TU WIEN Informatics

An SSA-based Register Allocator for the Glasgow Haskell Compiler

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Benjamin Maurer, BSc
Matrikelnummer 00826765

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 3. Oktober 2022

Benjamin Maurer Andreas Krall

Informatics

An SSA-based Register Allocator for the Glasgow Haskell Compiler

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Benjamin Maurer, BSc
Registration Number 00826765

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 3rd October, 2022

Benjamin Maurer	Andreas Krall

Erklärung zur Verfassung der Arbeit

Benjamin Maurer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Oktober 2022

Benjamin Maurer

Acknowledgements

I would like to thank my supervisor Prof. Dr. Andreas Krall, whose lectures first introduced me to the topic, for his patience and support.

My gratitude goes to my parents, Margit and Avram, who instilled a desire to learn and pursue higher education in me. The same goes for my older sisters, Daniela and Birgit, who were great role models and support while growing up and continue to be.

Furthermore, I'd like to thank the GHC developer community for accommodating me and for answering my many questions. In particular, I'd like to thank Andreas Klebinger, whose writing initially motivated me to start working on GHC, for his advice and expertise.

Finally, I want to express how grateful I am to my girlfriend Katrin. I want to thank her for her patience, generousness, all-round support and encouragement.

Kurzfassung

Registerallokation ist einer der letzten Schritte beim Übersetzen eines Programms. Dessen Aufgabe ist es Programmwerte den Prozessorregistern zuzuweisen und sie in den Arbeitsspeicher auszulagern, sollten davon nicht genügend verfügbar sein. Dieses leistungskritische Problem optimal zu lösen ist, im Allgemeinen, $\mathcal{NP}$-schwer. Die meisten, für die Praxis geeigneten Algorithmen, lassen sich als Graphenfärbe- oder Linear-Scan-Verfahren kategorisieren. Mehrere Forschungsgruppen haben 2005 entdeckt, dass die Konfliktgraphen von Programmen in Static Single Assignment (SSA) Form *chordal* sind. Die wichtigste Implikation dessen ist, dass dadurch eine klare Trennung der Allokationsphasen ermöglicht wird.

Der Glasgow Haskell Compiler (GHC) ist ein leistungsfähiger Übersetzer für die rein funktionale Programmiersprache Haskell. Sein "Native Code Generator" enthält sowohl einen Linear-Scan-, als auch Graphenfärbe-Allokator, wobei ersterer bessere Ergebnisse erzielt.

Diese Arbeit beschreibt Algorithmen zur SSA-basierten Registerallokation, insbesonders jenes Design, welches zur Umsetzung in GHC ausgewählt wurde. Eine detaillierte Evaluierung des neuen Allokators und ein Vergleich zu den Bestehenden wurde mithilfe GHCs *nofib* Benchmarksammlung durchgeführt. Während sich der neue, SSA-basierte Allokator dem Graphenfärbe-Allokator überlegen zeigt, kann er den Linear-Scan-Allokator nicht übertreffen. Ladebefehle wurden leicht verringert, wie auch Kopierbefehle, aber die Anzahl an Speicherbefehlen erhöht. Der SSA-basierte Allokator produziert Code, welcher im Durchschnitt 0.47% langsamer als jener des Linear-Scan-Allokators ist. Mögliche Ursachen und Verbesserungsmöglichkeiten werden besprochen.

Abstract

Register allocation is one of the last steps in the compilation pipeline. It is responsible for assigning program values to CPU registers and store them in memory when no free registers are available. Solving this performance critical task optimally is, in general, $\mathcal{NP}$-hard. Most practical algorithms fall in the categories of linear scan and graph coloring approaches. In 2005, several research groups discovered, that the interference graphs of programs in Static Single Assignment (SSA) form are *chordal*. The most important implication being, that this allows for a separation of the spill, assign and coalesce phases.

The Glasgow Haskell Compiler (GHC) is an industrial strength compiler for the purely functional programming language Haskell. Its Native Code Generator features both a linear scan and graph coloring register allocator, with the former delivering better performance.

This work discusses SSA-based register allocation algorithms, especially the design chosen for implementation in GHC. A detailed evaluation of the new allocator and comparison to the pre-existing ones is performed, using GHC's *nofib* benchmark suite. While the new SSA-based allocator proves superior over the graph coloring allocator, it does not surpass the linear scan allocator. Reloads are slightly decreased, as well as copies, but spills are higher. The SSA-allocator produced code is on average 0.47% slower than linear scan. Possible reasons for this, as well as opportunities for improvement are discussed.

Contents

CHAPTER 1

Introduction

Register allocation is one of the last steps in code generation, yet arguably one of the most important ones. It is responsible for allocating and assigning program values to CPU registers. Not only are CPU registers the fastest type of memory available, but - depending on CPU architecture - some or even most instructions can only operate on registers directly (as opposed to memory locations). A good register allocation, which holds the most frequently used values in registers and minimizes traffic to main memory, is vital for good program performance.

Register allocation and its sub-problems are very hard - in 1981 Chaitin et al.[CAC$^+$81] presented a graph coloring formulation of register allocation, where graph coloring is a famously $\mathcal{NP}$-complete problem. Therefore, clever heuristics are necessary to produce a good allocation within an acceptable time limit. Some of the most popular approaches[Per08] are graph coloring based[BCKT89, GA96] and linear scan[PS99, THS98] register allocation. The former is said to achieve better allocations, whereas the latter is considered faster (and thus suitable for Just-in-time (JIT) compilation).

Many extensions and improvements on both paradigms exist. An industrial grade compiler may have to tune its implementation to achieve competitive performance.

Haskell is a high-level, purely functional programming language with non-strict (or lazy) evaluation semantics. Due to the higher level of abstraction, Haskell programmers have to rely on the compiler to optimize their code and generate efficient machine instructions. The language's primary implementation, the Glasgow Haskell Compiler (GHC), supports several back-ends, with the default being the Native Code Generator backend (NCG), written in Haskell. NCG offers both a linear scan and a graph coloring register allocator. While the linear scan allocator is well optimized, the graph coloring allocator has received less attention and is not used by default at any optimization level, due to comparatively worse performance[1]

[1]https://gitlab.haskell.org/ghc/ghc/-/issues/7679 – Accessed 7.7.2022

It was observed, that the Graph Coloring (GC) allocator produces significantly slower output with more spills in a few benchmarks (e.g., *spectral/n-body*, ~62% slowdown, 2.3x spill code, see section 6.1), compared to the Linear Scan (LS) allocator (~0.8% slower overall). This is in part due to long live ranges and the "spill everywhere" approach of Graph Coloring Register Allocation (GCRA), i.e., a live range is spilled after each of its definitions and reloaded before each use. However, long live ranges may interfere with others at some point in the program, but not at different points.

One possible solution to this is *live range splitting*, where a live range is separated into two or more new live ranges, by inserting copy or memory instructions. Many approaches exist ([Bri92, CTS98, CE05, NIKN06]), which can be added onto the GCRA framework. However, when Briggs experimented with different splitting approaches in his dissertation, he saw mixed results. Therein, section 6.2.1, he writes:

> The key insight is that the interference graph captures none of the structure of the control flow graph. In reducing the allocation problem to a coloring problem, the compiler loses almost all information about the topography of the code. There is no representation for locality.

In 2005, several research independently discovered, that interference graphs of programs in SSA form are chordal [BDMS05, BDGR05, HG06, PP05] and such chordal graphs can be colored in polynomial time. More importantly, this enables new decoupled allocator designs, as lowering register pressure below k (for k available registers) with a spill phase guarantees that a coloring can be found without further spills (if register swaps are possible). This allows for the design of a spill algorithm, which does not follow a spill everywhere approach and takes program structure into account.

The goal of this work is to:

- Explore advantages and disadvantages of SSA-based register allocation and different allocator designs.
- Implement an SSA-based register allocator in GHC and compare it against existing allocators.

CHAPTER 2

Register Allocation

2.1 Task of a Register Allocator

A typical optimizing compiler can be divided into three stages.

The *front-end* parses the program text into some internal representation, e.g., an abstract syntax tree. Depending on the language, it may also perform type checking and other tasks. This stage is machine independent.

The *optimizer* (also called *"middle-end"*, analogous to the other stages, albeit being an oxymoron) takes an intermediate representation (IR) and performs optimizations on it, potentially translating it to other IRs more suited for the task at hand. The aim is producing a faster (or smaller) program, while preserving program semantics.

Lastly, the *back-end* (or *code generator*) performs instruction selection, instruction scheduling and register allocation, to emit target specific object code.

Each stage may translate the program to one or more IRs, specific to its objectives.

Typically, instruction selection emits instructions using an unlimited number of *virtual registers* (sometimes called "pseudo registers", or "pseudos"). As we want to generate valid machine instructions, these have to be replaced with references to actual machine registers. These physical, or "real", registers are small memory cells inside the CPU and as such the fastest type of memory available, but very limited in number. The number (and type) of machine registers available to the program are specified by the Instruction Set Architecture (ISA). While some ISAs allow a wide variety of instructions to access memory directly, in load-store architectures *only* load instructions can load a value from memory to register and vice-versa for store instructions.

To complicate matters further, some ISAs have instructions, which require *specific* registers as arguments and system calling conventions may require arguments to be passed in specific registers.

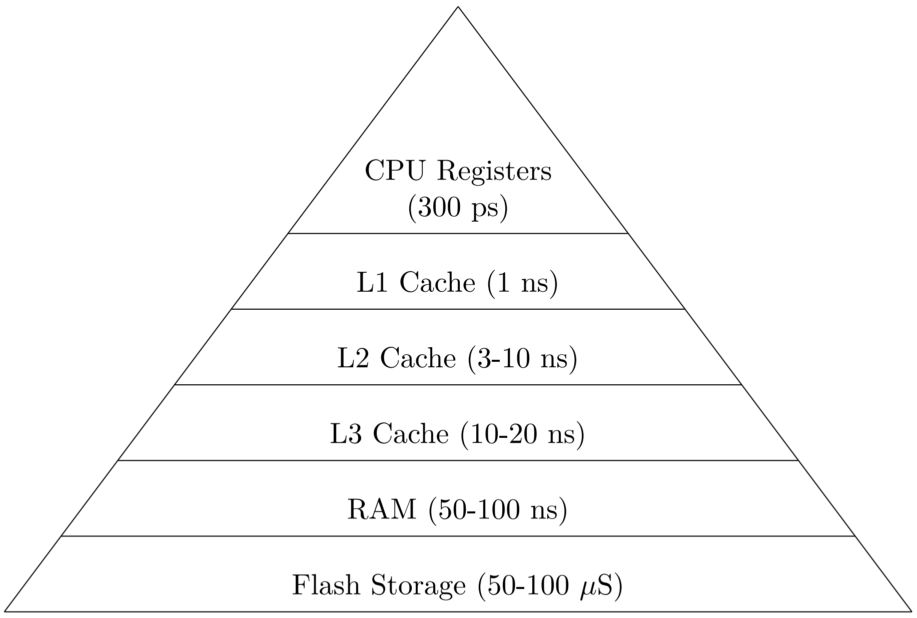


Figure 2.1: Memory hierarchy for a desktop

Register allocation also impacts program performance. CPU registers are the fastest type of memory available (see Figure 2.1, according to [HP17]) and so we want to minimize access to main memory, especially for frequently used values, e.g., in loops.

A good register allocation will also remove many register-to-register copies, by assigning the same physical register to the source and target of the copy, where possible.

The register allocator has to assign physical registers to instructions and insert spill/fill code, i.e., store values to memory and reload them from memory, such that these constraints are satisfied and program performance is as close to optimal as possible.

Speed of the allocator, i.e., time necessary to perform the allocation, is another factor. JIT compilers may impose strict time limits on compilation, but this is also a consideration for programmer productivity when using ahead-of-time compilers.

Register allocation can be performed *locally*, i.e., per basic block, or *globally* for the whole procedure at once. This thesis focuses only on global register allocation.

2.2 Live Ranges and Interference

Let us start by defining some terminology: A program may contain many *variables* and these may contain multiple *values*. During compilation, a program is translated into intermediate representations, analyzed and transformed. *Liveness analysis* constructs live ranges (LRs) - a variable v is said to be *live* at a point p in the program, if all paths from the start to p contain a *definition* of v and at least one path from p contains a *read* of v's value (without it being overwritten before). Or more simply put, a variable is live when it was defined and its value will be needed in the future. An LR for a variable v contains all the program points where it is *live* and may contain multiple definitions and uses in the general case. However, in Static Single Assignment (SSA)-form, a live range for an SSA-value may only contain one definition.

Two variables are said to *interfere* if they are both live at any point in the program. This means, that they cannot occupy the same physical register.

Since register allocation happens in the code generator, at a very "low level", symbolic names used in the code are called virtual registers (vregs), instead of variables. A vreg should only name one LR (otherwise a *renumbering* pass should rename them) and vregs are mapped to *real registers* by the register allocator.

2.3 Graph Coloring

One of the most prevalent formulations of register allocation is as a *graph coloring* problem. Chaitin et al.[CAC+81] proved that every undirected graph is the interference graph (IG) of some program. Thereby reducing register allocation to graph coloring, a known $\mathcal{NP}$-complete problem.

The graph coloring problem asks, whether we can color all vertices of a graph with at most k colors, such that no neighboring vertices have the same color. Or more formally:

A graph G with (V, E), is k-colorable, if we can find a color assignment $\rho(v)$, with at most k different colors, for all $v \in V$, s.t., no $\rho(u) = \rho(v)$, for $u, v \in V$ when $(u, v) \in E$.

The register allocation problem can be mapped to graph coloring in the following way:

First, the live ranges of all program variables have to be computed by liveness analysis. Then, the IG is built. The live ranges become the IGs vertices and edges between two vertices are added if the corresponding live ranges interfere, i.e., contain overlapping program points. Vertices connected by an edge can't be assigned the same color, i.e., interfering live ranges can't be assigned to the same register.

Formally, $IG = (V, E)$, where each variable $v \in V$ and for any two variables u, v live at the same time $(u, v) \in E$.

So k corresponds to the number of registers available and ρ is a register assignment. The *chromatic number* of a graph $\chi(G)$, is the smallest number of distinct colors necessary to color a graph. For many real-world programs, the number of available registers k will be less than $\chi(G)$.

In that case, variables have to be *spilled*. This means, that the value in the register is written to memory after definitions and read from memory before uses. New live ranges are constructed and vregs renamed ("renumbered"), then the IG is rebuilt and a new coloring attempt can start.

Inserting stores and loads for every definition/use of a live range is called the “spill everywhere” approach, which is often implicitly adopted by graph coloring register allocators.

Alternatively, live ranges can be *split*, e.g., by inserting copies, to create shorter live ranges which may fit into a register.

Interference region spilling tries to find the regions in which live ranges interfere and only place spill code there.

It can be cheaper to recompute a value, instead of storing it to and loading it from memory. This technique is called *rematerialization*.

2.3.1 Chaitin-Briggs Allocator

After building the IG, Chaitin’s allocator starts by **coalescing** copy-related nodes *aggressively*, if they do not interfere, thereby reducing copy instructions. However, this may lead to uncolorable graphs. Brigg’s[BCT94] improved allocator does this *conservatively*, only coalescing two live ranges if this does not impact the graph’s colorability.

In the **simplify** phase, nodes with degree less than k are removed from the graph and pushed onto a stack. If only nodes with degree greater or equal to k remain, Chaitin selects and removes one node for spilling (with a heuristic), whereas Briggs *optimistic coloring* places it also on the stack, hoping a suitable color can be found upon reinsertion.

The **select** phase pops nodes from the stack, selecting a color for them. Because in Chaitin’s allocator, each node has a degree less than k, this results in a valid coloring. If Brigg’s allocator cannot find a free color for a node in this step, it proceeds with to the next one. Any remaining nodes are spilled.

The **spilling** phase employs a *spill-everywhere* approach. Each definition of a variable v is followed by a `SPILL` instruction and every use preceded by a `RELOAD` instruction. This effectively splits LRs into extremely short segments.

After spilling, the reloaded LRs need to be assigned a new name. Chaitin calls this phase *renumbering*. Then, the IG is rebuilt and the cycle repeats, until all nodes can be colored.

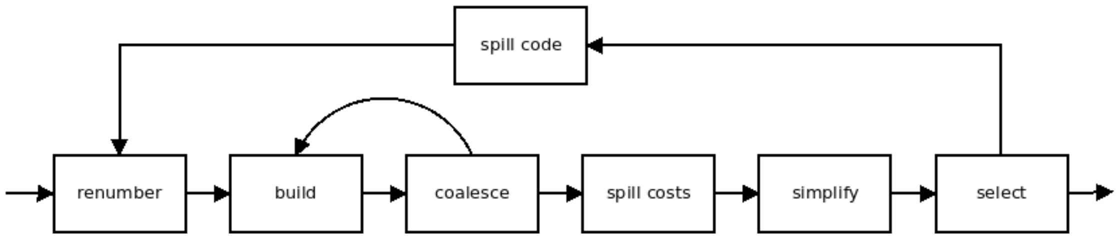


Figure 2.2: Phases of a Chaitin-Briggs-style allocator

2.3.2 Coalescing

George & Appel [GA96] improved on Brigg's work, by improving the coalescing criterion for *conservative coalescing* and by performing *Iterated Register Coalescing*, where *simplification* and *coalescing* are interleaved.

Park & Moon [PM04] describe *Optimistic Coalescing*, to remove more copies. They perform aggressive coalescing, but keep a set of former subsections of live ranges. When a node can't be colored, the merge decision can be undone. All subsets are checked, to find the most beneficial split.

2.3.3 Spilling, Splitting and Rematerialization

Bernstein et al.[BGG$^+$89] investigated ways to reduce inserted spill code. Most notably, they introduced a heuristic cost function that takes the best of three heuristics.

Bergner et al. introduced *Interference Region Spilling* in [BDEO97], trying to minimize the spill code inserted to regions where live ranges overlap.

Splitting live ranges, e.g., by inserting copy instructions, leads to shorter live ranges, which in turn may be colorable, whereas the whole one was not. Briggs experimented with aggressive splitting approaches in his PhD thesis [Bri92], that is splitting all live ranges around basic blocks, loops or at φ-nodes during SSA destruction, but saw mixed results.

Cooper et al.'s *passive splitting* [CTS98] splits live ranges *on-demand*, if it was selected for spilling and splitting seems profitable. In a follow up paper [CE05], they discuss issues with their original approach, suggest improvements and mention problems with the presented algorithm regarding 2-address instructions in CISC architectures (e.g., Intel's x86), however without presenting a solution.

Nakaike et al. [NIKN06] use profiling information to find suitable split points.

Another strategy, instead of spilling and reloading an LR to/from memory, is *rematerialization*. That is, to recompute a value if it is cheaper than storing and reloading it. Briggs describes adding rematerialization as an alternative to spilling in their allocator [BCT94].

2.3.4 Other Approaches

The Callahan-Koblenz allocator [CK91] performs hierarchical graph coloring. A tile tree is computed over the Control-flow graph (CFG) and allocation is performed for each tile separately, then results are combined. A comparison between the Callahan-Koblenz and Chaitin-Briggs allocators can be found in [CDE06].

Chow & Hennessy [CH90] use a priority based approach. Liveness information is not exact, but instead on the granularity of basic blocks. To that end, they limit basic block lengths, to lessen the impact on accuracy. LRs are colored in order of descending priority, promoting them from memory to a register. This is in contrast to Chaitin-Briggs allocators, that start by trying to hold all LRs in registers, spilling to memory when needed. The allocator uses live range splitting, when no register for the whole LR can be found. Cooper et al. [CHP] give a comparison of the Chow-Hennessy allocator to Chaitin-Briggs.

2.4 Linear Scan

An alternative paradigm is *Linear Scan Register Allocation*. The main advantage lies in its faster compilation speed as compared to GCRA, which makes it especially attractive for JIT compilation.

2.4.1 Classic Linear Scan

The original algorithm by Poletto & Sarkar [PS99] starts out by constructing *live intervals*. Instructions are expected to be numbered, e.g., in a depth-first traversal of the CFG, while the ordering scheme does not affect correctness, it does affect generated code quality. This numbering effectively flattens the control flow graph. To construct a *live interval*, we start at a definition for the variable v at program point i and extend it to the last use of v at point j. In other words, a live interval $[i, j]$ for a variable v means, that there is no point i' with $i' < i$ and no point j' with $j' > j$, s.t. v is live at i' or j'. Live intervals can be an *over approximation* of liveness, i.e., contain program points where v is not actually live. More precise live intervals can be beneficial (or necessary), depending on the linear scan approach used.

Live intervals interfere if they overlap. Overlap only changes at interval start and end points, so the algorithm maintains a list of intervals, sorted by end point. The *active* list contains at most k intervals, where k is the number of available registers. When a new interval starts and the active list is full, the interval with the furthest end point is spilled in its entirety.

2.4.2 Second-chance Binpacking

An approach which improves generated code quality is called “second-chance binpacking” by Traub et al. [THS98]. They observe, that live intervals may contain one or more sub-intervals, in which no useful value is maintained, calling these “lifetime holes”. Registers are viewed as “bins” which can contain one live interval at a time. Non-overlapping intervals can be assigned to the same bin, as well as live intervals that fit entirely into the lifetime holes of another interval.

Register allocation and rewriting happens in a single pass over the instruction stream. As the code is scanned, when a vreg is encountered that has not been assigned to a register yet, the algorithm tries to find a register (or “bin”) that is either free, or one with the smallest current lifetime hole large enough to fit the new interval inside.

If no such register can be found, the live interval with the furthest next use is spilled. Herein lies a key difference to the classic linear scan algorithm. The spill effectively *splits* the live interval and they are *not spilled everywhere*. It can be assigned a new register on its next use (a “second-chance”).

However, as thereby one live interval may be assigned to different registers in different blocks, the algorithm has to insert blocks with fixup code.

2.4.3 Other Approaches

Mössenböck and Pfeiffer [MP02] present a linear scan allocator that operates on SSA-form and makes use of lifetime holes. However, they do not use special properties SSA-form (such as parallel copy semantics), but insert copies for *phi*-nodes before the actual register assignment.

In [WM05], Wimmer and Mössenböck introduce various improvements to LS besides handling lifetime holes, such as handling architecture constraints with fixed intervals, optimal split positions, register hints for coalescing and spill store elimination.

Extended Linear Scan [SB07] presents a different formalization of register allocation from graph coloring. The algorithm explicitly prefer to insert copy and swap instructions, to avoid spills. They claim better compile and run times than GCRA, however, the algorithm spills intervals everywhere, i.e., does not support live range splitting.

Wimmer and Franz [WF10] exploit SSA properties for their simple and fast linear scan allocator. Liveness analysis is not needed to construct live intervals, due to the single definition property of SSA. SSA-form is deconstructed during the resolution phase, which creates fixup code for mismatched assignments due to liveness holes.

2.5 Optimal Register Allocation

The first integer linear programming (ILP) formulation of register allocation with spilling, rematerialization, live range splitting, coalescing and register targeting was proposed by Goodwin and Wilken in [GW96], with an improved approach published in [FWG05]. A generic ILP solver can be employed to solve such a formulation. According to the second paper, they were able to optimally allocate 98% of the functions in the SPEC CPU2000 integer benchmarks within 1024 seconds, with an average dynamic instruction count reduction of 10.6% over a Chaitin-style allocator.

Appel and George [AG01] split register allocation in two phases, namely spilling and coloring/coalescing. They provide an ILP formulation for the spilling problem on CISC machines, which includes instructions that can operate on memory directly. To make sure that the later coloring stage will find a coloring, they split all live ranges between adjacent instructions, using parallel moves. Optimistic coalescing ([PM04]) is used, to remove the large number of resulting moves. Limiting the ILP formulation to the spilling problem results in a much faster solution, as compared to Goodwin et al.

Colombet et al. [CBD14], too, focus on optimal spilling, their goal, however, is to study optimal spilling for SSA-based register allocation. Their ILP formulation can express φ-functions, rematerialization and the fact, that values can be in memory and register simultaneously (unlike Appel and George). They conclude that SSA-form does not provide clear advantages for the spilling problem (at least for non-Conventional Static Single Assignment (CSSA)-form) and that rematerialization can improve solutions significantly.

CHAPTER 3

Register Allocation on SSA-form

3.1 Static Single Assignment (SSA) Form

A program is in SSA form (or fulfills the SSA properties) [CFR+91], when every variable has exactly one definition and that definition dominates all of its uses.

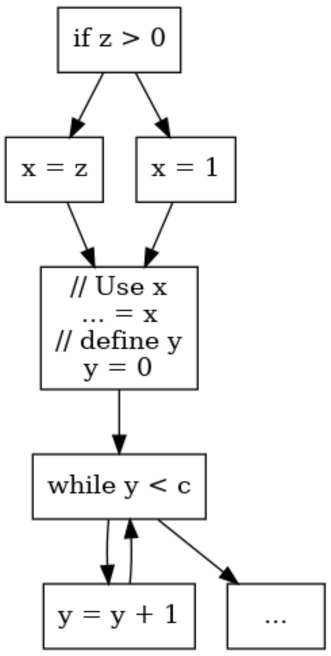
The program's CFG is a directed graph, where nodes represent basic blocks and edges represent jumps between them. A node d is said to *dominate* a node n, if all paths from the starting node to n must pass through d. A node d *strictly dominates* a node n, if d dominates n and $d \neq n$.

However, a non-trivial program has merging control-flow paths, on which data-flow merges too. For example, let's look at the program in Figure 3.1. The *if-then-else*-blocks would lead to several definitions for x, so we rename these two "versions" of x using subscripts. After the merge of control- (and data-) flow, we need a special instruction, the φ-node (or φ-function). It selects the correct value, depending on the branch taken, and we assign this to a new version of x, namely x_2.

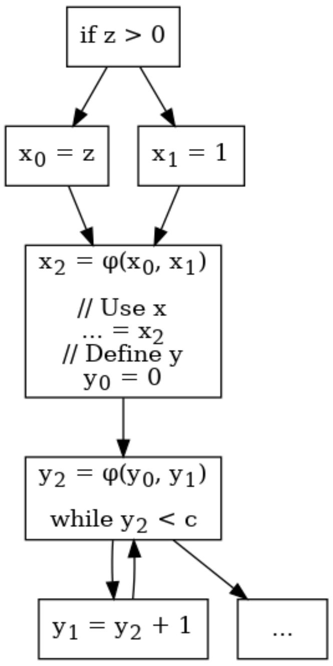
More formally, the value of a φ-function in block b with $n+1$ arguments, i.e., $\varphi(x_0, ..., x_n)$, is x_m when the control-flow comes from b's m-th predecessor.

φ-functions have *parallel copy semantics*: For a block with multiple φ-functions, all copies are performed in parallel at once and conceptually *along the CFG edges*. Therefore, the defined values are live-in to the block, but not the arguments.

A *φ-congruence class* [SJGS99] is the reflexive and transitive closure of all names connected by φ-functions. That is, it contains the φ-definition, all of its arguments and for those that are φs themselves, this continues recursively. One member can be representative of the whole φ-congruence class. These classes can be built using a union-find algorithm over the CFG.



(a) Non-SSA program



(b) SSA-form program

Figure 3.1: Example program before and after SSA transformation

Sreedhar et al. also define the following terms: After SSA-transformation, the code is said to be in CSSA form, i.e., there are no interferences inside a φ-congruence class. When optimizations were performed on the program in SSA form, it is said to be in Transformed Static Single Assignment (TSSA) and such interferences may exist.

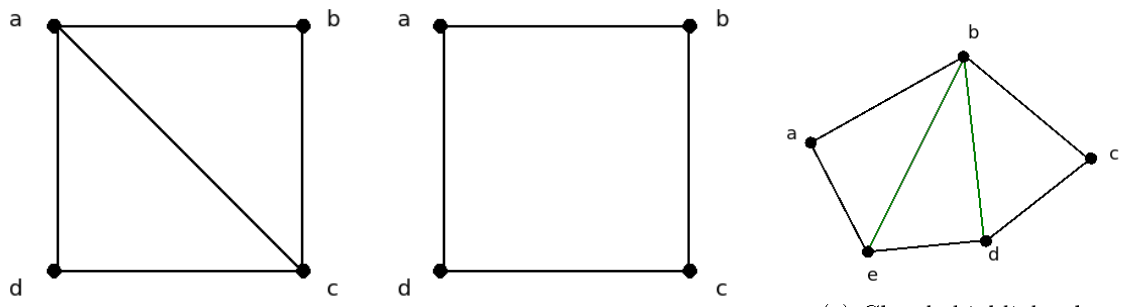
Translation out of SSA for a program in CSSA is straight forward: After computing the φ-congruence classes using union-find, every occurrence of a class member is replaced by the class representative and φ-functions can simply be removed.

For TSSA however, insertion of copies may be necessary [SJGS99, BDR+09].

An efficient algorithm for constructing SSA was described by Braun et al. [BBH+13].

3.2 Towards SSA-based Register Allocation

Pereira and Palsberg [PP05] examined the IGs of methods from the Java 1.5 standard library after SSA destruction, when compiled with the JoeQ compiler. They noticed, that 95% of these IGs were *chordal graphs*. Since the graph coloring problem can be solved in polynomial time on chordal graphs, they presented a heuristic which solved chordal instances optimally and still worked on non-chordal graphs.



(a) A chordal graph (b) A non-chordal graph (c) Chords highlighted

Figure 3.2: Examples of chordal and non-chordal graphs.

At the same time, three other research groups [BDMS05, BDGR05, HG06] discovered, that programs in SSA-form have chordal interference graphs.

3.2.1 Chordal Graphs

A graph is *chordal* (or *triangulated*, *rigid-circuit*), when each cycle with four or more edges has a *chord*, i.e., an edge connecting two vertices of the cycle, which itself is not part of the cycle.

Some examples are shown in Figure 3.2. The graph in Figure 3.2a is chordal, where the outer edges form a cycle and the edge in the middle is the chord. The second example in Figure 3.2b is not chordal, since the cycle of length four has no chord. Figure 3.2c shows a cycle of five edges, that can be turned into a chordal graph by adding the two green edges.

Many problems that are $\mathcal{NP}$-hard on general graphs, can be solved in polynomial time on chordal graphs, such as finding the maximum clique, maximum independent set and minimum coloring [Gol80]. Most importantly, a chordal graph $G = (V, E)$ can be colored in $O(|E| + |V|)$ time.

In section 3.1,the *dominance relation* was explained. Each block, except for the starting block, has a unique *immediate dominator*. By taking each node of the CFG, i.e., basic block of the program, and adding edges between a block and its immediate dominator, we can construct the *dominator tree*. The *dominance frontier* $DF(x)$ contains all the nodes that are direct successors of blocks that are strictly dominated by x. SSA construction places φ-nodes at exactly these dominance frontiers[CFR$^+$91], so the live ranges of SSA-values stretch from the definition dominating all uses, to just before the dominance frontier. That means every live range of an SSA-form program is a *subtree* of the program dominator tree.

Gavril showed in 1974 [Gav74], that chordal graphs can be characterized as intersection graphs of subtrees of a tree. The researchers mentioned in the introduction of this chapter used that property, to show that the interference graphs of SSA-form programs are chordal.

3.2.2 Consequences

The *chromatic number* of a graph $\chi(G)$, is the smallest number of distinct colors necessary to color a graph. Finding the chromatic number for a general graph is $\mathcal{NP}$-hard, but in chordal graphs, it is equal to the maximum clique size and can be found in polynomial time. Furthermore it is an *exact measure of register demand.*

So by lowering the register pressure to k though spilling, i.e., removing vertices from the graph and reducing the largest clique, we are guaranteed to find a valid coloring.

This enables register allocation in *separate phases*, such as spilling, coloring and coalescing. A decoupled design facilitates experimentation with new techniques and heuristics.

Note however, that this does not mean that register allocation can be solved in polynomial time. The solution is for the interference graph of an SSA-form program and cannot be converted to an optimal solution for a non-SSA-form program in polynomial time.

Many sub-problems of register allocation are still $\mathcal{NP}$-hard, even on SSA-form programs, such as optimal spilling and coalescing [Bou09].

3.2.3 Perfect Elimination Orders

For a chordal graph, so called Perfect Elimination Orders (PEOs) can be constructed. A PEO is built by iteratively removing *simplicial* vertices. A vertex v is called *simplicial*, if it forms a clique together with all its neighbors in a graph $G = (V, E)$. By removing a simplicial vertex, the graph stays chordal.

This can be used to find an ordering to color the graph. One can simply color and re-insert the vertices in the reverse order of the PEO. Since after spilling, the largest clique in the graph has size k, any vertex inserted has at most $k - 1$ neighbors, so a color must be available.

3.3 Spilling

The spilling phase's goal is to make sure, that register pressure at every point in the program is lower or equal to the number of available registers.

Generating efficient spill code is of paramount importance to program performance[KG09], however, most practically relevant formulations are $\mathcal{NP}$-hard[Bou09]. Bouchez et al. even show, that most of the complexity of the register allocation problem is due to the spilling problem. Furthermore, they conclude that SSA-form does not make the problem easier.

Nevertheless, the decoupled design of SSA-based register allocators allows for a spilling heuristics that is not constrained by the coloring heuristic used and that can take program structure into account.

Hack et al. employ a known algorithm by Belady et al., originally designed for virtual memory page replacement. It has been used for spilling in local register allocation before [FL98].

Belady's algorithm, or the "min" algorithm, works by scanning over instructions and keeping a set $InRegs$ of live ranges which are in a register. When this set is at capacity and a new value is defined, the LR with the *furthest next use* is spilled.

3.3.1 Global Next-Use Distance Analysis

Given a basic block B with i instructions, let's number these instructions from 1 to i. Each instruction is executed one after the other. If a variable v is used at instruction 1 and this is the instruction about to be executed, the next use distance is zero. On the other hand, if the current instruction is 1, but v is only used in instruction 5, the next use distance is 4, since we need to execute 4 instructions until we reach instruction 5.

It is very simple to compute the next use distances from the start of the block, by simply iterating backwards over the instructions.

To extend the notion of next use distance from basic blocks to the control flow graph, Hack et al. define it as the distance of *shortest path* from the current instruction to the next use, i.e., the *minimum distance.*

This can be computed with a standard liveness analysis pass. Instead of unioning live sets at merge points, maps of next use distance are merged using the minimum. Any variable that is *not live* has a next use distance of ∞. So it is easy to see that next use distance information entails liveness. Any variable with a finite next use distance ≥ 0 is *live.*

Braun and Hack [BH09] improve on this, by adding a *weight* (or length) to CFG edges. A loop analysis is used to discover loop nesting depth and edges exiting a loop are assigned a high weight (e.g., 100,000), to simulate that loops are executed often and uses "behind" a loop are further away.

A formal definition of the *join semi-lattice*, taken from [BH09]:

The analysis is performed on the domain:

$$\mathbb{D} = \mathbf{Var} \to \mathbb{N} \cup \{\infty\}$$

and the join of two maps $a, b \in \mathbb{D}$:

$$a \sqcup b = \lambda v.min\{a(v), b(v)\}$$

3.3.2 Belady's Algorithm Continued

The algorithm starts at the beginning of the basic block with a set W of values currently in registers. Each instruction I uses a number of (live) values, $use(I)$, and may define zero or more new values, denoted as $def(I)$. Any value used for the last time and is not live after the input phase of the instruction, is in the set denoted by $die(I)$. We assume that uses and definitions require registers for their values.

When the next instruction I is visited,

- Place reloads for all $v \in use(I)$ **not** in W before I

  Not enough space: If W does not have enough capacity, evacuate $|W| + |use(I)$ $W| - k$ values by placing spills before the reloads.

- Place definitions in registers: $|W = W \cup def(I)|$

  Not enough space: Spill $|W| + |def(I)| - k$ values before I.

Values with higher next use distance are spilled first. If a value has a next use distance of ∞, it won't be used anymore and no spill instruction has to be inserted at all. Any value with a next use distance of zero is used in the instruction and must not be spilled.

For *definitions*, we have to look at the next use distance from the *next instruction* on, i.e., the successor $succ(I)$ of I. Any instruction read for the last time in I is *dead* after it was read, so has an infinite next use distance in $succ(I)$. Again, no spill instruction is necessary.

Beyond Basic Blocks

To apply spilling to the whole procedure, the CFG is traversed in topological order. So we visit all predecessors of a block before the block itself, except for loop backedges.

The question is now, how do we initialize W and how do we connect assignments for different blocks?

In his thesis, Hack [Hac07] initializes W by sorting the live-in variables by next use distance, placing the first k in registers, thus optimistically assuming that they are passed by register from the predecessor blocks. The state at the end of the block W_B^{exit} is stored and fixup code is inserted to match up the exit sets of block predecessors with their entry set.

This simple scheme was improved by Braun & Hack [BH09] and chosen for implementation in this work.

Another set is introduced, namely S, which contains all the live values that have already been spilled on all incoming paths. This allows for the elision of redundant spill instructions.

Initialization of W and S distinguishes two cases:

In blocks **without incoming backedges**, that is non-loop-header blocks, we have already processed all predecessors P of B. W_B^{entry} is then initialized by looking at the exit states W_P^{exit}:

$$all_B = \bigcap_{P \in pred(B)} W_P^{exit} \qquad some_B = \bigcup_{P \in pred(B)} W_P^{exit}$$

W_B^{entry} is then set to all_B and filled up to capacity k by taking elements from $some_B\ all_B$, sorted by ascending next use distance.

The spilled set S_B^{entry} should be initialized to minimize inserted spill instructions. So it is set to all variables in a register at block start, that were spilled *on some path to B*:

$$S_B^{entry} = [\bigcup_{P \in pred(B)} S_P^{exit}] \cap W_B^{entry}$$

For **loop headers**, two considerations come into play. Due to the backedge, we do not know yet which values reside in registers at the end of the loop and secondly, we want to maximize register residency for values **used in** the loop.

Since we added a large weight to edges exiting loops, we can determine live-in values that are used inside the loop, as their next use distance will be lower than the exit edge weight. We fill W_B^{entry} with values v from $Live_{In}(B)$, sorted by next use distance $N(v)$, until a capacity of k is reached, or there are no longer values with $N(v) < w_{loop}$.

Any remaining space can be filled with values *live through* the loop, if they are guaranteed not to be spilled. We want to avoid unnecessary spills inside the loop, as they will be executed frequently.

To make this decision, we compute the *maximum register pressure* for loops, denoted by $RP(B)$, for any loop header B. We can simply compute this during next use distance analysis.

Let $T_B \subseteq Live_{In}(B)$ be the set with live through values. Then $RP(B) - |T_B|$ is the maximum register pressure in the loop, caused by values used in the loop. So we can put up to $k - (RP(B) - |T_B|)$ live through values into registers, being sure that they won't have to be spilled.

The exact initialization for S_B^{entry} in loop headers is not given in the paper, however this is the scheme used in this work:

We start by building S_B^{entry} just like in normal blocks, however, we do not have the spilled set coming from the loop backedge yet. So we add all values *used in the loop*, which are **not** in a register at the start of B. Since they are live-in to B and used inside the loop, they must be spilled on paths incoming to B, reloaded inside the loop for use and spilled again on a path back to B.

$$S_B^{entry} = ([\bigcup_{P \in pred(B)} S_P^{exit}] \cap W_B^{entry}) \cup \{v | v \in Live_{In}(B), v \notin W_B^{entry} and N(v) < w_{loop}\}$$

Connecting blocks Fixup code needs to be inserted between blocks, where exit and entry sets don't match. To line up a predecessor P of block B, everything in $(S_B^{entry} \S_P^{exit}) \cap W_B^{entry}$ needs to be spilled and in W_B^{entry}
W_P^{exit} reloaded from P to B.

For non-loop-header blocks, fixup code can be inserted right away, but for loop headers, this can only be done once the predecessor block connected by the backedge was processed.

3.3.3 Other Spilling Approaches

Pereira & Palsberg's [PP05] hybrid approach, which works both on chordal and non-chordal IGs, calculates all maximal cliques in the graph. They then determine the vertex v that is shared among the largest number of cliques and remove it. This is repeated until no clique of size larger than k remains.

Ebner et al. [ESK09] model spill code placement as a network problem. Nodes are placed for each CFG node where a given variable is live. Edges are placed between a node defining a value and its successors, to signify a spill and for every possible reload point. Costs can be associated with the edges, e.g., static costs, execution frequencies and rematerialization can be integrated as well, by modifying costs. The resulting constrained min-cut problem can be formulated as a linear integer program and be solved with an ILP solver. The paper also proposes a Lagrangian relaxation algorithm to progressively compute a solution, so as to choose how much compile time to spend for a desired code quality. For small register sets, of size 16 or less, the approach shows good speedups.

3.4 Preference Guided Register Assignment

In this section, we take an in-depth look at the combined approach, which was chosen for implementation. Other methods will be described in the next section.

Braun, Mallon and Hack [BMH10] present an approach, which combines register assignment and coalescing, while avoiding to materialize the IG. Because their results are close to a coloring approach, while much faster to compile (2.27 times faster than their recoloring approach) and appealing in its simplicity, it was chosen for implementation in this work.

The algorithm starts with the assumption, that spilling was already performed, i.e., register pressure is lower than k everywhere. In case register swaps are possible -

(whether by a dedicated instruction, or other means, e.g., 3 xor instructions) - no further spills will be introduced.

3.4.1 Register Assignment

The notion of *dominance* (see section 3.1) is central to what makes SSA-based register allocation work. When coloring an SSA IG with a heuristic like Chaitin's, that simplifies the graph by eliminating nodes with degree less than k, the vertex of a variable v will only be removed after all variables whose definitions dominate the definition of v and who interfere with v have been removed.

Any coloring order, which processes program points after all of their dominating program points have already been handled, leads to an optimal coloring.

The proposed algorithm scans over blocks, while performing assignment and rewriting of the code. At the start of a basic block, a bit set *occupied* is initialized with the registers assigned to live-in variables. Since in SSA-form, the definition of a variable dominates all points where it is alive and we visit blocks after all their dominators, all live-in variables were already assigned.

Next, φ functions are assigned. Their arguments are not live-in, so they are of no concern, but the variable they define needs to be assigned here.

Then the block's instructions are processed one by one. Any variable used, must have already been assigned, so we can rewrite the instruction. If a variable dies, its register can be released and marked as such in *occupied.* At a variable's definition, it is assigned a register (which is marked in *occupied*).

Register constraints

In the real world, not all registers can be used in each instruction and other constraints on registers used are caused by the Application Binary Interface (ABI).

To handle this, all LRs live across a constrained instruction are split by inserting a parallel copy right before it. This makes all registers available before the constrained instruction and the required one(s) can be assigned to the new live ranges.

3.4.2 Resolving Parallel Copies

Once all live ranges have been assigned registers, we have a register assignment for an SSA-form program. However, φ-functions (parallel copies) have to be eliminated to get an executable (non SSA) program.

To do this, we need to reconcile the assignments of the arguments and the destination of parallel copies. For example, let $\rho(x) : V \to R$ be a register assignment, i.e., a mapping from virtual register to real register and v_i the $i-th$ argument of $d = \varphi(v_0, .., v_i, .., v_n)$. Then the tuple $(\rho(v_i), \rho(d))$ describes a permutation of registers. This can be applied both to φs at the block start and to parallel copies before constrained instructions.

With a given list for register permutations, we can generate a series of moves, swaps or other instructions to resolve them. The algorithm is described in [Hac07].

To generate these permutations, we construct the *directed register transfer graph* T. Let Φ denote the set of φ-nodes at block B, $def(\Phi)$ all the values defined by φs in Φ and $\rho(x) : V \to R$ be a register assignment, i.e., a mapping from virtual register to real register.

The register transfer graph from the k-th predecessor of B is $T = (R, T_E)$, where R is the set of real registers and edges are drawn from the registers assigned to the k-th arguments of φ-functions in B to the assigned register of the φs definition:

$$T_E = \{(\rho(v_k), \rho(d)) | d \in def(\Phi), d = \phi(v_0, .., v_k, ..v_n)\}$$

Nodes may only have one incoming edge, since a result can only be written once to a register without overwriting it, but may have multiple incoming edges where virtual registers are copied.

The graph may contain fixed points, such as (r_x, r_x), which do not need to be resolved with instructions. More interestingly, we may find chains, like $r_x \to r_y \to r_z$ and cycles, as depicted in Figure 3.3.

To resolve the graph, the following steps are performed:

1. For any edge (r_x, r_y), with $r_x \neq r_y$ and where r_y has an out-degree of 0: Insert a move $r_x \to r_y$, remove r_y from T and change all paths starting from r_x to start from r_y instead (except for self-loops, to avoid unnecessary moves).
2. When no destination nodes with out-degree of 0 are left, T is either empty or contains only cycles of the form $C = \{(r_x, r_y), (r_y, r_z), (r_z, r_x)\}$.

   We can distinguish the following cases:

   a) A self-loop (r_x, r_x) does not require any moves and can be deleted from T.

   b) We have a free temporary register r_T and a cycle containing registers $r_1, .., r_n$. Then r_T can be used to break the cycle and issue a series of moves: $r_0 \to r_T$ $r_1 \to r_0$ ... $r_T \to r_n$

   c) No free register is available, so a register must be spilled temporarily, unless register swaps are possible. A spill slot can be used like a temporary register, in that a member of the cycle can be spilled and later reloaded into its destination register. If the platform supports register swaps - either through dedicated instructions, or arithmetic tricks (e.g., three xor-instructions) - then the permutations can be solved without spilling.
   For a cycle $(r_0, r_1), ..., (r_n, r_0)$, swapping r_0 and r_1 resolves the first edge, so it can be removed from T, but any outgoing edge at r_1 has to be moved to r_0 (Figure 3.3b).

$$\begin{tikzcd}
 & 0 \arrow[dr] & & & \\
3 \arrow[ur] & & 1 \arrow[dl] & & \\
 & 2 \arrow[ul] & & 4 \arrow[r] & 5
\end{tikzcd}$$

(a) Register transfer graph

$$\begin{tikzcd}
 & 0 \arrow[dd] & \\
3 \arrow[ur] & & 1 \\
 & 2 \arrow[ul] &
\end{tikzcd}$$

(b) Swapped 1 and 2

Figure 3.3: Register transfer graph

3.4.3 Coalescing with Preferences

While the approach described above may lead to a coloring that is optimal in the number of colors for an SSA-form program, this is not necessarily a good coloring in terms of performance.

When we look at register-to-register moves, e.g., $movv_1 \to v_2$, we notice that we can get rid of the instruction by assigning the same real register to both v_1 and v_2. This is particularly important to minimize the shuffle code inserted by resolving parallel copies.

However, optimal coalescing is $\mathcal{NP}$-hard, even on SSA-from programs[Bou09]. The heuristic described in [BMH10] collects register preferences and uses them during register assignment, trying to assign the preferred register to a live range where possible. Therefore, coalescing is performed together with register assignment and not as a separate pass.

Preferences are collected by a flow-insensitive analysis, which can be performed in one pass over the code. For every constrained use of a live range, an affinity (positive preference) is recorded for that register and a dislike (negative preference) for that register is recorded for all other live ranges at that instruction. This is so that a live range is preferentially assigned to a constrained register, while it is avoided for interfering live ranges.

So a preference is a mapping from virtual registers to a vector of weights: $pref(v) : V \to \mathbb{N}_0^k$ (note that weights may also be real numbers). We distinguish whether a live range is used or defined at an instruction i and we can associate different costs $\mathbf{c}_i^{def}$ or $\mathbf{c}_i^{use}$ with that. These costs are weighted by the estimated or measured execution frequency of the instruction, f_i. The preference function is then (adopted from [BMH10]):

$$pref(v) = \sum_{\{i|v \text{ is alive before } i\}} f_i * \mathbf{c}_l^{use}(v) + \sum_{\{i|v \text{ is alive after } i\}} f_i * \mathbf{c}_l^{def}(v)$$

Let $\Box \in \{use, def\}$, $\mathbf{e_i}$ the vector with all zeroes and a one in the i-th position and $\mathbf{1}$ the vector with all ones.

$$c_i^{\Box}(v) := \begin{cases} \mathbf{e_i} - \mathbf{1} & \text{if } v \in \text{ dom } constr_i^{\Box} \text{ and } i = constr_i^{\Box}(v) \\ -\sum_{i \in R} \mathbf{e_i} & \text{else} \quad \text{with } R = \{r|r \in constr_i^{\Box}\} \end{cases}$$

For example, for a machine with four registers r_0 to r_3 and a virtual register v with a use constrained to r_1 at instruction i, which is executed once, we get a preference vector $[-1, 1, -1, -1]$. A virtual register x, also live at i but with *no constrained use/definition*, would have a dislike for r_1 and thus $pref(x) = [0, -1, 0, 0]$.

Coalescing φs

As mentioned before, mismatched register assignments of φ-functions are another source of shuffle code. Since their register preferences are not fixed, they can't be recorded during register preference analysis.

Instead, as the register assignment algorithm is performed, φ affinities are propagated. At the start of a block, at least one predecessor must have already been processed (if there are any) and so at least one φ-argument is already assigned. A preference for the register(s) already assigned to a φ's arguments, weighted by the execution frequency of the argument's source block, is added for the φ-definition. Once the φ-definition was assigned a register, a preference for that (weighted by the block's execution frequency) is propagated to all yet uncolored φ-arguments.

In general, however, there may be interferences between φ arguments and definition. When that happens, propagating preferences won't lead to an *illegal* coloring, but maybe a less efficient one. This can be mitigated by using the live-in set to check for interferences.

Arguments to φs may be φs themselves and so we may build a whole graph of connected φs over which preferences are propagated. Any interfering live ranges are not connected to the graph by an edge. These φ-connected live ranges sharing affinities are called *affinity chunks* in the paper.

The liveness check above may prevent the addition of φs-components that interfere within one φ-function, but it does not result in the affinity chunk having no interferences at all. Interference free affinity chunks could be obtained by using *aggressive coalescing*, which is, however, an $\mathbb{NP}$-complete problem [Bou09]. A greedy algorithm could be employed instead, but it would require liveness checks against the whole affinity chunk.

Optimistic Moves

There are situations in which a live range's preferred register is already occupied by another live range. In this case, we may have to insert two moves before the constrained instruction - one to move the occupier away and another one to move to the target register. But it may beneficial to optimistically insert a move to free the register before the definition of our constrained live range. That way, we only insert one move and the live range is “born into” the right register. This is especially beneficial if the constrained use (and thus the two moves) lie in a more frequently executed block.

For this optimization, the register selection is modified. The algorithm goes through the variable's register preferences in order, searching for one that is free or can be freed through an optimistic move. When the variable's current preference is not free, we try to estimate whether we can find a beneficial new home for the current occupier.

To decide whether an optimistic move is beneficial, we take the *benefit* of the move, minus the *cost* of the move and check if it is larger than the current block's execution frequency. The *benefit* is the difference between the preference for the preferred register and the preference for the next available one. The *cost* is the difference between the occupiers preference for its current register and the next best available one:

$$\begin{aligned}
&benefit = pref(x, r) - pref(x, r^{x}_{next})\\
&cost = pref(other, r) - pref(other, r^{other}_{next})\\
&benefit - cost > f_B
\end{aligned}$$

Note that in [BMH10], this is instead given as:

$$(pref(x, r^{x}_{next}) - pref(x, r)) + (pref(other, r^{other}_{next}) - pref(other, r)) > f_B$$

However, this must be wrong, which can be shown with an example:

Let's assume the assignment for *other* was beneficial and it moderately prefers its current register r over the next best choice. E.g., let $pref(other, r) = 10$ and $pref(other, r^{other}_{next}) = 8$. Now x has a constrained use of r in a loop and has a *high preference* for r, but no preference for anything else, e.g., $pref(x, r) = 100$ and $pref(x, r^{x}_{next}) = 0$. Let us also assume that the current block is only executed once, i.e., $f_B = 1$. Such a scenario is a perfect case for optimistic moves. It is clearly beneficial to give r to x, instead of placing two moves into a loop.

But the equation from the paper gives us $(0 - 100) + (8 - 10) = -102 \not> 1$.

On the other hand, if we consider the benefit of giving r to x (100) and the cost for *other* yielding r (2), we get $100 - 2 = 98 > 1$.

3.4.4 Assignment Order

While any coloring order that is in dominance order can be used, Braun and Hack suggest an order based on execution frequency. This way, live ranges in more often executed code paths are more likely to receive their preferred register and shuffle code can be placed outside these paths.

The algorithm to compute the order is based on *execution traces*. A *trace* is a path in the CFG. The block's trace weights are computed by visiting them in reverse post-order and summing up the block's execution frequency (measured or estimated) with all its predecessors' execution frequencies (except for backedges).

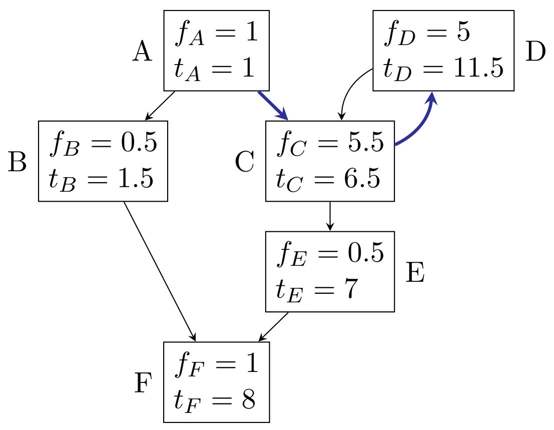


Figure 3.4: Example CFG with execution frequencies (f) and trace weights (t) from [BMH10].

Blocks are then sorted by their trace weight and traces are built greedily starting from the block with the highest weight, that is not part of a trace yet. From the chosen block, we build the trace by adding the unused predecessor with the highest weight and continuing the process from that block. We stop once the current block has no more predecessors, that are not yet part of a trace (or the start node, which does not have any predecessors). The path is then colored in reverse order.

Figure 3.4 shows an example CFG with execution frequencies and trace weights. The edges of the trace with the highest weight, which would be chosen first for assignment, are highlighted.

3.5 Coloring & Coalescing

In this section, we will look at various proposed heuristics for coloring and coalescing in SSA-based register allocators. The next section will handle one approach in-depth, which was chosen for implementation.

3.5.1 Recoloring

Finding a minimal coloring for a k-colorable chordal graph is simple. A PEO of the IG can be constructed by performing a post-order traversal of the dominance tree and a coloring can be obtained by reversing that order. Therefore, the IG can also be colored in order of a pre-order traversal of the dominance tree.

In [Hac07], a coloring is obtained this way and subsequently, φs are eliminated - both block-level φs and parallel copies inserted to handle register constraints. Parallel copy resolution is handled just like in the last section.

This may, however, leave many unnecessary copies in the program. Coalescing is treated as an (optional) optimization problem of the existing coloring, before φ-resolution. The IG is never modified to merge nodes, as to preserve its chordality, only the coloring is altered.

For each φ-function, an *affinity edge* is added to the IG for each φ-argument and definition pair, e.g., for $y_0 = \varphi(x_0, ..., x_n)$ we get $x_0 y_0, ..., x_n y_0$. Each edge is annotated with the cost of not fulfilling this affinity, e.g., by taking the execution frequency of the source block. We then try to find a register assignment that minimizes that cost.

Chunks of nodes connected by affinity edges want to be assigned the same color. However, there may be interferences within a chunk. So they are built bottom up, in a greedy fashion. Each node is first placed inside its own affinity chunk and affinity edges are then sorted by cost. For each edge, starting from the edge $x - y$ with the highest cost, the chunks of x and y are merged, if there are no interferences between their elements. Chunks are then placed in a priority queue, ordered by the aggregate cost of the affinity edges of the contained nodes.

Recoloring of chunks starts with the first element of the queue, i.e., the one with the highest cost and continues until the queue is empty. Nodes are recolored with every color, to find the one with minimal costs. Recoloring is performed recursively through the graph for a given node, changing interfering node's colors where they aren't fixed yet.

The new color may not be admissible for all nodes in a chunk, so these nodes are moved to a new chunk which is inserted into the priority queue. For the remaining nodes, the new color is *fixed*, so that it won't be changed in subsequent recolorings.

While this method can be performed by using the IG directly, Hack describes ways to *avoid building* the actual graph, which is expensive. This necessitates a different way to check for interference and enumerate a node's neighbors. Using the dominance relation, precomputed live-in and live-out sets, as well as a list of all uses of a variable, this can be performed on demand. Details can be found in [Hac07], section 4.5.2.1.

CHAPTER 4

The Glasgow Haskell Compiler

Haskell[1] is a purely functional, statically typed, general purpose programming language with non-strict (lazy) evaluation semantics.

Its primary implementation is the GHC[2], first released in 1992 [HHJW07], itself written mostly in Haskell (with some C and C–[JRR99], mostly for the run time system (RTS)).

GHC's compilation pipeline of intermediate representations can be seen in Figure 4.1. After parsing, the Haskell code is "desugared" into *Core*, a strongly typed, functional intermediate language. Next, Core is lowered to *STG*, the language for the Spineless Tagless G-Machine (STG)[Jon92], a virtual machine designed for Haskell's semantics. Following that comes *Cmm*, an implementation of C–[JRR99], a "portable assembly language that supports garbage collection" with C-like syntax. This is, of course, no longer a functional language.

There are three official backends which translate Cmm to either C-code, LLVM IR or directly to machine instructions. The last one is the default, performed by the NCG.

4.1 STG Registers

Haskell being translated to the STG abstract machine has an impact on the generated code and the register allocation problem tackled here.

Besides the regular call-stack, mostly used for register spilling and calls to functions not implemented in Haskell, a separate Haskell stack is maintained on the heap. The STG machine has several *registers*:

[1]https://www.haskell.org/ – Accessed 13.6.2022
[2]https://www.haskell.org/ghc/ – Accessed 13.6.2022

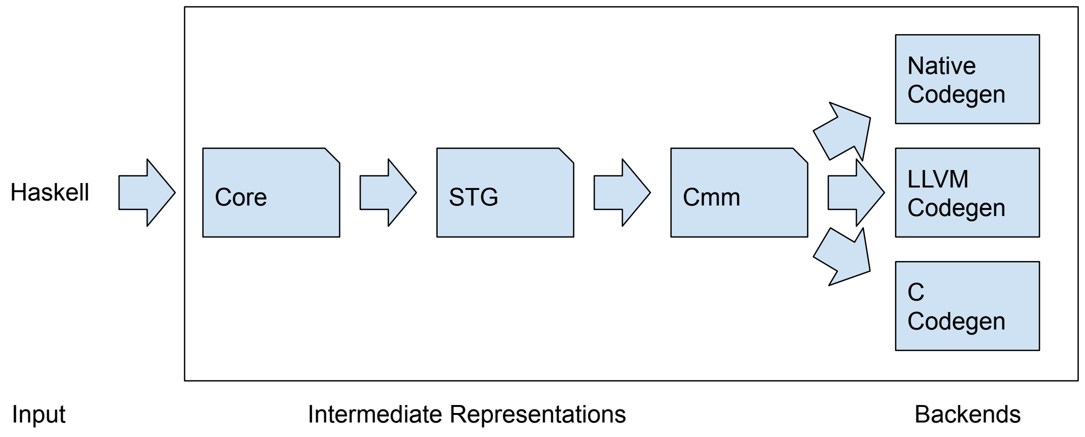


Figure 4.1: GHC compilation pipeline

- `BaseReg` - points to a static record, containing values of other registers if not allocated to real registers.
- `Sp` and `SpLim` - point to first and last available byte of Haskell stack.
- `Hp` and `HpLim` - point to first and last available byte of heap allocation space.
- `R1`..`R10`, `F1`..`F10`, etc. - temporary registers for argument and return value passing, per register class, e.g., words, floats, doubles, etc.

Some of these STG registers are mapped to hardware registers, depending on the architecture. Another subset of these hardware mapped registers are *not available* to the register allocator, raising register pressure further.

For example, the x86-64 architecture provides 16 general purpose registers, of which 14 can normally be used by a programmer (`rbp` and `rsp` are usually used to manage the call stack). However, a register allocator in NCG may only use 11.

4.2 Register Allocators

4.2.1 Linear Scan Allocator

GHC's linear scan allocator does not match any of the described algorithms directly, although it bares some resemblance to "second-chance binpacking" (see subsection 2.4.2). Contrary to what is typical for Linear Scan Register Allocation (LSRA), it does not build live intervals. Regular liveness analysis is performed to produce $Live_{In}$-sets and to annotate instructions with born/dies annotations. An example can be seen in listing 1.

The top-level Strongly Connected Components (SCCs) of the CFG are computed and blocks arranged in them. A list contains the topologically sorted SCCs, each of which is

```
c5gC:
    movq %vI_s4mr,%vI_n5HW
        # born:    %vI_n5HW
        # r_dying: %vI_s4mr

    andl $7,%vI_n5HW

    cmpq $1,%vI_n5HW
        # r_dying: %vI_n5HW

    jne _blk_c5gB

    jmp _blk_c5fh
        # r_dying: %vI_s4lr %vI_s4lQ %vI_s4lT %vI_s4lV %vI_s4m0
                   %vI_s4m1 %vI_s4m7 %vI_s4ma %vI_s4mu
```

Listing 1: A basic block with liveness annotated instructions.

either an *acyclic* SCC, containing one block, or a *cyclic* SCC, containing a topologically sorted list of blocks. Register allocation is performed in the order of these lists of lists.

The allocator starts at the entry block and as it processes instructions, it allocates new registers for definitions and assigns registers for already assigned vregs, keeping track of free registers in a bitvector and a mapping from vreg to *locations*. A location can be “in register”, “in memory” or “both”.

It tries to eliminate register-register moves where possible, by assigning the same register to the source and destination. When no free register can be found, the allocator has to *spill*. Unlike in the method of Traub et al., we do not have any “look-ahead”, i.e., no next-use distances or end points of live intervals. The spiller will prefer vregs that are in location “both”, because then it doesn’t have to insert a spill instruction. Otherwise an arbitrary register is evicted.

Reloaded values are preferentially assigned their first assigned register, to reduce fixup code, which is inserted between blocks in case of mismatching assignments and locations.

This simple algorithm is fast and performs surprisingly well.

4.2.2 Graph Coloring Register Allocator

The graph coloring register allocator in NCG is an implementation of the basic Chaitin-Briggs design, as described in subsection 2.3.1.

However, there is one big difference. It does not have a renumbering phase. Vregs are renamed *locally* when they are reloaded, i.e., an instruction `USE v` is replaced with the sequence `RELOAD s v'; USE v'`, where `s` is a stack slot and `v'` the new name for `v`. But the instruction selection pass may use the same vreg for disjoint live ranges.

This constrains the allocator too much, as it has to find a real register for the vreg's whole live range(s). These longer than necessary live ranges do sometimes result in unnecessary spills. In a previous project [Mau21], transforming into and out of SSA was used for renumbering, which lead to a modest performance improvement, at a disproportional compile time cost. The SSA transformation pass was, however, rewritten using a different algorithm for the new SSA-based register allocator.

The allocator does not perform live range splitting or rematerialization. Optional iterated coalescing can be activated using a compiler flag.

4.2.3 Performance Comparison

Using the evaluation setup described in chapter 6, we find that the code produced by the GCRA is 2.47% slower, on average (geometric mean). The minimum and maximum of relative performance are -15.19% and 60.76%.

Such slowdowns of 60% can be observed in *imaginary/kahan* and *shootout/n-body*. The GCRA produces more spills and reloads in these benchmarks, ostensibly due to its "spill everywhere" approach.

The improvement of 15% comes from the benchmark *spectral/hartel/wave4main*. However, the measurements for this benchmark show a relatively high standard error of the mean of 3.1% and 4.1%.

Only 27 out of 109 benchmarks compiled with the GCRA show an improvement in (wall clock) runtime over the linear scan allocator. It produces more spill code on average, but it is more effective at removing (coalescing) register-to-register copies.

Introducing a renumbering phase for the GCRA, to rename vregs of disjoint live ranges and using the classic spill cost heuristic by Chaitin[Cha82] does not remedy the situation.

CHAPTER 5

Implementing an SSA-based Register Allocator

5.1 SSA Transformation

SSA transformation was implemented using the Braun et al. algorithm [BBH$^+$13]. At this point in the code generator, that is after instruction selection, the intermediate representation consists of an abstract data type representing architecture specific assembly instructions, with some meta-instructions added. These type constructors take operands which represent immediate values, address modes or registers. Registers can be either *virtual registers* (type `VirtualReg`) or *real registers* (type `RealReg`). They are further distinguished by *register class* (integers, floats and doubles).

```
data PhiFun = PhiF !VirtualReg !VirtualReg [VirtualReg]
                   -- ^ old      def         args

data SSABasicBlock instr
               = SSABB [PhiFun] !(GenBasicBlock instr)
               deriving (Functor)
```

Listing 2: φ representation

Listing 2 shows the representation of φs in the code. A φ is a simple data constructor, with three fields. The "old" field contains the name before SSA transformation and is mostly there for SSA construction and debugging. The second field is the name defined by the φ and lastly, its arguments. Type invariants are, that all vregs must have the same register class and that the list of arguments is *positional*, i.e., must be sorted in order of CFG predecessors of the containing block.

At the basic block level, φs are block arguments and not part of the regular instruction stream. This makes handling them much easier, since accessing them is necessary to check whether live ranges reach a φ, whether a φ is defined in the local block or just live-in, etc.

5.1.1 SSA in a code generator

Using SSA-form in the code generator creates a few issues, stemming from ISA constraints. For example, instructions may have register constraints, i.e., require specific registers as arguments or for their return value.

Some have *implicit* inputs/outputs, e.g., the x86 `DIV` instruction, which takes the dividend from `rdx:rax` and the divisor as an argument, then writes the results back into `rdx:rax`.

Which leads to the next complication, namely *modifying instructions*, or *2-address instructions*, as they are used in many CISC architectures (like x86). This means that one operand is both an input *and* an output.

All of these properties pose a problem for SSA. Blindly issuing new names for constrained registers may break these assumptions. 2-address instructions make it impossible to rename the output register, unless they are modeled as 3-address instructions and later translated.

A deeper discussion of this issue and some solutions can be found in [LS99, dD14].

For the purpose of this allocator however, the solution is quite simple. The SSA-transformation only ever touches *virtual registers*, any register constraints are handled by the instruction selection pass inserting values of type `RealReg`, including copies from virtual registers to the constrained real registers.

2-address instructions are also left as-is and handled specially, where necessary.

This is possible, because we do not perform any optimizations other than register allocation on the code. Special care is only needed for 2-address instructions during SSA-transformation and repair, as no new name must be introduced for the input/output vreg and during spilling, where any modification removes the vreg from the *spilled set*. Treating LRs used in modifying instructions simply as one long live range is also done in [BDGR06].

5.1.2 SSA Repair

The implemented algorithm is also suitable for SSA repair, i.e., restoring the SSA properties, e.g., after spilling.

The repair function takes a set of reloaded vregs as an additional argument. This is an optimization, so the algorithm only has to look at relevant vregs that have actually changed.

When a φ's arguments are spilled, a reload is necessary to use the φ's value. That means the φ's definition can't be in a register at block start, i.e. is no longer live-in. The spiller filters out such φ instructions before calling the repair routine. φ arguments are also expected to be in order of the block's predecessors and the spiller must restore this invariant if it inserted any fixup blocks.

Spilling may also introduce new φ nodes. For example, take a variable x with one definition, that reaches block B via both of B's incoming edges. The SSA construction algorithm does not place trivial φs, like $x_1 = \varphi(x_0, x_0)$. But when x is spilled and reloads are placed on both paths to B, those reloads constitute new definitions and we have to place a φ node at B to unify those values.

5.2 Global Next-Use Distance Analysis

This analysis produces a map of basic block IDs to set of next-use distances at the start of the block. The set contains vregs with the number of instructions until their next use. φ-definitions are considered live-in, their next-use distance corresponds to their actual first use and any non-φ vreg used in the first instruction has a distance of zero.

Standard liveness analysis can be extended to compute global next-use distances. Instead of using the classical iterative data-flow analysis algorithm, which iterates until a fixed-point is reached, the *worklist algorithm* from [CHK04] was used and extended.

To go beyond basic blocks, liveness analysis unions live-sets at control-flow splits, but here we merge next-use distance maps by taking the *minimum* distance. This ensures convergence of the algorithm.

φ-functions are not considered uses. When calculating the next-use distance for a vreg v at a branch to a successor block s, we have to check whether v is live-in to s, or whether it is an argument to a φ-function p. The resulting next-use distance is then one plus the next-use distance at for either v or p at the start of s.

In order to reflect dynamic program structure, i.e., loops iterating for a long time, control-flow edges leaving loops get an added weight (e.g., 100.000). So any live vreg only used *after the loop* will have a high next-use distance just before and inside the loop. Thus the spiller will prefer to spill those vregs over ones used inside the loop.

5.2.1 Register Pressure

In order to decide which vregs should be kept in a register in loops, the spiller needs to know the *maximum register pressure* for each loop.

This can simply be computed alongside the next-use distance analysis. We just keep track how many vregs (per register class) are live at most, for each basic block. Since we already have loop analysis information available (used to identify edges leaving loops), we can create a mapping from loop-header to maximum loop register pressure.

We do not need to walk the whole next-use distance set for each instruction, but instead we can simply add vregs born and subtract those "dying" (last use). However, special care has to be taken for *CALL-instructions*. NCG models *call-clobbered* registers as writes, so liveness information carries a "dies" annotation for these registers. That creates a lower register pressure before and after the instruction, however, in this case we care about the register pressure "during" the instruction, i.e., how many registers can "survive" the function call, without being spilled.

This is a special case of a not *"register pressure faithful"* instruction, as described in [Hac07], section 4.6.2. That is, the register pressure and register *demand* do not agree. Another example is a pre-colored move instruction, e.g., `copy r1 -> r2`, where both are real registers. Even if `r1` dies at this instruction, the register demand is not one, but clearly two. Hack et al. handle this, by adding dummy arguments to those instructions. This, however, does not work with the existing data types used in NCG, so CALL- and pre-colored copy instructions are handled as special cases in the next-use analysis and spiller.

5.3 Spilling

We start out by computing φ-congruence classes in a pass over the program. This is done, so that we can spill all members of a φ-congruence class to the same stack slot. An invariant for the code is, that it is in CSSA, so no interferences between φ-arguments exist and this is safe.

Blocks are processed in reverse post-order, the *in-register* and *spilled* sets are initialized (see section 3.3.2) and then processes the blocks instructions. Spills and reloads are inserted and for branch instructions at the end of the block, the current sets are stored per outgoing CFG edge. These saved states are used to initialize the sets for successor blocks and to insert fixup code for non-backedges.

Handling of different register classes is intertwined. The in-register set W is partitioned by register classes and all classes are handled for each instruction.

After the whole CFG was processed, another pass inserts fixup code for backedges. Then the CFG is updated with any inserted fixup blocks.

The algorithm spills all or none of a φ's arguments. Since the arguments of a φ are not live-in, but its definition is, it is the defined vreg that gets to live in a register (or not). So, if the defined vreg is in a register at block start, its arguments were not spilled, or reloads were inserted before the block. On the other hand, if the defined vreg is **not** in a register, it must be spilled on all paths to the current block.

This means that φ-functions can become redundant. So after spilling, each block's φs are compared to the W sets saved at block starts. Any dead φ is removed.

Since φ-arguments are positional, this ordering is fixed for all φs in blocks where new fixup blocks were inserted as predecessors.

5.4 Preference Guided Register Assignment

The general workflow of the algorithm is this: Trace block orders are built, using predicted execution frequencies, computed by a pre-existing routine based on [WL94]. The assignment function is then applied in trace order. For each basic block, we fetch the incoming state, i.e., current register assignments, register preferences and a bitvector of free registers. There can only be one incoming state, as it is propagated along a trace.

The paper describes inserting shuffle code to fix φ assignments between blocks, where necessary. If the register assignment of any of the arguments of a φ-node and of the definition disagree, this inserts moves to adjust assignments. However, the SSA-transformation algorithm that was implemented here, does not place "trivial" φ-nodes. For a diamond shaped CFG, with a definition of x in the top node A and a use of x in the bottom node D, no function $x_1 = \varphi(x_0, x_0)$ gets inserted. This becomes a problem when we assign that graph in two traces, for example $A - B - D$ and then C separately. If C contains a constrained instruction and `enforceConstraints` move x to a different register, then there is no point to fix this assignment again.

So such a trivial φ is placed preemptively at the current block for each live-in variable, that is not a φ already, if that block has multiple predecessors.

We then proceed to bias the non-trivial φ-definitions towards the registers of their already assigned arguments. Due to the semantics of φs as parallel copies, there is no *inherent ordering* of φ functions. The paper does not mention it, but the sequential ordering chosen can affect the quality of the resulting assignment. When several φs compete for a specific register, *not assigning* it to a particularly φ will incur some cost. The reference implementation in libFirm[1] uses the Hungarian Algorithm to solve the assignment problem[Mun57]. In this case, the problem is formulated as a matrix of vregs and their preferences (weights) towards real registers. A solution which maximizes weight is sought.

This implementation simply sorts φ by their highest preference. A solution using the Hungarian method was tested, improving some benchmarks, worsening others and with a negligible impact on average (+0.16% CPU cycles), so the simpler solution was chosen.

After assigning registers to φ-definitions, a preference for that register is propagated to not yet assigned arguments.

[1]https://pp.ipd.kit.edu/firm/ – Accessed 23.6.2022

Then, the assignment function is applied to each instruction in the block. The function `enforceConstraints` is applied to each instruction (see subsection 5.4.2), then any *use* is rewritten using the assigned register and a register is assigned for any definition.

The destination operand of moves from virtual register to virtual register, where the source operand *dies*, are biased towards the register of the source operand. The preference (execution frequency) is reduced to 50%, to favor register constraints over coalescing moves. Weight factors can be adjusted, but 0.5 proved beneficial in experiments.

When a vreg's preferred real register is not available, `tryOptimisticMove` move tries to "steal" one. Optimistic moves are *not* performed recursively.

5.4.1 Recording Preferences

Live range affinities and dislikes are collected by scanning over all blocks and their instructions. A sparse representation of preferences is used, namely the `IntMap`[2] based finite map and set that is ubiquitous in GHC's code base. The analysis results in a map from `VirtualReg` to a map from `RealReg` to `Double`.

In the paper, the algorithm works by adding *negative weights* for all but the preferred register(s), to a live range's preference and adding a negative weight for the preferred register to *all other live ranges*, when encountering a constrained use. However, this is incompatible with a sparse representation, because one constrained use causes $k - 1$ values to be set. So instead, for a constrained use, a *positive weight* is added to the preferences of the affected live range, and a *negative weight* to all other active live ranges.

As NCG adds moves from virtual registers to real registers before constrained instructions and moves from real registers to virtual registers after them, those are the main sources of preferences before register assignment. Only if the source operand of such a move dies do we record a preference for the real register, hoping to get rid of the move. Otherwise a dislike is added to all live ranges for that real register.

Any other type of instruction defining real registers causes a dislike for those registers in all current live ranges.

Preferences for already assigned registers have to be propagated to φ-connected vregs during assignment. The libFirm implementation uses a graph based SSA intermediate representation, so φ-connections are already encoded within the IR. This is not the case in NCG, so we use the preference computation to also generate a φ *graph*, mapping the φ defined vreg to a list of tuples of φ-argument and the execution frequency of its source block (weight).

The data types used can be seen in 3, where `UniqFM` is an `IntMap` based finite map of *Uniques*, which are GHC internal IDs.

[2]https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-IntMap.html

Algorithm 5.1: Record Preferences

```
Input: Preferences prefs, Live-Set live, Instruction instr
Output: Updated Preferences
switch instr do
    case MOVE (RegVirtual src) (RegReal dst) do
        if $src \in dies(instr)$ then
            addPref(src, dst, weight)
        end
        else if $dst \in born(instr)$ then
            addPrefs(live, dst, -weight)
        end
    case MOVE (RegReal src) (RegVirtual dst) do
        if $src \in dies(instr)$ then
            addPref(dst, src, weight)
        end
    otherwise do
        foreach realReg in born(instr) do
            addPrefs(live, realReg, -weight)
        end
    end
end
```

```
type Weight = Double
type PhiGraph = UniqFM VirtualReg [(Weight, VirtualReg)]
type RegPrefs = UniqFM VirtualReg (UniqFM RealReg Weight)
```

Listing 3: Preferences and φ-Graph

5.4.2 Register Constraints

Section 2.2 of [BMH10] describes dealing with ABI constraints, e.g., instructions requiring their arguments to be in specific registers. The paper assumes that one splits all live ranges around constrained instructions, inserts parallel copies and resolves them to find a valid assignment.

Those parallel copies have to be implemented with real instructions, i.e., a sequence of moves and swaps. To find a good permutation, libFirm uses the Hungarian Algorithm again (just as for assigning φs).

However, NCG's instruction selection already inserts pre-colored move instructions from vregs to real registers, so that sequence is already predetermined.

Algorithm 5.2: Enforce constraints

```
Input: Instruction instr, Live-Set Live, Register assignment $\rho$, Bitvector of
       occupied registers occupied
Output: Updated: $\rho$, occupied; List of shuffle instructions
1  $constrainedRegs \leftarrow \{r | r \leftarrow written(instr), \text{where r is RealReg}\}$
2  $liveThrough \leftarrow live - die(instr)$
3  $badAssigs \leftarrow \rho(liveThrough) \cap constrainedRegs$
4  $sortedPrefs \leftarrow sortDesc(prefs(badAssigs))$
5  $shuffleInstructions \leftarrow \{\}$
6  foreach (vreg, prefs) in sortedPrefs do
7      $(\rho', occupied', shuffles) \leftarrow getRegister(\rho, occupied, constrainedRegs, vreg)$
8      $\rho \leftarrow \rho'$
9      $occupied \leftarrow occupied'$
10     $shuffleInstructions \leftarrow shuffleInstructions + shuffles$
11 end
12
```

Algorithm 5.2 shows the function *enforceConstraints*, which is applied to every instruction. If that instruction has any constrained registers, we insert any moves or swaps necessary to free those registers and record the new assignments.

Instead of formulating and solving an assignment problem, new register assignments are simply assigned in order of highest preference by any vreg. Since NCG inserts moves to real registers for constrained instructions, the routine usually just has to deal with a single assignment anyway. A set of *forbidden* registers, namely those with constrained uses, is employed to avoid reassignment.

Shuffle code may have to be inserted if an assignment was changed, that is expected in a successor block. However, fixup-code generation is based on φ-nodes, which poses a problem. For example, imagine a Y-shaped control flow, where the left branch and stem were already processed, while the right branch is the trace path currently in assignment. If we have a variable v, which was defined before the Y-region and is live across it, but there are no writes to v, then the SSA construction algorithm won't place a φ-node at the merge point. This is because the algorithm from [BBH+13] does not place “trivial” φ-functions, i.e., φs that do not merge at least two distinct values, excluding the φ-definition itself. But for v, we would get $p = \varphi(v, v)$.

To solve this issue, a trivial φ is placed in the successor block for each changed assignment of a variable v if:

- The successor block(s) have more than one predecessor.
- v was live-in at that successor to begin with.

5.4.3 Connecting Traces

When we assign different registers to the arguments and definition of φ-functions, we need to insert fixup code.

The fixup function is applied to each block, checking every edge from the current source block to some destination block. We extract the arguments reaching φs in the destination from the source, as well as the φs definitions and map them to their assigned real registers. To that end, φ-assignments are saved during assignment, as well as register assignments for each control flow edge.

When zipping the assignments of argument and definition together, we get register permutations as a list of tuples. For example, (r_1, r_0) means that the value in r_1 needs to be moved to r_0 before the destination's instructions are executed. The most common case, however, are fixed points, e.g., (r_0, r_0).

Permutations are then resolved by the algorithm described in [Hac07]. To efficiently implement this, the ST (State Thread) Monad [LJ94] is used, so that mutable state can be used inside a pure function, e.g., mutable arrays.

The resolution function uses two arrays, one for the permutations and one for the out-degree, that is, to how many destination registers a source register has to be copied. In the permutation array, the array index represents the destination register number and the element value the source register number.

The degrees array is initialized to all zeroes, then while iterating over the permutation tuples, permutations are written to the permutation array and for each source register the value at the corresponding index in the degrees array is incremented.

Resolving Non-Cyclic Permutations

We then iterate over the permutation array to resolve *non-cyclic* permutations. When source and destination differ and the destination degree is zero, a move from source to destination is recorded and the source's out-degree is reduced by one. To resolve the permutation, we set the source of destination to destination itself in the permutation array and record the destination register as occupied. If the source's degree has fallen to zero, we can mark the register as available.

To illustrate this, take the example in Table 5.1. At the start, registers r_0 and r_1 are occupied, but their contents need to be copied to r_1 and r_2 respectively. The permutation array shows that register number zero is a fixed point - nothing has to be moved to zero, so source and destination are the same. However, index 1 has source 0 and index 2 has source 1. Only the out-degree of 2 is zero, because it doesn't have to be moved anywhere.

Iteration begins at 0, but only at index 2 do we find and element with out-degree of zero. In the permutation array, source and destination differ, so we issue a move from register 1 to 2 and decrement the out-degree of the source (register 1). Register 2 is marked as occupied and since the source degree has fallen to zero, we can release register 1.

As the source degree is zero and its index is less than the current index, we loop back to source's index, performing the same steps again.

Graph	Permutations	Out-Degrees	Occupied
$0 \to 1 \to 2$	[0, 0, 1]	[1, 1, 0]	$\{r_0, r_1\}$
$0 \to 1$	[0, 0, 2]	[1, 0, 0]	$\{r_0, r_2\}$
	[0, 1, 2]	[0, 0, 0]	$\{r_1, r_2\}$

Table 5.1: Resolution of non-cyclic permutation.

Resolving Cycles

After performing the last steps, we are left with fixed points and cycles. We again iterate over the permutation array, checking whether destination (index) and source (value) differ. If so, this edge is part of a cycle and we start resolving it.

Two versions of this routine were implemented. The first one is a generic resolver using moves. This works for all register classes and on all platforms, but we need to perform a spill and a reload when no temporary register is available. The second one is platform specific and may use special instructions to swap values.

Starting with the generic routine (Figure 5.1): The first index is recorded as `start` index. If there is a free temporary register, we move the value in "start" to the temporary location, then call the resolver sub-routine, which moves the source to the destination, then continues the process visiting the source's element, issuing the next move, until the new source is equal to "start". This last, or "end", index is returned, so that the outer routine can issue a move from the temporary to the "end" register.

$$\begin{tikzcd}
 & 0 \arrow[dr] & \\
3 \arrow[ur] & & 1 \arrow[dl] \\
 & 2 \arrow[ul] &
\end{tikzcd}$$

(a) Cyclic permutation.

$$\begin{tikzcd}
 & & & & \text{tmp} \arrow[dr] & & \\
\text{start} & 0 \arrow[d] & & 0 & & 1 \arrow[d] & \text{end} \\
 & \text{tmp} & & 3 \arrow[u] & & 2 \arrow[ll] &
\end{tikzcd}$$

(b) Cycle broken with temporary.

Figure 5.1: Resolving cyclic permutation with temporary.

When there is no free register available, the process is the same, with the difference that we issue a spill instruction of "start" and a reload into "end".

As a possible optimization, which was not implemented, one could instead resolve cycles by register class and issue only one spill and reload per class.

The second, architecture specific function, may use special instructions or tricks to avoid the spilling. An optimized routine was implemented for the Intel x86_64 architecture, which provides `XCHG` instructions to swap the contents of general purpose registers.

Instead of resolving one cycle at a time, it is simpler to resolve all cycles for general purpose registers. Consider the cycle $x \to y \to z \to x$. Iterating over the permutation array from the beginning, we see that the current index x and element value z a different, i.e., have to be resolved. We look at the index z to find that y wants to move into z. A swap is issued between z and y, which *exchanges their contents*. The value that was in y is now in z, so we don't have to move that value anymore and we can mark it as resolved (setting the element at z to z). The *former value* in z, which wanted to move to x is now in y, so we update the permutation array by setting it to y at index x.

The `XCHG` based routine is *only used*, when no temporary register is available. Even with a temporary available, it would seems that for a 2-cycle, a single swap instruction would be more efficient than three register-to-register moves. However, benchmarks showed that this is in fact *slower*.

Since there is no swap instruction for floating-points, a different trick was used. On x86_64, the *sse* registers are used for floating-point values. These are 128 bits wide, so they can contain 4 32-bit single precision floating-point values, or 2 64-bit double precision values.

One can consider these as 2 "slots" for doubles, a "low" and a "high" one. The "low" slot is used for all scalar operations and since NCG does not support vectorized instructions yet, the "high" slots are always free. We can treat them as free temporary registers and resolve the cycle like in the generic case.

No performance comparison can be given however, since *not even a single benchmark* triggered this case. Given that there are 16 Streaming SIMD Extensions (SSE) registers available, of which none are reserved for the STG-machine and programs in general have lower register pressure for floating-point registers, this is not a huge surprise.

CHAPTER 6

Evaluation

6.1 Nofib Benchmark Suite

The GHC project maintains the *Nofib Benchmark Suite* (see [Par92]) to monitor changes in the compiler's performance. The main four categories of benchmarks included in the suite were used, as described in Table 6.1. Not included in the evaluation were the categories *gc*, which tests different garbage collector configurations, and *smp*, which exercises the scheduler of the Haskell threaded runtime.

Category	Description
Imaginary	Toy programs and puzzles, like "n-Queens".
Spectral	Small programs not large enough for "Real" and algorithmic kernels like FFT.
Real	Small, self-contained programs, like a raytracer.
Shootout	Programs from Debian's "The Computer Language Benchmarks Game" (formerly known as "Language Shootout").

Table 6.1: Nofib-Benchmark Categories

The benchmark-runner supports three *modes*, which affect the input size and runtime of the programs. They are "fast" (0.1-0.2 seconds), "normal" (1-2 seconds) and "slow" (5-10 seconds). However, these times are aspirational and the runtimes between programs in the same mode can vary widely. A few programs are not even parameterized at all and always execute with the same (small) input.

Therefore, Benchmarks with a runtime of less than 0.5 seconds in "normal" mode were excluded, as these short running benchmarks are very noisy and susceptible to interference by the system's background activity. It is also questionable what these benchmarks measure, as some spend more time starting up and shutting down, than performing real work.

The following programs were excluded:

- spectral/scc - Computes strongly connected components, but graph is hard-coded and tiny (wall time 1.05E-2s).
- spectral/pretty - Pretty printer: tiny, hard-coded amount of text (wall time 1.06E-2s).
- spectral/last-piece - Solves a hard-coded, small puzzle (wall time 0.48s).
- real/eff - Contains seven benchmarks of algebraic effect systems. Each benchmark simply adds one to an integer 10 million times, within the effect system. This is supposed to measure the overhead of the effect system, however 5 out of 7 benchmarks finish in less than 0.5 seconds and all spend between 60% and 80% of their runtime in a built-in routine for adding boxed integers. They do not require spill code and contain less than 100 register-register-moves. That makes them a very poor test for register allocation and any measurement likely contains more noise than signal.

Furthermore, the benchmark-runner supports running programs a given number of iterations and using programs like *perf*[1] and *valgrind*[2] to collect metrics, beyond the integrated ones (e.g., wall-clock time, garbage collector statistics, mutator time).

6.2 Test Setup

Taking precise performance measurements is hard. Many papers have identified various sources of measurement bias and ways to combat them ([MDHS09], [CB13], [KJ13], [CR16]).

A description of the test system, setup and measures taken against measurement bias follows.

Tests were performed on a Laptop with AMD Ryzen 7 4700U CPU (8 cores, 8 hardware threads (no hyper-threading), 2 GHz base clock), 16 GB (DDR4) RAM, running 64-Bit Ubuntu 20.04 (kernel 5.15.0-33-generic).

The system was booted into "recovery mode", i.e., a terminal environment without graphical user environment. *Dynamic overclocking* was **deactivated**. The Linux kernel's *CPU Scaling Governor* was set to **performance**. *Address Space Layout Randomization (ASLR)* was **deactivated**. Linux virtual memory *"swappiness"*, i.e., propensity to swap pages from RAM onto mass storage, was reduced to **10**, from the default of 60. Valgrind's

[1]perf: Linux profiling with performance counters - `https://perf.wiki.kernel.org/index.php/Main_Page`

[2]Valgrind instrumentation framework - `https://valgrind.org/`

cachegrind, which simulates a program's interaction with a simple cache hierarchy, was used to gather instruction counts and cache misses.

Using the *cpuset*-utility[3], a "shielded" set of 4 cores was created. The Linux scheduler will not assign normal threads to shielded CPU cores, and kernel threads only if necessary. While the benchmarks are single threaded, 4 cores were chosen to speed-up recompilation of all dependencies before each benchmark run.

A minimal shell, without environment variables, except for a 9 byte HOME- and a 102 byte PATH-variable, was launched inside the CPU set for each benchmark run. This is due to the effects the environment can have on memory alignment, as described in [MDHS09].

Benchmarks were run in *"normal"*-mode for *30 iterations* each.

Throughout this section, the following names for the compared allocators will be used:

- Linear - The linear scan allocator described in subsection 4.2.1.
- Graph - The graph coloring allocator described in subsection 4.2.2.
- Chaitin - The graph coloring allocator, but extended with an SSA-based renumbering phase before allocation and using the classic spill heuristic by Chaitin.
- SSA - The SSA-based allocator described in this work.

For each allocator, the compiler and base libraries were also compiled with the given allocator, with the exception of *Chaitin*. Using *Chaitin* to compile GHC itself triggered an unresolved error, so the GHC compiled with *Graph* was used.

6.3 Static Spills and Copies

To evaluate the new allocator, we will first look at the number of inserted spill code and remaining register-to-register copies. These static instruction counts are not a good predictor of runtime performance. For example, a single spill instruction inside a loop, which is executed one million times, will have a bigger impact than ten spill instructions in straight line code.

However, these numbers can give a rough idea of the efficacy of the allocators heuristics. Table 6.2 gives a summary of the inserted spills, reloads, combined spills and reloads, as well as remaining register-to-register copies, across all benchmarks.

Figure 6.1 shows plots for comparison of allocators. The linear scan allocator performs surprisingly well in terms of spill code inserted, but is least effective in removing copies. Performing renumbering through SSA, which splits some disjoint live ranges and using a

[3]https://github.com/lpechacek/cpuset

Allocator	Spills	Reloads	Spills+Reloads	Copies
Linear	4686	4670	9356	93942
SSA	5888	4581	10445	77013
Graph	9040	7622	16662	69166
Chaitin	6330	5702	12032	65882

Table 6.2: Spill code inserted and remaining copies.

more sophisticated spill cost heuristic proves beneficial for the graph coloring allocator ("Chaitin").

When comparing the SSA-based allocator to the linear scan allocator for all 117 benchmarks, the SSA-based allocator inserts fewer or equal amounts of spill instructions 68.22% (strictly fewer 19.62%) of the time and load instructions 79.43% (29.90%) of the time. Remaining copies are reduced in 92.52% of cases (no benchmark without decrease or increase exists).

6.4 Instruction Counts

Instruction counts, as reported by Cachegrind, give us a better idea of dynamically executed instructions. Runtime not only depends on the raw number of instructions, as a CPU generally takes a different number of cycles to complete different instructions, as well as on the latency incurred by cache misses and branch mispredictions, etc. This measure is, however, very robust and much less susceptible to noise and disturbances.

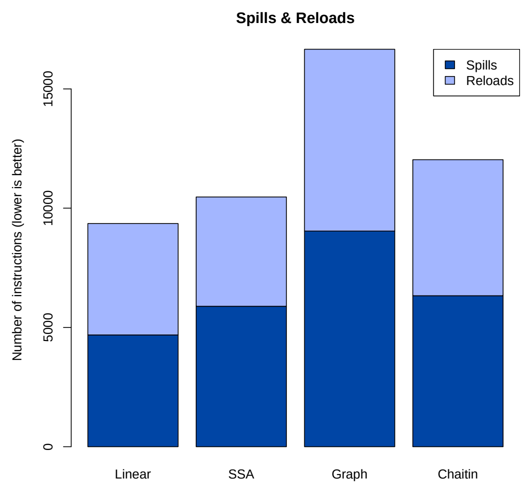
Allocator	imaginary	spectral	real	shootout	overall
Graph	1.53%	2.82%	3.18%	0.89%	2.63%
Chaitin	1.93%	2.50%	2.69%	-0.42%	2.29%
SSA	-0.02%	-0.43%	-0.07%	1.88%	-0.11%

Table 6.3: Change in instructions executed (linear scan baseline)

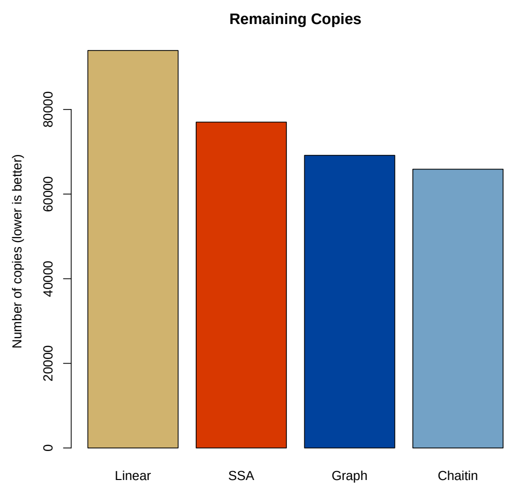
Table 6.3 shows the relative change in instructions executed, in percent, using the linear scan allocator as a baseline. Changes are given for the four benchmark categories and the suite overall.

The SSA-based allocator performs better than the graph coloring allocators in most cases. Its performance is almost on a par with the linear scan allocator, with the exception of the `shootout` benchmarks, where it performs much worse.

Looking at the constituent benchmarks in detail, Figure 6.2b shows, that this is largely due to a 18.54% increase in `shootout/fannkuch-redux`. In fact, all three allocators perform much worse than the linear scan allocator on this benchmark. This difference is less pronounce when we look at actual runtimes, but still present (e.g., +4.44% for SSA).



(a) Spills and Reloads



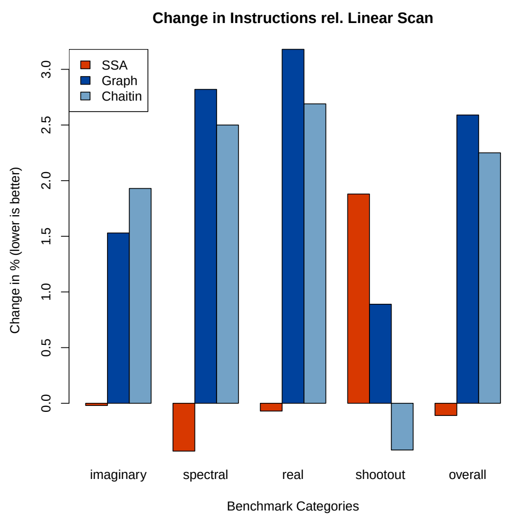
(b) Remaining Copies

Figure 6.1: Static Instruction Counts

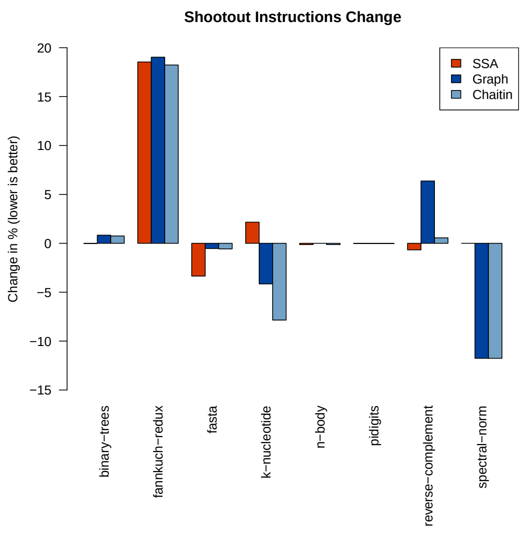
Static spill code counts are also higher for all other allocators, compared to linear scan (e.g., SSA produces ca. 11.5% more spill code).

6.5 Runtime

The metric we most care about is runtime. At the end of the day, we want our programs to run faster, no matter how many instructions are executed. What is measured here



(a) Change in instructions executed



(b) Shootout benchmarks in detail

is wall clock time. Unfortunately, this metric is also most susceptible to noise and disturbances.

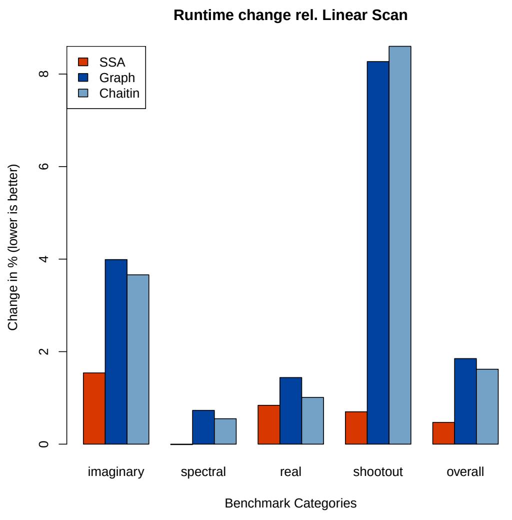
The test setup described in the introduction of this chapter is intended to reduce noise and control the environment. Out of the thirty measurements, the mean and the standard error of the mean are calculated. Some benchmarks proved to be very volatile, despite the countermeasures. All programs, that had test runs with a standard error greater 1% were excluded from the analysis (see Table 6.4). All of the excluded programs, except for `real/infer`, showed an improvement in runtime for the SSA-based allocator.

Benchmark	Linear	SSA	Graph	Chaitin
imaginary/wheel-sieve1	8.9e−2%	2.70%	3.00%	0.20%
real/infer	4.6e−2%	1.10%	1.10%	2.60%
shootout/reverse-complement	2.20%	1.17%	2.20%	2.10%
spectral/hartel/wave4main	3.10%	4.10%	3.90%	4.10%

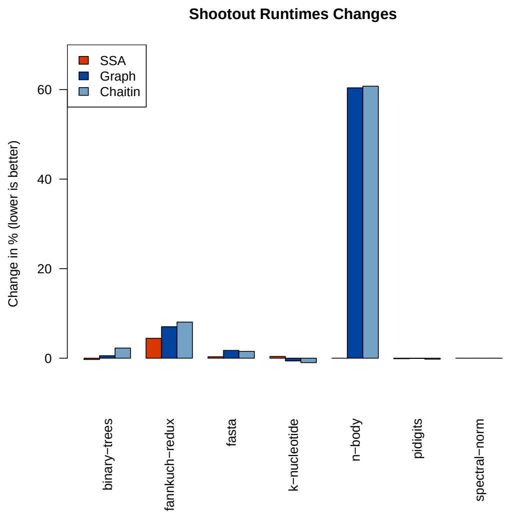
Table 6.4: Benchmarks with runtime standard error of the mean $> 1\%$

Figure 6.3a shows a comparison of changes in runtime as compared against the linear scan allocator baseline. On average, we see a slowdown of 0.47% for programs compiled with the SSA-based allocator over the linear scan allocator. The graph coloring allocators show slowdowns of 1.85% and 1.62% respectively.

Zooming in on the `shootout` category again (Figure 6.3b), we see a 4.44% slowdown in `shootout/fannkuch-redux`. This correlates with the 18.54% increase in executed instructions, reported in the last section. Compared to linear scan, the SSA-based allocator inserts more spills (92 vs. 161) and a lot more copies (361 vs. 577). Of these



(a) Change in runtime



(b) Shootout benchmarks in detail

copies, 10 were inserted as optimistic moves. Additionally, 12 `XCHG` instructions were inserted for swaps. These are, in general, slower than `MOV` instructions and only used when no free scratch register is available.

The graph coloring allocators fare even worse on this benchmark. Noteworthy is also the massive, ca. 60% slowdown on `shootout/n-body` for the graph coloring allocators. The linear scan allocator does not produce spill code for this benchmark and neither does the SSA-based one. The latter even removes more copies, but the runtimes are almost equal. Strangely, while the graph coloring allocator *does* insert spills and reloads, which may explain the slowdown, the chaitin variant *does not* insert spill code. The number of copies remaining is even *one less* than for the SSA-based allocator. Static instruction counts seem not to be able to explain this behavior. Interestingly, neither do instructions executed as measured by Cachegrind correlate with the observed runtimes. As the focus of this work lies on the allocator presented and implemented therein, no further investigation was performed.

Figure 6.4 shows a scatter plot of all runtime changes for the SSA-based allocator over the linear scan allocator. The biggest outlier is `imaginary/kahan`, a slowdown of 24.37%. This calls for a closer look at what is happening here.

6.5.1 `imaginary/kahan`

This microbenchmark is an implementation of the Kahan summation algorithm [Kah65]. It compiles down to a tight loop, in which it spends 99% of execution time. This loop contains a C-function call (which has different calling conventions from Haskell internal calls), which necessitates storing the caller-saves registers on the stack before the call and reloading them afterwards.

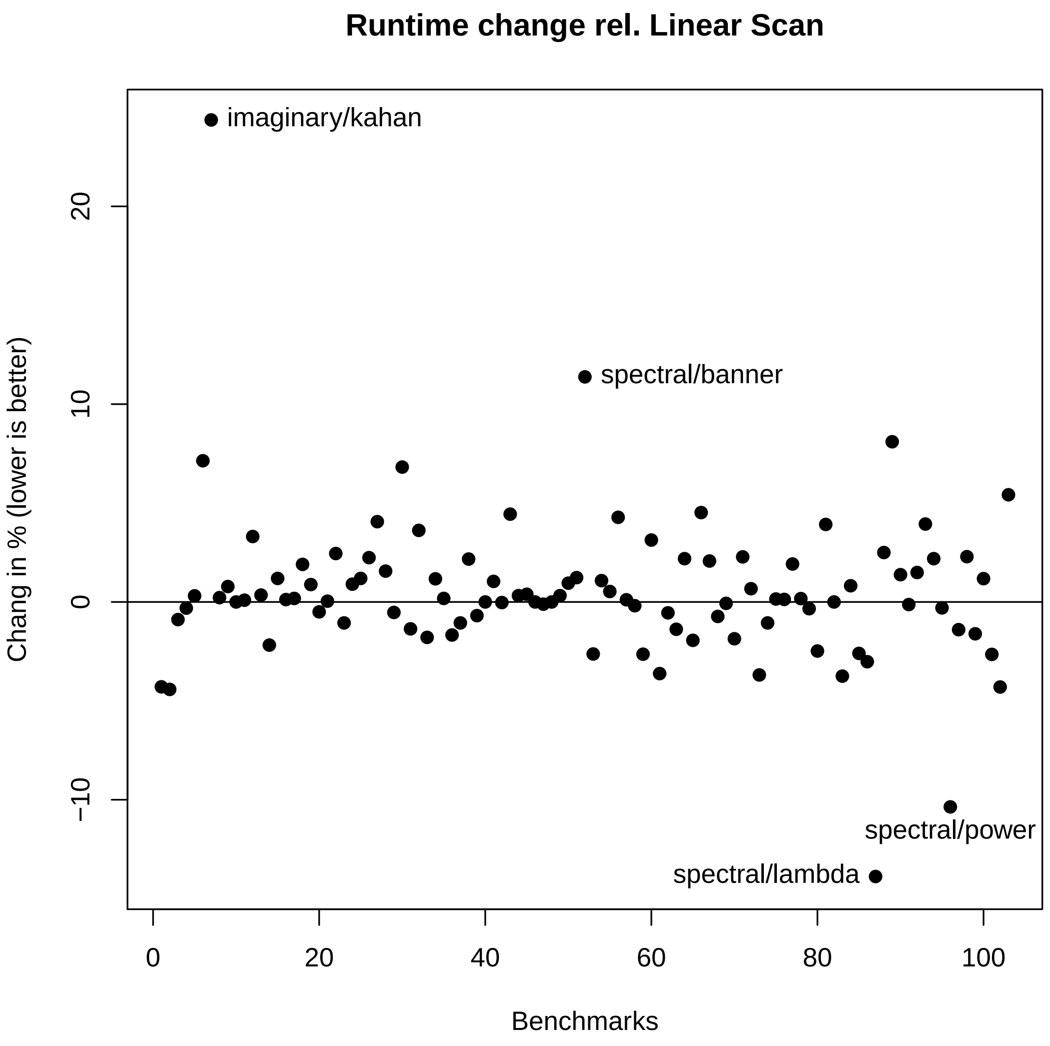
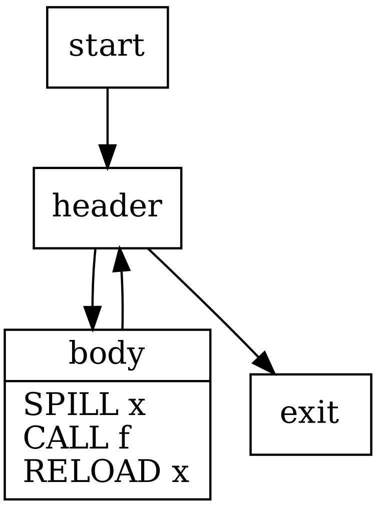
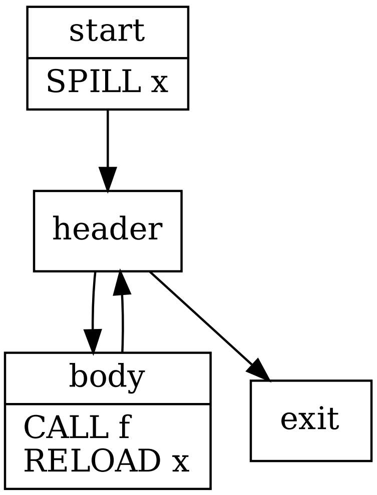


Figure 6.4: Runtime change of all benchmarks for SSA



(a) Schematic CFG of `imaginary/kahan`



(b) `imaginary/kahan` CFG with hoisted spill

Both the linear scan and SSA-based allocator insert the same number of spill and reload instructions, however they do spill different live ranges. At first, these choices seemed insignificant, one of the live range chosen by the SSA-based allocator is even a better choice. But upon closer investigation, using the uiCA CPU simulation tool [AR22], the spill choice lengthened the critical path of the loop, thereby lowering its throughput. Because the program spends almost all of its time in this loop, even this small difference created a big performance regression.

In the paper describing the second-chance binpacking algorithm [THS98], the “furthest next use” spill heuristic is employed. GHC’s linear scan allocator, however, simply picks the next live range in a list, which is sorted by the internal ID used. Therefore, it performs better simply by chance. Loop carried dependencies seem like an interesting factor to consider for spill cost heuristics in future research.

Further analyzing the body of `imaginary/kahan`’s inner loop, it can be observed, that some of the live ranges which cannot survive the function call in a register are never modified inside the loop. This can be used to hoist the spill out of the loop and reload the value after the call, as depicted in Figure 6.5b.

Several attempts to integrate an ad-hoc heuristic for spill hoisting showed mixed results. At the loop header, the spiller can consider which live ranges are used inside and the maximum register pressure in the loop. Any of the unused live ranges can be spilled for the loop. But even if we know, that some of the live ranges used in the loop will have to be spilled at some point, we don’t know which one is beneficial to spill beforehand. For example, spilling a live range that ends before the point of maximum register pressure won't help. To complicate things further, as described in subsection 5.1.1, we may have vregs that are modified within the loop.

In one attempt, used-in-loop live ranges were spilled before the loop, yet also given a register and the spiller inside the loop was biased to prefer evicting already spilled live ranges. This approach is not targeted, as there is no guarantee that this will spill the live ranges that are live across the point of maximum register pressure. In the case where a vreg gets modified, we have to spill it again afterwards, rendering the spill before the loop useless.

Another attempt tried to hoist inserted spills after they were placed. This has several downsides. Given two spill candidates, the spiller may choose to spill the live range whose spill instruction cannot be hoisted. Both approaches can be complicated by nested loops, e.g., how far out should we try to hoist the spill?

Results were very mixed. While most of the experiments fixed the regression in `imaginary/kahan`, results overall got slightly worse.

Future work could investigate how to integrate such a mechanism into a spiller. For example, one could tag "hoistable" live ranges during an analysis pass, similar to rematerialization tags.

6.5.2 `spectral/banner`

Another outlier, with a slowdown of approximately 9.8%, is the benchmark `spectral/banner`. This program takes its text input and turns it repeatedly into an ASCII art banner.

The slowdown can mostly be attributed to a library function for encoding UTF8 text (`GHC.IO.Encoding.UTF8.utf8_encode`). GHC generates a CFG here, which proves challenging for the spiller. From a central node, let's call it H, two outgoing edges turn into a long and ramified part of the graph. Ultimately, six edges branch back to H, additionally to the one incoming non-backedge.

The SSA allocator ends up inserting copious amounts of fixup code (spills, loads, moves, swaps). For the complete `GHC.IO.Encoding.UTF8` module, out of 247 register to register copies, 189 were inserted by the allocator. Of those, 132 are optimistic move instructions and 57 shuffle instructions. Additionally 155 swap instructions were inserted as shuffle code.

While the linear scan allocator also has to spill and insert fixup code, it ends up needing much less.

6.6 Compile Times

Compile times with the different allocators were evaluated by compiling the *Cabal* package manager library with *-O1* and the respective allocator flags. The metrics were reported

Allocator	Spills	Reloads	Spills+Reloads	Copies
Linear	143	470	613	499
SSA	388	688	1076	247
Graph	199	592	791	226
Chaitin	257	708	965	262

Table 6.5: Spill code in `UTF8.hs`.

by the Haskell Runtime System (RTS).

The first column contains the approximate number of total allocations (in MiB). This is important, because many short-lived allocations put more strain on the garbage collector and can be a source of performance problems in Haskell programs. The second column contains the peak amount of memory allocated by the RTS. *MUT* stands for *mutator* and measures the time spent in the actual program (the compiler in this case), whereas *GC* stands for the time spent in the *garbage collector*. The last column, *Total*, contains both *MUT* and *GC*, but also initialization and exit times.

Allocator	Allocations (MiB)	Peak (MiB)	MUT	GC	Total
Linear	$\approx 171,600$	1113	111.790s	22.909s	134.702s
SSA	$\approx 226,566$	1071	135.197s	24.440s	159.639s
Graph	$\approx 206,194$	1264	125.417s	23.439s	148.860s
Chaitin	$\approx 222,993$	1130	133.099s	24.672s	157.772s

Table 6.6: Cabal compilation metrics.

It is important to note, that the SSA-based allocator is a prototype and not optimized for compile time or memory usage in any way. By comparing *Graph* and *Chaitin*, one can get a sense of the overhead involved in translating into (and out of) SSA-form.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

This master thesis explores SSA-based register allocation and compares implementations of an SSA-based design, with traditional linear scan and graph coloring register allocators in the Native Code Generator backend (NCG) of the Glasgow Haskell Compiler (GHC).

One of the key benefits of register allocation on SSA-form is, that it enables a separation of allocation phases. This more modular design allows for different combinations of approaches to sub-problems. In particular, the implemented spiller design takes program structure into account.

Colombet et al.[CBD14] conclude, that SSA-form may provide advantages during register assignment, but introduces complications during spilling. While these problems did not occur in this particular implementation, as the code is guaranteed to be in CSSA, SSA-form did complicate many parts of the code. Correctly mapping φ-arguments to their definition and propagating information across branches proved very error prone. However, this may be due to the SSA representation implemented.

The spill-everywhere approach employed by Chaitin-Briggs-allocators seems unappealing, because this may cause spills in areas without excessive register pressure and within loops. Hack and Braun's approach, on the other hand, attempt to spill where necessary and specifically tries to avoid spills in loops. However, when inspecting some of the worst performing benchmarks, it seems like the code generated by GHC does not lend itself to this approach. At least for high optimization levels, GHC produces CFGs with complicated loops, where one block may participate in several loops and loop headers may have several backedges. Here, the amount of shuffle and fixup code inserted seems to impact performance negatively.

The published literature seems to contain very few spilling heuristics, despite there certainly being more avenues to explore.

Preference guided register assignment proved to be a relatively simple solution, taking care of register assignment, coalescing and out-of-SSA transformation all in one. It provides adequate reduction of copy instructions at moderate compile time cost. While other approaches may be able to coalesce more live ranges, remaining copies showed less impact on runtimes than other factors. The biggest source of implementation complexity were certainly the resolution of register permutations.

7.1.1 Architecture specific aspects

Properties of the CPU Architecture play an important role here. Swap instructions (`XCHG`) on x86-64 were slower than copy instructions with a temporary register, but performed better than spilling.

It seems that, in general, architecture specific characteristics can have a major impact on performance, while not being considered by the underlying cost models. For example, during development of the register assignment phase, some benchmarks seemed strangely slow. As it turned out, register candidates were sorted in reverse, so that *higher* register numbers were preferred. But using "new" registers, which were added for the 64-bit extension of the x86 architecture, requires an additional byte in the instruction encoding, in some cases. This small change caused a slowdown of 50% in one benchmark.

The observed slowdown of *imaginary/kahan* can also be attributed to spilling a live range on the critical path of the loop. Whether data dependencies can be reasonably incorporated into a spill cost heuristic is questionable.

Most architectures have some form of irregularity in their register sets, like aliasing, where multiple names may refer to parts of a register, or register pairs. These irregularities are not handled by the implemented algorithms. While this poses no problem for this implementation, since NCG does not produce code requiring such an ability (the pre-existing allocators do not handle these irregularities either), it may well be a consideration for other compiler implementors.

7.2 Future Work

Colombet et al.[CBD14] conclude, that their formulations of optimal spilling show, that there is still room for improvement of spilling heuristics. More work in this area could explore spill placement for loop-invariant live ranges, the effect of spill code on critical path lengths and focus on integrating rematerialization.

While the specific handling of loops in the Braun and Hack[BH09] spill heuristic seems like a good way to take program structure into account, it did not fare well on the complex CFGs generated by GHC. A new approach may borrow from the register assignment algorithm implemented in this work and spill along *program traces*. Therefore trying to avoid spills on frequently executed program paths, without taking the complete loop structure into account.

List of Figures

List of Tables

List of Algorithms

Acronyms

ABI Application Binary Interface. 19

CFG Control-flow graph. 7, 8, 11, 13, 15, 16, 18, 23, 24, 28, 32, 34, 35, 52, 55, 56

CSSA Conventional Static Single Assignment. 10, 12, 34, 55

GC Graph Coloring. 2

GCRA Graph Coloring Register Allocation. 2, 8, 9

GHC Glasgow Haskell Compiler. 1, 27, 28, 36, 45, 51, 52, 55–57

IG interference graph. 5, 6, 12, 18, 19, 24, 25

ILP integer linear programming. 9, 10

IR intermediate representation. 3, 36

ISA Instruction Set Architecture. 3, 32

JIT Just-in-time. 1, 4, 8

LR live range. 4–8, 15, 19, 32

LS Linear Scan. 2, 9

LSRA Linear Scan Register Allocation. 28

NCG Native Code Generator backend. 1, 27–29, 34, 36–38, 41, 55, 56

PEO Perfect Elimination Order. 14, 24

RTS Runtime System. 53

SCC Strongly Connected Component. 28

SSA Static Single Assignment. 5, 9, 11, 12, 14, 15, 31, 32

SSE Streaming SIMD Extensions. 41

STG Spineless Tagless G-Machine. 27, 41

TSSA Transformed Static Single Assignment. 12

vreg virtual register. 5, 8, 29–38, 52

Bibliography

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 243–253. ACM, 2001.

[AR22] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In Lawrence Rauchwerger, Kirk Cameron, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos, editors, *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, USA, June 27-30, 2022*, ICS '22, pages 1–12. ACM, June 2022.

[BBH$^+$13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction*, volume 7791, pages 102–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, pages 275–284. ACM, 1989.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems, Volume 16, Issue 3*, 16(3):428–455, 1994.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill Code Minimization via Interference Region Spilling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 287–295. ACM, 1997.

[BDGR05] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under SSA. Technical report, ENS Lyon, 2005.

[BDGR06] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? Or revisiting register allocation: Why and how. In *Languages and Compilers for Parallel Computing, 19th International Workshop, LCPC 2006*, volume 4382 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2006.

[BDMS05] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, 2005.

[BDR$^+$09] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *2009 International Symposium on Code Generation and Optimization*, pages 114–125, Seattle, WA, USA, March 2009. IEEE.

[BGG$^+$89] David Bernstein, Dina Q. Goldin, Martin Charles Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–263. ACM, 1989.

[BH09] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for SSA-form programs. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, volume 5501 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2009.

[BMH10] Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-guided register assignment. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010*, volume 6011 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2010.

[Bou09] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases.* PhD Thesis, École normale supérieure de Lyon, France, 2009.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992.

[CAC$^+$81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[CB13] Charlie Curtsinger and Emery D. Berger. STABILIZER: statistically sound performance evaluation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013*, pages 219–228. ACM, 2013.

[CBD14] Quentin Colombet, Florian Brandner, and Alain Darte. Studying optimal spilling in the light of SSA. *ACM Trans. Archit. Code Optim.*, 11(4):47:1–47:26, 2014.

[CDE06] Keith D. Cooper, Anshuman Dasgupta, and Jason Eckhardt. Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 1–16, Berlin, Heidelberg, 2006. Springer.

[CE05] Keith D. Cooper and Jason Eckhardt. Improved Passive Splitting. In *Proceedings of The 2005 International Conference on Programming Languages and Compilers, PLC 2005*, pages 115–122. CSREA Press, 2005.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, pages 98–105. ACM, 1982.

[CHK04] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative Data-flow Analysis, Revisited. Technical Report TR04-100, Rice University, March 2004.

[CHP] Keith D Cooper, Timothy J Harvey, and David M Peixotto. Chow and Hennessy vs. Chaitin-Briggs Register Allocation: Using Adaptive Compilation to Fairly Compare Algorithms. In *SMART 2008: Second workshop on statistical and machine learning approaches to Architecture and compilation.*

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 192–203, New York, NY, USA, May 1991. Association for Computing Machinery.

[CR16] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *CoRR*, abs/1608.04295, 2016.

[CTS98] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, volume 1383, pages 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.

[dD14] Benoît Dupont de Dinechin. Using the SSA-Form in a Code Generator. In *Compiler Construction*, volume 8409, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Series Title: Lecture Notes in Computer Science.

[ESK09] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. Progressive spill code placement. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2009, Grenoble, France, October 11-16, 2009*, pages 77–86. ACM, 2009.

[FL98] Martin Farach and Vincenzo Liberatore. On local register allocation. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, page 564573, USA, 1998. Society for Industrial and Applied Mathematics.

[FWG05] Changqing Fu, Kent D. Wilken, and David W. Goodwin. A faster optimal register allocator. *J. Instr. Level Parallelism*, 7, 2005.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[Gav74] Fnic Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[Gol80] Martin Charles Golumbic. *Algorithmic Graph Theory And Perfect Graphs.* Academic Press, 1980.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Softw. Pract. Exp.*, 26(8):929–965, 1996.

[Hac07] Sebastian Hack. *Register allocation for programs in SSA form.* PhD thesis, Karlsruhe Institute of Technology, Germany, 2007.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150–155, 2006.

[HHJW07] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1–55, San Diego, California, USA, jun 2007. ACM.

[HP17] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach (6th Edition)*. Morgan Kaufmann, 2017.

[Jon92] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. *J. Funct. Program.*, 2(2):127–202, 1992.

[JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection. In *Principles and Practice of Declarative Programming, International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 1999.

[Kah65] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, page 40, jan 1965.

[KG09] David Ryan Koes and Seth Copen Goldstein. Register allocation deconstructed. In *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '09, pages 21–30, New York, NY, USA, April 2009. Association for Computing Machinery.

[KJ13] Tomas Kalibera and Richard E. Jones. Rigorous benchmarking in reasonable time. In *International Symposium on Memory Management, ISMM 2013*, pages 63–74. ACM, 2013.

[LJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 24–35. ACM, 1994.

[LS99] Allen Leung and Mercer St. Static Single Assignment Form for Machine Code. 1999.

[Mau21] Benjamin Maurer. SSA transformation for GHCs native code generator, 2021.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009*, pages 265–276. ACM, 2009.

[MP02] Hanspeter Mössenböck and Michael Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Proceedings of Compiler Construction, 11th International Conference*, volume 2304 of *Lecture Notes in Computer Science*, pages 229–246, April 2002.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 1957.

[NIKN06] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-Based Global Live-Range Splitting. 2006.

[Par92] Will Partain. The nofib benchmark suite of haskell programs. In *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992.

[Per08] Fernando Magno Quintao Pereira. A Survey on Register Allocation. Technical report, UCLA, October 2008.

[PM04] Jinpyo Park and Soo-Mook Moon. Optimistic Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 26(4):31, 2004.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Programming Languages and Systems, Third Asian Symposium, APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2005.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems, Volume 21m Issue 5*, 21(5):895–913, 1999.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended Linear Scan: An Alternate Foundation for Global Register Allocation. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction*, Lecture Notes in Computer Science, pages 141–155, Berlin, Heidelberg, 2007. Springer.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *Static Analysis*, volume 1694, pages 194–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices*, 33(5):142–151, May 1998.

[WF10] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10*, page 170, Toronto, Ontario, Canada, 2010. ACM Press.

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 111, New York, NY, USA, 1994. Association for Computing Machinery.

[WM05] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 132–141, New York, NY, USA, June 2005. Association for Computing Machinery.