Research and Analysis of Compiler Optimization Techniques
Independent Study Report for Semester 2148
Kevin Bradley (kmb3398) - Advised by Prof. Fluet

Introduction
Earlier in my college career I had taken compiler construction, a class which I found to be extremely interesting. During the course the students make a compiler for a relatively small language: LangF. The course breaks this task down into five main projects, namely the five stages of compilation: scanning, parsing, type-checking, optimization, and code generation. Following this course, I devoted my MS project to implementing additional SML language features in the MLton compiler. Adding in these additional language features involved working with the first three stages of compilation. This course, Research and Analysis of Compiler Optimization Techniques, is an extension of the compiler construction course, focusing on the optimization stage, or fourth stage of compilation. The goal of this course is to learn and better understand the various optimizations that compilers use today. Additionally, this includes the subcategories of both proving compilers and specific optimizations correct and the various techniques applied to do this. Another, smaller subcategory of this course is register allocation. This category includes the methods for doing this along with how this can be done optimally and with what cost this optimality comes.

This course consisted of reading and comparing various papers of the topics described above. Additionally, each week there would be a two hour meeting to discuss what I had learned and any questions or related topics with Prof. Fluet. Many times we would discuss how the techniques learned in the reading were or were not applicable in the context of MLton. Throughout the course, approximately 21 hours were spent meeting with Prof. Fluet, with an additional 56 hours spent doing the reading and research. This report is presented in the chronological order of the topics covered and papers read throughout the course (with the exception of the Mitigating Compiler Optimization Problem using Machine Learning which was read earlier, but falls into the miscellaneous category). The references to both the textbook and papers are included at the end.

Background (Weeks 1 & 2)
The first couple of weeks of this course were spent reading chapters eight through ten of the textbook Engineering a compiler, second edition by Keith D. Cooper and Linda Torczon. [1]

Chapter 8
Chapter eight covered an introduction to optimizations in which they describe how the goals of optimizations vary, but generally try to induce: faster execution, less memory utilization, less energy cost, less memory traffic, etc. The different scopes of optimizations are also introduced. Local method optimizations work on single basic blocks (maximal length sequences of branch-free code); regional methods optimizations work on a scope larger than a single basic block but smaller than a full procedure; global method optimizations work on an entire procedure; and interprocedural optimizations work on a scope larger than a single procedure [1]. Furthermore, chapter eight discusses some basic optimizations such as local value numbering, loop unrolling, global code placement, inline substitution etc. These optimizations are given in a fairly basic form to give both an introduction to the optimization along with a better understanding of which of the categories they fall into.

Chapter 9
Chapter nine of the book discusses Static Single Assignment (SSA) form, an intermediate representation commonly used when performing optimizations. In SSA form each definition has a unique name and each use refers to a single definition. Also discussed in this chapter is the idea of data-flow analysis and related terminology including dominance, availability, and anticipable expressions. Minimal, pruned and semi-pruned SSA forms are also discussed and how they differ in where and how their phi functions are placed.

Chapter 10
Chapter ten talks about scalar optimizations, that is optimizations which focus on a single thread of control [1]. More specifically, this chapter covers useless control flow which can be removed by: folding redundant branches, removing an empty block, combining blocks, and hoisting branches. Along with removing useless control flow, removing redundant and partially redundant expressions, coalescing, parameter promotion, and strength reduction are other scalar optimizations which are described. An important thing to note is that when performing a transformation, this may enable another transformation to be performed as well or better. This leads to the question of how to order the different optimizations that will be performed on the program known as the optimization phase ordering problem. This problem was both discussed in how it is handled in MLton along with a paper in which machine learning is used to try to determine the best ordering.

Overall these first weeks gave me a solid background of how optimizations are set up, what their goals are, how their scopes vary, forms generally used when working on optimizations, and a large amount of terminology likely to appear in the papers to be read in the following weeks.

Additional SSA Background (Week 3)
Detecting Equality of Variables in Programs [2]
For a bit of extra background on SSA form I read the paper Detecting equality of variables in programs. This paper obviously outlines how to detect equivalence between variables when in SSA form, which they describe as, in general, being undecidable. The notion of equivalence of variables is that variables are “equivalent at a point p if they contain the same value at p whenever the program reaches p during any possible execution” [2]. This process is conservative in the fact that the equivalent variables detected will always actually be equivalent but not all equivalent variables will be detected. The notable contribution of this paper is to determine equivalence of variables in the presence of control flow structures such as ifs and loops. This is done by their notion of congruence and activeness, and utilizing their conclusion that two active congruent variables have the same value, and are equivalent. Variable equivalence is useful in register allocation, common sub-expression elimination, movement of invariant code, branch elimination, branch fusion, etc. This paper was useful in showing me how SSA form can be utilized and understanding how equivalence of variables can be determined, which is important to many other optimizations.

In the context of MLton, Prof. Fluet argues that SSA is better represented as functional programming, and instead of phi-functions gotos with arguments are used. This leads to a more clear and natural understanding of the IR, but this also means that there is no variable versioning which makes partial redundancy elimination (PRE) harder.

SSA Form Optimization Techniques (Weeks 3 & 4)

Following the background weeks three and four were used to learn about various common optimizations which use SSA form. These optimizations included global value numbers and redundant computations, operator strength reduction, array bounds checking on demand, and lazy code motion. For each of these optimizations I read and discussed a corresponding paper.

Global Value Numbers and Redundant Computations [3]
This paper describes an algorithm for determining and replacing redundant computations in O(C * N * E) time where C is the number of computations, N is the number of nodes in the control flow graph (CFG), and E is the number of edges in the CFG. Additionally, this algorithm will move computations out of loops and place them in the landing pad of the loop (directly before the loop). This has the effect of shortening the live ranges of the operands for the computation but lengthening the live range of the result of the computation. Another key lesson from this paper is that many optimizations create second order effects; they allow for more optimizations to be performed. In this case, by identifying computations that compute the same value, this in turn can allow others to be found as well. This paper, though fairly old, also describes and utilizes a couple of techniques that are commonly used in other optimizations, such as placing landing pads at the beginning of loops and the splitting of what are known as critical edges in control flow graphs.

Operator Strength Reduction [4]
This paper outlines the methodology behind operator strength reduction, which is the reformulation of costly computations to equivalent computations in terms of less costly ones. In general this gives way to two improvements: reduction in the number of operations and/or the use of less costly operations. A very simple example of weak strength reduction would be to reformulate 2*x to be x + x or more commonly x << 1. This paper's algorithm is described as an improvement of the Allen-Cocke-Kennedy (ACK) algorithm which works on the CFG, whereas this algorithm works mainly on SSA form. Much of operator strength reduction is based around finding region constants, which are variables that don't change within a loop and induction variables, which are variables that increase or decrease by a constant amount each loop iteration. In general, candidates for OSR are of the form $x \leftarrow i*j$, $x \leftarrow j*i$, $x \leftarrow i \pm j$, or $x \leftarrow j+i$ where i is an induction variable and j is a region constant [4]. Again, it is important to note that similar to many other optimizations, this is intended to be followed by other optimizations. In this case the authors specifically mention that their algorithm should be followed by a global value numbering and dead code elimination passes and that this algorithm should be preceded by constant propagation and code motion passes. These passes can increase the size of the induction variable and region constant sets and thereby allow operator strength reduction to potentially be performed in more places.

ABCD: Eliminating Array Bounds Checks on Demand [5]
The motivation behind this paper is that using powerful boundscheck optimizations at runtime is too heavyweight for dynamic compilation. In the dynamic compilation setting described by this paper, the compile time cost is constrained and the hot statements are known. The purpose of this paper is to eliminate redundant bounds checks. A bound check of A[x] is redundant if and only if $0 \leq x < A.length$ whenever the check is executed, equivalently $x - A.length < 0$ should always be true [5]. This paper expands upon using SSA form by adding in pi functions, similar to phi functions to the representation. As phi functions are used to identify the possible values of a variable incoming into a block, pi functions are placed at the end of blocks to identify what is known about certain variables. This additional information is then used to be able to determine

which bounds checks are redundant and can be removed. This paper also takes advantage of global value numbering to determine equivalence such that if the check of A[x] is found and it is known that x < B.length if B is actually indeed equal to A, this check can then too be eliminated. In their context of the Jalapeño optimizing compiler they state on average their technique was able to remove 45% of dynamic bounds checks.

Lazy Code Motion [6]
The goal of this paper is to perform code motion, but in a lazy fashion, that is not to just move code upwards to the earliest possible position it can be placed. Traditionally, computations are placed as early as possible by code motion passes, even if unneeded, this leads to unnecessary register pressure. The algorithm presented by this paper is as efficient with respect to computationally optimal results, as others, but does this in a manner to avoid the unnecessary register pressure. In this context computationally optimal results mean the number of computations cannot be reduced any further. With this instead of placing the computation as early as possible, it is instead placed as late as possible while still maintaining this computationally optimality.

In this part of the course I learned about how many common optimizations use SSA form and how the optimizations work. Some of the papers were a bit older and give some history towards the optimization such as the GVN and LCM papers while OSR and ABCD papers provide improvements to some previous techniques. In addition to utilizing SSA form, common techniques such as splitting critical edges are used in multiple different optimizations. Many of these papers enforce the concept that the phase ordering problem of optimizations is very critical and that optimizations such as dead code elimination are likely to be run multiple times during the full optimization pipeline. In particular I found the idea of expanding upon SSA form with the pi functions to be a novel idea to be able to more easily determine which array bounds checks are redundant. SSA form is widely used both historically and currently in the optimization process which was illustrated throughout this portion of the course.

Verifying SSA Optimizations (Week 5)
A Verifiable SSA Program Representation for Aggressive Compiler Optimization [7]
This paper also expands upon SSA form, this time with the goal of being able to verify that the safety of a program after optimizations. The main contributions are their representation which combines SSA form with safety dependencies, a type-system to verify the memory safety, and a demonstration of the utility of this methodology in the StarJIT compiler. A potentially unsafe instruction is any instruction that might fault or cause illegal memory access, and a contextually safe instruction is one such that its safety dependencies guarantee its safety [7]. The authors argue that the Java bytecode is not a good IR, regardless of whether it is erasure style or refinement style. In erasure style, safety dependencies are hidden, making it easy to optimize, but hard to verify the safety of the optimizations. In refinement style, safety information is indirectly encoded making it easy to verify optimizations, but hard to actually perform them. The way the safety information is added into SSA form is via the use of proof variables. For instance, branches assign proofs to proof variables that reflect the conditions being branched upon. One nicety of this technique is that many optimizations required little adjustment to work with this new SSA form, in most cases the optimization can simply transform both the computation and proof at the same time. For some optimizations such as OSR, new proofs must be created for the original facts before the transformation has been completed.

Formal Verification of SSA-Based Optimizations for LLVM [8]
In this paper, the authors worked to formally verify optimizations in the context of LLVM. In particular they created a new optimization vmem2reg which is verified version of the mem2reg optimization. Mem2reg promotes un-aliased local variables & stack based temporary values into registers to improve performance. The reason mem2reg was not directly verified is because it has intermediate stages which break SSA invariants, which are key to their verification. The way the verification of vmem2reg was performed was by composing the transformation of what they call mictrotransformations which were much more easily proven correct. It is worth noting that similar to many other transformations, the final stages of this one are followed by phi-elimination, dead store elimination and dead alloca elimination. Their version, vmem2reg, on average showed a 77% speedup, slightly less than the average 81% speedup of the original mem2reg. This decrease in speedup is attributed to mem2reg promoting alloca's used by LLVM intrinsics whereas vmem2reg considers these to potentially escape and does not promote them. While only the results of their vmem2reg are discussed thoroughly, the technique of reconstructing other transformation via these proven microtransformations could be applied to other LLVM transformations.

Both of these papers showed a common similarity in taking advantage of SSA form and extending to be to able to more easily accomplish their goals, namely verifying optimizations. It is also interesting to see the dominance of SSA form across many different compilation contexts, both in JIT compilation and in LLVM. Unfortunately, both of the techniques presented here require a fair amount of work effort to adjust current transformations into their corresponding extended SSA or microtransformation composition forms in order to be verified.

Register Allocation (Weeks 6-8)
Being unfamiliar with the topic of register allocation, before starting to read various papers, I read the corresponding chapter in textbook about register allocation. This enabled me to gain some background information on the topic and to understand some of the more classic and common methods along with terminology which would likely been seen in the coming papers.

Chapter 13 of Engineering a Compiler [9]
Chapter 13 of the textbooks outlines the idea of register allocation and a couple of the common approaches to the problem. The register allocator itself is what is responsible for determining at each point in the program which values will be in registers and which register will hold each of the values. In general, it is very unlikely that all values throughout the program can be kept in registers, simply because there are not enough registers. When this occurs values are spilled into memory meaning additional stores and loads are inserted to place the value in and retrieve the value from memory respectively. Based on the representation of the program going into the register allocator two differing forms of register allocators are used. In a memory to memory case, the register allocator promotes memory-bound values to registers. In the case where the program representation assumes an infinite number of registers, register to register allocation is performed, where the register allocator decides which values will live in real registers or be spilled to memory. Spilling has several undesirable impacts: there is the increased execution time of the spill code, the additional code space for the spill operations, and the data space used for the spilled values [9]. These impacts result in address computation costs and memory operation costs. The book outlines two general approaches to register allocation: the top-down approach and the bottom-up approach. In the top-down approach, one counts the number of references to a value in a block, and these frequencies are used to determine which values will

go into registers. In the bottom-up approach, one walks over the code and places values in registers, and spills as needed, different techniques are applied in determining which registers should be spilled. In practice the most common techniques for register allocation are based on either graph coloring or what is known as linear scan register allocation which will be discussed later.

A Generalized Algorithm for Graph-Coloring Register Allocation [10]
As implied by the name this paper presents a generalized algorithm for register allocation, with the generalization being the key contribution. In many register allocation techniques it is assumed that registers are interchangeable and independent. Registers are interchangeable if they are equally suitable in any program context [10]. Registers are independent if writing to one register cannot change the value of another register [10]. In practice it is not likely that either of these assumptions hold true. Some common problems which break these properties are that a single registers name appears in multiple register classes and multiple register names may be aliases for a single hardware register. A register class is a set of register names that are interchangeable in a particular role, by definition registers within a single class are interchangeable, but not necessarily independent [10]. The key insight behind this algorithm is that it allows for simultaneous allocation of multiple register classes even with register aliasing occurring.

This paper extends upon the standard graph-coloring approach to register allocation in which nodes are register candidates and the edges between them represent candidates that can be allocated to the same register known as interferences. In this application of graph coloring they base their technique on the concept of the squeeze of a node which represents the maximum number of names from this registers class (colors) which could be denied to a register because of an assignment of colors to its neighbors. The implementation of their new coloring methodology was done in a mere 1215 lines of code, and in order to apply their generalization it only needed slight adjustments in 25 places. In their presented results the generalizability of their technique comes at a cost, in which allocation time did increase, though on average not more than 15%. Much of this cost comes from the fact that spilling causes them to have to rebuild the interference graph. The paper does provide a convenient way to apply a register allocation technique to multiple architectures easily where the only difference is that they must specify the register class tree and aliasing that occurs in the architecture. This avoids the common problem of the compiler-writer having adjust the allocator by hand to handle various different architectures that they wish to support.

Linear Scan Register Allocation on SSA Form [11]
In contrast to the close to if not optimal register allocation produced by graph coloring, linear scan allocation tends to not be as close to optimal but it is generally done faster. Due to this, linear scan allocation tends to be used frequently in just-in-time compilers. Usually SSA form is deconstructed before register allocation; in contrast, in this paper SSA form is used. For this paper they modified the linear scan register allocator of the Java HotSpot compiler showing that their “simpler and faster version generates equally good or slightly better machine code” [11]. They note that the interference graph of a program in SSA form is chordal, that is for every cycle with four or more edges, there is an edge connecting two vertices of the cycle. Based on this notion they are able to construct lifetime intervals without a full dataflow analysis. Furthermore, the SSA form deconstruction can be integrated into the resolution phase of the linear scan algorithm. From their results they observed an overall compilation time savings between 4-8%

and the compiler code size itself was actually 200 lines smaller. The time savings can be attributed to a savings in both lifetime analysis (due to no dataflow analysis) and in the low-level intermediate representation (LIR) construction (due to no need for SSA decomposition). The resolution phase became slightly slower however because of the SSA decomposition being integrated at this stage.

An Optimal Linear-Time Algorithm for Interprocedural Register Allocation in High Level Synthesis Using SSA Form [12]
This paper utilizes a graph coloring approach to register allocation. In general graph coloring is non-deterministic and it takes polynomial time to complete. In this paper by utilizing SSA form, which ensures a chordal graph, their algorithm takes O(|V|+|E|) time. Extending SSA form by adding pi functions at the end of basic blocks with multiple successors, this running time can be improved further to be colored in O(|V|) time. SSA form with the additional pi functions is known as Static Single Information (SSI) form. Their technique takes advantage of the fact that by definition a chordal graph is hole-free. A hole is a chordless cycle of at least length four. They prove that the chromatic number of a chordal graph is the cardinality of the largest clique. With this they then prove that their algorithm is optimal with respect to the number of registers allocated. What is not discussed in the paper however, is how spilling is handled where they refer to an optimal spill test provided by Pereira and Palsberg which they claim can be applied without sacrificing optimality.

Register Allocation via Coloring of Chordal Graphs [13]
This paper, while it does not use SSA form also takes advantage of some of the ideas of the previous paper. Mainly they also utilize the fact that optimal coloring of a chordal graph can be done in O(|E|+|V|) time. In this case however they are utilizing this idea in the context of the JoeQ compiler being run on the runtime library of Java 1.5. Since they are not using SSA form, not all of the interference graphs that they are working on are chordal, however, 95% of the methods in the Java 1.5 library do have chordal interference graphs anyway. They reason about their argument by saying a one-perfect graph is a graph in which the chromatic number is the size of the largest clique. A perfect graph is a one-perfect graph, but also every induced subgraph is one-perfect. By proving every chordal graph is perfect, which means every chordal graph is one-perfect, the chromatic number is known to be the size of the largest clique. They use a technique known as a maximum cardinality search to acquire the Simplicial Elimination Ordering (SEO) which in turn when given to a greedy coloring algorithm will yield the optimal coloring. Should the graph not be chordal, there will be no SEO, but the order returned by the maximum cardinality search will still be used to give a non-optimal coloring.

This paper does discuss spilling along with when and what methods they use to perform spilling. When the maximal clique size is greater than the number of available colors, they utilize pre-spilling to make the graph colorable. They spill based on two-different heuristics, they spill either the least used color, or the highest color assigned. Spilling a color refers to the fact that when given a k-colored graph if one removes all the vertices sharing a color, the graph inherently becomes a k-1 colored graph. This process can be performed until the graph does not require more than the amount of available colors.

Optimal Register Allocation in Polynomial Time [14]
This paper focuses on performing register allocation in the context of embedded systems. They not only describe their technique but implement it, and it is now used as the primary register

allocator for the Small Device C Compiler (SDCC). Due to the nature of linear scan algorithms being faster, but further from optimal, this technique is also based on graph coloring. In their case they utilize tree decompositions of the CFG, which they then transform into a nice-tree decomposition, expressing the same tree but modified such that nodes can only be of four forms: leaves, introduces, forgets, or joins. By having the nodes in just these four forms they can utilize a cost function to assure optimality, in this case optimality refers to the code size which they claim to be critical for embedded systems. It is worth noting that their methodology is only done in polynomial time when the tree decomposition is of minimum width. Should the tree decomposition not be of minimum width there will be an increase in runtime, or optimality of the allocation can be sacrificed. When compared to the previous register allocator of SDCC, which was a linear scan algorithm, they observed a 10% reduction in code size.

I found this particular section of the course to be very interesting, mostly due to the fact that I had no knowledge of register allocation beforehand. Many of these papers shared various common themes in how they attacked the problem. In the case of graph coloring many of these papers utilized the fact that many interference graphs are chordal, or when expressed in SSA form are guaranteed to be chordal. Again, this gives merit to the usefulness and versatility of SSA form as an IR for programs. It was also interesting to see the similarities and differences between the goals of the register allocator. In some circumstances where compile time was critical, such as in JITs, linear scan algorithms are used. In other circumstances the amount of registers be optimal or closer to optimal is more critical, such as in embedded systems, graph coloring techniques are used. It will be interesting to see if either of these methods, graph coloring or linear scan can be improved enough with regard to time and optimality respectively, such that one becomes dominant in practice.

Proving Correctness of Optimizations and Compilers (Weeks 9-10)
Provably Correct Peephole Optimizations with Alive [15]
In this paper the authors developed and utilized a framework named Alive to be used in conjunction with LLVM. Alive has syntax very similar to LLVM which allows for easy translation of the LLVM peephole optimization to Alive. The way Alive is designed to be used is that the compiler writer writes the peephole optimization in the Alive IR representation and the tool proves or disproves the the optimization with satisfiability modulo theory; if the optimization is proven correct, then Alive will automatically generate the corresponding C++ code; if the optimization is proven incorrect then Alive provides a counterexample demonstrating why the optimization is incorrect. A couple of the key features of Alive is that it allows for abstraction over the choice of constants, over the bit-widths of operands, and over LLVM instruction attributes that control undefined behavior. In its current state, and due to the nature of peephole optimizations, Alive does not support branches.

In order to prove the optimizations correct they introduce the definitions for undefined behavior, poison values, and a notion of correctness based around these. Undefined semantically stands for the set of all possible bit patterns for a specific type [15]. Poison values “are used to indicate that a side-effect-free instruction has a condition that produces undefined behavior” [15]. With this, correctness of an optimization needs to meet three criteria, namely the target is defined when the source is defined, the target is poison free when the source is poison free, and the source & target produce the same result when the source is defined and poison free [15]. These three criteria must hold for all valid typing assignments. The authors of the paper translated approximately 300 LLVM peephole transformations and found eight of which were

incorrect. It is worth mentioning that the cases in which the source and target would produce different values are very rare (such as dividing by negative min-int). Regardless, the contribution from Alive is very helpful to aid LLVM developers in confirming the correctness of their peephole optimizations.

Compiler Validation via Equivalence Modulo Inputs [16]
This paper takes a different approach to compiler validation. In this case, they describe how an input program can be manipulated to an equivalent program. This class of the variant programs that are equivalent is called equivalence modulo inputs. By generating these equivalent modulo inputs and then compiling them as well, if there is any difference between the original compiled version and this compiled variant, then there is a compiler bug. They describe three main ways in which the program can be modified; inserting code, modifying code, or stochastically pruning code which is guaranteed to not execute. In the paper they focus mainly on the latter, in what they describe as a profile and mutate strategy. They implemented this idea in a tool they call Orion which is approximately 500 lines of shell scripts and 1000 lines of C++ code. Orion thus far has lead to 147 confirmed GCC and LLVM bugs (110 of which have since been fixed). One of the key features of their technique is its easy extensibility from C to C++ or other domains. Currently they are working on applying this to the JVM JIT compiler.

It is interesting to see two very different techniques being applied to the same problem, both of which have had fairly good results. In the case of Alive, the syntax itself is different and is used as a means to prove via satisfiability modulo theory whether the optimization is truly correct and if not; provide a counterexample. In the latter case, bugs are found by throwing test cases at the compilers which should exhibit specific known behavior. Both of these papers are very recent, and I expect the work in field proving compilers and optimizations to grow. Additionally, both of these papers have referenced CompCert in their related work sections which is a certified correct C compiler. The second paper, though it tested against CompCert, was unable to discover any bugs in its verified portions (some of the front-end stages are not verifiably correct).

Misc. (Week 10)
Mitigating the Compiler Optimization Phase-Ordering Problem Using Machine Learning [17]
As previously discussed, many optimizations can transform code such that other optimizations can be performed either better or worse. Many optimizations assume others had been run before-hand and/or others will be run following it. In many instances the compiler will use a fixed order of optimizations and always apply this order regardless of the code being presented to the optimizer. This paper extends this by introducing the idea of a method specific optimization ordering: that is each method, based on the method itself, will have a different ordering and length of optimizations applied to it. In this instance the authors utilize machine learning to determine how to go about this, and implement their technique in the Jikes RVM Java JIT compiler. They argue that the information they can acquire about a function exhibits the Markov property, that is the current state of the method represents all the information required to choose the next optimization that will be the most beneficial given its current point.

In general, the method is examined by the neural network (after having been trained) and based on its current information fed into the network, it determines what is the best optimization to be performed next. The best optimization to be performed next could be to stop performing optimizations such that the method will not be indefinitely stuck in the optimization process. In

the case of optimization A being applied then optimization B being applied, and it is determined A should be applied again and so forth, after a predetermined amount of cycling back and forth (5 times), the cycle is forcibly broken. There was a large upfront time needed to train the neural network but the results presented were enough to justify it. When compared to the adaptive compilation setting of Jikes (where Jikes decides the level of optimization to use) their neural network showed an average speedup in runtime of 8% and 4% in total time (runtime and compilation time). When compared to the optimizing compilation setting (all methods are compiled at the highest level of optimization) they observed an average speedup in runtime of 8.2% and 6% in total time. The speedup in runtime was the general goal, meaning the optimization sequence determined resulted in faster executing code. The interesting fact is that the compilation time was also improved in some cases as the number of optimizations which were applied by the neural network would be less than the static number that would have normally been applied. In these cases the neural network performed fewer optimizations in a more effective ordering.

Rigorous Benchmarking in Reasonable Time [18]
While benchmarking itself is not a compilation phase, it is a very important part when trying to identify and measure the improvement of an optimization, or the lack thereof. This paper deals with being able to properly setup benchmarking experiments to capture what is actually trying to be measured along with how many repetitions are needed at each level of the experiment to gain meaningful results. The paper covers independent states, in which execution times are independent identically distributed and initialized states in which iterations no-longer are subject to initialization overhead. Some graphing techniques I was unfamiliar with such as lag plots and autocorrelation plots are used to identify whether an independent state has actually been reached. Lag plots are used to determine if the previous run has any impact on the next run, and this can be identified by any noticeable pattern in the lag plot. The autocorrelation plot is used to identify dependencies across several runs. For instance the first run impacts the time of not the next run but perhaps the following run. The techniques presented in this paper should prove to be very useful in my future work as measuring and comparing times is a very common theme across many disciplines in computer science.

Conclusion
Throughout this course I examined many compiler optimization techniques and how they are performed. I spent the beginning of the course learning about compiler optimization techniques in general along with learning about SSA form and why it is beneficial to be used as an IR during compilation. Following this I learned about many SSA form optimizations included GVN, OSR, ABCD and lazy code motion. Additionally, I read about how SSA form can be extended to be able to verify specific optimizations. Following this, I spent three weeks learning about the topic of register allocation. This included learning about both linear scan register allocation, and register allocation which is performed via graph coloring. I concluded the final weeks of the course with reading papers regarding proving optimizations correct formally via satisfiability theory and via generating equivalent modulo inputs to test against. Throughout the course a couple of miscellaneous but related topics were covered, namely how machine learning was used to deal with the optimization phase ordering problem, and how to effectively benchmark. This course provided me a great learning experience where I could expand upon my interest in compilers despite there only being one course offered on the topic. I would like to thank Prof. Fluet for his teaching and insight throughout this course.

References

[1] Engineering a compiler, second edition by Keith D. Cooper and Linda Torczon. SIGSOFT Softw. Eng. Notes 37, 1 (January 2012), 36-37. DOI=10.1145/2088883.2088908 http://doi.acm.org/10.1145/2088883.2088908

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting equality of variables in programs. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88). ACM, New York, NY, USA, 1-11. DOI=10.1145/73560.73561 http://doi.acm.org/10.1145/73560.73561

[3] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88). ACM, New York, NY, USA, 12-27. DOI=10.1145/73560.73562 http://doi.acm.org/10.1145/73560.73562

[4] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction.ACM Trans. Program. Lang. Syst. 23, 5 (September 2001), 603-625. DOI=10.1145/504709.504710 http://doi.acm.org/10.1145/504709.504710

[5] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00). ACM, New York, NY, USA, 321-333. DOI=10.1145/349299.349342 http://doi.acm.org/10.1145/349299.349342

[6] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy code motion. SIGPLAN Not. 27, 7 (July 1992), 224-234. DOI=10.1145/143103.143136 http://doi.acm.org/10.1145/143103.143136

[7] Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. 2006. A verifiable SSA program representation for aggressive compiler optimization. SIGPLAN Not. 41, 1 (January 2006), 397-408. DOI=10.1145/1111320.1111072 http://doi.acm.org/10.1145/1111320.1111072

[8] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. SIGPLAN Not. 48, 6 (June 2013), 175-186. DOI=10.1145/2499370.2462164 http://doi.acm.org/10.1145/2499370.2462164

[9] Engineering a compiler, second edition by Keith D. Cooper and Linda Torczon. SIGSOFT Softw. Eng. Notes 37, 1 (January 2012), 36-37. DOI=10.1145/2088883.2088908 http://doi.acm.org/10.1145/2088883.2088908.

[10] Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A generalized algorithm for graph-coloring register allocation. SIGPLAN Not. 39, 6 (June 2004), 277-288. DOI=10.1145/996893.996875 http://doi.acm.org/10.1145/996893.996875

[11] Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and

optimization (CGO '10). ACM, New York, NY, USA, 170-179. DOI=10.1145/1772954.1772979 http://doi.acm.org/10.1145/1772954.1772979

[12] Philip Brisk, Ajay K. Verma, and Paolo Ienne. 2010. An optimal linear-time algorithm for interprocedural register allocation in high level synthesis using SSA form. Trans. Comp.-Aided Des. Integ. Cir. Sys. 29, 7 (July 2010), 1096-1109. DOI=10.1109/TCAD.2010.2049060 http://dx.doi.org/10.1109/TCAD.2010.2049060

[13] Fernando Magno Quintão Pereira and Jens Palsberg. 2005. Register allocation via coloring of chordal graphs. In Proceedings of the Third Asian conference on Programming Languages and Systems (APLAS'05), Kwangkeun Yi (Ed.). Springer-Verlag, Berlin, Heidelberg, 315-329. DOI=10.1007/11575467_21 http://dx.doi.org/10.1007/11575467_21

[14] Philipp Klaus Krause. 2013. Optimal register allocation in polynomial time. In Proceedings of the 22nd international conference on Compiler Construction (CC'13), Ranjit Jhala and Koen Bosschere (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-20. DOI=10.1007/978-3-642-37051-9_1 http://dx.doi.org/10.1007/978-3-642-37051-9_1

[15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2015). ACM, New York, NY, USA, 22-32. DOI=10.1145/2737924.2737965 http://doi.acm.org/10.1145/2737924.2737965

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.* 49, 6 (June 2014), 216-226. DOI=10.1145/2666356.2594334 http://doi.acm.org/10.1145/2666356.2594334

[17] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.* 47, 10 (October 2012), 147-162. DOI=10.1145/2398857.2384628 http://doi.acm.org/10.1145/2398857.2384628

[18] Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on memory management* (ISMM '13). ACM, New York, NY, USA, 63-74. DOI=10.1145/2464157.2464160 http://doi.acm.org/10.1145/2464157.2464160