REGISTER ALLOCATION BASED ON CHORDALITY OF SSA-FORM PROGRAM INTERFERENCE GRAPHS

Cyrill Gössi, Edwin Dornbirer

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

This report describes the structure, implementation and performance evaluation of a non-iterative register allocator making use of program properties induced by the Static-Single-Assignment (SSA) form. Most salient, SSA renders a programs interference graph chordal. Since chordal graphs are a subset of perfect graphs and perfect graphs can be recognized and optimally colored in polynomial time, allocating registers for programs in SSA form can be optimally done in polynomial time. Together with the possibility to destroy SSA in a color-preserving manner, those properties allow to diverge from the classical iterative register allocation process, as exploited in our implementation and described in this report.

1. INTRODUCTION

Register allocation is the process of creating a function which maps an unbounded number of source code variables to a bounded number of machine registers. This function has to be such that no two variables, whose values are simultaneously needed at one or multiple points in the execution of a program, are mapped to the same register. To construct a map satisfying this constraint, one generally has to define certain variables not to be mapped to registers but memory locations. Since accessing values located in memory has a bigger latency than accessing values in registers, a register allocator has to compute the set of register-located variables such that the overall sum of latency-costs introduced by accessing values of memory-spilled variables will be kept as low as possible.

Quiet prominently, this optimization problem is reducible to graph coloring[1]. In this reduction, a graph is constructed whose nodes represent source code variables and whose edges represent overlapping liveness-ranges. Applying a coloring algorithm to the graph and identifying colors with machine registers will conclude the register allocation process.

However, graph coloring has long been known to be an NP-complete problem.[2]. In order to deal with this complexity, polynomial-time heuristics were developed, some of the more successful ones being iterative spilling and coalescing algorithms[3][4][5].

In 2006 the interference graphs of SSA-Form programs were proven to be chordal[6]. As a subset of perfect graphs, chordal graphs can be optimally colored in polynomial time.

In this report, we describe the structure and implementation of both a non-iterative register allocator exploiting the chordality of SSA-Form program interference graphs as well as a classical allocator which was built in order to asses the quality of the results produced by the SSA based allocator.

The report is organized as follows: Section 2 introduces the relevant graph and compiler theoretical notions. Section 3 is where the structure and algorithmic details of both the SSA based as well as the classical allocator are described. Section 4 presents experimental results and Section 5 will conclude the report.

2. BASIC GRAPH AND COMPILER THEORY

This section introduces graph and compiler theoretical notions and concepts essential for the understanding of the subsequent chapters in this report.

Graph Theory. A graph $G = (V, E)$ is an ordered pair of sets where $E \subseteq V \times V$. A $v \in V$ is called a vertex, whereas an $e \in E$ is called an edge. If $(p, q) \in E$ then p and q are said to be adjacent. A cycle in a graph is a sequence of vertices $(v_0, v_1, .., v_n)$ where $v_i \in V$ for $0 \leq i \leq n$, $v_0 = v_n$ and $(v_i, v_{i+1}) \in E$ for $0 < i < n-1$. A graph is chordal iff every cycle of four or more vertices in the graph has an edge joining two nodes that are not adjacent in the cycle. A vertex coloring of a graph $G = (V, E)$ is a map $c : V \to S$ such that $c(p) \neq c(q)$ if $(p, q) \in E$. An $s \in S$ is called a color. A graph G is called $k - colorable$ if there exists a vertex coloring for it with $|S| = k$. The smallest k for which a vertex coloring exists for G is called its chromatic number and is denoted as $\chi(G)$. A *clique* in graph G is a subset $C \subseteq V$ such that for every $p, q \in C$ it holds that $(p, q) \in E$. The size of the biggest clique in graph G is called its clique number and is denoted as $\omega(G)$.

Theorem 1. *If graph G is chordal, then $\omega(G) = \chi(G)$.*

Proof. Chordal graphs are a subset of perfect graphs[7] for which the conclusion holds by definition. □

The neighborhood of a vertex v in a graph G is the induced subgraph of G consisting of all vertices adjacent to v and all edges connecting two such vertices. A vertex v of graph G is called simplicial if its neighborhood in G is a clique. A simplicial elimination ordering SEO of graph G is a bijection $V(G) \rightarrow 1, .., |V|$ s.t. every vertex v_i is a simplicial vertex in the subgraph induced by $v_1, ..., v_i$.

Theorem 2. *Chordal graphs $G = (V, E)$ can be optimally colored in $O(|V| + |E|)$.*

Proof. For chordal graphs, the Maximum Cardinality Algorithm MCS, Alg. 1, computes a SEO in $O(|V| + |E|)$[9]. The greedy coloring algorithm, Alg. 2, with a SEO as input gives an optimal coloring in $O(|V| + |E|)$[10]. □

input : $G = (V, E)$
output: Seq. of nodes σ. If G chordal, then σ is SEO
foreach $v \in V$ **do**
  $\alpha(v) \leftarrow 0$;
end
for $i \leftarrow 1$ **to** $|V|$ **do**
  let $v \in V$ be s.t. $\forall u \in V, \alpha(v) \geq \alpha(u)$;
  $\sigma(i) \leftarrow v$;
  foreach $u \in V$ **do**
    $\alpha(u) \leftarrow \alpha(u) + 1$;
  end
  $V \leftarrow V - \{v\}$;
end

Algorithm 1: Maximum Cardinality Search MCS

input : $G = (V, E)$, Sequence of nodes σ
output: A coloring c. If σ is SEO then c is optimal, otherwise
$0 \leq c(v) \leq maxDeg(V), \forall v \in V$
foreach $v \in \sigma$ **do**
  $c(v) \leftarrow \perp$;
end
foreach $i \leftarrow 1$ **to** $|\sigma|$ **do**
  let r be the lowest color not used in $N(\sigma(i))$;
  $c(\sigma(i)) \leftarrow r$;
end

Algorithm 2: Greedy Coloring

Compiler Theory. A variable x is *live* at some point p in a program if there is a possible path of execution from p to some point p' where x is used and the path from p to p' does not contain a definition of x. Two variables x and y *interfere* iff there is a point in the program where both are *live*. A graph $G = (V, E)$ with V as the set of all variables in a program P and $E = \{(x, y)|x, y \in V \wedge x, y\, interfere\}$ is called the interference graph of P. A strict program is one in which every path from the initial basic block to a usage of variable v passes through a definition of v. A program in Static Single Assignment (SSA) form, discussed in depth in [8], assigns to each variable exactly once. If a variable in the original source code is assigned multiple times then the compiler has to transform the source such that this variable will get a new version for every occurring assignment. Usages of the original variable have to be transformed such that they will use the appropriate version of the variable reaching that point. The selection of the correct version is done via so called Phi-Nodes. A Phi-Node is a purely virtual construct and will have to be turned into simultaneously executed assignments in the predecessor basic blocks.

Theorem 3. *Interference graphs of strict SSA-Form programs are chordal.*

Proof. See [6] □

Theorem 4. *A valid register allocation for a program P in SSA form can be turned into a valid register allocation of P's non-SSA form.*

Proof. See [6] □

Theorem 5. *Register Allocation can be reduced to graph coloring.*

Proof. Informal: Interfering program variables cannot be placed into the same processor register. On input of program P, the compiler constructs the interference graph. Computing a valid coloring on this graph mean that all interfering variables of P will be held in different registers. □

Theorem 6. *Register Allocation of SSA-Form programs can be done in polynomial time.*

Proof. Theorem 2 and 3 show that interference graphs of strict SSA-Form programs can be colored in $O(|V| + |E|)$. With theorem 5 this is equivalent to allocating registers for an SSA program in $O(|V|+|E|)$. Using theorem 4, the SSA form can be color preserving destroyed. □

3. STRUCTURE AND IMPLEMENTATION OF OUR REGISTER ALLOCATOR

In this chapter we describe how we implemented a register allocator exploiting the chordality of SSA-form program interference graphs. Furthermore, we also describe the structure of a classical register allocator we implemented and whose building blocks are the same like those used in the

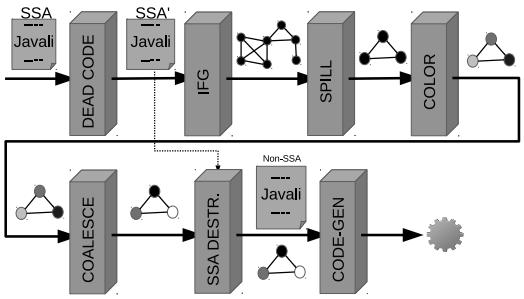

Fig. 1. Structure of the SSA-Form program interference graph chordality exploiting register allocator.

chordality exploiting allocator, just in a different order of execution.

SSA Interference Graph Chordality Exploiting Register Allocator.

Our allocator consists of several independent phases, namely dead code elimination, interference graph creation, spilling, coloring, coalescing, color preserving SSA destruction and code generation. An illustration of this process can be seen in Fig.1. As input to our allocator we expect a Javali file in SSA form. As output we generate assembly code, respecting the allocated registers.

In what follows, all the phases of the allocator in Fig. 1 will be described and their purposes and internal structures analyzed.

Dead Code Eliminator. Dead code elimination takes a Javali in SSA form and removes all Phi-Nodes and instructions that involve variables which are only written but never read. The removal is only done if no side-effects can occur from it. The purpose of this phase is to remove variables with point-like liveness ranges and with it easing the logic involved in the final code generation phase. In itself, dead code elimination is a fix-point algorithm, computing liveness ranges of variables and iterating over the input program until no more code elements can be removed.

Interference Graph Creation. In this phase, the interference graph (IFG) of the SSA form Javali is created. First, for each instruction in the program, a set of live variables is computed. Those live sets are then analyzed and adjacency lists representing the interference of variables are computed. Since the Javali is still in SSA form, SSA destruction has to be mimicked and interferences created from resolving Phi-nodes into assignments have to be inserted into the IFG as well. Additionally, the IFG also stores the largest live set found. This set equals the largest clique in the IFG and will simplify the subsequent spilling phase.

Spilling. The spilling phase takes an IFG and removes variables from it until a subsequent coloring stage is guaranteed to compute a valid coloring. The nodes removed represent the variables that will be spilled to memory during the code generation phase. Since spilling a variable to memory will increase the time needed to access its value, selecting the nodes to be spilled has to be done with care. In our implementation, the next variable v to be spilled is computed according to the following equation:

$$v = \arg\min_{v_1 \in V(G)} \left(\frac{1}{|N(v_1)+1|} \sum_{l \in usage(v_1)} 10^{Depth(l)}\right)$$

Here, $N(v)$ denotes the set of neighbors of v in G, $usage(v)$ the set of basic blocks where variable v is read at least once and $Depth(l)$ the nesting depth of basic block l. The intuition behind this equation is, that loops are modeled to run 10 iterations. Therefore, for every variable v we estimate the number of times v is read. Then, if this number is the same for two variables, we choose the one that removes the most interferences from G. Theorem 1 and 2 give rise to a simple spilling algorithm, outlined in Alg. 3, assuring a subsequent coloring phase to succeed.

input : IFG G, Integer k
output: IFG G' s.t. $\omega(G') \leq k$, set sp of spilled variables
$sp := \varnothing$;
while $\omega(G) > k$ **do**
  $c := largestClique(G)$;
  $v := chooseNext(c)$;
  $sp := sp \cup \{v\}$;
  $removeNode(G, v)$;
end

Algorithm 3: Spilling on Interference Graphs

Coloring. The coloring stage takes a chordal IFG as input and computes an optimal coloring for it. This stage is an application of theorem 2 and therefore is split up in two phases; In the first phase, a SEO is computed for the input IFG via MCS (Alg. 1). In the second phase, this SEO is given to a greedy coloring algorithm (Alg. 2). Since the previous spilling stage modified the initial chordal IFG s.t. $\omega(IFG) \leq k$, Alg. 1 is proven to produce a SEO upon input of a chordal graph and Alg. 2 is proven to produce an optimal coloring, the coloring stage computes in $O(|V| + |E|)$ a valid coloring for the input IFG which uses at most k colors.

Coalescing. Coalescing tries to modify an existing coloring such that variables involved in copy or move instructions end up having the same color. On success, such an instruction becomes obsolete and execution time can be saved. A potentially large number of assignment instructions results from a later stage resolving SSA phi-nodes. Our implementation therefore has to mimic an SSA destruction phase by which all final assignment instructions become visible. The heuristic we used for this stage can be found as pseudo code in [11]. This heuristic attempts a greedy recursive recoloring and extends the input IFG with a second type of edges, called affinity edges. Affinity edges are

inserted between every pair of variables that are involved in a copy/move instruction and whose colors are different. Those affinity edges will get a value assigned which reflects the cost incurred by having the value of the variables stored in different registers. In our implementation, the cost c for an affinity edge resulting from a copy/move instruction located in basic block b is computed as follows: $c = 10^{Depth(b)}$. This approximates the number of times the instruction has to be executed if the two variables involved do not have the same register assigned. After computing all the affinity edges, chunks are built in which pairs of affinity related but non-interfering variables are collected. Then the chunk with the highest total cost of affinity edges residing in it is taken and its variables are tried to be recolored to the same color. Recoloring is attempted with all available colors and the color removing most affinity costs is stored as the final color. After all colors have been tried, the final color is applied to variables where applicable. The recolored variables are marked as final and the coalescing heuristic moves on with the remaining variables.

In our implementation we coalesce both the register located variables as well as the memory-spilled variables. In order to coalesce the spilled variables, we first assign each variable a symbolic color and then after coalescing transform each final color into a final stack offset. The modified coloring, as output from the coalescing heuristic, will then be the final assignment variables to registers (or memory locations).

SSA Destruction. In this stage, the SSA form of a program is destroyed in a color preserving manner. Our implementation of this stage is an application of theorem 4. The semantics of SSA dictates that Phi-nodes in a basic block are replaced with assignments in predecessor blocks and that these assignments have the same effect when executed in sequence as if they were executed simultaneously. This replacement can be done while preserving the coloring and preventing register values from being corrupted. To see this, consider that the dead code eliminator removed Phi-nodes whose left-hand sides were only written but never read. Additionally, the spilling phase made sure that at every point in the program at most k variables are live. This means that the number of Phi-nodes present in a basic block is at most k since all the left-hand sides of the Phi-nodes immediately start to live. On the other hand, also at most k different variables can reach the point where the assignment-instructions of the Phi-node will be placed. Therefore, replacing the Phi-nodes in a basic block boils down to a l to m mapping of registers where $l \leq m \leq k$. All assignments with target registers not needed as an argument in another Phi-node can directly be inserted into the predecessor. After those assignments have been removed, either all Phi-nodes have been resolved, or we are left with a mapping of j to j registers. The corresponding instructions will not be turned into assignments but will be solved by swapping the registers involved and inserting this instruction into the predecessor. At the end of this process, all Phi-nodes in a basic block are resolved.

Code Generation.

This stage deals with generating an assembly file from the parsed Javali file in abstract syntax tree AST form. The control flow is given in a control flow graph CFG for each method, consisting of basic blocks and the links between them. A basic block is composed of several statements and an optional condition which always have to be executed as is, meaning that the only jump allowed into a basic block is a jump to its first statement. The condition determines what block to execute after the basic block has finished executing its last statement. From this outset thus, each basic block can be handled separately, with jumps to other blocks and methods handled with labels. Since the control flow part of our code generator does not veer significantly from the framework we were given as a starting point, we omit its discussion here.

The bulk of the time we spent on implementing the code generator was spent on the generation of code for a single statement or a condition. These two cases are henceforth referred to as expressions, and each expression has a corresponding AST. There are a large number of different cases to consider when one has variables in an expression AST that could be in memory or already in registers, with the additional need for free registers should the need arise, coupled with the added complexity of constant leaf nodes. To simplify the implementation, we split the code generation into two distinct phases for each expression AST. The first stage traverses the entire AST of a given statement in order to find out key characteristics of the given AST. In this phase, no actual code is emitted. The characteristics we were interested in are the following:

- The number of free registers used by the different operations (this depends on the operation itself as well as the results received from the children)
- The number of uses for each register
- The place of last usage of a dying register (explained below)

The first item should be straight forward, since this directly determines the number of registers we need to free (push on the stack and pop after the AST has been traversed) before we can emit the code corresponding to the AST. The second item is a simple heuristic to determine which of the registers should be freed. If a register is never used in a expression for example (but is used later on in the code and therefore still holds valid data), it will preferentially be put on the stack during the execution of said expression. Finally, the last item warrants some further explanation. By dying

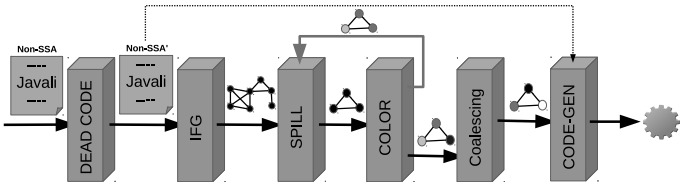

Fig. 2. Structure of the Classical Register Allocator.

register we denote registers that are live up to and including a given expression, but are not live in the next expression. Put simply, the given expression is the last use of the variable that is assigned to this register. This means that in an expression that would usually call for an additional register, such as (x + y), we can circumvent the need for a free register to store the result if this expression is the last usage of either x or y. To determine the last use of a register more easily, we traverse the AST in reverse order (with respect to the second phase), where the last use becomes the first occurrence of the register.

With the characteristics of an expression determined, we enter the second phase of our code generator. We first push the appropriate number of registers on the stack. Then we traverse the AST in a similar fashion as in the first phase, but this time we emit the actual code as we traverse the AST. Finally, the registers that were freed up before the traversal are popped from the stack to restore the initial state regarding live registers.

Classical Register Allocator. In order to compare the performance of the binaries produced by the SSA exploiting register allocator, we also implemented a basic classical allocator. This classic allocator is made up of the same building blocks like the SSA based allocator and an outline of its structure can be seen in Fig. 2. As opposed to the SSA based allocator, the classic version is missing the SSA destruction stage since for a classic register allocation SSA already has to be destroyed beforehand. Another structural difference lies in the loop from coloring back to spilling. The spilling stage still spills according to Alg. 3, therefore modifying the initial interference graph such that $\omega(IFG) \leq k$. However, since the interference graphs of non-SSA programs are not necessarily chordal, the coloring stage with MCS and greedy coloring (Alg. 1 and Alg. 2) is not guaranteed to produce an optimal coloring anymore. After greedy coloring, we therefore check whether the number of colors used is less than k. If so, we go on to the code generation stage, if not we go back to the spiller, telling it to spill one more variable and then make another attempt to color it with less than k colors. An outline of this spill-and-color iteration can be seen in Alg. 4. After the loop in Alg. 4 terminates, we proceed to the coalescing stage as in the SSA based allocator.

input : IFG $G = (V, E)$, Integer k
output: A coloring c s.t. $0 \leq c(v) < k, \forall v \in V$
$spill(IFG, k)$;
$k' = color(IFG)$;
while $k' > k$ **do**
  $spillOneMore(IFG)$;
  $k' = color(IFG)$;
end

Algorithm 4: Iterative Spill-Color Algorithm used in the Classic Register Allocator.

ID	# Files	Avg LOC	Avg |V|	Max |V|
B0	200	77	9	19
B1	200	135	24	73
B2	200	356	115	259
B3	200	104	2	6

Table 1. Characteristics of Benchmarks used for Evaluation

4. EXPERIMENTAL EVALUATION

In order to evaluate our implementation, we created 4 different benchmarks, B0 to B3, out of which B0 to B2 are sets of randomly generated Javali files and B4 is a collection of real world programs. Some characteristics of the benchmarks can be found in table 1.

LOC denotes lines-of-code and V is the set of variables of the initial interference graphs reaching the register allocators. The randomly generated benchmarks serve the purpose of analyzing the performance of our algorithms when facing arbitrarily complex interference graphs. We analyzed the compilation times when using both types of register allocators as well as if the existing framework is used. To asses the quality gain by using the SSA based allocator compared to a classical allocator, we kept track of the percentage of variables kept in registers for a given initial interference graph. The quality of the coalescing stage was measured by means of computing the total eliminated affinity edges as well as the total eliminated affinity costs. Finally, the run times of the binaries produced by both versions of allocators as well as the framework were measured and compared.

Setup. All measurements were conducted on an Intel Core i7 2820QM 2.3GHz PC with 4GB RAM and running Debian with kernel version 3.11.0. The general workload on the system was reduced to the minimum and the benchmarks were run with highest CPU and IO priorities.

Compile Time Behavior.

The compile time behavior of the framework as opposed to using the SSA register allocator is roughly equal, as seen in Fig. 3. Classic allocation however is seen to have mostly

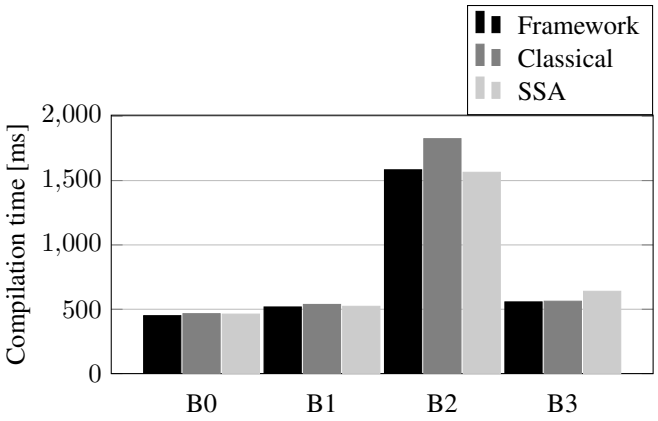

Fig. 3. Compilation Times

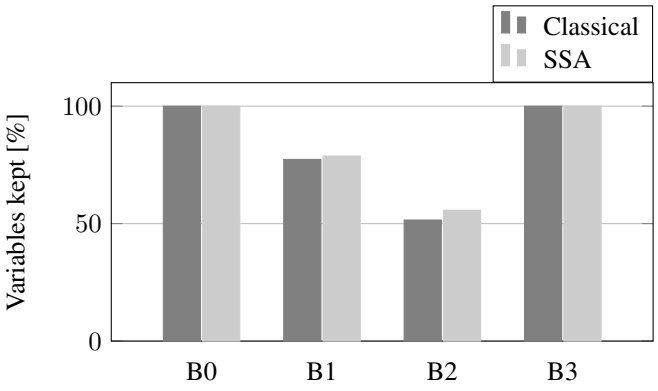

Fig. 4. Spilling: Percentage of Variables kept in Registers

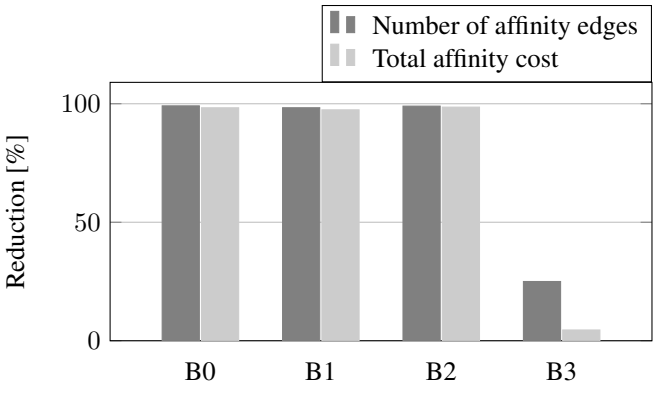

Fig. 5. Coalescing: Percentage of Affinity Edges and Costs Removed

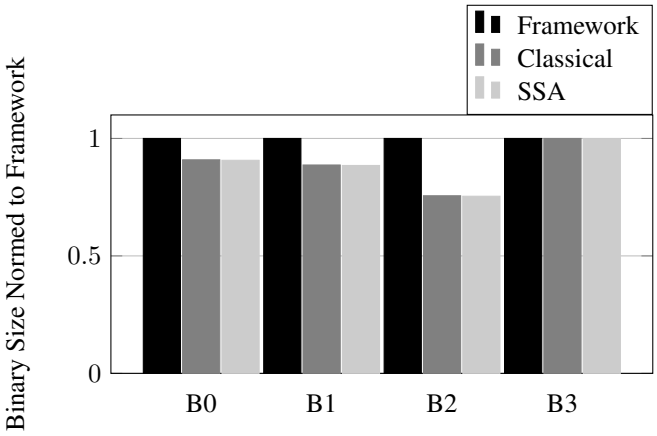

Fig. 6. Sizes of final Binaries Generated

bigger compilation times than both other types of implementation. Especially benchmark 3 showed that the classical version frequently colored graphs sub-optimally and therefore had to execute the spill-and-color iteration repeatedly. Furthermore, the coalescing heuristics implemented is optimized for an SSA context and it was observed that the heuristic took noticeably longer in a non-SSA setting.

Spilling Behavior.

Since spilling under SSA involves an optimal coloring, it was expected to observe a different percentage of variables kept in registers as opposed to a classical allocator which possibly had to spill more variables due to a sub-optimal coloring. Indeed, for big enough Javali files and interference graphs, the difference was quite remarkable, with the SSA allocator in benchmark 3 keeping roughly 10 % more variables in registers as the classical version. See figure 4 for a full comparison.

Coalescing Behavior.

The performance of the coalescing heuristic was measured without differentiating the underlying type of register allocator. As described above, coalescing introduces weighted affinity edges to the interference graph. We kept track of the number of initial affinity edges introduced as well as their total sum of weights and compared them with the final output of the coalescing heuristic. For the benchmarks 0 to 3, we observed that the number of affinity edges removed through coalescing are an impressive 99% of the initial number and also the total costs removed by coalescing was always around 98%. (See figure 5 for details.) Those numbers differ for the benchmark with real world programs. The reason for it is that their interference graphs generally were to small for the heuristic to have much potential to optimize.

Code Generation.

With colored and coalesced variables as well as a non-trivial register-usage analysis in the code generator, we expected the final binaries to measurably differ in size. And indeed, a code size reduction of around 10% could be measured on average. See figure 6 for more details.

Runtime Behavior.

Measuring the run times was done by considering 100

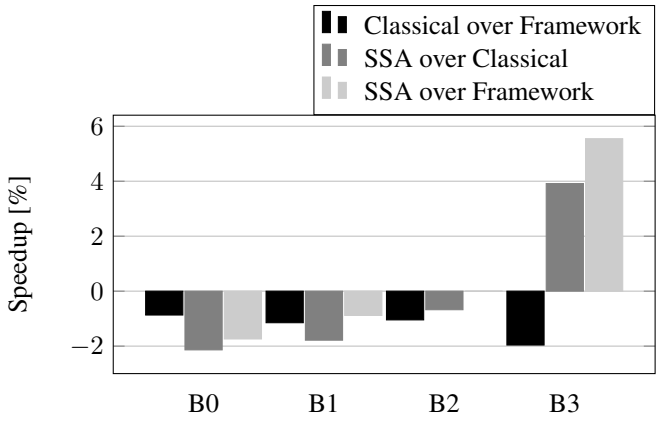

Fig. 7. All Speedups for all Benchmarks and Implementations

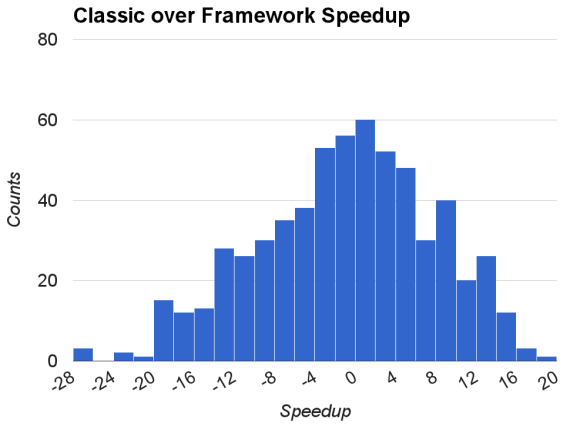

Fig. 8. Speedups between Framework and Classic Implementation

executions of a file as one super-execution and then executing the super-execution 200 times remembering the best super-execution execution time in micro-seconds. Differences between all three types of implementation were slim. A figure containing the speedup in percentage compared between all implementations and for each benchmark separated can be seen in figure 7. Histograms of the speedup between all implementations over the entire benchmark suite can be found in figures 8, 9 and 10.

5. DISCUSSION

Implementing an SSA based allocator targets at simplifying the structure of the register allocator as well as potentially improving the run-times of programs generated. While the simplification was directly achieved in our implementation, we weren't able to measure any notable difference in the run-time behaviour of all of our randomly generated bench-

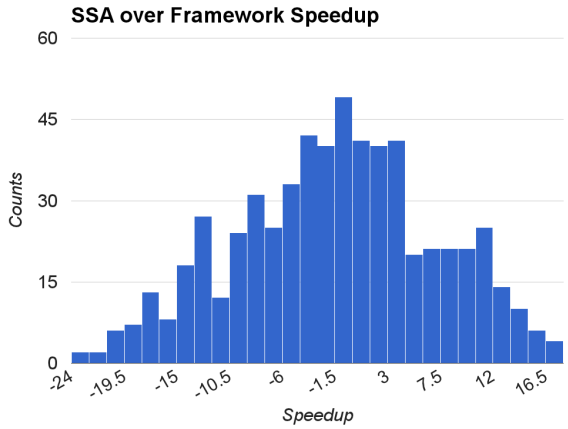

Fig. 9. Speedups between Framework and Classic Implementation

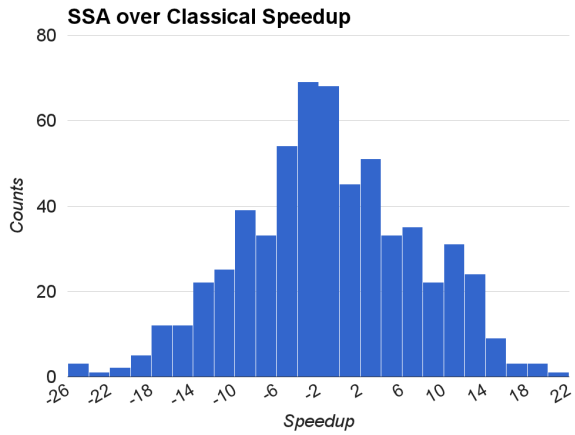

Fig. 10. Speedups between Framework and Classic Implementation

marks. However, inspecting the generated code shows that there is a significant number of cycles saved by having coalesced both register and memory-spilled variables. The lack of being reflected in the run-time measurements might be due to the structure of the benchmark files which contain a proportionally big number of IO code, possibly dwarfing all computational gains. With a more evolved set of benchmark files, a more sophisticated benchmark environment and a more fine grained timing resolution we are sure that the efforts spend on implementing an SSA register allocator would show up in the run-time behaviour of the final executables.

6. FINAL REMARKS

With regards to our implementation, we would like to point out the following aspects:

- Floats are always spilled to memory. Moving a float to an SSE register requires it to reside in memory anyhow and time didn't permit us to spend much more thoughts on this matter.

- Testing had a focus on programs involving computations on primitive types. However, all of Javali is expected to be compiled correctly.

7. REFERENCES

[1] Chaitin, Gregory J., et al. "Register allocation via coloring." Computer languages 6.1 (1981): 47-57.

[2] Karp, Richard M. Reducibility among combinatorial problems. Springer US, 1972.

[3] Appel, Andrew W., and Lal George. "Optimal spilling for CISC machines with few registers." ACM SIGPLAN Notices. Vol. 36. No. 5. ACM, 2001.

[4] Briggs, Preston, Keith D. Cooper, and Linda Torczon. "Improvements to graph coloring register allocation." ACM Transactions on Programming Languages and Systems (TOPLAS) 16.3 (1994): 428-455.

[5] George, Lal, and Andrew W. Appel. "Iterated register coalescing." ACM Transactions on Programming Languages and Systems (TOPLAS) 18.3 (1996): 300-324.

[6] Hack, Sebastian, and Gerhard Goos. "Optimal register allocation for SSA-form programs in polynomial time." Information Processing Letters 98.4 (2006): 150-155.

[7] Kloks, T. advanced graph algorithms. Kloks, 2012.

[8] Cytron, Ron, et al. "Efficiently computing static single assignment form and the control dependence graph." ACM Transactions on Programming Languages and Systems (TOPLAS) 13.4 (1991): 451-490.

[9] Tarjan, Robert E., and Mihalis Yannakakis. "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs." SIAM Journal on computing 13.3 (1984): 566-579.

[10] Gavril, Fanica. "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph." SIAM Journal on Computing 1.2 (1972): 180-187.

[11] Hack, Sebastian. "Register Allocation for Programs in SSA Form" Universitätsverlag Karlsruhe, (2007).