 16GB of memory. Evaluation was performed using the SPEC CPU 2017 benchmarks and optimization level -O3. These benchmarks test a variety of features and contain integer as well as floating point benchmarks. As mentioned in Section 1.5.1, clang is distributed with four register allocators: basic, greedy, pbqp, and fast. As fast can only be used for unoptimized code, it was not included in comparisons. We evaluated the three distributed clang register allocators, gcc, and our pseudo-Boolean Optimization register allocator with different parameters.

Each register allocator was evaluated on the 505.mcf_r benchmark with input of size test which is part of the SPEC 2017 benchmarks (nc). The notation (nc), for non-compliant, is used because our compiler does not meet SPEC's requirements for general availability. The executable resulting from compilation was run five times with the results being reported for each run.

In addition to runtime, the clang register allocators were evaluated for the number of static memory operations added to the functions during register allocation. This is one indication of the performance of register allocation as it can show how many expensive memory operations were added as a result of the register allocation and could lead to a worse execution time. These statistics were not collected for the gcc compiler.

We compiled the benchmark with three different PBO Allocators: The first (clang PBO 1) which had a spill weight of 10000 and timeout of 1024 seconds (1̃7 minutes), the second (clang PBO 2) which had a spill weight of 200 and a timeout of 21000 seconds (6 hours), and the final (clang PBO 3) which had a spill weight of 10000 and a timeout of 21000 seconds (6 hours).

3.1 Results

3.1.1 Static Memory Operations

Table 3.1 outlines additional memory operations (spills/reloads) that were added by each of the register allocators during compilation of a file. Five files were omitted as no register allocation method added additional memory operations during their compilation. Initial analyis indicates that the same register allocation was found for functions in these files regardless of the register allocator used, but due to differences in statistics reportings we cannot definitively say that this is true. The number of static spills serves as a baseline comparison for the gap between the register allocation found in the time limit and current heuristics. Because in general, minimization of memory accesses is our goal, register allocations that add fewer memory operations tend to per-

Compiler	1	2	3	4	5	6	7
clang greedy	36	144	491	2	14	2	17
clang pbqp	31	169	744	9	13	3	14
clang basic	67	210	929	9	22	2	19
clang PBO 1	55	255	1197	13	47	10	29
clang PBO 2 *	55	260	1180	13	47	9	34
clang PBO 3	55	295	1197	13	47	10	29

Table 3.1: Static Memory Operations

form better. We note that none of the PBO Allocators had the same number of static memory operations added. For a specific function in file 2, PBO Allocator 3 was able to make an additional assignment to a register which contrary to expectations led to an increase in the number of additional static memory operations added to the function. However, runtime confirms that despite these additional static memory operations, the PBO Register Allocator 3 outperforms the other PBO Register Allocators.

3.1.2 Runtime

Table 3.2 provides information about the different register allocators and their respective running times. We note that gcc outperformed all of the clang register allocators. Additionally, we note that the median for the best PBO Register Allocator was 3% greater than the median of the clang greedy Register Allocator. This is good performance considering the vast difference in the number of memory accesses inserted by the two register allocators. It is possible that the additional memory operations may have been added to basic blocks that are for error handling or low probability branches or that good memory locality led to better cache performance which did not significantly impact the runtime. We also note that despite having added more additional memory operations in each of the above functions except the first, the PBO Allocators perform better than the clang basic register allocation. Also of note is that

Compiler	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	**Median** (s)
gcc	11.0	11.2	11.3	11.1	11.1	11.1
clang greedy	12.5	12.5	12.4	12.5	12.5	12.5
clang pbqp	12.3	12.5	12.2	12.2	12.6	12.5
clang basic	14.5	14.5	14.5	14.5	14.4	14.5
clang PBO 1	13.9	14.2	14.1	14.1	14.1	14.1
clang PBO 2	14.6	14.5	14.6	14.5	14.6	14.6
clang PBO 3	13.1	12.9	12.8	12.9	13.1	12.9

Table 3.2: Runtimes

the PBO Allocator 1 outperformed PBO Allocator 2 despite PBO Allocator 2 having a larger timeout value. This indicates more thought must be put into the spill weights and parameters for the PBO problem.

3.2 Discussion

3.2.1 PBO Problem

The benchmark compiles a total of 37 functions. The PBO Register Allocation was able to find register allocations for each of these functions. Of the 37 functions, optimal register allocations were found for 30 of the functions (81.1%). Each of the optimal solutions to the PBO problem for the 30 functions was found in under 65 seconds. Thus, finding the optimal solution for most functions is achievable and a reasonable practical solution. It is mainly a small set of large functions that cause a spike in compilation time and alter execution time from the other register allocation methods. The problems for which optimal solutions were found in general had hundreds of logical variables and thousands of constraints. The seven problems that could not be solved optimally within a six hour time limit ranged from tens of thousands of constraints to the largest problem generated which had 93741 logical variables and 1461249 constraints. Due to the large number of constraints and PBO solving being NP-hard, solving the problem as it is currently formulated seems intractable, but future work could investigate longer timeouts for the problem and what improvements the increased time brings to program execution.

3.3 Future Work

Further work will be devoted to further optimizations and better approximation of the cost model. Towards this end, one area is to convert programs from SSA form to Static Single Use (SSU) Form. In this form, a virtual register in the program has a single definition and a single use in the program. This would allow for better optimization of coalescing. Additionally, programs in SSU form would essentially add live-range splitting to the PBO Register Allocator as the cost model would reward extended chains of coalescing which is the same effect as live-range splitting. This addition would allow for a better comparison between the clang register allocators all of which use live range splitting in some capacity.

Another area to explore is the constraints generated for the PBO problem. Evaluation of the cost model could be performed to determine other possible terms for the cost model or fine tune the weights of the model to better reflect the architecture or target instruction set. These parameters could be altered depending on the register set (e.g. register preference) or known information about the memory layout.

Further work will be aimed at reducing the number of constraints generated. This will increase the likelihood that the solver is able to find the optimal assignment before reaching the timeout value. In many cases, the PBO Register Allocator was not able to find the optimal register allocation with respect to the cost model. This is due to the timeout and a complicated cost model that made it harder to find the optimal solution. Being able to simplify the cost model and constraints would allow for faster solving which would hopefully lead to a faster solving time which would lead to a better Register Allocation found.

Finally, one line of research could be to formally verify the PBO register allocator. This would guarantee correctness of the register allocation produced as well as be a step towards a formally verified compiler.

4

Related Work

4.1 Heuristic Methods of Register Allocation

Register Allocation was first performed systematically by Chaitin in 1982. The proposed graph coloring method of register allocation relied on construction of an interference graph for the program based on the live ranges of virtual registers. This graph is transformed in a number of steps before a satisfying assignment of variables to registers is found [3]. Additionally, this method maintains a stack of virtual registers as they are assigned physical registers. This is an iterative process whereas our approach is to find the physical memory location for every virtual register at the same time.

In 1999, Linear Scan was developed as a new method of register allocation. The linear scan algorithm provides a greedy method of performing register allocation that could be performed in linear time. While faster to perform, the generated code using Linear scan was initially worse than that produced by graph coloring [16]. Traub et al. simultaneously were working on a linear scan algorithm that added live-range splitting of variables to improve the results of the Linear Scan register allocation. This live-range splitting helped improve the results of the Linear Scan register allocator due to a better fitting of registers into ranges that could not be covered in the original linear scan algorithm [18]. Linear Scan and its variants are viewed as the best heuristic and a version of linear scan is included in LLVM as the default register allocator.

More recently, the idea of reasoning about register allocation as a jigsaw puzzle was developed by Pereira and Palsberg [15]. This implementation ran in linear time and produced code that was as efficient as was produced by linear scan with coalescing. This paper outlined recent practices of testing compiler implementations and suitable benchmarks.

Hames and Scholz discuss the use of the NP-hard problem of Partitioned Quadratic Boolean Programming for Register Allocation in their 2006 paper [7]. The paper focuses on branch-and-bound techniques for trimming the solver's search space. Additionally, they use an iterative coloring approach reminiscent of Chaitin's approach of removing registers as they are successfully colored. In our approach we directly map the solution to the PBO problem to a register allocation.

4.2 Complexity of Register Allocation

The question of the complexity of Register Allocation has been widely studied with seemingly contradictory results. In 2006, Hack et al. proved that Register Allocation for programs in SSA form could be completed in polynomial time [6]. This result seemed to indicate that general register allocation could be performed in polynomial time by converting a program to SSA form in linear time, performing register allocation on the SSA graph and then converting the SSA program into machine code. However, a later result by Pereira and Palsberg showed that this mapping from SSA register allocation to general register allocation is NP-hard [14].

In 1998, Kannan and Proebsting found that the restriction of register allocation to structured programs was still NP-hard by reduction from circular-arc graph coloring. They developed an approximation algorithm that was within a factor of 2 of the optimal solution [8]. In 2013, Krause developed a seemingly optimal register allocator for structured programs by using tree decomposition, but this result was called into question by later work by Krause about the complexity of register allocation [9, 10].

4.3 Optimization

Boros and Hammer survey pseudo-Boolean Optimization (PBO) in their 2002 paper. They discuss the extension of PBO to the NP-hard problem of MAXSAT [1]. The paper focuses on approximations of the NP-hard problem by using approximations of PBO problems and generation of binary vectors. In this work, we solve the PBO problem directly to find an optimal register allocation with respect to the cost model rather than finding an $\alpha-$approximation of the optimal register allocation with respect to our cost model.

An optimal register allocation algorithm by 0-1 Integer Programming was attempted by Goodwin and Wilken [5]. Their initial model focused on the lattice of control-flow graphs that could be constructed for each possible assignment of virtual registers to physical registers. Follow-up work by Fu and Wilken introduced decision variables to optimize the spilling and loading of virtual registers [4]. Both of these works limit the solver time to 1024 seconds and rely on an integer program to solve the register allocation. Goodwin and Wilken's Optimal Register Allocator allocated 82% of programs optimally in that time limit which is a similar result to our findings. However, they are only able to find solutions for a subset of the functions while in our study, we were able to find some solution to the PBO problem within a 1024 timeout. Goodwin and Wilken use a model of expected execution that follows Kirchoff's current Law (i.e. the sum of incoming expected executions is the sum of the outgoing expected executions). In this paper, we rely on LLVM's Block Frequency which guarantees that the sum of outgoing probabilities is 1 but does not ensure that incoming probability is 1.

5
Conclusion

Register Allocation has now been studied for decades. In this thesis, we outlined how to construct a pseudo-Boolean Optimization problem whose solution encodes a register allocation for a given program. We showed how any correct register allocation can be represented as a solution to this PBO problem. Our constructed cost model gives an optimal register allocation goal for a PBO solver to find. We found that even with a 6 hour optimization time, modern heuristics surpassed the PBO Register Allocator in both runtime and in the number of static memory operations added by the program. The PBO Register allocator implemented came within 3% of the default clang register allocation method. Additionally, the greedy allocation of clang coalesces the same virtual registers as our optimal allocation. Future additions of live range splitting to the PBO cost model and increasing the timeout value for the PBO solver may allow the PBO method of register allocation to surpass the default clang register allocation method, but in these trials we found that modern heuristics are able to produce code on par with an optimal register allocations. While we cannot conclude that Register Allocation is a solved problem, we have confirmed that current heuristics find register allocations that are optimal or very near optimal.

A

PBO Problem Specification

Let n range over virtual registers, m range over physical registers. The decision variable $alloc_{n,m}$ is a decision variable representing the assignment of virtual register n to physical register m for the duration of the virtual register. The decision variable $spill_n$ represents the need to spill virtual register n at some point during the life of the virtual register. The decision variable $use^p_{n,m}$ represents the assignment of virtual register n to physical register m for the use that occurs at program point p.

A.1 Constraints

A.1.1 Valid Assignments

1. A register is spilled or assigned to a physical register for its lifetime:

$$\forall n \in \mathbf{VR}, \mathit{spill}_n + \left(\sum_{m \in \mathbf{PR}} \mathit{alloc}_{n,m} \right) = 1$$

2. If a register is spilled, then it has a physical register assignment for each use:

$$\forall n \in \mathbf{VR}, p \in \mathit{Uses}(n). - s_n + \sum_{m \in \mathbf{PR}} \mathit{use}^{p}_{n,m} = 0$$

3. Registers can only be assigned to the correct register class:

$$\begin{aligned} \forall n \in \mathbf{VR}. \forall m \in \mathbf{PR}. \mathit{RegisterClass}(n) \neq \mathit{RegisterClass}(m) \rightarrow \\ \mathit{alloc}_{n,m} = 0 \wedge \forall p \in \mathit{Uses}(n). \mathit{use}^{p}_{n,m} = 0 \end{aligned}$$

A.1.2 Live Range Conflict Constraints

1. Virtual registers with overlapping live ranges cannot be assigned to the same physical register:

$$\begin{aligned} \forall n_0, n_1 \in \mathbf{VR}. \forall m_0, m_1 \in \mathbf{PR}. \mathit{LiveRangeOverlap}(n_0, n_1) \wedge \\ \mathit{PhysRegOverlap}(m_0, m_1) \rightarrow \mathit{alloc}_{n_0,m_0} + \mathit{alloc}_{n_1,m_1} \leq 1 \end{aligned}$$

2. A virtual register should not be assigned to a physical register that has a conflicting live range:

$$\forall n \in \mathbf{VR}. \forall m \in \mathbf{PR}. \mathit{LiveRangeOverlap}(n, m) \rightarrow \mathit{alloc}_{n,m} = 0$$

A.1.3 Usage Constraints

Here the set R_p is the set of all registers used at program point p and the set L_p is the set of registers live at program point p.

1. If two virtual registers are used at the same program point (i.e. in the same instruction), at most one may be assigned to a physical register (or its aliases):

$$\forall n_0 \in \mathbf{VR}.\forall p \in \mathit{Uses}(n_0).\forall n_1 \in R_p \cap \mathbf{VR}.\forall m_0, m_1 \in \mathbf{PR}.$$
$$\mathit{PhysRegOverlap}(m_0, m_1) \rightarrow \mathit{use}^p_{n_0,m_0} + \mathit{use}^p_{n_1,m_1} \leq 1$$

2. If a physical register is used in the instruction at program point p, then a virtual register cannot be assigned to that physical register for use at p:

$$\forall n \in \mathbf{VR}.\forall p \in \mathit{Uses}(n).\forall m_0 \in R_p \cap \mathbf{PR}.$$
$$\forall m_1 \in \mathbf{PR}.\mathit{PhysRegOverlap}(m_0, m_1) \rightarrow x^p_{n,m_1} = 0$$

3. If a virtual register is live at program point p, it cannot be assigned the same physical register as a register that is used at p:

$$\forall n_0 \in \mathbf{VR}.\forall p \in \mathit{Uses}(n_0).\forall n_1 \in L_p \cap \mathbf{VR}.\forall m_0, m_1 \quad \in \mathbf{PR}.$$
$$\mathit{PhysRegOverlap}(m_0, m_1) \rightarrow \mathit{use}^p_{n_0,m_0} + \mathit{alloc}_{n_1,m_1} \leq 1$$

4. If a physical register is live at program point p, a virtual register cannot be assigned that physical register for use at p:

$$\forall n \in \mathbf{VR}.\forall p \in \mathit{Uses}(x).\forall m_0 \in L_p \cap \mathbf{PR}. \quad \mathit{forall} m_1 \in \mathbf{PR}.$$
$$\mathit{PhysRegOverlap}(m_0, m_1) \rightarrow \mathit{use}^p_{m_1} = 0$$

A.2 Cost Model

The constant MOP represents the cost of a memory operation (a spill or load) and the constant MOV represents the cost of a move instruction. We let the weight of the spill of virtual register n, denoted $w(n)$, be the following:

$w(x) = \mathbb{E}[U_n] * MOP$, where $\mathbb{E}[U_n]$ is the expected number of uses of n during execution.

The sets **VR**, **PR**, and **CS** are the sets of Virtual, Physical, and Callee-save Registers respectively and **MP** is the set of move related pairs of registers.

The weight function $w : \mathbf{VR} \to \mathbb{R}$ is the approximate cost of spilling a virtual register. We define it as follows:

$$w(n) = \mathbb{E}[U_n] * MOP$$

where $\mathbb{E}[U_n]$ is the expected number of uses of n during execution.

Our cost model is the following:

$$min : \sum_{n \in \mathbf{VR}} w(n) \cdot spill_n + \left(2 \cdot MOP \cdot \left(\bigvee_{n \in \mathbf{VR}, m \in \mathbf{CS}, p \in Uses(x)} (alloc_{n,m} \vee {}^*use^p_{n,m}) \right)\right) -$$
$$\left(MOV \cdot \sum_{(n_0,n_1) \in \mathbf{MP} \cap (\mathbf{VR} \times \mathbf{VR}), m \in \mathbf{PR}} alloc_{n_0,k} \cdot alloc_{n_1,k}\right) - \left(MOV \cdot \sum_{(n,j) \in \mathbf{MP} \cap (\mathbf{VR} \times \mathbf{PR})} alloc_{n,j}\right)$$

We note that there is no disjunction operation in the .opb file format which is the required input for the sat4j solver. Thus, we add a decision variable *disjunction* and add constraints:

$$-\left(\sum_{i=0}^{n} x_i\right) + n \cdot \mathit{disjunction} \leq n - 1 \tag{A.1}$$

$$-\left(\sum_{i=0}^{n} x_i\right) + n \cdot \mathit{disjunction} \geq 0 \tag{A.2}$$

For $x \in \{0, 1\}$, these constraints will make the new decision variable take on the value of the disjunction: if there is at least one variable in the disjunction that has value 1 then *disjunction* must be 1 otherwise Constraint A.2 would not hold. If no variable has the value 1, then *disjunction* must be 0 or Constraint A.1 would not hold. Thus, this is a suitable encoding of disjunction using the operations available in the format.

A.3 Size Analysis

The number of constraints is $O(|\mathbf{VR}|+|\mathbf{CS}|+|\mathbf{VR}||\mathbf{PR}|+|\mathbf{VR}|^2+|P|(|\mathbf{VR}|^2|\mathbf{PR}|+|\mathbf{PR}|))$ where P is the set of all program points. This sum is dominated by the final term, so we conclude that the number of constraints is $O(|P||\mathbf{VR}|^2|\mathbf{PR}|)$.

References

[1] Boros, E. & Hammer, P. L. (2002). Pseudo-Boolean Optimization. *Discrete Applied Mathematics*, 123(1), 155–225.

[2] Bryant, R. E. & O'Hallaron, D. R. (2016). *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition.

[3] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., & Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6(1), 47–57.

[4] Fu, C. & Wilken, K. (2002). A Faster Optimal Register Allocator. *International Symposium on Microarchitecture: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, (pp. 245–256).

[5] Goodwin, D. W. & Wilken, K. D. (1996). Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. *Software: Practice and Experience*, 26(8), 929–965.

[6] Hack, S., Grund, D., & Goos, G. (2006). Register Allocation for Programs in SSA-Form. In A. Mycroft & A. Zeller (Eds.), *Compiler Construction* (pp. 247–262). Berlin, Heidelberg: Springer Berlin Heidelberg.

[7] Hames, L. & Scholz, B. (2006). Nearly Optimal Register Allocation with PBQP. In *Modular Programming Languages: 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006 Proceedings*, volume 4228 of *Lecture Notes in Computer Science* (pp. 346–361). Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Kannan, S. & Proebsting, T. (1998). Register Allocation in Structured Programs. *Journal of Algorithms*, 29(2), 223–237.

[9] Krause, P. K. (2013). Optimal Register Allocation in Polynomial Time. In R. Jhala & K. De Bosschere (Eds.), *Compiler Construction* (pp. 1–20). Berlin, Heidelberg: Springer Berlin Heidelberg.

[10] Krause, P. K. (2014). The complexity of register allocation. *Discrete Applied Mathematics*, 168(C), 51–59.

[11] Larabel, M. (2019). Icelake Compilers GCC Clang.

[12] Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO* (pp. 75–88). San Jose, CA, USA.

[13] Le Berre, D. & Parrain, A. (2010). The Sat4j library, release 2.2: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3), 59–64.

[14] Pereira, F. M. Q. & Palsberg, J. (2006). Register Allocation After Classical SSA Elimination is NP-Complete. In L. Aceto & A. Ingólfsdóttir (Eds.), *Foundations of Software Science and Computation Structures* (pp. 79–93). Berlin, Heidelberg: Springer Berlin Heidelberg.

[15] Pereira, F. M. Q. & Palsberg, J. (2008). Register Allocation by Puzzle Solving. *PLDI '08: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (pp. 216–226).

[16] Poletto, M. & Sarkar, V. (1999). Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5), 895–913.

[17] Proebsting, T. (1998). Proebsting's Law.

[18] Traub, O., Holloway, G., & Smith, M. D. (1998). Quality and Speed in Linear-scan Register Allocation. *ACM SIGPLAN Notices*, 33(5), 142–151.