NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2022-01

January, 2022

Number of Execution Analysis: Novel Relations to Facilitate Middle-End Instruction Scheduling

Drew Kersnar

Abstract

Automatic compiler optimizations are limited by how much information can be derived from a program with analysis. We introduce Number of Execution Analysis (NEA), a framework that statically compares the execution counts of basic blocks in a program. We apply this analysis technique to instruction scheduling optimizations, moving instructions globally between basic blocks that are guaranteed to execute the same number of times. We evaluate the robustness of NEA as well as the effectiveness of our global scheduling technique using the Parsec3.0 benchmark suite. We find that a scheduler utilizing NEA can easily make a significant number of global moves, enabling any optimization that benefits from moving instructions between basic blocks.

Keywords

Compilers, static analysis, optimization, instruction scheduling

Number of Execution Analysis: Novel Relations to Facilitate Middle-End Instruction Scheduling

Drew Kersnar
Department of Computer Science, Northwestern University
drewkersnar2021@u.northwestern.edu

ABSTRACT

Automatic compiler optimizations are limited by how much information can be derived from a program with analysis. We introduce Number of Execution Analysis (NEA), a framework that statically compares the execution counts of basic blocks in a program. We apply this analysis technique to instruction scheduling optimizations, moving instructions globally between basic blocks that are guaranteed to execute the same number of times. We evaluate the robustness of NEA as well as the effectiveness of our global scheduling technique using the Parsec3.0 benchmark suite. We find that a scheduler utilizing NEA can easily make a significant number of global moves, enabling any optimization that benefits from moving instructions between basic blocks.

1 INTRODUCTION

A compiler's primary objective is to translate code from programming language to machine code. However, this is only half the story, and modern compilers also have a second, arguably more important objective: automatic optimization.

Due to the eventual ending of Moore's Law [20], hardware advancements can no longer be relied on to meet ever-growing performance demands alone. Additionally, as programming languages become higher level, developers are further separated from the hardware on which they run code, and they cannot manually optimize code with low-level details in mind. The compiler can help on both fronts by automatically optimizing the programs it is tasked with translating. As it lowers the program from language to architecture, it can improve the performance of the machine code it generates. Compiler optimizations are pivotal to the computing stack.

However, automatic optimization is not easy. For a compiler to optimize a piece of code, it needs to prove that the transformations it wants to attempt are *guaranteed* not to break the semantics of the original code. Due to this conservatism, optimizations are bounded by how much can be proven about the code through analysis techniques. The more complex the transformation, the more analysis is needed to prove that the transformation is legal. A lack of useful analysis leads to compiler optimizations that are severely stunted compared to the theoretical optimal.

Over the years, various analysis frameworks have been created to collect and organize information to enable better transformation of code. The Dominator Tree and Post Dominator Tree, the PDG, Data Flow Analysis (DFA), and many more give the compiler knowledge it needs to transform and optimize the code safely [2] [11] [3].

To add to the compiler's toolkit, we introduce a novel analysis framework called Number of Execution Analysis, or NEA. NEA provides information about the relative execution counts of basic blocks and generates relations so that compilers can attempt more ambitious optimizations. Using NEA, we implement a scheduling engine that is capable of moving instructions across basic blocks in a program and apply this engine towards the use case of unlocking thread-level parallelism with HELIX [7].

This work is outlined as follows. Section 2 provides background and motivation for this work. Section 3 introduces Number of Execution Analysis. Section 4 discusses our application of NEA to the instruction scheduling optimization. Section 5 outlines the implementation of the concepts discussed in this paper. Section 6 discusses evaluation techniques and results. Section 7 discusses related work. Section 8 provides options for future work.

2 BACKGROUND AND MOTIVATION

This section provides a primer on basic compiler concepts used in this paper. First, it outlines a traditional compiler stack. Then, it defines some common compiler terminology. Finally, it introduces the *instruction scheduling* optimization and motivates a need for new analysis to enable scheduling.

2.1 Compiler Stack

Traditionally, compilers are implemented in three layers, referred to as the front-end, middle-end, and back-end. The front-end performs programming-language-specific optimizations and lowers code from the source language to an **Intermediate Representation**, or IR. The back-end performs architecture-specific optimizations and lowers code from the IR to the target instruction set architecture (ISA). The IR is designed to enable optimization, and the middle-end takes advantage of this to perform programming-language-and-architecture-*agnostic* optimizations. The advantage of this model is two-fold. First, a common IR allows N programming languages to be translated to M architectures without implementing $N * M$ different compilers. Instead, N different programming languages can be funneled into the IR by N

front-ends, and machine code can be generated for M different architectures by M back-ends. Second, a common IR allows middle-end optimizations to be implemented *once* and used with any programming language or architecture. However, this means the middle-end can only perform optimizations that do not require the context of the source language or the target architecture. This paper discusses analysis and optimization in the middle-end of the compiler.

2.2 Compiler Terminology and Structures

In order to properly analyze and transform code, compiler tool chains organize the memory representation of a program into specific control structures.

Basic Block: Instructions are grouped into structures called basic blocks. All instructions in a basic block run as a unit. Control flow can only enter at the top of the block, after which every instruction in the block is executed in order, concluding with a terminator such as a branch or return. Grouping instructions into basic blocks allows analysis and transformation to operate at a larger granularity than individual instructions.

Control Flow Graph: All basic blocks in a function can be organized into a structure called a Control Flow Graph, or CFG [2]. A CFG is a directed graph where every node in the graph represents a basic block, and every edge represents a block terminator that can transfer flow to one or more successor blocks. Upon execution of a function, control flow enters at a single "entry" block and flows through the graph until it reaches one or more "exit" blocks. Given this, an entire program can be represented hierarchically. A program is a collection of CFGs representing functions, each of which is made up of basic blocks, each of which contains a list of instructions.

Dominance: From the CFG, it is often useful to define relationships between basic blocks. A block, A, is said to **Dominate** a block, B, if every path from the entry of the CFG to B contains A. Conversely, a block, B, is said to **Post-Dominate** a block, A, if every path from A to an exit of the CFG goes through B. When A dominates B and B post-dominates A, A and B are **Control Flow Equivalent** [2].

Program Dependence Graph: The Program Dependence Graph, or the PDG, explicitly defines dependences between instructions [11]. A **Data Dependence** signifies that two instructions write or read to the same location. A **Control Dependence** signifies that the execution of an instruction is dependent on the result of another; for example, instructions in the body of an "if" statement are control dependent on the condition.

2.3 Instruction Scheduling

Instruction Scheduling is a compiler term with varying definitions. It primarally refers to an optimization found in many compiler back-ends that targets modern ISAs and out-of-order processors. In particular, this optimization reorders instructions (barring dependences) to exploit instruction-level parallelism found in the processor pipeline [1] [10]. Reordering instructions can produce an instruction stream that better fits pipelining while reducing stalls in the core due to memory accesses or cache misses.

In this paper, we use a broader, more general, definition of *instruction scheduling*. We define *instruction scheduling* as the reordering of instructions to satisfy an objective function without causing any changes to program semantics. This broad definition a) decouples the concept of instruction scheduling from the tight back-end-processor design space, b) allows instruction scheduling to extend to any section of the compiler stack that is capable of performing code motion, and c) offers the option of configuring the objective function beyond optimizing the use of processor resources.

Broadening the definition also highlights how constrained the design space and objective functions are in traditional compiler implementations. In this paper, we use this definition to expand instruction scheduling into the middle-end (taking advantage of the powerful analysis techniques the middle-end offers), explore configurable objective functions (like thread-level parallelism), and design general compiler relations, analysis techniques, and scheduling mechanisms to facilitate new optimizations.

2.4 Maintaining Dependences while Scheduling

When performing instruction scheduling, a given schedule is legal if it does not break the semantics of the original program. Whether a given move maintains semantics is very difficult to prove, so instruction schedulers typically restrict the set of possible moves to a provable subset.

One traditional way to ensure moves are legal is to check that neither *data* nor *control* dependences are broken.

Data Dependence: Instructions can be both consumers and producers of data. A data-producing instruction cannot be moved after its corresponding consumer, and a data-consuming instruction cannot be moved before its producer. In either case the correct data would not be available when the consumer is run.

Control Dependence: While data dependences restrict the *order* that instructions are executed, control dependences restrict *whether or not* a given instruction is executed, and indirectly, *how many times* that instruction is executed. At first glance, one might assume that if no data dependences are broken, it does not matter if an instruction is executed greater

or fewer times. However, some instructions might have *side effects* that can break the semantics of the code if they execute an unexpected number of times [1]. For example, a function call to `printf` might not have any data dependences, but if it is moved outside the body of an "if" statement, the behavior of a program may change.

Following this data and control dependence framework becomes increasingly difficult as the scope of an attempted schedule increases, ranging from **Local Code Scheduling** to **Global Code Scheduling**.

Local Code Scheduling, or scheduling within the scope of a basic block, is the easiest and most widely adopted method. Every instruction in a basic block is guaranteed to run if the block is executed, so they all share the same control dependences and only data dependences need to be considered.

Global Code Scheduling, however, is more complicated because a scheduler must avoid invalidating control dependences. Conservatively, most global moves would be invalidated by completely maintaining control dependences.

As explained, if both data and control dependences are maintained, a move is legal. However, the converse is not necessarily true; if a move is legal, this does not always mean data and control dependences are maintained. In other words, while data and control dependences are a useful framework to ensure move validity, they do not define the set of all possible legal moves.

Because of this, a lot of global code scheduling techniques *relax* the control dependence assertion. For example, loop invariant code motion breaks control dependences by hoisting loop invariants to the pre-header of a loop. Other methods focus on moving instructions without side effects upstream to dominating blocks [1]. Other optimizations, like loop fusion and distribution, make ad-hoc schedules that maintain semantics in specific ways.

Given the difficulty of maintaining program semantics while making aggressive scheduling optimizations, there is a need for new analysis to enable more powerful and general scheduling.

3 NUMBER OF EXECUTION ANALYSIS

Number of Execution Analysis (NEA) is a novel compiler analysis framework that can be used to compare the number of times basic blocks are guaranteed to execute in a program *relative* to other basic blocks. Notably, NEA is not trying to determine if a given basic block executes a constant number of times, as that information is almost always unknown at compile time.

Using NEA, we can derive a few different relations between blocks:

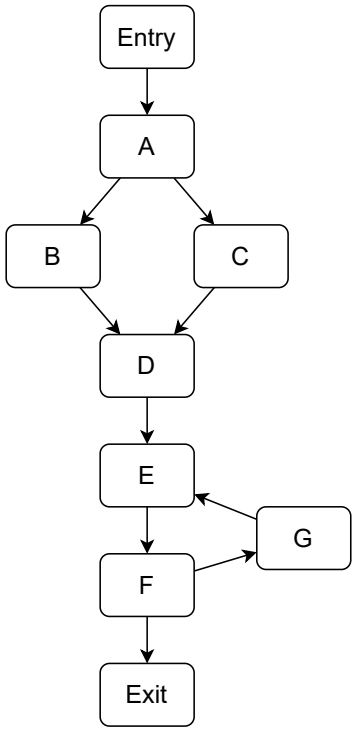

Figure 1: Example CFG

- Two blocks are Number of Execution Equivalent (NEE) if they are guaranteed to execute the same number of times.
- Two blocks are Number of Execution Less Than or Equal (NELE) if one of them is guaranteed to execute a lesser or equal number of times than the other.
- Similarly, we can derive other traditional comparison operators: NEGE, NEL, and NEG.

Figure 1 shows an example CFG. Figure 2 shows the NEA relations that might be derived from that CFG, represented by standard comparison operators. Blocks A and D are intuitively NEE, as every time A is executed, execution flow is guaranteed to reach D. Additionally, E and F are intuitively NEE, as each iteration of the loop will execute both one additional time. Finally, G is intuitively NEL to E and E is NEG to G, as G will execute one fewer time than E when the back edge is not taken upon loop exit. For a simple example like this, we can pick out these relations with ease.

For a compiler to be able to derive these relations on its own, as well as relations for much more complex CFGs, we have created an algorithm to implement NEA called NE Flow. NE Flow analyzes the CFG and other information from the code to solve basic block execution count variables in terms of each other.

The output of NE Flow is a mathematical expression for each basic block, which we call an NE Expression. A basic

	A	B	C	D	E	F	G
A		>=	>=	==	<=	<=	?
B	<=		?	<=	<=	<=	?
C	<=	?		<=	<=	<=	?
D	==	>=	>=		<=	<=	?
E	>=	>=	>=	>=		==	>
F	>=	>=	>=	>=	==		>
G	?	?	?	?	<	<	

Figure 2: Expected NEA relations

block's NE Expression represents the number of times that block will execute. These NE Expressions can be trivially compared to the NE Expressions of other basic blocks to derive any of the NEA relations.

3.1 NE Flow Algorithm

The NE Flow algorithm is broken into two main steps, which we summarize here:

(1) Find the Number of Executions (NE) for each edge in the CFG relative to other edges.
   - Give each edge in the system a variable, representing the number of times the edge will execute at runtime.
   - Define relationships between edges as equations in a linear system.
   - Solve the linear system, generating a mathematical expression for each edge in the CFG, which we call an NE Expression.

(2) Use the NE Expressions generated for each edge to find an NE Expression for each basic block.
   - Sum the NEs of all incoming edges to get the NE Expression for a given block.

By solving the system of equations, we reduce the output NE Expressions to the minimal number of independent variables, allowing for comparisons to be made across blocks.

Next, we dive into this algorithm with more detail using a concrete example.

3.1.1 Example NE Flow. Figure 1 shows the example CFG the algorithm will target. Figure 2 shows some relations we intuitively expect this algorithm to be able to derive.

Starting with Step 1, we define a NE variable for each edge in the CFG. Each variable will represent the number of times that edge will execute in one run of the function. Figure 3 labels our CFG with these edge variables, from $e1$ to $e10$.

Next, we define equations between edges using the following framework. Given that execution flow must start at the entry point and follow the graph until it reaches the exit point, a given execution path that enters a block must flow out of it. Thus, the total execution flow into a block must

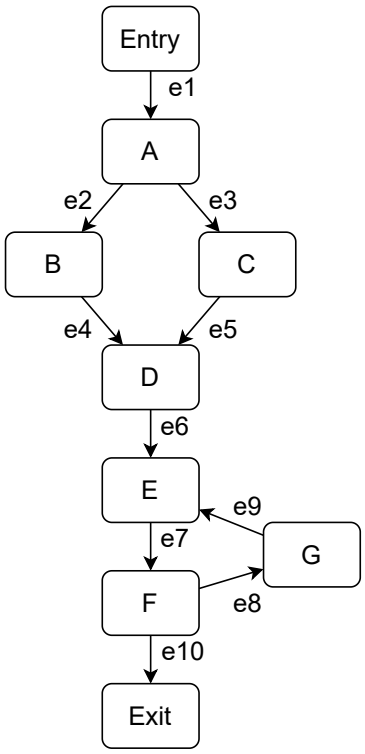

Figure 3: CFG with Edge Annotations

***Outgoing* = *Incoming* Equations**
e2 + e3 = e1 (A)
e4 = e2 (B)
e5 = e3 (C)
e6 = e4 + e5 (D)
e7 = e6 + e9 (E)
e10 + e8 = e7 (F)
e9 = e8 (G)

Extra Equations
e1 = 1

Figure 4: Edge Equations

equal the total flow out of a block. Therefore, for each block, *the sum of the NEs for each incoming edge must equal the sum of the NEs for each outgoing edge.*

With this, we can derive equations between edges in our example, as shown in Figure 4. There is one equation generated for every basic block. Additionally, since these NE variables are all in the context of one function run and every function is guaranteed to have exactly one entry point, we create the extra equation "$e1 = 1$" (details in Section 3.3).

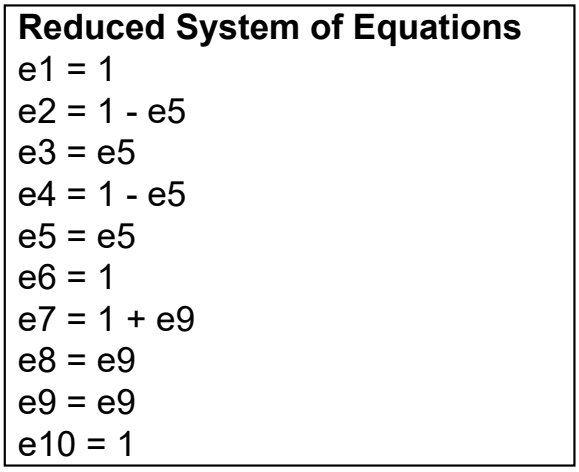

Figure 5: Solved System

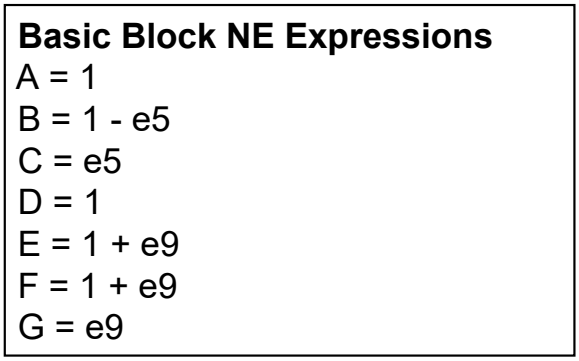

Figure 6: NE Flow Result: Mapping between Basic Blocks and reduced NE Expressions

These equations represent relationships between the execution count of the edges in our program, but they are not particularly useful in their current state. The next step of NE Flow is to solve the system of equations, reducing it to as few independent variables as possible. Figure 5 shows the solved system of equations for our example, containing only two remaining independent variables.

Having generated reduced NE Expressions for each edge in the CFG, we proceed to Step 2, where we generate NE Expressions for each basic block. When an incoming edge of a block is taken, that block is executed; *the NE of a given block is equal to the sum of the NEs of each incoming edge.*

In our example, adding together the NE Expressions for the incoming edges of each block results in the equations shown in Figure 6. We now have the desired output of NE Flow: An NE Expression for each basic block in terms of the other basic blocks in the CFG.

3.1.2 NE Flow Complexity. Although the system formed by NE Flow can be solved using standard methods such as Gaussian Elimination, which would have a complexity of $O(n^3)$, the equations themselves possess properties that we can exploit to speed up computation. In particular, the equations we generate exhibit the following properties:

(1) The outgoing flow of each equation is limited to two variables. This property is derived from CFGs themselves; every basic block is guaranteed to have, at most, two outgoing edges.
(2) Unlike outgoing edges from a basic block, the number of incoming edges to a basic block is theoretically unbounded. However, this typically occurs with loop nests, and the number of back edges in a loop nest is typically small. As a result, in the *common case*, there will usually be a small number of incoming edges for each basic block in the CFG.

Using these two observations, we conclude that the system in NE Flow is likely a sparse matrix. Consequently, we can utilize a reordering algorithm for sparse matrices to solve the system, leading to a better worst-case complexity for the NE Flow algorithm.

Regardless of the validity of this hypothesis, $O(n^3)$ is still practical, as many compiler optimizations have larger worst-case complexities. Additionally, NE Flow only needs to run once, and the results of the analysis can be reused multiple times, and do not need to be recomputed unless changes are made to the CFG.

3.1.3 Deriving NEA Relations. Using the output of NE Flow, deriving the desired relations is trivial and can be approached in a couple of different ways.

First, querying two specific blocks to compare their execution counts using NE Expressions is a trivial operation.

From this, we can generate a comprehensive structure called an NEE Graph, where nodes in the graph represent blocks and an edge between two nodes represents a known NEE relation. Figure 7 shows the NEE Graph generated from our example. This graph can be queried at any time to get sets of NEE pairs or all NEE pairs associated with a given block. Similar structures can be generated for the other NEA relations, all of which will instead be directed graphs. These graphs can be generated by doing a pairwise comparison between all basic blocks in a function.

3.2 Conservatism of NEA

Notably, NEA is a *conservative* analysis. This means that if it cannot derive anything concrete between two blocks in a function, it will not draw any relations between those blocks, signifying that they *may or may not* hold any of the NEA relations. For example, the lack of an edge in the NEE Graph does *not* rule out the possibility of two blocks being NEE. Optimizations that use NEA must keep this property in mind and only rely on relations that exist, not the lack of a relation, when proving the legality of a transformation.

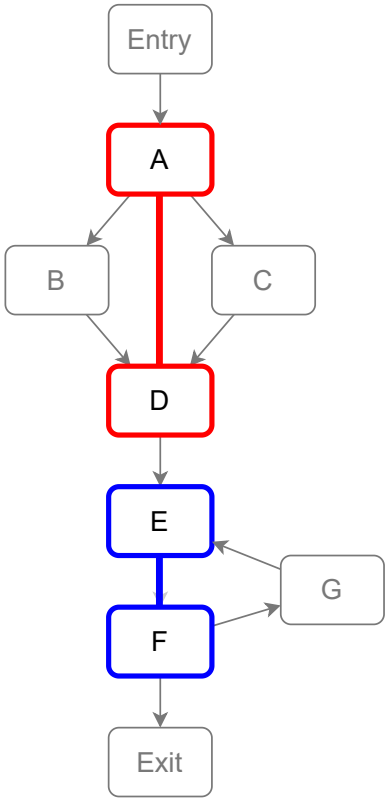

Figure 7: NEE Graph overlaid onto CFG

3.3 Extra Equations

The equations generated in our example Figure 4 are correct, but not necessarily exhaustive. The more information the compiler can derive about the execution count of edges, the stronger NE Flow becomes. For example, if the number of times the loop executes is constant and can be derived at compile time, an equation like "$e8 = 3$" may be appended to the system. If there were a function with two loops, A and B, and the compiler determines that both loops iterate X number of times, the equations "$backedge_A = X$" and "$backedge_B = X$" may be appended to the system.

In these cases, the extra information encoded in the system only helps to improve the results of NE Flow, allowing it to find more relations between blocks that otherwise (conservatively) would not find.

In this way, NEA is designed to be extensible as analysis techniques improve. NEA is conservative, but as more powerful analysis techniques generate more equations for the NE Flow algorithm, the generated relation graphs will approach perfection.

4 NEE SCHEDULER

We apply Number of Execution Equivalence to the problem of instruction scheduling. The NEE relation ensures that instructions, even those with side effects, can be moved across basic blocks safely. By ensuring that each moved instruction will run exactly the same number of times as it did pre-move, we can make more aggressive schedules that do not invalidate program semantics *while completely ignoring control dependences*. Thus, NEE enables construction of a simple global code scheduling engine in the middle-end, which we call the NEE Scheduler.

The NEE Scheduler is designed around the principle that moving an instruction is valid and will not break program semantics if the following conditions are met:

(1) The source block and the destination block are NEE.
(2) There are no data dependences "in between" the source and the destination.

By enforcing these constraints on every attempted move, the scheduler ensures that program semantics are maintained while allowing select inter-basic-block moves.

4.1 In Between Analysis

Notably, the NEE Scheduler still needs to check data dependences when moving an instruction. It can utilize the PDG to find all data dependences in the function. A move will only be blocked if any of those dependences would be invalidated by the move.

In other words, if there are any data dependences of the instruction "in between" the source and destination of a move, the move is invalid.

We define "in between" as follows: An instruction C is "in between" instructions A and B in either of the following cases:

(1) There exists a path from A to B that contains C.
(2) There exists a path from B to itself (not passing through A) that contains C.

Algorithms 1, 2, 3, and 4 shows the In Between Analysis algorithm, which finds all blocks in all paths from the source to the destination or vice versa depending on the direction of the attempted move.

Algorithm 1 In Between Analysis

```
1: procedure GetInBetweenBasicBlocks(A, B)
2:     return GetBlocksAToB() + GetBlocksBToB()
```

4.1.1 Description of In Between Analysis. In Between Analysis's goal is to generate the set of basic blocks that could possibly execute "in between" a run of the first block (A) and a run of the second block (B). It does this by recursively finding every block in every possible path from A to B (Algorithm 2), and every cycle from B to itself (Algorithm 3), with a DFS-style search. However, it does not actually traverse every possible path. The goal of the algorithm is not to find ordered paths, but simply the set of basic blocks that make

Algorithm 2 In Between Analysis - A to B

```
1:  procedure GetBlocksAToB(A, B)
2:      visited, currPath, inBetweenBasicBlocks ⇐ {}
3:      allPaths, predNodes ⇐ {{}}
4:      FindAllPathsAToB(
5:          A, B, visited, currPath, allPaths, predNodes)
6:      inBetweenBasicBlocks ⇐ {}
7:      for path in allPaths do
8:          for pathBlock in path do
9:              inBetweenBasicBlocks.insert(pathBlock)
10:     CollectAllPredNodes(inBetweenBasicBlocks, predNodes)
11:     if !predNodes[A].contains(A) then
12:         inBetweenBasicBlocks.remove(A)
13:     return inBetweenBasicBlocks
14:
15: procedure FindAllPathsAToB
16:     (currNode, B, visited, currPath, allPaths, predNodes)
17:     if currNode == B then
18:         allPaths.push(currPath)
19:         return
20:     visited[currNode] ⇐ true
21:     currPath.push(currNode)
22:     for succ in successors(currNode) do
23:         if !visited[succ] then
24:             FindAllPathsAToB(
25:                 succ, B, visited, currPath, allPaths, predNodes)
26:         else
27:             for predNode in currPath do
28:                 predNodes[succ].insert(predNode)
29:     currPath.pop()
```

Algorithm 3 In Between Analysis - B to B

```
1:  procedure GetBlocksBToB(A, B)
2:      visited, currPath, inBetweenBasicBlocks ⇐ {}
3:      allPaths, predNodes ⇐ {{}}
4:      FindAllPathsBToB(
5:          B, A, B, visited, currPath, allPaths, predNodes)
6:      inBetweenBasicBlocks ⇐ {}
7:      for path in allPaths do
8:          for pathBlock in path do
9:              inBetweenBasicBlocks.insert(pathBlock)
10:     CollectAllPredNodes(inBetweenBasicBlocks, predNodes)
11:     return inBetweenBasicBlocks
12:
13: procedure FindAllPathsBToB
14:     (currNode, A, B, visited, currPath, allPaths, predNodes)
15:     if currNode == A then
16:         return
17:     visited[currNode] ⇐ true
18:     currPath.push(currNode)
19:     for succ in successors(currNode) do
20:         if succ == B then
21:             allPaths.push(currPath)
22:         else if !visited[succ] then
23:             FindAllPathsBToB(
24:                 succ, A, B, visited, currPath, allPaths, predNodes)
25:         else
26:             for predNode in currPath do
27:                 predNodes[succ].insert(predNode)
28:     currPath.pop()
```

Algorithm 4 Collect All Predecessor Nodes

```
1:  procedure CollectAllPredNodes
2:      (inBetweenBasicBlocks, predNodes)
3:      while inBetweenBasicBlocks has not converged do
4:          for node in inBetweenBasicBlocks do
5:              for predNode in predNodes[node] do
6:                  inBetweenBasicBlocks.insert(predNode)
```

up all the paths. The algorithm utilizes this fact to only visit each node only once, greatly reducing the time complexity of the search. Upon reaching a visited node, N, the algorithm caches the current path in N's "*predNodes*" set, representing blocks in alternative paths that can reach N from A. If N ends up in a path that leads to B, all of N's cached predecessors are added to the result, as well as N's predecessors' predecessors, and so on. This process of collecting predecessors and adding them to the result is detailed in Algorithm 4.

4.1.2 Edge Cases. There are some edge cases worth mentioning, which are encoded in the algorithms. These edge cases result in slight differences between the two DFS algorithms (2 and 3).

(1) A is considered "in between" A and B if there is a path from A to B that includes another visit to A (caused by a cycle). Therefore, A is only "in between" if it is in its own predecessor nodes set, implying there is a self-cycle on the way to B.

(2) Cycles from B to itself are only "in between" if they do *not* pass through A, because data dependences before A would not be affected by a move.

4.1.3 Constraints of In Between Analysis. The algorithm, and therefore the entire scheduler, relies on the source and destination being "ordered" to identify a block from which to start searching for paths. To achieve this constraint, we also conservatively assert that the source and destination blocks are control flow equivalent in either order. The basic block that dominates the other is the "upstream" block and will be the origin of the paths derived in In Between Analysis. This constraint also allows for specific placement within the destination block to be trivial: A *downstream* move should

result in the moved instruction ending up directly before the destination instruction, and an *upstream* move should result in the moved instruction ending up directly after the destination instruction.

4.1.4 *In Between Analysis Proof of Complexity.*

Proof. This algorithm maps to two depth first searches, with a time complexity of $O(V + E)$. □

4.1.5 *In Between Analysis Proof of Correctness.*

Definition: a block C is said to be "in between" A and B if there exists a path from A to B that contains C.

Claim 1: If the algorithm outputs a block C, it is "in between" A and B.

Proof.

- Case 1: C is added to the output as part of a direct traversal from A to B. The algorithm traversed from A to B and found C on the way. Therefore, there exists a path from A to B that contains C.
- Case 2: C is added to the output as part of a direct traversal from B to B. The algorithm traversed from B to B without passing through A and found C on the way. Therefore, there exists a path from B to B that contains C.
- Case 3: C is added to the output as part of a $predNode$ set for a node D, which is part of a direct traversal from A to B. If C is part of D's predecessor node set, there exists a path from A to D that contains C. Additionally, because D was visited in a direct traversal from A to B, there is a path from D to B. Therefore, there is a path from A to C to D to B, and so C exists "in between" A and B.
- Case 4: C is added to the output as part of a $predNode$ set for a node D, which is part of a direct traversal from B to B. If C is part of D's predecessor node set, there exists a path from B to D that contains C. Additionally, because D was visited in a direct traversal from B to B, there is a path from D to B. Therefore, there is a path (that doesn't include A) from B to C to D to B, and so C exists "in between" A and B.

□

Claim 2: If the algorithm *does not* output a block C, it is *not* "in between" A and B.

Proof.

Proof by contradiction. Assume the algorithm *does not* output a block, C, that *is* "in between" A and B. Given the definition of "in between", this means that there is a path from A to B or from B to B that includes C. This is a contradiction, as the algorithm traverses all paths from A to B and B to B, collecting all blocks along those paths. If C were on said path it would be output by the algorithm. □

4.2 Application of NEE Scheduler: SCC Squeezer

The NEE Scheduler was designed as a general purpose scheduling engine that can be applied to any problem. Any optimization that benifits from reordering instructions can take advantage of the NEE Scheduler to do so safely and easily.

In this paper, we apply the NEE Scheduler to HELIX [7], a parallelizing compiler, to extract thread-level parallelism (TLP).

HELIX is a parallelizing compiler that schedules statically identified sequential and parallel code segments of loop iterations onto separate cores. The shorter the sequential segments are, the more expansive the space becomes to parallelize code. As a result, shortening sequential segments is of utmost importance.

Sequential segments are primarily bottlenecked by strongly connected components (SCCs) of dependences that cannot be broken. These SCCs are defined by instructions that can span several basic blocks of a loop, meaning that the dependences themselves lie in instruction streams where instructions in between SCC boundaries may not depend on the SCC itself. These parallelizable instructions are considered "trapped" by the SCC. By "squeezing" the SCC's instructions (moving then together) using a scheduling mechanism, we can release the trapped instructions inside, allowing them to be parallelized by HELIX.

Consequently, a scheduler can define its objective function as follows: two instructions A and B make up an SCC that "traps" instructions inside a sequential segment. The scheduler can move A to B, or vice versa, to release the trapped instructions. We call this scheduler the SCC Squeezer.

The SCC Squeezer is built on top of the NEE Scheduler, which 1) checks for *data dependences* in between A and B and 2) checks that A and B are NEE before moving them together.

5 IMPLEMENTATION

This section outlines some specifics about our implementation of the concepts in this paper. First, it defines the software this project was built on. Then, it explains some design choices we made. Next, it outlines some additional implementation constraints we followed. Finally, it discusses some details about our SCC Squeezer implementation.

5.1 Software

LLVM: LLVM [16] is a widely used compiler framework that enables sophisticated code analyses and transformations. LLVM is used in heavily in both industry and academia due to

its customizability and modularity. For our purposes, LLVM allows us to slot custom compiler passes directly into the middle-end while reusing the rest of the tool chain. LLVM also provides some abstractions to represent aspects of the program, including Instruction, BasicBlock, and Function.

NOELLE: The concepts discussed in this paper were implemented in Noelle [18], a compiler framework built on top of LLVM. Noelle provides high-level abstractions to compiler developers, enabling research concepts to be implemented much faster than in vanilla LLVM.

5.2 Separation of Concerns

When implementing this project, our goal was to use clearly defined APIs to separate clients from implementation details.

- The SCC Squeezer is a client of the NEE Scheduler, which it uses to move instructions in SCCs closer together.
- The NEE Scheduler is a client of both In Between Analysis and NEE Graph, which are used to verify move legality.
- The NEE Graph is a client of the NE Flow algorithm, which it uses to generate comparable NE Expressions for basic blocks.
- NE Flow is a client of a MATLAB system solver.

The MATLAB integration is done as follows: In C++, equations are generated, each of which contains a left and right expression. Each expression is made up of multiple variable terms and one constant term. Once all the equations are constructed, the system solver converts the system into a templated MATLAB program, executes the program, and parses the result back into our C++ NE Flow representation. From this result, it generates a mapping from basic block to reduced expression.

However, we also designed NE Flow to allow the use of any linear system solver. There is an isolated translation layer between the NE Flow memory representation and the system solver, so solvers can be swapped in and out. We chose MATLAB for convenience. A more scalable engineering solution might use the C++ Boost library.

5.3 LLVM-Specific NEE Scheduler Constraints

In addition to the general constraints that the NEE Scheduler uses to validate a move, there are also some LLVM-specific constraints that we follow to maintain LLVM intrinsics. Specifically, a given move is not legal if any of the following conditions are met.

- The instruction to move is a Phi instruction.
- The instruction to move is a Terminator.
- The destination is in between two Phi instructions.

5.4 SCC Squeezer Improvement

The SCC Squeezer's use in HELIX is not a novel optimization, and it has been described and implemented in a limited capacity in NOELLE. However, this paper marks the *first* attempt to generalize the scheduling engine beneath the SCC Squeezer, reducing its dependence on an ad-hoc scheduling optimization, and to expand the optimization beyond the scope of a basic block, allowing for squeezing of global SCCs.

To test our implementation against the existing ad-hoc squeezer, we created a test case with an SCC that spans across basic block boundaries. This test represents a problem that the NEE Scheduler, but not the ad-hoc squeezer, is able to solve.

Figure 8 shows the CFG for this test case. Same-color outlines indicate NEE pairs; black boarders indicate blocks with no known NEE pairs. Blocks 37 and 94 contain the SCC instructions and blocks 39 and 41 contain the "trapped" sequential segment. By using the NEE Scheduler to move the SCC instructions together, we free the sequential segment. Because 37 and 41 are NEE, the move is successful, and we are able to achieve 12 times speedup over the ad-hoc squeezer. However, this test case was designed for the NEE Scheduler, so it is also necessary to test the system on real benchmarks.

6 EVALUATION

Evaluation of this work is split into two parts. First, we evaluate NE Flow and the NEE pairs it generates. Second, we evaluate the NEE Scheduler, and its use in the SCC Squeezer. We use the Parsec3.0 [6] benchmark suite for all evaluation.

6.1 NE Flow

The NE Flow implementation was evaluated for its accuracy compared to a profile-generated NEE Graph.

6.1.1 Comparison to Oracle. Theoretically, for any function, there is a perfect NEE Graph. This graph correctly identifies every NEE pair and has no false positives, therefore representing the optimal output of NEA. We call this hypothetical graph the "oracle", and would ideally use it to measure the success of our implementation of NEA. The pairs in the oracle would be a superset of the pairs in the NEE Graph we generate from NE Flow. However, the oracle is not possible to generate: if we could generate it, we would already have a perfect implementation of NEA. Instead, we compare NE Flow to the next best thing: a profile-generated NEE Graph.

6.1.2 Comparison to Profiler. A pseudo-NEE Graph can be generated using a profiler by analyzing the execution patterns of the code at *runtime*. While every NEE pair in the profile-based NEE Graph is not provably correct, the oracle

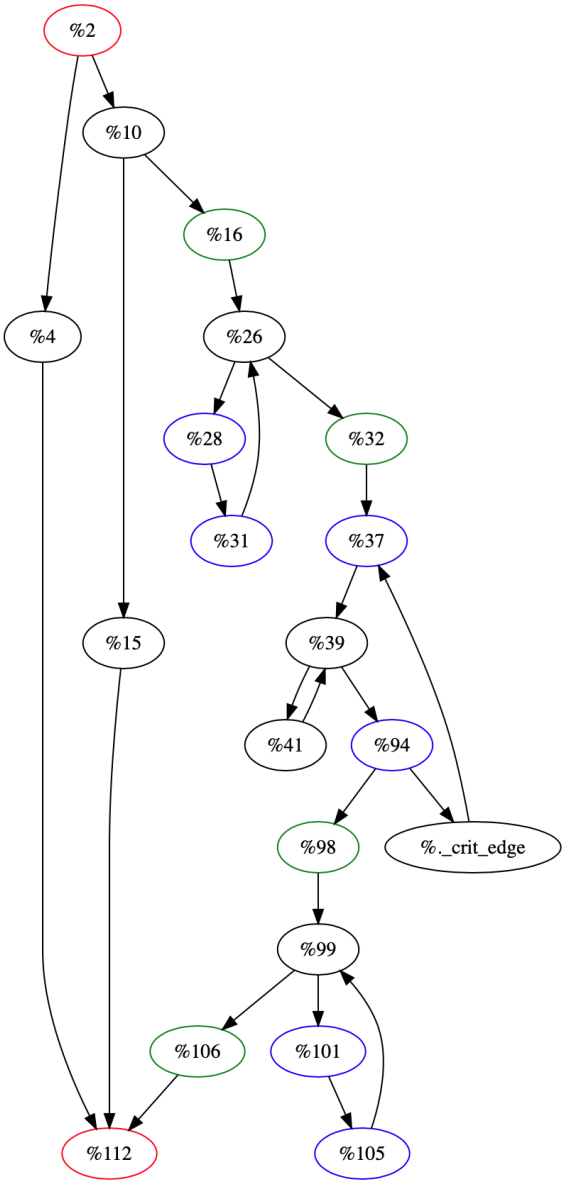

Figure 8: CFG for the SCC Squeezer Test Case annotated with NEE Pairs

NEE Graph is guaranteed to be a subset of the profiler's version of the graph. Because of this, we can easily evaluate the accuracy of the NE Flow algorithm and the resulting NEE Graph by comparing them to this profiler. Notably, matching the profiler graph 100% is not a feasible goal because the profiler can generate *false positives*. For example, if the behavior of two "if" statements depends on user input and both bodies happen to be skipped at runtime, the profiler will pair those bodies as NEE even though they were not guaranteed to run the same number of times. However, the profiler does give us an upper bound to compare against. The closer our

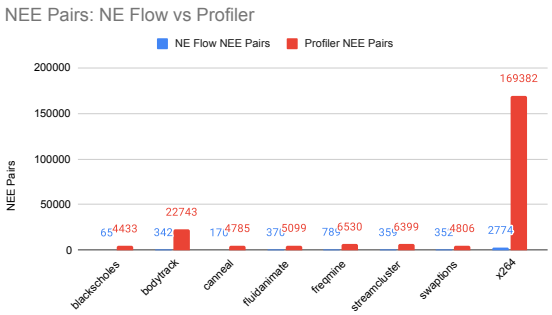

Figure 9: Total NEE pairs found in Parsec3.0, NE Flow vs Profiler

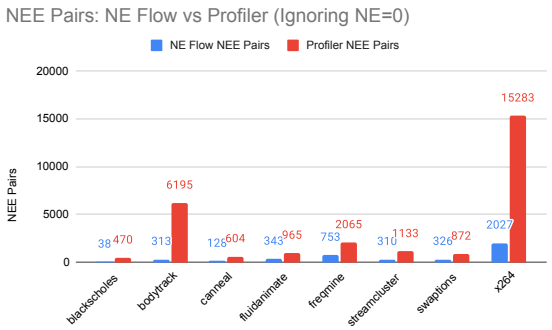

Figure 10: Total NEE pairs found in Parsec3.0 (ignoring NE = 0), NE Flow vs Profiler

algorithm can get to profiler's output, the better.

To quantify this gap, we run analysis on each benchmark in Parsec3.0. We start with all NEE pairs the profiler finds. Then, we count how many of those are present from NE Flow. Figure 9 compares the total number of pairs that are contained in each version of the NEE Graph. Figure 10 only examines pairs in which the NE is not 0.

6.1.3 Analysis of Results. Clearly, the profiler is not an ideal comparison for NE Flow. The profiler has access to information that the oracle will never have access to, and so it generates far too many false positives. Figure 11 shows the profiler-generated NEE Graph for the SCC Squeezer test case previously discussed. Each block in the CFG is annotated with the number of times it executed at runtime (NE). In contrast to Figure 8, this NEE Graph contains many NEE pairs that are not provably correct, such as 4 and 15, which are only NEE because they both happen to run zero times. This relationship is not provable at compile time, thus this pair is a false positive.

Regardless of the usefulness of the comparison, this data is still promising. NE Flow is able to generate a non-trivial number of pairs, and therefore the NEE Scheduler that relies on it has lots of candidate destinations for moving instructions.

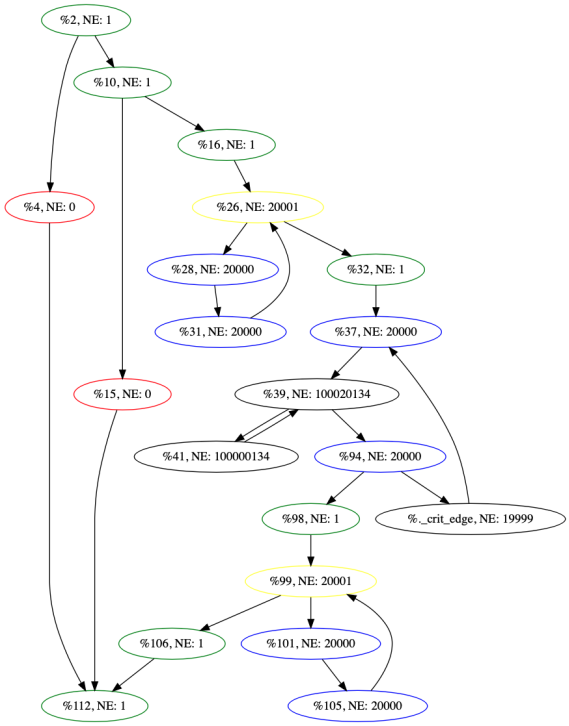

Figure 11: SCC Squeezer test case CFG annotated with profiler-generated NEE Pairs

Benchmark	SCCs Squeezed	Moves Attempted	Moves Succeeded
blackscholes	0	0	0
bodytrack	20	16060	5
canneal	14	147754	80
fluidanimate	4	67262	144
freqmine	14	204330	211
streamcluster	9	39090	37
swaptions	2	200262	618
x264	43	954772	263

Figure 12: Number of moves attempted and made for Parsec3.0 benchmarks

6.2 Scheduling

We evaluate the NEE Scheduler as a scheduling mechanism used by the SCC Squeezer. We evaluate the NEE Scheduler in two ways.

6.2.1 NEE Scheduler - Number of Moves Made. The NEE Scheduler does not decide which moves get attempted: it leaves that up to the client (in this case, the SCC Squeezer). Thus, to evaluate the effectiveness of the NEE Scheduler in a vacuum, we want to evaluate the *quantity* of attempted moves that pass all the constraints imposed by the NEE Scheduler, not the *quality* of the moves made. Figure 12 shows the number SCCs we attempt to squeeze, the number of moves attempted, and the number of successful moves made for each benchmark.

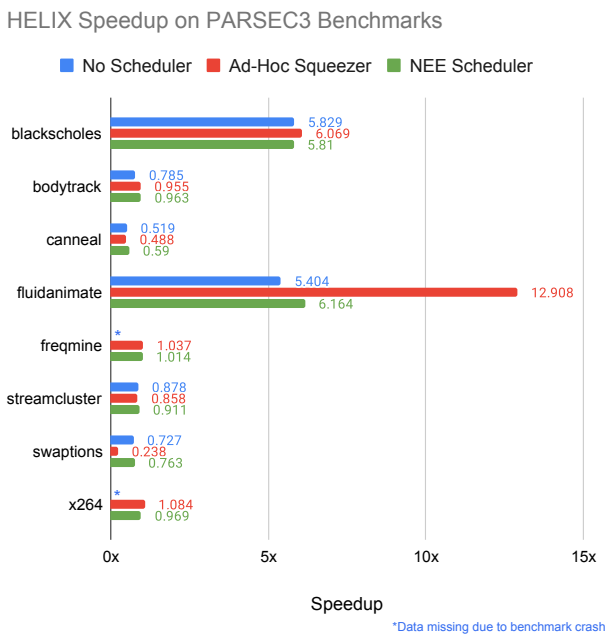

Figure 13: Helix Speedups for Parsec3.0

6.2.2 SCC Squeezer - HELIX Speedup. To evaluate the SCC Squeezer, we run it on Parsec3.0 benchmarks to see if any speedup can be gained with HELIX. We compare our SCC Squeezer with the ad-hoc squeezer/scheduler already available in Noelle, and we hope to improve upon it in two ways:

(1) Our SCC Squeezer is built using the general NEE Scheduler instead of implementing an ad-hoc scheduling solution specific to this use case. If we can match the performance of the ad-hoc squeezer, we can claim our scheduling engine brings generality with no sacrifice to performance.
(2) The NEE Scheduler allows inter-basic-block moves, while the ad-hoc solution stays within the scope of a basic block. We achieve speedup on the unit test designed to take advantage of inter-basic-block squeezing, but achieving speedup on actual benchmarks would prove the impact of this strategy.

Figure 13 shows the results. We compare three different versions of the compiler: 1) with no squeezing implementation, 2) with the ad-hoc squeezer, and 3) with our SCC Squeezer using the NEE Scheduler.

6.2.3 Analysis of results. These results show that, despite the NEE Scheduler being quite conservative, it does allow a significant amount of moves for each benchmark.

Unfortunately, based on the lack of significant speedups, it is clear that the system has some room to improve and is not very impactful in its current state. However, we argue

that the lack of impact speaks to a naive SCC Squeezer implementation, rather than an overly-restrictive NEE Scheduler. Given that the ad-hoc squeezer only allows moves within basic blocks, the NEE Scheduler allows all the same moves and more. Therefore, the weakest link is the *quality* of the moves attempted by the SCC Squeezer. Given this, we outline some potential improvements in the Future Work section.

7 RELATED WORK

This paper introduces novelty on two fronts: it presents 1) a new analysis framework to analyze relative execution counts to enable optimization, and 2) a new instruction scheduling technique built in the middle-end. Given this, we present related work in both areas.

7.1 Related to NEA

Prior works also introduce analysis techniques and frameworks to facilitate optimization, but they aim to gather different kinds of information from a program.

Some works express relationships between instructions in a program. For example, the PDG organizes dependences between instructions and DFA analyzes how data flows through a program.

Some operate at a larger granularity, deriving relationships between basic blocks in a program. Dominance and Post-Dominance, Control Flow Equivalence, and Control Dependence Equivalence are all examples of relations between basic blocks [2] [15].

No existing analysis techniques specifically compare execution numbers between basic blocks to the extent that NEA does. There are, however, some works that explore closely related concepts.

NEE can be compared to Single-Entry Single-Exit (SESE) regions [15]. SESE regions are defined by a pair of blocks, one entry block and one exit block, surrounding a contained region. These pairs are derived by finding instructions that are Control Dependence Equivalent (in other words, share the same control dependences). If one instruction executes, the other does as well. Thus, the entry and exit blocks of an SESE region are provably NEE. However, NEE is broader than SESE regions; the NEE Graph is a superset of all SESE pairs, especially with the introduction of extra equations in NE Flow. Theoretically, this means that SESE regions could be translated to a limited or conservative NEE Graph.

Another work utilizes a concept it calls "conforming loops", which are pairs of loops that run the same number of times [4]. This has parallels to some of the extra equations we show improve NEA.

Profile-guided optimization also has connections to NEA [8]. In fact, this paper utilizes a profiler for evaluation of NEA. However, they are not truly comparable. Profilers are useful for generating heuristics that lead an optimization down a certain path, such as finding hot loops, but as discussed in the evaluation section, execution count information derived from a profiler does not provide any provable guarantees about how many times basic blocks will always execute. In contrast, NEA only provides relationships between blocks that it can statically prove will always hold true, no matter what happens at runtime, so it is far more useful as a tool to help optimizations preserve program semantics.

7.2 Related to Instruction Scheduling

Instruction scheduling has been heavily explored by prior works, but there are some key differences that separate this paper.

Local Code Scheduling: Many works (for example, [13]), as well as most industry compilers, restrict instruction scheduling to Local Code Scheduling, staying within the scope of a basic block. Our work tackles a broader scope in Global Code Scheduling.

Extracting Instruction-Level Parallelism: For many prior works that perform Global Code Scheduling, the focus is on extracting instruction-level parallelism in the back-end [14] [17] [23] [5] [22] [12]. In contrast, we designed a scheduling engine that is not tied to any specific optimization and applied this engine to a unique objective function.

Middle-End: While most instruction scheduling happens in the back-end, some work also explores its use in the middle-end using different objective functions and different methods of maintaining program semantics [21] [19].

Maintaining Semantics: When scheduling globally, there are different ways to maintain semantics. Many works follow the traditional model of maintaining control and data dependences [1] [10] [5] [9]. Some conservatively move instructions upstream or downstream from their starting block [1] [10] [22]. Some utilize a technique called Trace Scheduling [23] [14] [12]. We maintain program semantics in differently: relaxing control dependences via NEE.

Ad-Hoc Code Motion: Some prior work involving code motion maintains semantics in hyper-specific and ad-hoc ways. For example, Loop Invariant Code Motion [1] only moves loop invariants, which are a strictly defined subset of instructions. In contrast, our work provides a general-purpose engine (not tied to a specific optimization) that can work on almost all instructions, even those with side effects.

Profiler: One work uses a profiler to utilizes "frequencies of execution paths" in scheduling [23]. However, as discussed, profile-based NEA does not make any guarantees, and as such can not be used to prove that program semantics are maintained. This work, along with others [17], instead use the profiler to generate heuristics for more effective scheduling.

8 FUTURE WORK

There are several ways to continue this research direction.

Improving NE Flow with Extra Equations: As explained, the NE Flow algorithm is designed to be extensible, improving with further equation generation techniques. There are likely more techniques worth exploring beyond this paper.

Improving the NEE Scheduler Implementation: In addition to the NEE Scheduler, this project also involved the construction of a Local Code Scheduler to facilitate intra-basic-block scheduling. Given the limited scope of this scheduler, implementation of complex features was far easier than in its global counterpart. Certain features could greatly improve the NEE Scheduler, including:

- **Close Enough Movement:** If a move of an instruction to a destination is not legal, offer the option to move the instruction as close as possible.
- **Dragging Chains of Dependences:** If there are data dependences blocking a move, drag them along as far as possible.

Improving the SCC Squeezer Implementation: As discussed in the evaluation section, despite being more constrained, the ad-hoc squeezer outperforms our SCC Squeezer on some benchmarks. Further work is required to understand this discrepancy and improve our SCC Squeezer to match or exceed the speedup of the ad-hoc squeezer by attempting smarter moves.

Application of the NEE Scheduler to Other Use Cases: The NEE Scheduler was designed to be a general purpose scheduling engine, applicable to any optimization that benefits from moving select instructions across basic blocks. In this paper, we apply the NEE Scheduler to the SCC Squeezer and HEIX, but recognize the potential of this engine in other applications. For example, we envision the construction of a Liveness Shrinker that uses the NEE Scheduler to shrink the live range of variables in a function.

Use of Other NEA Relations Outside Scheduling: This paper introduced the NEA framework, which generates various relations between basic blocks. However, we focused exclusively on NEE and its application in scheduling. The other relations (NELE, NEGE, NEL, and NEG) were not applicable to the problems presented here, but they are worth experimenting with in the future. Additionally, the information derived from the NEA framework has potential uses outside the domain of instruction scheduling.

Many-to-Many NEA: In addition to one-to-one pairings of blocks, Number of Execution Analysis also enables one-to-many or many-to-many relationships. For example, a set of two basic blocks A and B could be determined to be NEE with a third basic block, C. This relationship implies that the number of times C will execute is equal to the sum of times A and B will independently execute. This direction is conceptually sound, but will require further engineering effort.

9 CONCLUSION

This work introduced Number of Execution Analysis, a new analysis framework to enable compiler optimizations. We used NEA to create the NEE Scheduler, a middle-end scheduling engine that allows global movement between basic blocks that are guaranteed to execute the same number of times. We attempted to use the NEE Scheduler to help HELIX extract automatic parallelism, and despite the lack of speedup derived from benchmarks, the raw quantity of global moves allowed by the NEE Scheduler demonstrate its power and potential impact. Future work should improve the applications of the NEE Scheduler and apply NEA to other problems.

10 ACKNOWLEDGEMENTS

I would like to thank Simone Campanoni for advising this project, and Peter Dinda for serving as co-chair on my thesis committee. I would also like to thank Souradip Ghosh for his significant contributions to this work. This work was supported in part by [Insert grant name/number].

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*. Addison Wesley, 1986.

[2] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[3] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[4] Christopher Mark Barton. *Code transformations to augment the scope of loop fusion in a production compiler*. 2004.

[5] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 241–255, 1991.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. HELIX: Making the extraction of thread-level parallelism mainstream. *IEEE Micro*, 32(4):8–18, July 2012.

[8] Pohua P Chang, Scott A Mahlke, and Wen-Mei W Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[9] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 246–257, 1995.

[10] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[11] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[12] Stefan M Freudenberger, Thomas R Gross, and P Geoffrey Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1156–1214, 1994.

[13] Philip B Gibbons and Steven S Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 11–16, 1986.

[14] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: An effective technique for vliw and superscalar compilation. In *Instruction-Level Parallelism*, pages 229–248. Springer, 1993.

[15] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, 1994.

[16] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] Uma Mahadevan and Sridhar Ramakrishnan. Instruction scheduling over regions: A framework for scheduling across basic blocks. In *International Conference on Compiler Construction*, pages 419–434. Springer, 1994.

[18] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. NOELLE Offers Empowering LLvm Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*, 2022.

[19] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 254–264, 2018.

[20] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[21] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 171–184. IEEE, 2017.

[22] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 43–55. IEEE, 2007.

[23] Cliff Young and Michael D Smith. Better global scheduling using path profiles. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 115–123. IEEE, 1998.