Howest.be

Digital Arts and Entertainment

THE IMPACT OF COMPILER OPTIMIZATIONS

by

Krikilion Laurens, DAE, Game Development

Advisor(s): Vanden Abeele Alex

Graduation Work

2020-2021

TABLE OF CONTENTS

Page

TABLE OF CONTENTS i

ABSTRACT iii

INTRODUCTION 1

i LITERATURE STUDY **2**

1. IR-Level optimizations **2**

1.1 Control-Flow Graph 2

1.1.1 Implementation 2

1.1.2 Examples 3

1.1.3 Usages 4

1.2 Constant Propagation 5

1.2.1 Constant Folding 5

1.2.2 The Algorithm 7

1.2.3 Complexity and Completeness 7

1.3 Dead code elimination 8

1.3.1 The Algorithm 8

1.3.2 Complexity and Completeness 10

2. Code Generation **11**

2.1 Instruction Selection 12

2.1.1The algorithm 14

1.3.2 Complexity 15

2.2 Instruction Scheduling 16

2.2.1 General Approach 16

2.2.2 Conclusion 16

2.3 Register allocation 17

2.3.1 Global allocation 17

2.3.2 Linear scan 17

2.3.3 Graph Coloring........ 18

2.3.4 Conclusion........ 18

2.4 Peephole optimization........ 19

2.4.1 Use cases........ 19

2.4.2 Conclusion........ 19

ii CASE STUDY **21**

3 Methodology **21**

3.1 Front End........ 21

3.2 Implemented Optimizations........ 22

3.2.1.Dead Code Elimination........ 22

3.2.2 Constant Propagation........ 22

3.2.3 Instruction Selection........ 23

3.2.4 Register Allocation........ 23

3.2.5 Peephole Optimization........ 24

4 Evaluation **25**

4.1 Compile time........ 25

4.2 Micro Operations........ 30

4.2.1 Specific Optimization Comparisons........ 32

4.2.2 Combined IR level optimizations........ 33

CONCLUSION & FURTHER WORK **35**

Appendix A **36**

ASM - x86 **36**

A.1 Stack........ 36

A.2 Functions........ 36

A.2.1 Prologue........ 37

A.2.2 Function call........ 38

BIBLIOGRAPHY........ 40

ABSTRACT

This paper will go over the impact of compiler optimizations in a small compiler. We take a look at intermediate representation level optimizations such as dead code elimination, constant propagation and code generator optimizations like instruction selection, register allocation, and peephole optimization. We explain the implementation and complexity of these optimizations, and evaluate them in terms of compile time and micro operations. We compare our results to state-of-the-art compilers GCC and TCC(Tiny C Compiler) and determine the individual impact of each optimization and optimal combinations of optimizations. We provide results with the potential to improve the performance of small compilers.

INTRODUCTION

Compiler optimizations are often disregarded, we tend to get more frustrated with our compiler than thank it for some of the wonderful things it does. Writing low level optimized code is both hard to maintain and understand. Therefore we write programs in ‘high-level’ languages, as these are easier understood by a human. However computers do not understand these high-level languages so we use compilers to translate these high level languages to low-level languages that machines can understands. These compilers take care of both translating these high-level languages to low-level languages that and optimizing the code in the process. This allows the programmer to focus on writing clear maintainable code. These compiler optimization have been a focus of research for a long time [2]. In this paper we will take a look at techniques used in optimizing compilers and the impact each of these different optimizations have in terms of performance and compilation time. To do this we compared our result to results of some well knowns compilers using a variety of source files and measured these in terms of complexity and micro operations of the generated output

i LITERATURE STUDY

Before a compiler can convert an expression into machine code that can be executed, it needs to understand both its syntax and meaning.[3, Ch. 1.3.1] The front end of the compiler first checks if the input code is syntactically correct. If this is the case, it creates a representation of the code in the form of an Intermediate-Representation(IR). If there are errors, it reports error messages often in combination with the identified problem and location of that error in the source code to identify the problems with the code. The optimizer then analyses the intermediate representation of the code and uses this to rewrite the code more efficiently. After this, the back end of the compiler goes through the intermediate representation of the code and produces code for the target machine[4, Ch. 6]. It does this by choosing the target machine operations needed to implement each IR operation together with the order in which the operations will execute efficiently and decides where values will be stored in registers or memory. Finally, this results in an optimized version of the code that the machine can understand.[3, Ch. 1.3.3] In the following sections we will discuss this intermediate representation and code generation stage, we will discuss various algorithms at each stage and their applications.

1. IR-Level optimizations

1.1 Control-Flow Graph

A control flow graph(CFG) is a graphical representation of the flow of control in a program. It is commonly used in compilers and other tools to analyse and optimize the structure of a program.[5]

In a control flow graph, each node represents a basic block of code, which is a sequence of instructions that is executed without any branching or jumping.[4] Edges between nodes represent control flow between basic blocks, and can be labeled with conditions or other information.[6]

1.1.1 Implementation

Control flow graphs can be implemented as doubly-linked graphs, which are graphs in which each node is connected to its successors and predecessors by two pointers. Each node has a set of pointers pointing to its dynamic successors and a set of pointers pointing to its dynamic predecessors. This allows algorithms working with the control flow graph to easily navigate through the graph, which will prove especially useful when working with algorithms that use data flow equations, like Sparse Conditional Constant propagation described in the later section about constant propagation.[4, pp. 264–265], [7, Ch. Data-Flow Analysis]

1.1.2 Examples

Figure 1,2,3,4 shows an example of the control flow for different statements. The basic blocks are represented by the circular nodes, and the edges represent the flow and how a computer would read trough the given source code.

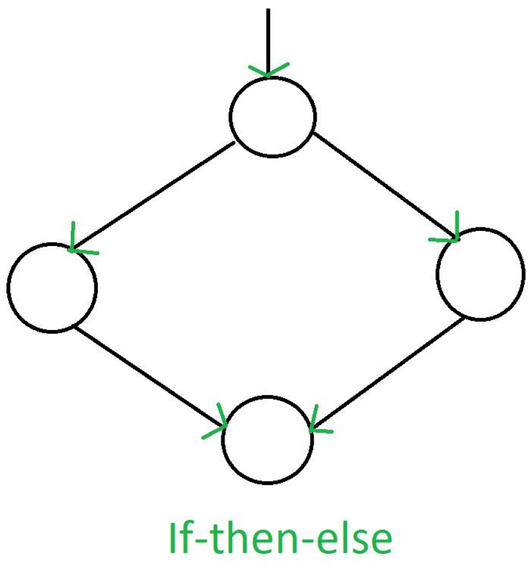


Figure 1. Graphical representation of If Else -statement

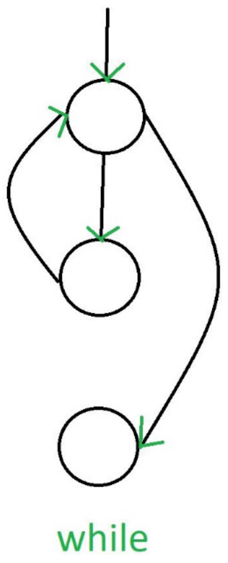


Figure 2. Graphical representation of while statement

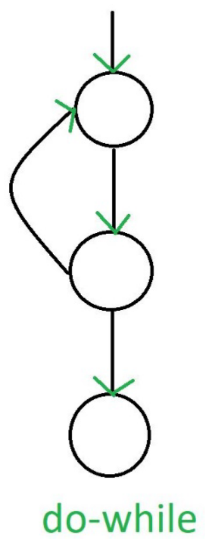


Figure 3. Graphical representation of do while -statement

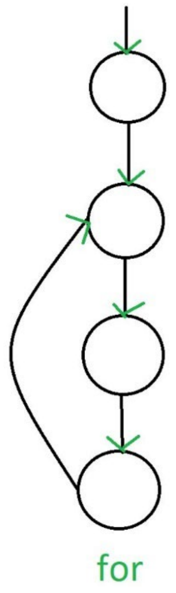


Figure 4 . Graphical representation of a for loop

[8, Figs. 1, 2, 3, 4]

Images retrieved from https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/

1.1.3 Usages

Control flow graphs can be used for a variety of purposes in compilers and other tools.[5] For example, they can be used to:

Analyse the structure of a program: They are used to understand the detail of a process, showing where control starts and ends.[9] By examining these, it is possible to identify patterns and relationships within the code, such as in loops and conditional statements. This can be useful for later optimizations to the code.[4, pp. 360, 465, 595]

Optimizing the code: By analysing the control flow graph, it is possible to determine whether variables are being used at specific points in the program. A variable is considered "live" at a particular node in the control flow graph if its value is used on at least one path through the control flow graph from that node. If a variable is not used on any path from a particular node, it is considered "dead" at that node. Note it is important to consider the use of a particular value of a variable, rather than just the variable itself. During code generation, it can be important to know if a variable is live or dead at a given node in the code since if it is dead, the memory allocated to it can be

Reused for another variable. This is especially important if the variable resided inside a register since from that given node on, the register can be reused for variable and purposes. [4, pp. 285–289]

Conversion to dependency graph: control-flow graph is acyclic and can easily be transformed into a data dependency graph, which is advantageous for code generation. This is because a dependency graph is much less restrictive to instruction scheduling than the control-flow graph.[4, p. 390] (Dependency graphs and Instruction scheduling are treated more extensively in the chapter about code generation)

These are just a few of the possible usages of control flow graphs but they can be used for a variety of other applications. The use of control flow graphs in compilers is an effective 'guide' for compilers to analyse and optimize code.[10]

1.2 Constant Propagation

Constant propagation is a basic but sometimes powerful optimization in compilers. It consists of evaluating constants assigned to a variable and propagating these through at compile time.[11]–[13] This technique can be applied during various stages of a compiler[14], including the intermediate representation stage, and is typically used to improve the performance of a programs runtime by reducing the total number of instructions.

1.2.1 Constant Folding

Constant folding is an optimization technique that involves replacing expressions with their constant values at compile time.[15] Just like constant propagation this can be done to improve the performance of a program by reducing the instructions needed. For example, if the source code contains the expression `"2 + 2,"` the constant folding algorithm would replace it with the constant value `"4"`.

The difference between constant folding and constant propagation is that constant folding only looks at the operands of the current expression and if it is able to calculate the result at compile time it will, whereas constant propagation will propagate constant values trough and substituted that value everywhere that variable is used.[16]

Constant folding combined with constant propagation can be applied to a wide range of expressions, including but not limited to, arithmetic expressions, logical expressions, and conditional expressions. For example, consider the following code snippets:

```
int x = 2 + 2;
int y = x * 3;
int z = (x > 0) ? 1 : 0;
```

After constant propagation, the code could be transformed as follows:

```
int x = 4;
int y = 12;
int z = 1;
```

In this example, the expressions "`2 + 2`" and "`(x > 0)`" have been replaced with their constant values, and the expression "`x * 3`" has been simplified to "12."

Constant propagation can also be used in combination with other optimization techniques, such as common subexpression elimination and strength reduction.[7] As shown in the following code snippets:

```
int a = 2;
int b = 2;
int c = a * x + 2 * y;
int d = b * x + 2 * z;
```

After applying constant propagation and common subexpression elimination, the code could be transformed as follows:

```
        …
int tmp = 2 * x;
int c = tmp + 2 * y;
int d = tmp + 2 * z;
```

Here, the constant value "`2`" has been propagated trough replacing both "`2 * x,`" this allows common subexpression to eliminate the "`2 * x`".

1.2.2 Sparse Conditional Constant Propagation

Sparse Conditional Constant Propagation (SCC)[17] is one example of an algorithm to perform constant propagation, this algorithm uses a Static Single Assignment form(SSA)[18]. SCC uses this SSA to propagate constants through a control flow graph, taking branches into account and computes their values at compile time rather than at runtime. Sparse Conditional Constant Propagation is a more powerful form of constant propagation than Simple Constant Propagation as it takes control flow into account and can make conclusions about the values of variables based on the control flow.[18, Pt. Motivation]

1.2.3 The Algorithm

To perform Sparse Conditional Constant propagation(SCC), the compiler first constructs the control flow graph of the program. From here this CFG is converted in to SSA form[18, Pt. SSA] and annotates each basic block with a set of "in" and "out" variables, which represent the values of variables that are live (used) at the beginning and end of the basic block.[19]

During constant propagation, the compiler traverses the CFG and performs the following steps at each basic block:

1. Compute the "in" set of the block by intersecting the "out" sets of all predecessor blocks.
2. Compute the "out" set of the block by adding the "in" set to the set of variables defined or used in the block.
3. Replace any expressions that are constant within the block with their constant values.

1.2.4 Complexity and Completeness

The complexity of SCC is O(n*m), where n is the average number of predecessor blocks and m the average number of variables inside of that block, thus in practice this depends on the specific implementation and the size of the program being optimized. In general, SCC algorithms involve involves propagating constant values as far as possible through a program. This can be a time-consuming process, especially for larger programs.

Therefore, it is typically used in combination with other optimization techniques to achieve the best trade-off between performance and compilation time.[7], [17]

One of the challenges of SCC is achieving completeness, or the ability to identify all possible constant values in a program. This can be difficult because programs often have complex control flow and may include loops or conditional statements that can change the values of variables. While it is generally possible to achieve high levels of completeness SCC does not guarantee full completeness but does guarantee to find at least as many constants as Conditional Constant propagation(CC).[17, p. 194]

1.3 Dead code elimination

Dead code elimination is a optimization technique that removes code that will never be executed during runtime, called 'deadcode'. This can improve performance by reducing the size of the generated machine code and opening up further opportunities for more optimizations to the remaining code.[3, Ch. 10.2]

There are multiple ways in which code can become dead in a program. For example, code might be surrounded by an "if" statement that can be evaluated at compile time and is always true or false, or a loop that is never entered. In these cases, the code will be unreachable, and can thus be safely removed.[3, Ch. 10.2.1]

To perform dead code elimination, the compiler must first identify the unreachable code. This can be done using a variety of techniques, such as static analysis (analysing the code without executing it) or dynamic analysis (executing the code and observing its behaviour).

1.3.1 The Algorithm

One way to eliminate dead code using static analysis is to mark all instructions that are "useful" in some sense and remove any that are not marked. This process is similar to how mark-sweep garbage collectors work[20], with the IR code serving as the data. (For a more detailed explanation of mark-sweep garbage collectors, see [3, Ch. 6.6.2]) The algorithm performs two passes over the code.

In the first pass, the algorithm marks certain "critical" operations as useful. These are operations that are considered important for the correct execution of the code e.g. things like setting return values of a function, performing input/output statements, or affecting the value of other ‘critical’ variables or operations. [3, Ch. 10.2.1]

Once all the critical operations have been marked as useful, the algorithm also marks all of the operands of that operation as ‘useful’ as well and continues this process until no more instructions can be marked. For example, if you have the operation `‘c = a + b;’` in case `‘c’` is marked as critical both `‘a’` and `‘b’` should also be marked as critical as these are necessary to compute the value of the critical `‘c’`. [21]

In the second pass, the algorithm walks through the code again and removes any operation that is not marked as 'useful'. This can include entire blocks of code that are not reachable (for example unreachable branches) as well as individual instructions that are not used by any ‘useful’ operations.

It is worth noting that static analysis algorithms for dead code elimination have some limitations. They can only identify and remove code that is unreachable under certain conditions, such as when the values of variables are known at compile time. If the values of variables are not known until runtime, the compiler will not be able to determine whether a block of code is reachable or not, and it will need to leave the code in the intermediate representation. Therefore it is often used in combination with constant propagation, as both of these techniques involve removing or replacing code in the intermediate representation of a program. In particular, constant propagation can assist to identify unreachable code in the IR, for example it can allow the compiler to determine what path it might take in the CFG according to the value of a constant.

For example, consider the following code:

```
int x = 5;

if (x > 0) {
  // Code…
}
```

In this case, the value of ‘x’ is known to be 5 at compile time, so the condition "x>0" will always evaluate to be true. This means the compile can safely assume that the code inside the if statement will always be executed, and it cannot be removed. This allows us to predict part of the control flow off our program and optimize accordingly by safely removing any basic blocks or operations that are known to be never executed.[4, p. 323]

Dead code elimination, while seemingly trivial, is a complex optimization technique that can significantly improve the performance of compiled code by reducing its size and allowing further optimizations to the remaining code. While some programmers might question the need for dead code elimination as they “do not write useless code”[3, p. 551] there are some other optimization like SCC as mentioned previously, that can create dead code and thus even when one does not explicitly write dead code, dead code elimination can still have a profound effect for achieving optimal results.

1.3.2 Complexity and Completeness

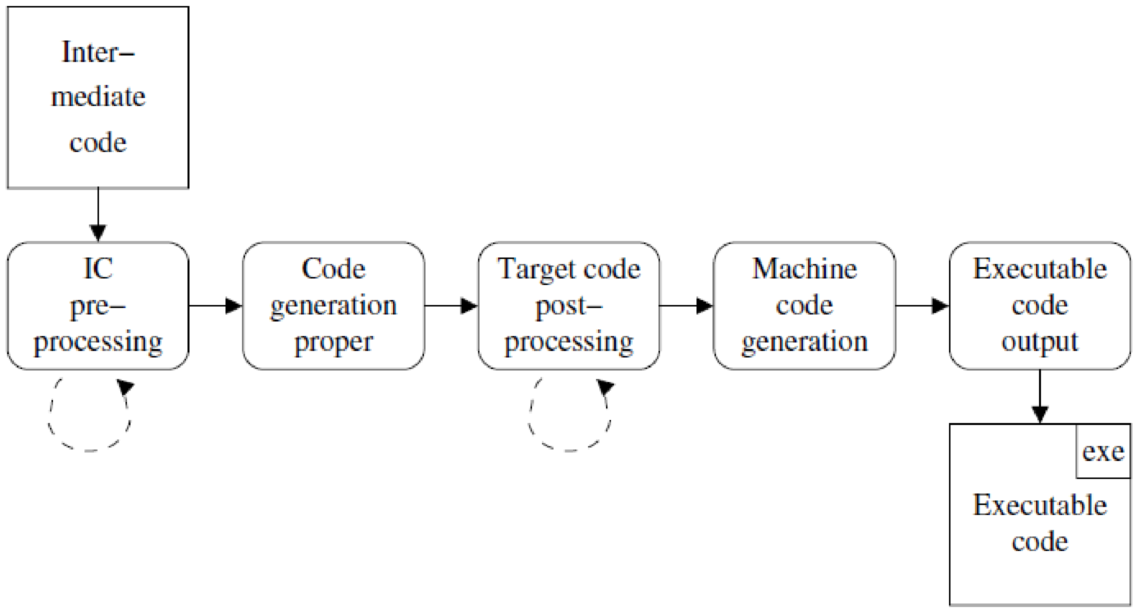
The algorithm consists of two main phases: the mark phase and the sweep phase. The mark phase has a linear time complexity of O(n) as it traverses the CFG and identifies all critical code and marks these as useful. Similarly, the sweep phase also has a linear time complexity as it walks through the code and removes any operation that is not marked as useful. Thus the overall time complexity of the algorithm is therefore O(n).

When defining completeness of dead code elimination as the ability to find all dead code in a program; every algorithm for dead code elimination will be incomplete. The problem of detecting and eliminating dead code is closely related to the Halting problem[22], which was proven to be unsolvable by Alan Turing. The Halting problem states that it is impossible to write a general algorithm that can determine, for any given program and input, whether the program will halt or continue running indefinitely.

As finding dead code requires determining which parts of a program will never be executed, this means that it is also impossible to write a complete algorithm for dead code elimination because if it were to be possible to write an algorithm for dead code elimination, it could be used to solve the Halting problem, which has been proven to be impossible by Alan Turing.[23]

2. Code Generation

Code generation is the process of converting an intermediate representation of a program, such as an abstract syntax tree(AST), into machine-readable code. It involves replacing nodes and subtrees of the AST with target code segments, in a way that preserves the semantics of the program. [4, Ch. Code Generation] This replacement process is known as tree rewriting, The gradual transformation from AST to target code, where semantics remains unchanged, helps in designing accurate code generators.



[4, p. 321] In summary: code generation is performed in three phases:

Preprocessing: Which refers to the process of analysing and transforming the AST of a program before it is converted into machine-readable code. The goal of preprocessing is to improve the performance of the generated code by replacing AST node patterns with other, more efficient patterns. We will not discuss preprocessing in detail as a large part of this overlaps with intermediate representation optimizations and is more aimed towards compilers that don't have an explicit IR stage.

Code Generation Proper: This is the core of the code generation process. The target code sequences generated in this phase will be in the machine-readable format, such as assembly or machine code and it typically involves Instruction Selection, Register

Allocation, and Instruction Scheduling which are the three main issues in code generation.[3, p. 600], [4, p. 319]

Postprocessing: Which is the final step of the code generation process and it involves making further optimizations to the generated code after it has been converted from an intermediate representation to machine-readable code. This step can be done trough techniques like peephole optimization or branch optimization etc. [3, p. 383], [4, p. 353]

Both preprocessing and postprocessing steps may open new opportunities for further optimizations, as a result some compilers may repeat these steps multiple times to ensure that the generated code is as efficient as possible.

2.1 Instruction Selection

Instruction selection can be a important step in the compilation process, during instruction scheduling each instruction will be mapped to equivalent constructs of the machine's target instruction set architecture(ISA). This process can involve various techniques, such as deciding weather or not to store a variable in a register and combining multiple IR operations into a single machine instruction. In this section, we will discuss the details of instruction selection and its role in the compilation process.[3, Ch. Instruction Selection]

One simplistic approach to instruction selection is to provide a single target ISA sequence for each IR operation, resulting in a template-like expansion that produces correct code. However, this method may result in poor utilization of target machine resources. To address this issue, more advanced approaches consider multiple possible code sequences for each IR operation, and choose the sequence with the lowest expected cost. [3, Ch. Instruction Selection], [4, Ch. 9.1.4]

The complexity of instruction selection stems from the fact that processors often provide various distinct ways to perform the same computation.

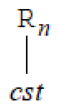
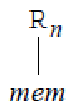
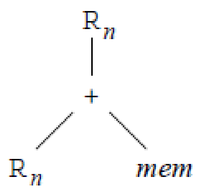
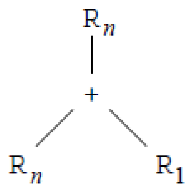
	`Load_Const` cst, R_n	load constant	cost = 1
	`Load_Mem` mem, R_n	load from memory	cost = 3
	`Add_Mem` mem, R_n	add from memory	cost = 3
	`Add_Reg` R_1, R_n	add registers	cost = 1

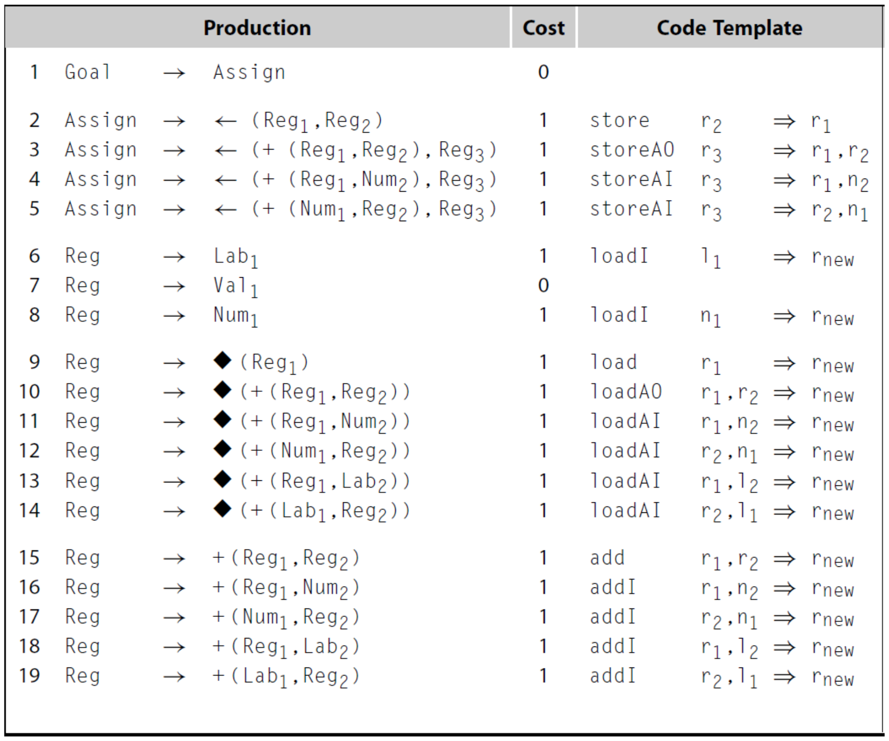
Figure 6 Sample instruction patterns [4, Fig. 9.23]

For example, the operation of `'x + y'`, where both `'x'` and `'y'` are variables stored in memory. A naive approach to implementing this operation might involve loading both `'x'` and `'y'` into registers and performing the addition operation on these registers. As shown in figure 6, the cost of this naive implementation would be the sum of the cost of loading both `'x'` and `'y'` from memory, plus the added cost of the addition operation between two registers, resulting in a total cost of 7 (Note that the cost of execution in figure 6 is measured in arbitrary units and are not representative of real-world values, as the cost of execution for each instruction is dependent on the specific machine architecture, for a list of accurate costs according to specific architectures you can refer to [24, Sec. 3]).

However, alternative approaches may also be considered. For instance, one could choose to only load `'x'` into a register and perform the plus operation between the register containing `'x'` and `'y'` in memory, resulting in a cost of 6. Furthermore, if one of the variables `('x'or'y')` were to be a constant, the cost of loading that value into a register would be only 1 instead of 3 for loading a memory address into a register. Additionally to that, the size of the variables also plays a significant role in the cost and order of the instructions. A human programmer may quickly discard most, if not all, of these alternate sequences, but an automated process must consider all possibilities and make the appropriate choices. The ability of a specific instruction set architecture to accomplish the same effect in multiple ways increases the complexity of instruction selection. [3, Ch. Instruction Selection]

2.1.1The algorithm

The optimum tiling algorithm is a technique used in instruction selection. It uses dynamic programming to find the most efficient combination of instructions on a target architecture for each operation. Each instruction is given a ‘tile’ which is a sort of blueprint for a operation and these tiles together form a instruction sequence. The goal is to minimize the total execution time of the code by finding the optimal series of instructions to tiles. The algorithm considers all possible combinations of instructions to tiles and selects the one that minimizes the total execution time using a set of rewrite rules and optionally previous nodes. A rewrite rule consists of a production in a tree grammar, a code template, and an associated cost. [3, p. 610], [4, p. 411], [25]



	Production	Cost	Code Template
1	Goal → Assign	0	
2	Assign → ← (Reg$_1$,Reg$_2$)	1	store r$_2$ ⇒ r$_1$
3	Assign → ← (+ (Reg$_1$,Reg$_2$),Reg$_3$)	1	storeAO r$_3$ ⇒ r$_1$,r$_2$
4	Assign → ← (+ (Reg$_1$,Num$_2$),Reg$_3$)	1	storeAI r$_3$ ⇒ r$_1$,n$_2$
5	Assign → ← (+ (Num$_1$,Reg$_2$),Reg$_3$)	1	storeAI r$_3$ ⇒ r$_2$,n$_1$
6	Reg → Lab$_1$	1	loadI l$_1$ ⇒ r$_{new}$
7	Reg → Val$_1$	0	
8	Reg → Num$_1$	1	loadI n$_1$ ⇒ r$_{new}$
9	Reg → ◆(Reg$_1$)	1	load r$_1$ ⇒ r$_{new}$
10	Reg → ◆(+(Reg$_1$,Reg$_2$))	1	loadAO r$_1$,r$_2$ ⇒ r$_{new}$
11	Reg → ◆(+(Reg$_1$,Num$_2$))	1	loadAI r$_1$,n$_2$ ⇒ r$_{new}$
12	Reg → ◆(+(Num$_1$,Reg$_2$))	1	loadAI r$_2$,n$_1$ ⇒ r$_{new}$
13	Reg → ◆(+(Reg$_1$,Lab$_2$))	1	loadAI r$_1$,l$_2$ ⇒ r$_{new}$
14	Reg → ◆(+(Lab$_1$,Reg$_2$))	1	loadAI r$_2$,l$_1$ ⇒ r$_{new}$
15	Reg → +(Reg$_1$,Reg$_2$)	1	add r$_1$,r$_2$ ⇒ r$_{new}$
16	Reg → +(Reg$_1$,Num$_2$)	1	addI r$_1$,n$_2$ ⇒ r$_{new}$
17	Reg → +(Num$_1$,Reg$_2$)	1	addI r$_2$,n$_1$ ⇒ r$_{new}$
18	Reg → +(Reg$_1$,Lab$_2$)	1	addI r$_1$,l$_2$ ⇒ r$_{new}$
19	Reg → +(Lab$_1$,Reg$_2$)	1	addI r$_2$,l$_1$ ⇒ r$_{new}$

Figure 7 Rewrite Rules for Tiling [3, Fig. 11.4]

The result is the schedule that minimizes execution time across all of the tiles.

```
void matchExpr (Tree.Exp e) {
  for (Tree.Exp kid : e.kids())
    matchExpr (kid);

  cost = INFINITY
  for each pattern P_i
    if (P_i.matches (e)) {
      cost_i = cost(P_i)
             + sum ((wildcard (P_i, e)).mincost)
      if (cost_i < cost) { cost = cost_i; choice = i; }
    }
  e.matched = P_{choice}
  e.mincost = cost
}
```

Figure 8 pseudo implementation of Optimum Tiling[25]

The optimum tiling algorithm allows the compiler to take advantage of the characteristics of the target architecture to schedule instructions in the most efficient way possible. However, it can be quite complex and expensive to compute.

1.3.2 Complexity

The complexity of the algorithm using dynamic programming is O(m + n)k.

The first term 'm', represents the maximum amount of nodes(machine code instructions) needed to check for the algorithm to find a match for each operation. This is dependent on the specific instructions being used and the target architecture.

The second term 'n', indicates the average time needed to find the optimal pattern(combination of instructions) for each of the operations and dependents on the number of possible combinations for each instruction. Bigger architecture with more possible combinations take longer to find a best fit match than smaller architectures with less available instructions.

The third term 'k', specifies the total number of instructions in the source code.[25]

2.2 Instruction Scheduling

Instruction scheduling is used to optimize the order of execution of machine instructions to maximize overall parallelism and minimize data dependencies and thus reducing the number of cycles required to execute a given set of instructions.[3, Ch. 6.12.2], [26, Ch. 10.4] For this a list scheduling algorithm can be used. However like Cooper K and Torczon L describe in their book Engineering a compiler "list scheduling is an approach rather than a specific algorithm."[3, Ch. 12.3] There are many variations in how it can be implemented and we will describe a general approach for list scheduling.

2.2.1 General Approach

We start by analysing the instructions in the basic block to be scheduled and identify the data dependencies between them, this way we can construct a dependency graph where each node contains an operation and its newly created value from each of these nodes together with a rank that indicates its priority these priorities are used as a guide for the scheduler often using one as the primary ordering rank and the others to break ties between equally ranked nodes. We continue by adding edges from each node to every other node that uses its value combined with the latency required to execute set operation.[3, Ch. 12.3.1]

When an instruction is ready we can schedule instructions by ordering them based on their dependencies and priorities. Instructions that are independent of each other are placed at the beginning of the list, while instructions that have dependencies are placed later. An instruction is ready if it itself has not yet been scheduled and all of its predecessors have been scheduled and the depended latencies have passed.[27]

2.2.2 Conclusion

List scheduling is a widely employed technique in modern compilers for optimizing the execution of machine instructions on a processor.[28] The algorithm is highly adaptable, and its effectiveness can be fine-tuned by modifying the priority schemes, tie-breaking rules, and even the direction of scheduling.[3, Ch. 12.6] List scheduling is known for its robustness, as it consistently generates good results[3, Ch. 12.6].

2.3 Register allocation

Register allocation is the process of assigning and managing local variables and expression results in registers.[29], [30] It is an important method in the final phase of the compiler, as registers are located in the processor and provide the fastest way to access data[31]. Register allocation algorithms can be divided into two categories: global and local.[3, Ch. 13], [32] Global register allocation attempts to optimize across multiple expressions, while local register allocation works on one expression at a time.[33], [34] We will primarily focus on global allocation algorithms, as they are more complex and we think require further explanation. For a more in depth explanation of local allocation, interested readers can refer to the following article. [33].

In recent years, there has also been a interest in using machine learning and hybrid algorithms that combine the strengths of different techniques for register allocation [29], [35]–[38] .These new techniques have the potential to improve the performance of register allocation trough techniques like reinforcement learning.[39] They are trained to find optimal register allocation and prevent as much spilling(writing a variable from a register to memory[29]) as possible.[39]

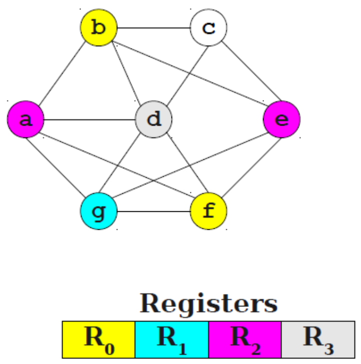
2.3.1 Global allocation

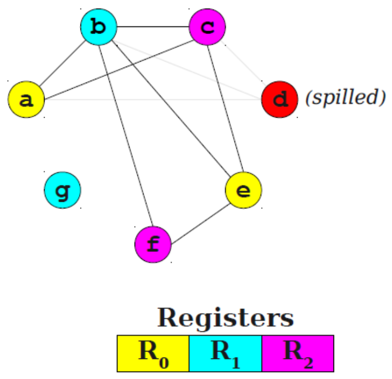
We will consider Linear Scan and Graph Coloring; both are global allocation algorithms and share the same goal of minimizing the number of register spills while maximizing the number of variables stored in registers.[40] Each of these algorithms has its own advantages and disadvantages[33] and the choice of algorithm will depend on the goals of the compiler and the target programs.

2.3.2 Linear scan

Linear scan is an algorithm that scans the instructions in a basic block in linear order and assigns registers to variables as they are encountered. The algorithm starts with a empty set of available registers and a set of live variables. As the instructions are scanned, the algorithm checks if a variable is being defined or used, and if it is being defined, the algorithm checks if there is a register available to hold the value. If there is no available register, the algorithm spills the oldest live variable that is not currently being used. [40], [41]

2.3.3 Graph Coloring





Graph coloring is an algorithm that uses the principles of graph theory to assign registers to variables. The algorithm starts by building a interference graph, where each node represents a variable and an edge between two nodes represents a data dependency between the two variables. The algorithm then uses graph coloring techniques to assign a register to each variable, while ensuring that no two adjacent nodes (variables with a data dependency) are assigned the same register. If there is not enough registers to assign to all the variables, the algorithm spills the variable with the least number of interfering nodes.[4, p. 429]

The problem is that it's NP-complete meaning that even the best algorithms cannot find optimal coloring within a polynomial time. However it can be solved in polynomial time using non-deterministic algorithms these can solve the problem in almost linear time and usually do a good to very good job. [4, Ch. 9.1.5.2]

2.3.4 Conclusion

Graph coloring is more complex to implement but can produce better register assignments, as it takes into account the data dependencies between variables while linear scan does not do this. Which one is best depends on what machine you're working with and the expected size and complexity of programs that will be compiled. While graph coloring produces more efficient code it also has a higher overhead than linear scan.[42]

2.4 Peephole optimization

Even moderately sophisticated code generation techniques can produce ‘stupid‘ instruction sequences like:

```
Load_Reg R1,R2
Load_Reg R2,R1
```

or

```
Store_Reg R1,n
Load_Mem n,R1
```

One solution to this problem would be to go over the generated assembly and identify stupid instructions like these. This solution is called peephole optimization [3, Ch. 11.5.1], [4, Ch. 7.6.1]. Peephole optimization is a technique that analyses short sequences of adjacent operations called ‘peepholes’ [26, Ch. 8.7] and uses these peepholes to identify and remove inefficiencies like these. [43]

2.4.1 Use cases

The main idea behind peephole optimization is to identify and eliminate redundant or unnecessary instructions in the code. This can include removing unnecessary loads and stores, common subexpression elimination, constant folding, and strength reduction[26, Ch. 8.7.4].

Peephole optimization also allows the replacement of slow instructions with faster equivalents.[3, p. 624], [4, p. 350], [26, Ch. 8.7.4], [43] For example, a peephole optimizer might replace a slow instruction with a sequence of faster instructions that produce the same result.[43] For example On some processors, multiplication is extremely expensive, and it is worthwhile to replace all multiplications with a constant by a combination of left-shifts, additions, and/or subtractions. E.g.

```
‘3 * V -> (V << 1) + V’
```

2.4.2 Conclusion

Peephole optimization also has its limitations. The optimization is done on the assembly or machine code level[43], so it can only be performed after the source code has been translated into assembly or machine code. This means that the optimization cannot be applied to high-level constructs such as loops and branches. Additionally, as mentioned in the book Modern compiler design by Grune D *“Peephole optimization is inversely proportional to the quality of the code yielded by the code generation phase. A good code* *generator requires little peephole optimization, but a naive code generator can benefit greatly from a good peephole optimizer."*[4, Ch. 7.6.1.3]

ii CASE STUDY

In the following chapters, we describe our study on the effects of different compiler optimizations. We often refer to our own compiler under the abbreviation ‘Crk’ coming from the name of our compiler ‘Cricket’. We start off by comparing the overall results of our compiler to other existing compilers to give us a better understanding of the position of our compiler in the current state of the art in terms of compile time and quality of the generated code in terms of micro operations.

3 Methodology

In this chapter we describe the various algorithms implemented and how we evaluated them. Later in chapter 4 we will evaluate the resulting data and conclusions.

3.1 Front End

For the front end of the compiler we used ANTLR (ANother Tool for Language Recognition). ANTLR generates a parse tree from a provided grammar file, which holds the set of rules that define a language. Additionally ANTLR provides us a visitor pattern which we can use to traverse this parse tree and create our Intermediate representation in the form of a Control Flow Graph(CFG) and our symbol table that holds every variable and function in relation to its scope. The CFG is used to split the source code in various basic blocks and keep track of each expression in the code. This can then be used to analyse and optimize the source code as discussed in the literature study section.

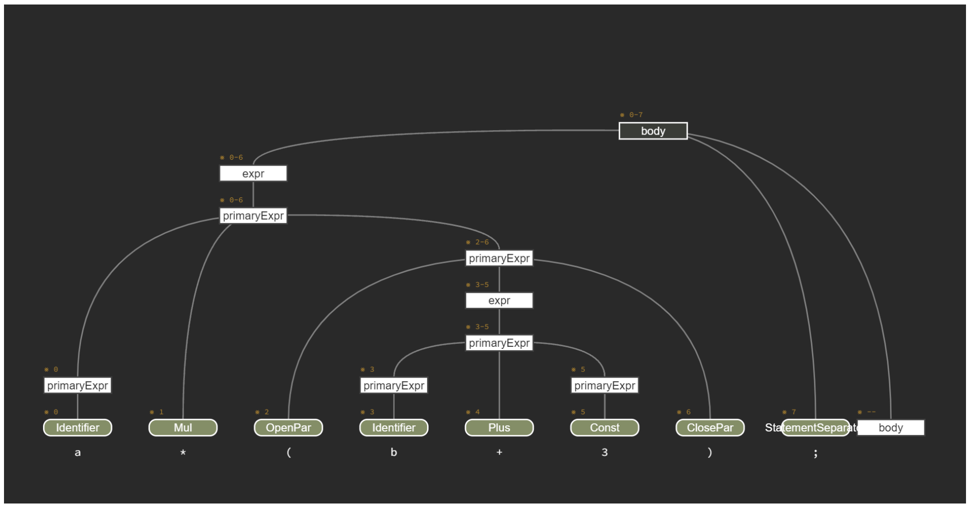


Figure 11 visualization of the parse tree generated by ANNTLR for the expression a(b+3);*

3.2 Implemented Optimizations

When compiling with the O0 flag, our compiler has only instruction scheduling turned on in contrast to this most widely-used compilers like GCC, MSVC, and Clang have a relatively high baseline; GCC 12.2 for instance has over 50 options turned on when compiling with O0 , which can be viewed by using the flags "-Q --help=optimizers" therefore it is easy to underestimate how big of an impact some of these optimization may have compared to a fully unoptimized compiler. One example of these 'baseline' optimizations that are turned on at O0 is dead code elimination.

3.2.1.Dead Code Elimination

In order to measure the possible improvements of the performance we have implemented dead code elimination in our compiler, using the mark and sweep algorithm as described in section 1.3.1 of our paper. As described there and later confirmed through our own experiments, presented in section 4.2.2. We concluded that dead code elimination is most effective when used in combination with constant propagation as one opens up new optimization opportunities for the other.

3.2.2 Constant Propagation

With this in mind we decided to implement constant propagation. To preform this optimization step we have implemented a slightly modified version of Sparse Conditional Constant propagation(SCC). Rather than on a on a static single assignment(SSA) form our SCC works with the control flow graph. The control flow graph represents the program's control flow as a set of nodes and edges, where each node corresponds to a basic block as described in section 1.1. Our implementation of SCC uses the control flow graph to identify expressions that are constant within a given basic block and save its value and last assignment instruction. This way we can ensure that every time the variable is changed to a unknown value inside a conditional branch, that its last assignment will still be written ensuring the correct value is in memory for all possible paths.

3.2.3 Instruction Selection

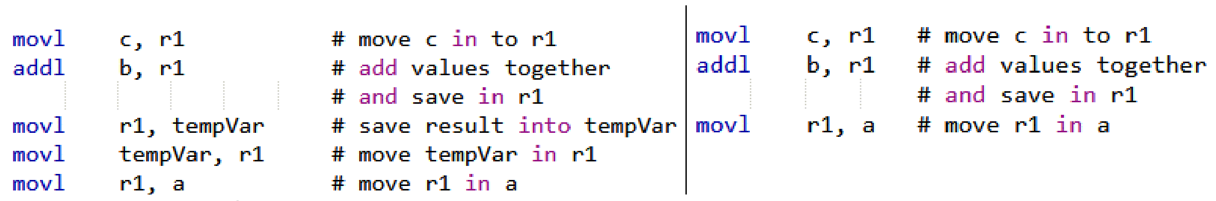
Both dead code elimination and constant propagation are IR level optimization In order to achieve results across multiple optimization phases we decided to implement code generation optimizations such as instruction selection register allocation and peephole optimization.

For instruction selection we implemented the optimum tiling algorithm as described in in chapter 2.1. However, there is still room for improvement in fully utilizing this algorithm in our compiler. Currently, it only supports optimization of basic operations. Further research and development is required to extend the implementation and its capabilities of this algorithm in our compiler.

3.2.4 Register Allocation

For register allocation we have implemented a version of the naïve Register Allocation algorithm as described in this article.[44] By default without any register allocation algorithm we have to store and load every intermediate result in memory as if they were a variable, this is done because when creating the CFG operations like ‘a = b + c;’ are split up in to multiple instruction, in this case “add()” and “store()”. As there is no guarantee of which register will be used to hold the result of the “add” operation we create a new temporary variable and then store that temporary variable in to ‘a’. However to prevent all intermediate results to be spilled while still keeping the algorithm simple, our version of the algorithm starts of by assuming that every intermediate result fits in a register and instead of every intermediate result having its own memory location we use peephole optimizations to store these in the desired location this way spilling is only required when the correct location can not be determined or when there are no available registers left.

The following is a simplified side by side comparison of the operation 'a = b + c;' with and without the algorithm:



```
movl    c, r1           # move c in to r1
addl    b, r1           # add values together
                        # and save in r1
movl    r1, tempVar     # save result into tempVar
movl    tempVar, r1     # move tempVar in r1
movl    r1, a           # move r1 in a
```

```
movl    c, r1    # move c in to r1
addl    b, r1    # add values together
                 # and save in r1
movl    r1, a    # move r1 in a
```

Figure 12 Comparison of expression a = b + c; with and without register allocation

As we can see from the above example this effectively saves us a load and store instruction for every operation like this.

3.2.5 Peephole Optimization

The peephole optimization technique as implemented in our compiler, is not based on any specific algorithm and our results as shown in chapter 4.2.1 indicate that this optimization step has the most room for improvement. In our peephole optimization step we examine our generated assembly code for useless operations as described in section 2.4.

4 Evaluation

In this section, we evaluate the implemented optimizations described in section 3.2. we compare these in terms of Compile time and Micro operations, no measurements of execution time are presented in this paper as most of our source files where too small and fast to present any meaningful data.

4.1 Compile time

We compared the time it took to assemble a range of source files to assembly and compared our results to GCC and TCC(Tiny C Compiler) to do this in GCC we used the flags “-Q -ftime-report” in combination with the -s flag to compile to assembly for a more detailed explanation about these compiler flags we will refer to [44]. For TCC we used their specialized "-bench" flag which prints some detailed information about the compilation process.

The source files we used can be found on the GitHub page of this graduation work[45] These include a Fibonacci sequence, a file to test dead code elimination, a single integer square root function, a file that consists multiple square root function but halfway there’s a constant assignment making the fist half unused this is in order to test dead code elimination and constant propagation trough functions, a basic math file that tests the compiler across many basic operations and expression such as basic operators, branches, function calls, conditionals, and while loops, and lastly the most complicated file which is a version of the susan principle described here [46] simplified to work with our compiler.

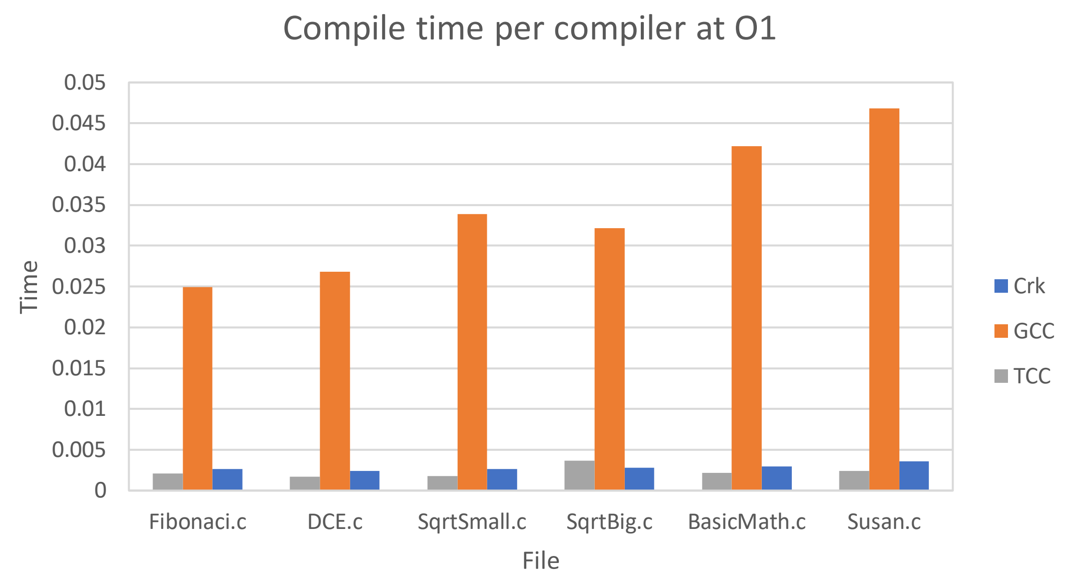


Figure 13 Compile time per compiler

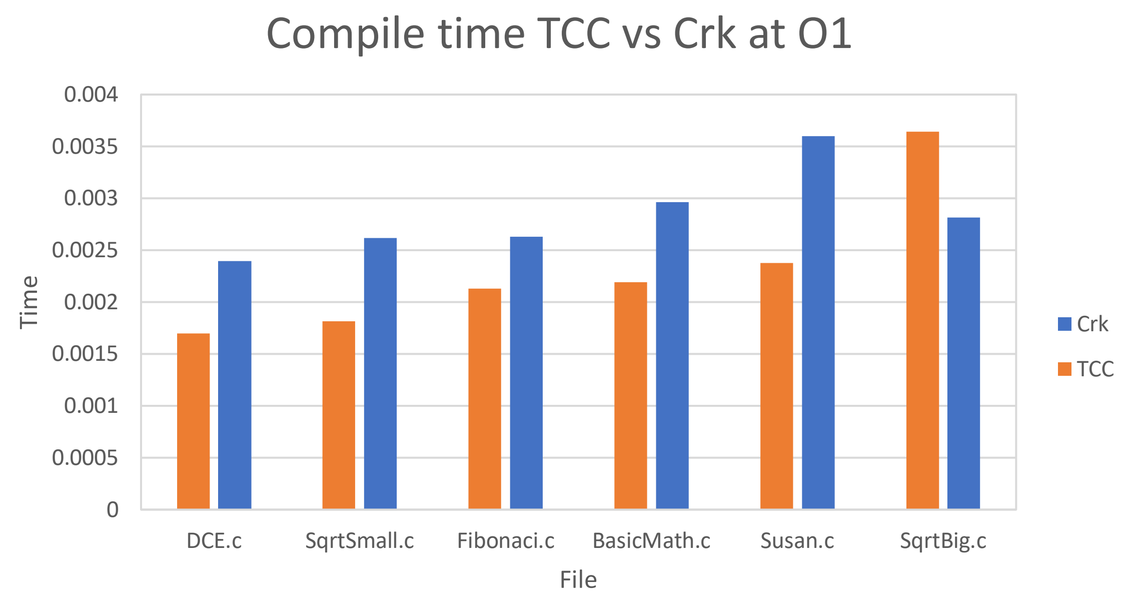


Figure 14

As you can see in Figure 13 GCC by far takes the longest time to compile, this is not very surprising as this is also the most sophisticated of the 3. Figure 14 shows a more detailed comparison of our compiler and TCC which is a better comparison as TCC is also a rather small compiler so it is expected to have compile times that are more on par with our compiler. We can conclude that overall TCC is faster than our compiler except for one file "SqrtBig" this is likely due to the high amount of function calls inside that file and is not surprising to take longer as most of our optimizations do not propagate trough functions and thus require less computation.

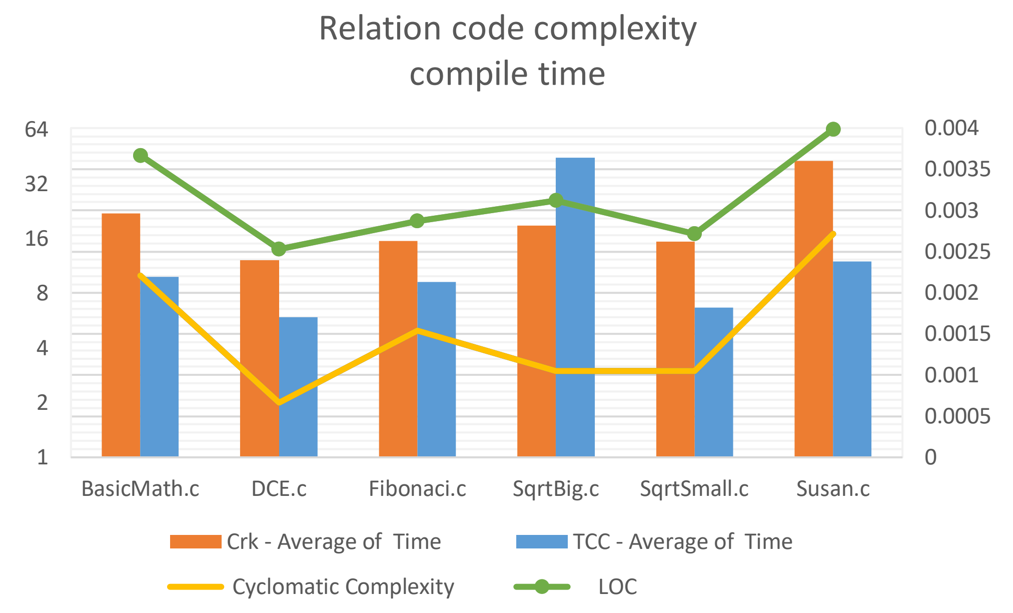


Figure 15 relationship compile time and code complexity in terms of Cyclomatic Complexity and Lines Of Code

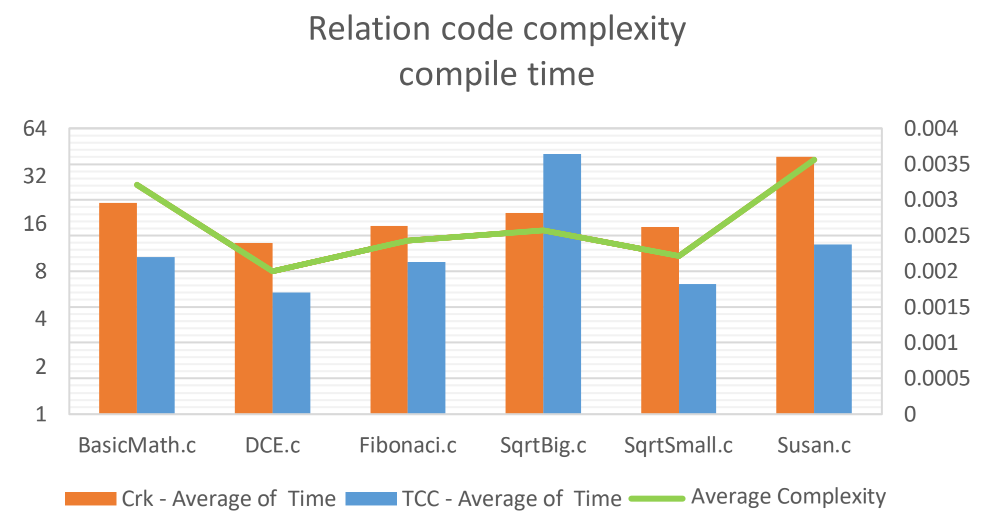


Figure 16 relationship compile time and average code complexity

In figure 15 and Figure 16 we compare each file's complexity to its compile time. We determine the complexity of each file in terms of Cyclomatic Complexity which is "*the amount of decision logic in a source code function*"[47] and LOC (Lines Of Code) we can see a clear correlation to the complexity of each file and its compile time, this correlation becomes even more apparent when we calculate the average complexity by assigning equal weights to both the Cyclomatic Complexity and LOC and combing these.

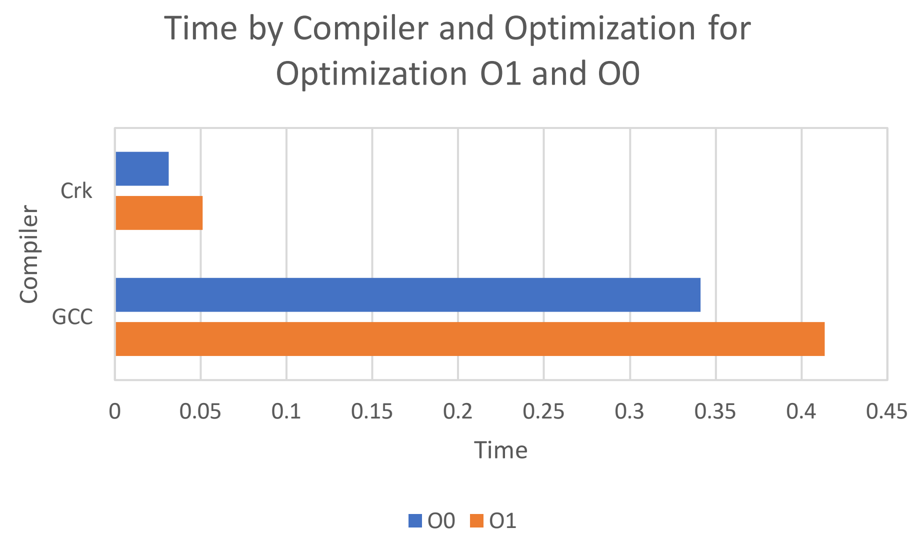


Figure 17

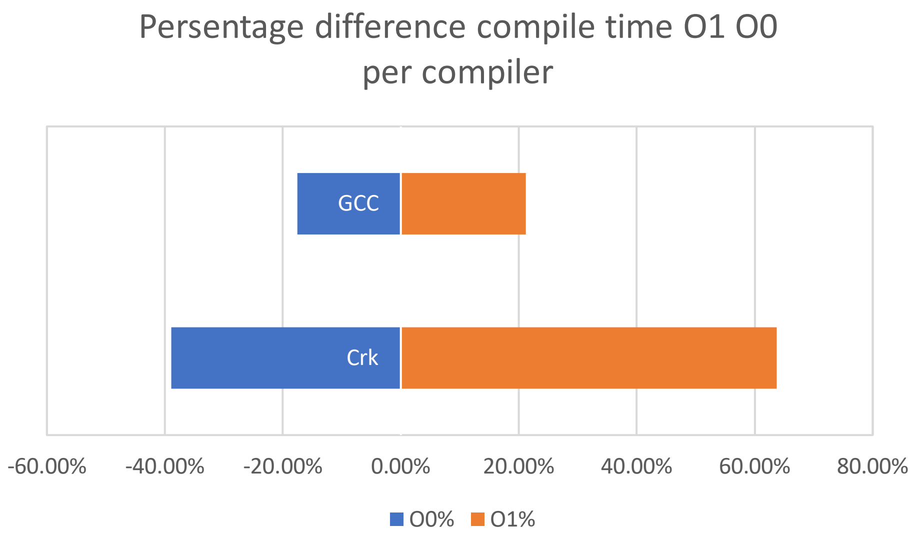


Figure 18

Figure 17 and 18 demonstrate the difference in compile time when compiling at different optimization levels. In Figure 17 it might look like compiling at different optimization levels in GCC has a relatively bigger impact on compile time than ours but when we look in terms of percentages as shown in Figure 18, we can conclude that difference in compile time between O0 and O1 in our compiler is greater than the difference in GCC. This is likely due to the fact that the 'baseline' (the amount of optimizations turned on at O0) is greater in GCC than ours as mentioned previously in chapter 3.2.

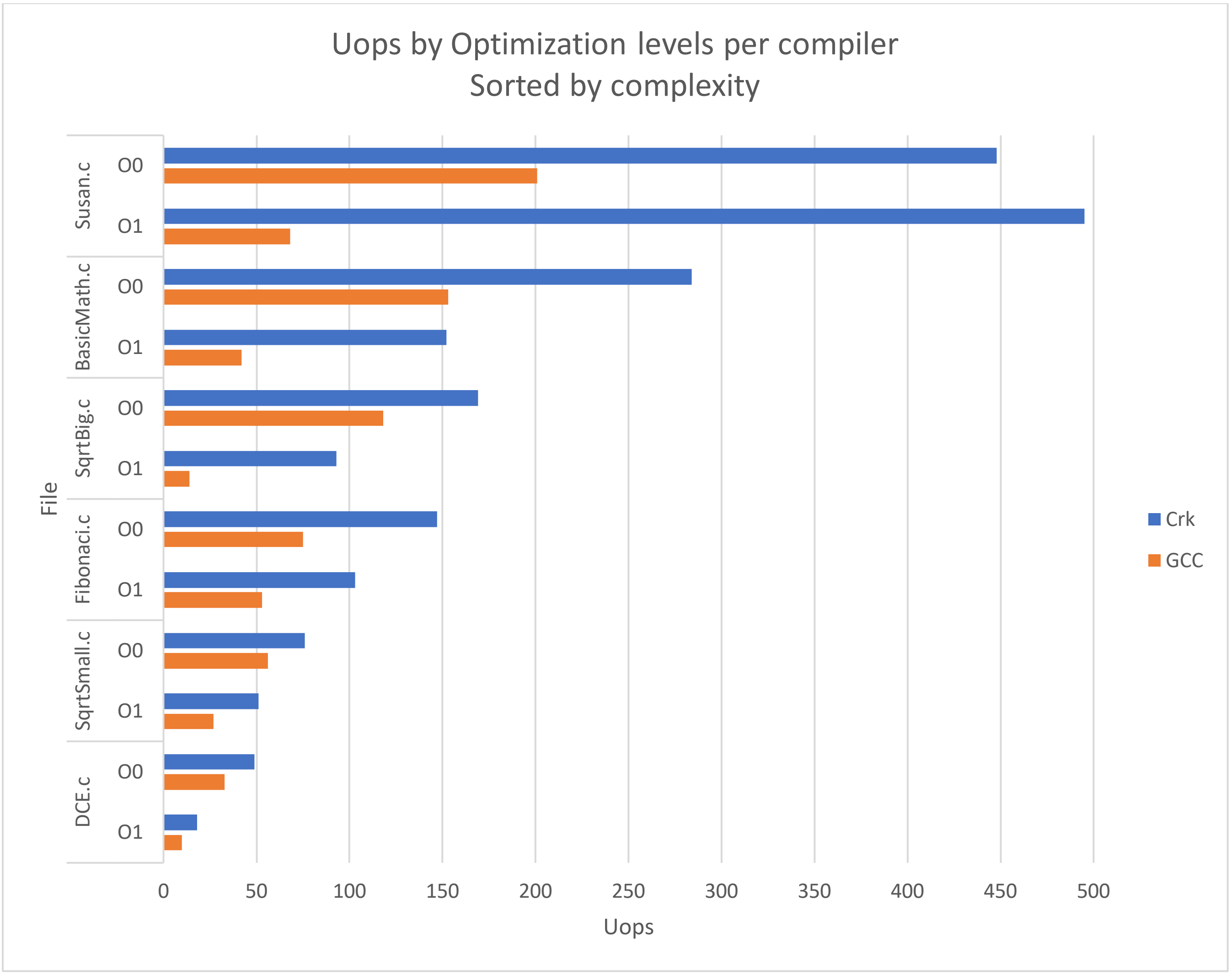


Figure 19

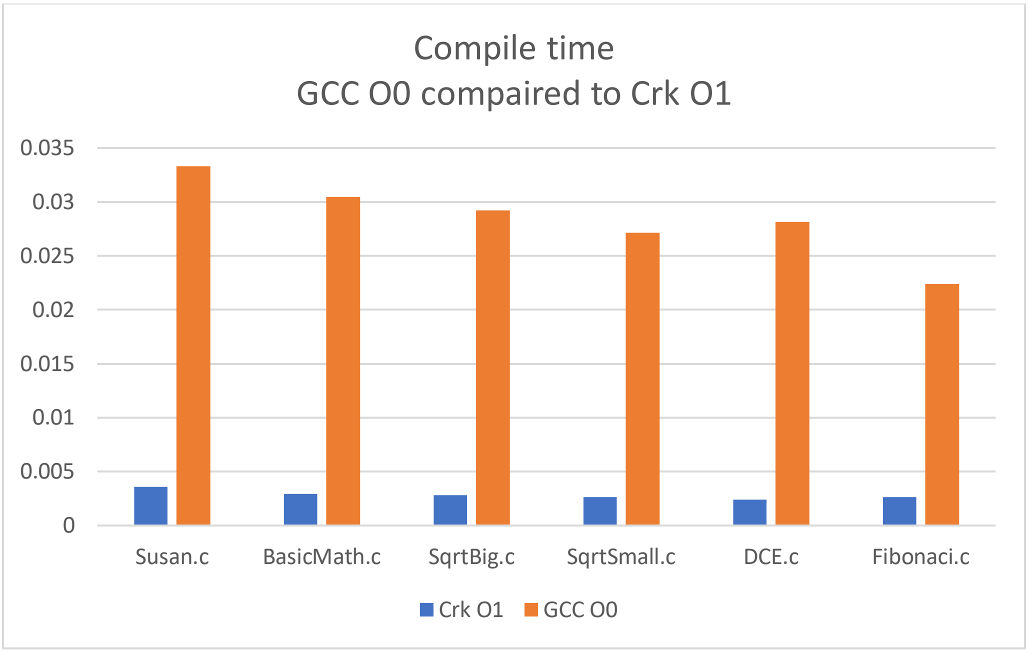


Figure 10

4.2 Micro Operations

Figure 19 shows us a comparison of the total micro-operation of each generated assembly file.

"*Execution of a program consists of sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter sub-cycles(example – fetch, indirect, execute, interrupt). The performance of each sub-cycle involves one or more shorter operations, that is,* micro-operations.*"*

These micro ops will be used as one way to compare the quality of the generated assembly files. We compared our compiler to GCC only, as we were unable to get the required data from TCC to conduct this research. When we look at Figure 19 we can immediately notice that GCC generates a lot less micro ops than our compiler, this is also to be expected from a compiler that has been worked on as a mass collaboration project that exists since1984[48] However when we look at the comparisons of the BasicMath, Sqrt and DCE file we can see that our compiler ran at O1 generates about as much and sometimes less micro operations than GCC where GCC and when we look at Figure 20 which compares the compiler time of GCC at O0 to our compiler at O1 we can see that our compiler still compiles a lot faster . We also notice that for the file Susan we generate more micro operations when compiling at O1 than at O0 this is not as intended and is further analysed in chapter 4.2.2

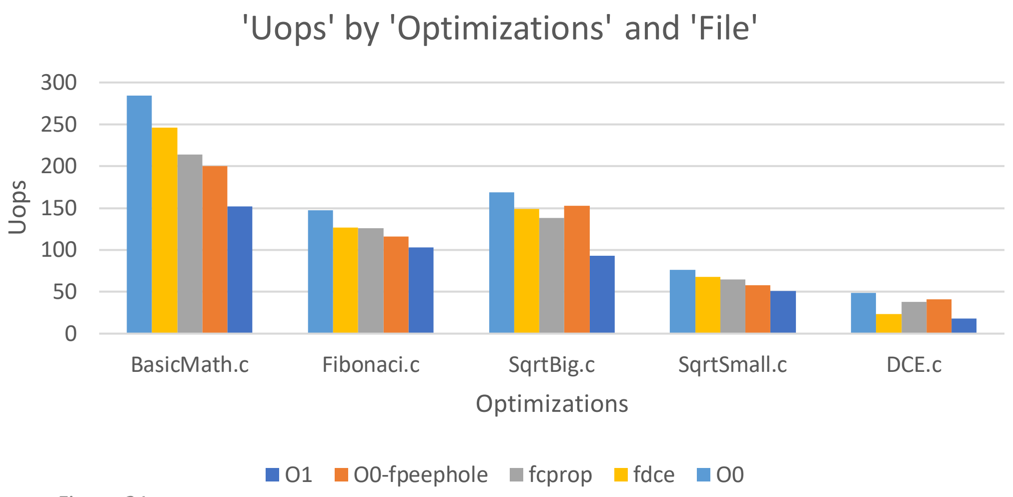


Figure 21

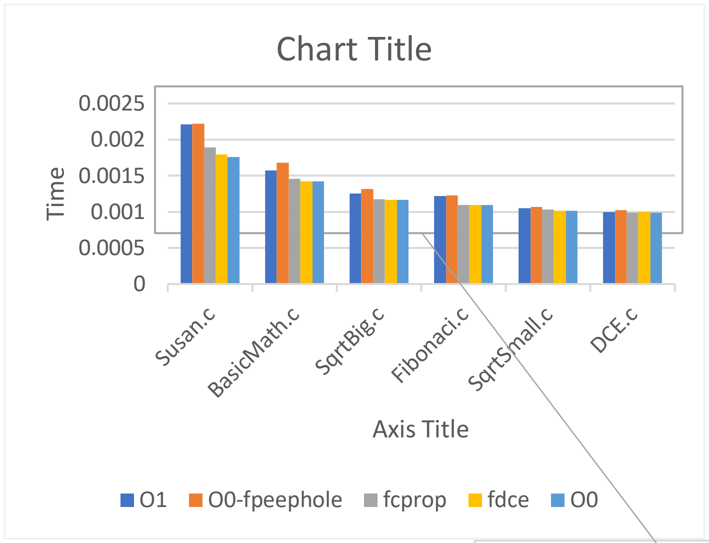


Figure 22

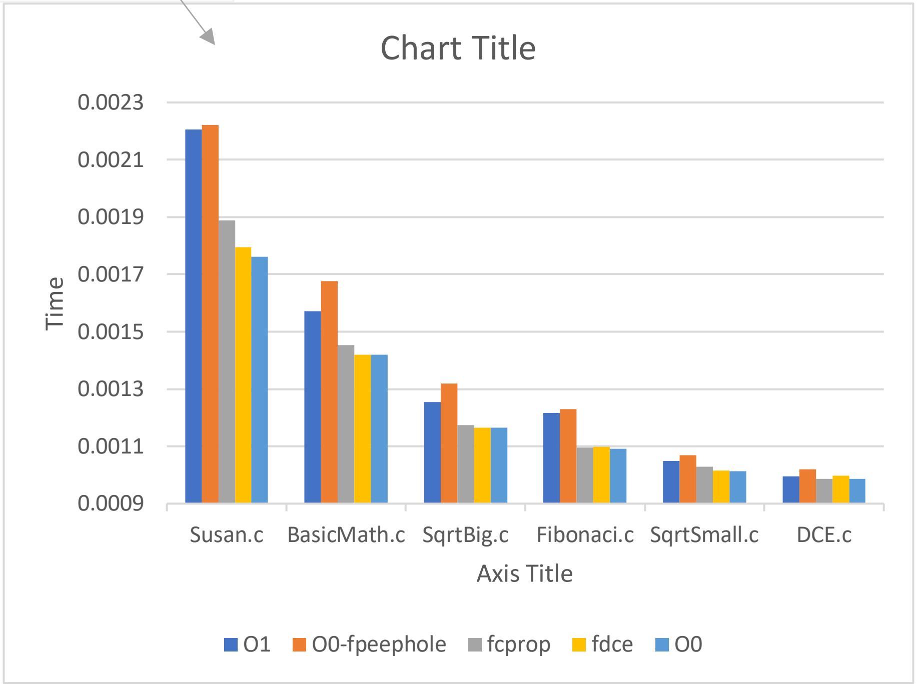


Figure 23

4.2.1 Specific Optimization Comparisons

When looking at the generated micro ops for each individual optimization we see about what we expected. Overall dead code elimination on its own has the least impact, except in case of the file specifically intended towards dead code elimination. As mentioned before dead code eliminations is often best used in combination with other IR-level optimizations therefore we will test this more in depth in the next section

When looking at the compile time impact of each optimizations in Figure 22 we notice not a lot difference in terms of compile time, this is because the compile time of each of these files is so small that we have to zoom in to get a better visualization of the impact of each of these. In Figure 23 we notice that peephole optimizations has a considerably large overhead compared to the other optimizations. This is likely due to a combination of multiple factors such as the lack of a concrete algorithm used for this optimizations and as Grune D mentioned "*Peephole optimization is inversely proportional to the quality of the code yielded by the code generation phase. A good code generator requires little peephole optimization, but a naive code generator can benefit greatly from a good peephole optimizer.*"[4, Ch. 7.6.1.3] This also explains why peephole optimization in multiple cases takes more time than when compiling at O1 which includes peephole optimizations.

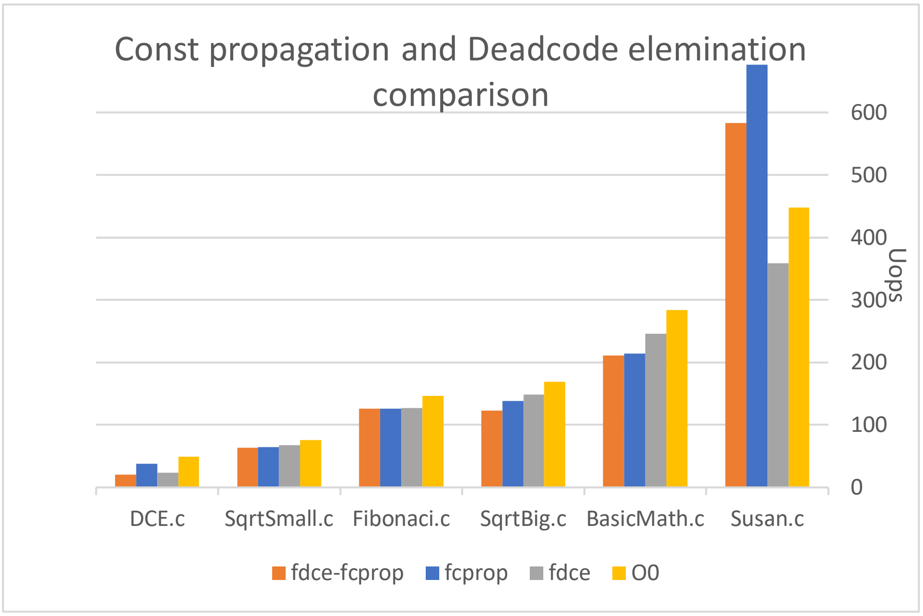


Figure 24

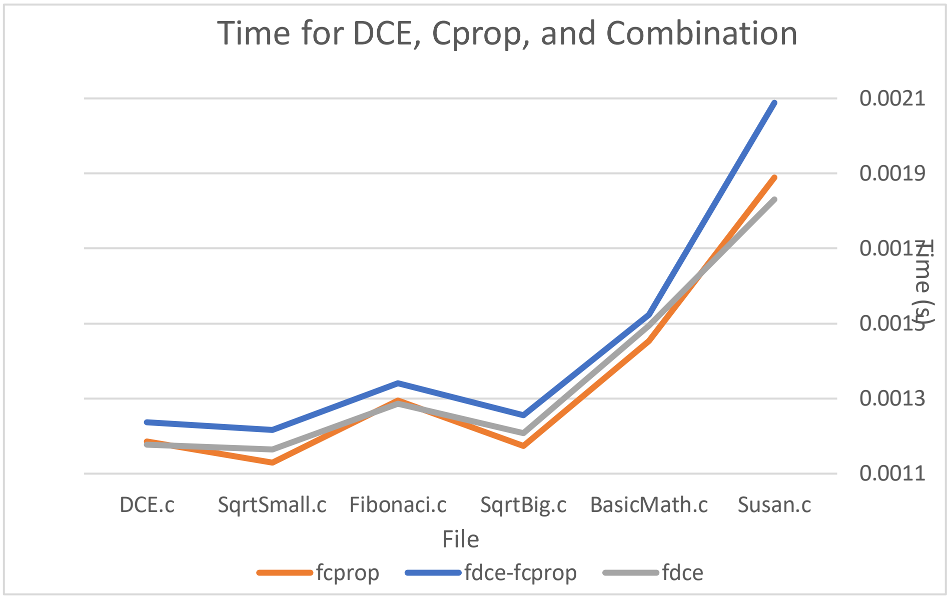


Figure 25

4.2.2 Combined IR level optimizations

In Figure 24 we can clearly see that dead code elimination in combination with constant propagation results in the best performance while in Figure 25 we can see that the impact on the compile time is rather low. Here we also notice that constant propagation has a negative impact when compiling the Susan program this is likely due to a bug in our implemented constant propagation algorithm. However finding the exact cause of the problem is something that requires further research.

5 Conclusion

In conclusion, our research has shown that GCC generates the least micro operations but has longer compile times, even at low optimization levels and in contrast, smaller compilers such as ours and TCC generally have faster compile times. Furthermore as discussed in chapter 4.2 of our paper, when comparing our compiler at high optimization levels to GCC at low optimization levels, our compiler was able to generate better results for smaller, less complicated files while remaining compile times several times faster than GCC. Besides that we have also been able to provide some data on the individual impact of certain optimizations and their combined benefit. Overall we can conclude that when choosing a compiler for smaller projects that it can be valuable to consider both performance and the compile time.

CONCLUSION & FURTHER WORK

We have examined the impact of intermediate representation level optimizations and code generation optimizations in a small compiler. We have concluded that a combination of dead code elimination and constant propagation can lead to improvements while having a relatively low impact on compile time compared to their individual implementation. We have also found that peephole optimization has the most room for improvement within our compiler and have confirmed their impact inversely proportional to the quality of the code.

While we have provided a better understanding into the design and development of compilers, there are several areas where further research could be performed. One area for future work is to expand the scope of the study to include more language features and benchmarks to evaluate the optimizations on a wider range of source files and measurements. Additionally, further research could be done on the implementation and optimization of peephole optimizations to improve its performance. Additional research should also be done to investigate the unexpected behaviour of constant propagation in the Susan file and lastly, additional optimization techniques could be implemented and evaluated to further improve and determine the performance impact in a small compiler.

Appendix A

ASM - x86

x86 (both 32- and 64-bit) has two alternative syntaxes available *Intel, and AT&T*.[49] Some assemblers can only work with one or the other like NASM, MASM, etc. uses Intel syntax exclusively, while a few can work with both GNU Assembler uses AT&T syntax by default but supports Intel syntax for x86 and x86_64.

	Intel	**AT&T**
Comments	;	#
Instructions	Untagged add	Tagged with operand sizes: addq
Registers	eax, ebx, etc.	%eax,%ebx, etc.
Immediates	0x100	$0x100
Indirect	[eax]	(%eax)
General indirect	[base + reg + reg * scale + displacement]	displacement(reg, reg, scale)

[50]

Besides the syntactic differences the main differences are rather superficial[50], [51] like making the sizes of the instruction operands explicit or implicit, AT&T makes them explicit, by appending suffixes to the instruction name, while Intel leaves them implicit. A good example of this is add[Intel] vs. addq[AT&T] (q meaning quad[52]) or reversed operands, for example:

> In AT&T syntax to *move ebp* in to *esp* you have to write **“mov %ebp, %esp”**
> The same in Intel syntax (MASM in this case) would look like **“mov esp, ebp”**

A.1 Stack

Pushing a value (not necessarily stored in a register) means writing it to the stack.
Popping means restoring whatever is on top of the stack into a register.
Those are basic instructions:

A.2 Functions

Whenever a function gets called that function is set up with a stack frame[53]. The stack frame stores all the local variables of that function.

Functions in assembly have a prologue and epilogue, typically these are a set of instructions that 'set up' the context for the function – prologue for when it gets called and epilogue cleans up when it returns.

A.2.1 Prologue

In AT&T syntax, a function prologue is a set of instructions that are executed at the beginning of a function to set up the stack and prepare for the execution of the function's code.[54], [55]

Here is an example of a function prologue in AT&T syntax:

```
.globl my_function
my_function:
    push %ebp      # Save the old base pointer
    mov %esp, %ebp # Set the base pointer to the current stack pointer
    sub $8, %esp   # Make room on the stack for local variables
```

The first line, `.globl my_function`, declares `'my_function'` as a global symbol, meaning it can be called from other files. The second line, `'my_function:'`, labels the start of the function.[56], [57]

The next three lines are the actual prologue instructions. The `'push'` instruction saves the old base pointer on the stack. The `'mov'` instruction sets the base pointer to the current stack pointer. Finally, the `'sub'` instruction makes room on the stack for local variables.

This prologue sets up the stack frame for the function, which will be used to access local variables and pass arguments to other functions. It also saves the old base pointer, which will be used to restore the previous stack frame when the function returns.

In AT&T syntax, the `'.type'` directive is used to indicate the type of a symbol. The syntax for the `'.type'` directive is `'.type symbol_name, @type'`.[58] `'@type'` can be specified as of the following:

- `@function`: Indicates the symbol as a function
- `@object`: Indicates a data object (such as a global variable)
- `@file`: Indicates a file name
- `@section`: Indicates a section name

For example, here is how you might use the `'.type'` directive with function declaration:

```
.globl my_function
.type my_function, @function
my_function:
    # function code goes here
```

The '.globl' directive declares 'my_function' as a global symbol, and the '.type' directive specifies that it is a function. This allows the linker to properly link calls to 'my_function' from other files.

You can use the '.type' directive to specify the type of a data object[58], like this:

```
.data
.globl my_variable
.type my_variable, @object
my_variable:
    .long 0
```

'@object' declares that this global variable called 'my_variable' is a data object. Often you will also see additional directives, such as '.globl' and '.size', which provide extra information about symbols in the object file. This information is later used by the linker to properly link the object file with other object files and create the final executable.

A.2.2 Function call

To call a function in assembly, you will need to use the 'call' instruction. The 'call' instruction causes a subroutine/function to be executed at the specified address. This address can be specified as a label in the code, or as an immediate value. Below is an example of how you can use the 'call' instruction to call a function:

```
call labelname
```

You can also specify the address of the function as an immediate value:

```
call 0x12345678
```

This will cause the processor to jump to the code at the label 'labelname' or memory address.[57], [59] As soon as the subroutine is finished, the processor will return to the instruction below the 'call' instruction.

It's also important to properly set up the stack and register values appropriately to prevent a SEGFAULT. This usually involves moving the necessary arguments in to registers, setting the base pointer to the current stack pointer followed by setting the stack pointer to point to the top of the stack, and optionally saving the values of any registers that the function may modify.

To properly set up the stack for a function call in assembly, you will need to follow a few steps:

1. Allocate space on the stack for the function's arguments and local variables. This is done by decrementing the stack pointer (`esp` on x86) by the amount of space needed(This is only required when passing more than 6 parameters, as for up to 6 parameters we can use registers[10]).
2. Push the function's arguments onto the stack; Traditionally, arguments were passed by *pushing them onto the stack*. This has been changed by the x86-64 Application Binary Interface[10] who sets the standard and now the first 6 arguments should be moved in to registers still in the reverse order, with the last argument being pushed first and the first argument being pushed last.[10]
3. Save the values of any registers that the function may modify. This is usually done by pushing the values of the registers onto the stack.
4. Set up a stack frame for the function. This involves setting the frame pointer (`ebp` on x86) to the current value of the stack pointer, and then decrementing the stack pointer by the size of the stack frame.

[59]

Here is an example of how you might set up the stack for a function call in x86 assembly:

```
# (Not required in this case)Allocate space on the stack for the function's
arguments and local variables
#sub esp, 4

# Pass the function's arguments in reverse order
movl arg2, %edi
movl arg1, %rsi

# (Optionally)Save the values of any registers that the function may modify
push %ebx

# Set up a stack frame for the function
mov %esp, %ebp
sub $4, %rsp
```

This code will allocate 4 bytes of space on the stack for the function's arguments and local variables, arguments 1 and 2 are saved in registers and the value of the '`ebx`' register can be optionally pushed on the stack to preserve its value. It will then set up a stack frame for the function by setting the frame pointer (`ebp`) to the current value of the stack pointer and decrementing the stack pointer by 4 bytes making room for the local variables and function parameters.

After setting up the stack, you can then use the `call` instruction to call the function. When the function returns, you will need to restore the values of any saved registers and deallocate the space on the stack in case anything was pushed.

BIBLIOGRAPHY

[1] "TCC : Tiny C Compiler." https://www.bellard.org/tcc/#speed (accessed Jan. 23, 2023).

[2] J. Cong, B. Liu, R. Prabhakar, and P. Zhang, "A study on the impact of compiler optimizations on high-level synthesis," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7760 LNCS, pp. 143–157, 2013, doi: 10.1007/978-3-642-37658-0_10/COVER.

[3] K. D. (Keith D. Cooper and L. Torczon, *Engineering a compiler*. Accessed: Jan. 06, 2023. [Online]. Available: http://www.sciencedirect.com:5070/book/9780128154120/engineering-a-compiler

[4] D. Grune, "Modern compiler design," p. 736, 2000.

[5] F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1970, doi: 10.1145/390013.808479.

[6] J. Yousefi, Y. Sedaghat, and M. Rezaee, *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2015. doi: 10.1109/ICCKE.2015.7365827.

[7] Jr. Clifford Noel Click, "Combining Analyses, Combining Optimizations."

[8] "Software Engineering | Control Flow Graph (CFG) - GeeksforGeeks." https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/

[9] "Control Flow Diagram in Software Engineering: Symbols & Example - Video & Lesson Transcript | Study.com." https://study.com/academy/lesson/control-flow-diagram-in-software-engineering-symbols-example.html (accessed Jan. 18, 2023).

[10] "Control Flow Graphs - Cornell University - lec25."

[11] "Constant Folding and Constant Propagation in Compiler Design." https://iq.opengenus.org/constant-folding-and-propagation/

[12] S. Boldo and G. Melquiond, "Computer arithmetic and formal proofs : verifying floating-point algorithms with the Coq system", [Online]. Available: https://dokumen.pub/computer-arithmetic-and-formal-proofs-verifying-floating-point-algorithms-with-the-coq-system-computer-engineering-1785481126-9781785481123.html

[13] "Constant Propagation." http://compileroptimizations.com/category/constant_propagation.htm

[14] C. Verbrugge, P. Co, and L. Hendren, "Generalized constant propagation a study in c," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1060, pp. 74–90, 1996, doi: 10.1007/3-540-61053-7_54.

[15] "Constant Folding." http://compileroptimizations.com/category/constant_folding.htm

[16] "Constant Folding and Constant Propagation in Compiler Design." https://iq.opengenus.org/constant-folding-and-propagation/

[17] M. N. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches."

[18] "CS 6120: Sparse Conditional Constant Propagation." https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/sccp/ (accessed Jan. 24, 2023).

[19] "Global constant propagation." https://iitd.github.io/col728/lec/global_constant_propagation.html

[20] "A Survey of Data Flow Analysis Techniques. 1981. | ArchivesSpace Public Interface." http://archives.library.rice.edu/repositories/2/archival_objects/211490 (accessed Jan. 06, 2023).

[21] S. S. Muchnick and N. D. Jones, "Program flow analysis : theory and applications," p. 418.

[22] "Halting problem - Wikipedia." https://en.wikipedia.org/wiki/Halting_problem (accessed Jan. 14, 2023).

[23] "Why can't dead code detection be fully solved by a compiler? - Stack Overflow." https://stackoverflow.com/questions/33266420/why-cant-dead-code-detection-be-fully-solved-by-a-compiler (accessed Jan. 14, 2023).

[24] "Software optimization resources. C++ and assembly. Windows, Linux, BSD, Mac OS X." https://agner.org/optimize/ (accessed Jan. 16, 2023).

[25] P. Thiemann and M. Keil, "Compiler Construction Instruction Selection," 2016.

[26] M. Lam, R. Sethi, J. Ullman, and A. Aho, "Compilers: Principles, Techniques, and Tools," p. 1038, 2007, [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Compilers+Principles,+Techniques,+&+Tools#0

[27] RICE university, "Instruction Scheduling."

[28] “Instruction scheduling - Wikipedia.” https://en.wikipedia.org/wiki/Instruction_scheduling (accessed Jan. 14, 2023).

[29] “Register allocation.” https://en.wikipedia.org/wiki/Register_allocation

[30] “What is Register Allocation? - Definition from Techopedia.” https://www.techopedia.com/definition/21664/register-allocation (accessed Jan. 14, 2023).

[31] “Processor register - Wikipedia.” https://en.wikipedia.org/wiki/Processor_register (accessed Jan. 18, 2023).

[32] “Register Allocations in Code Generation - GeeksforGeeks.” https://www.geeksforgeeks.org/register-allocations-in-code-generation/ (accessed Jan. 22, 2023).

[33] “REGISTER ALLOCATION.” https://pages.cs.wisc.edu/~horwitz/CS701-NOTES/5.REGISTER-ALLOCATION.html (accessed Jan. 18, 2023).

[34] “Register Allocation - Stanford University”, [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/17/Slides17.pdf

[35] Q. Liang, Y. Z. Wang, and Y. H. Zhang, “Resource virtualization model using hybrid-graph representation and converging algorithm for cloud computing,” *International Journal of Automation and Computing*, vol. 10, no. 6, pp. 597–606, Dec. 2013, doi: 10.1007/s11633-013-0758-1.

[36] “RL4ReAl: Reinforcement Learning for Register Allocation | DeepAI.” https://deepai.org/publication/rl4real-reinforcement-learning-for-register-allocation (accessed Jan. 18, 2023).

[37] “MLGO Framework Brings Machine Learning in Compiler Optimizations.” https://www.infoq.com/news/2022/07/MLGO-framework-machine-learning/ (accessed Jan. 18, 2023).

[38] S. Asghar, A. Amd, and K. Venkataramanan, “Deep Learning-based Hybrid Graph-Coloring Algorithm for Register Allocation Dibyendu Das Consultant.”

[39] “MLGO: A Machine Learning Framework for Compiler Optimization – Google AI Blog.” https://ai.googleblog.com/2022/07/mlgo-machine-learning-framework-for.html (accessed Jan. 18, 2023).

[40] M. Poletto, “Linear Scan Register Allocation.”

[41] S. Kananizadeh and K. Kononenko, “Improving on Linear Scan Register Allocation,” *International Journal of Automation and Computing*, vol. 15, no. 2, pp. 228–238, Apr. 2018, doi: 10.1007/S11633-017-1100-0/METRICS.

[42] O. Traub, G. Holloway, and M. D. Smith, “Quality and speed in linear-scan register allocation,” *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, pp. 142–151, 1997, doi: 10.1145/277650.277714.

[43] “Peephole Optimization in Compiler Design - Coding Ninjas CodeStudio.” https://www.codingninjas.com/codestudio/library/peephole-optimization (accessed Jan. 16, 2023).

[44] “Register Allocation Algorithms in Compiler Design - GeeksforGeeks.” https://www.geeksforgeeks.org/register-allocation-algorithms-in-compiler-design/ (accessed Jan. 22, 2023).

[45] “Slowlor1ss/Compiler.” https://github.com/Slowlor1ss/Compiler (accessed Jan. 22, 2023).

[46] “The SUSAN Principle for Feature Detection.” https://users.fmrib.ox.ac.uk/~steve/susan/susan/node2.html (accessed Jan. 22, 2023).

[47] A. H. Watson and T. J. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric ,” in *(NIST Special Publication 500-235)*, 1996.

[48] “GNU Project (Using the GNU Compiler Collection (GCC)).” https://gcc.gnu.org/onlinedocs/gcc/GNU-Project.html (accessed Jan. 23, 2023).

[49] R. Narayam, *Linux assemblers: A comparison of GAS and NASM*. 2007. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html

[50] “Intel vs. AT&T syntax.” https://staffwww.fullcoll.edu/aclifton/courses/cs241/syntax.html

[51] “AT&T Syntax versus Intel Syntax.” https://sourceware.org/binutils/docs-2.19/as/i386_002dSyntax.html#i386_002dSyntax

[52] “Learning to Read x86 Assembly Language - Pat Shaughnessy.” https://patshaughnessy.net/2016/11/26/learning-to-read-x86-assembly-language

[53] “x86 Assembly Crash Course - YouTube.” https://www.youtube.com/watch?v=75gBFiFtAb8
[54] “Function Prologue and Epilogue in C - Sanfoundry.” https://www.sanfoundry.com/c-tutorials-prologue-epilogue-terms-context-functions-program/
[55] “Function prologue - HandWiki.” https://handwiki.org/wiki/Function_prologue
[56]. “global symbol, .globl symbol.” https://web.mit.edu/rhel-doc/3/rhel-as-en-3/global.html
[57] “Symbols.” https://web.mit.edu/rhel-doc/3/rhel-as-en-3/symbols.html#LABELS
[58]. “type.” https://web.mit.edu/rhel-doc/3/rhel-as-en-3/type.html
[59] J. Rexford, “Assembly Language: Function Calls".”