UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Rafael Alvarenga de Azevedo

Idempotent Backward Slices: A GSA-Based Approach to Code-Size Reduction

Belo Horizonte
2025

Rafael Alvarenga de Azevedo

Idempotent Backward Slices: A GSA-Based Approach to Code-Size Reduction

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira
Co-Advisor: Rodrigo Caetano de Oliveira Rocha

Belo Horizonte
2025

[Ficha Catalográfica em formato PDF]

A ficha catalográfica será fornecida pela biblioteca. Ela deve estar em formato PDF e deve ser passada como argumento do comando `ppgccufmg` no arquivo principal `.tex`, conforme o exemplo abaixo:

```
\ppgccufmg{
     ...
     fichacatalografica={ficha.pdf}
}
```

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Idempotent Backward Slices: A GSA-Based Approach to Code-Size Reduction

RAFAEL ALVARENGA DE AZEVEDO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Fernando M Q Pereira

Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

Rodrigo Caetano de O. Rocha

Prof. Rodrigo Caetano de Oliveira Rocha - Coorientador
Huawei Research - Centro de Investigación de Huawei - Edimburgo - Reino Unido

[illegible]

Prof. Sandro Rigo
Instituto de Computação - UNICAMP

[illegible]

Doutor Yann Herklotz
Verification and Computer Architecture Laboratory - EPFL

Belo Horizonte, 28 de novembro de 2025.

À minha família, com todo meu amor. Agradeço por me ensinarem o valor do estudo e por me darem a liberdade de me dedicar àquilo que verdadeiramente me inspira.

Acknowledgments

I would like to express my sincere thanks to the Minas Gerais State Research Foundation (FAPEMIG) for the essential financial support provided through the research grant, without which this project would not have been possible.

I am especially grateful to my advisor, Professor Fernando Magno Quintão Pereira, whose disciplined guidance, intellectual rigor, and academic excellence were crucial for the development of this thesis. His availability and insightful advice were invaluable.

My gratitude extends far beyond this program to every teacher and professor who has guided me on my academic journey. From my earliest school days to my most advanced studies, each one laid a stone in the path that has led me here. They instilled in me a love for learning that has been my constant motivation, and for their collective wisdom and encouragement, I am profoundly thankful.

“*The day you stop learning is the day you begin decaying.*”
(Isaac Asimov)

Resumo

Otimizações de compiladores são cruciais para melhorar a eficiência de programas, especialmente para software implementado em sistemas com recursos limitados, onde o tamanho do código é uma preocupação primordial. Esta dissertação introduz uma nova técnica para a redução do tamanho de código, identificando e extraindo *Program Slices* recorrentes. A nova abordagem utiliza a representação intermediária Gated Single Assignment (GSA) form, que torna explícitas tanto as dependências de dados quanto as de controle, para permitir a extração precisa de lógicas de programa autocontidas e executáveis, as quais denominamos *Idempotent Backward Slices.* O algoritmo proposto é implementado como um pass completo, funcional e de código aberto para a infraestrutura de compiladores LLVM. Para avaliar sua eficácia, conduziu-se um rigoroso estudo empírico, compilando 2007 programas da coleção de testes de LLVM. Os resultados demonstram que a nova técnica alcança reduções significativas no tamanho do código em casos específicos em que outras técnicas publicadas previamente falham. Concluímos que o *slicing* baseado em GSA é uma ferramenta viável, porém especializada, mais adequada para domínios onde o tamanho do código é fundamental e as bases de código contêm os padrões computacionais recorrentes que nosso algoritmo foi projetado para identificar.

Palavras-chave: Otimização de Compiladores. *Program Slices.* Gated Single Assignment Form.

Abstract

Compiler optimizations are critical for enhancing the efficiency of programs, particularly for software deployed on resource-constrained systems where code size is a primary concern. This thesis introduces a novel technique for code-size reduction by identifying and outlining recurrent *program slices*. Our approach leverages the Gated Single Assignment (GSA) form, an intermediate representation that makes both data and control dependencies explicit, to enable the precise extraction of self-contained, executable program logic, which we term *Idempotent Backward Slices*. The proposed algorithm is implemented as a complete, functional, and open-source, *out-of-tree* pass for the LLVM compiler infrastructure. To evaluate its effectiveness, we conducted a rigorous empirical study, compiling 2007 programs from the LLVM Test Suite. The results demonstrates that our technique achieves significant code-size reductions in specific, targeted cases where other optimizers fail. We conclude that GSA-based slicing is a viable but specialized tool, best suited for domains where code footprint is paramount and code bases contain the recurrent computational patterns our slicer is designed to identify.

Keywords: Compiler Optimizations. *Program Slices.* Gated Single Assignment Form.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Compiler optimization is a foundational area of computer science, focused on transforming programs to improve their use of computational resources. These improvements are typically guided by a specific objective, such as minimizing execution time, reducing power consumption, or decreasing memory footprint. In an era of ubiquitous computing, from resource-constrained IoT devices to large-scale data centers, the efficient use of these resources is key. This work focuses on space optimization, a critical concern for applications deployed on systems with limited memory and storage.

Specifically, we address the challenge of code-size reduction by leveraging a program analysis technique known as *Program Slicing*. First defined by Weiser [24], a program slice consists of the parts of a program that potentially affect the values computed at some point of interest, referred to as the *slicing criterion*. The concept has proven to be widely applicable.

Despite decades of research, efficiently extracting precise and executable slices from arbitrary programs remains an open challenge [3]. For instance, recent approaches in the related field of code-size reduction are often limited to identifying contiguous instruction sequences, thereby failing to capture optimizations across semantically related but discontiguous code fragments [18].

To address the complexities of generating executable slices, particularly in the presence of intricate control flow, we leverage the *Gated Single Assignment* (GSA) form [7], which is an extension of the widely-used Static Single Assignment (SSA) [17] representation. While SSA uses ϕ-functions to merge values from different control flow paths, GSA introduces three explicit gating functions that preserve control flow information by encoding the specific predicate governing each incoming edge. Specifically, the γ-function acts as a guarded conditional selector that assigns v_{true} or v_{false} based on a predicate p at control-flow join points. The μ-function operates at loop headers to select the initial value v_1 upon entry and the loop-carried value v_2 for all subsequent iterations. Finally, the η-function determines the value of a variable after loop termination by capturing the definition v_3 according to the exit predicate p. By utilizing these gating functions, both data and control dependencies are made explicit, providing a richer substrate for program analysis.

We term the resulting subprogram an **Idempotent Backward Slice**. It is a *backward slice* because it is formed by tracing data dependencies backward from a specific instruction, known as the slicing criterion. The slice is also a *pure* function: it is self-contained, has no external side effects such as memory writes, and is referentially transparent, always producing the same output for the same inputs.

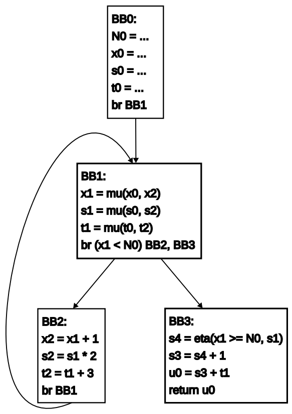
A key structural constraint applies when the slicing criterion is inside a loop. In this case, the extracted slice is bounded by the loop's body, capturing the computation for a single conceptual iteration. The resulting function may still contain its own internal loops if they are essential to compute the criterion's value.

Example 1.0.1. We will now demonstrate the process of extracting an **idempotent backward slice**. Consider the program shown in Listing 1.1 and its corresponding GSA control-flow graph in Figure 1.1. For this example, we select the instruction `s = s + 1;` (represented as `s3 = s4 + 1;` in GSA form) as our slicing criterion. The dependencies for this criterion are highlighted in Figure 1.2. Finally, Figure 1.3 presents the complete idempotent backward slice that is extracted into a new function based on these dependencies.

```c
int foo(int N) {
    int x = 0;
    int s = 1;
    int t = 0;
    while (x < N) {
        x += 1;
        s *= 2;
        t += 3;
    }
    s = s + 1;
    u = s = t;
    return u;
}

```

Listing 1.1: Example C program to analyze.

Figure 1.1: The program's CFG in GSA form.



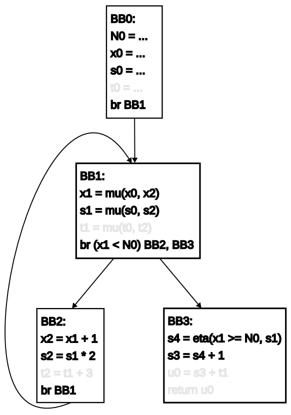
```
BB0:
N0 = ...
x0 = ...
s0 = ...
t0 = ...
br BB1

BB1:
x1 = mu(x0, x2)
s1 = mu(s0, s2)
t1 = mu(t0, t2)
br (x1 < N0) BB2, BB3

BB2:
x2 = x1 + 1
s2 = s1 * 2
t2 = t1 + 3
br BB1

BB3:
s4 = eta(x1 >= N0, s1)
s3 = s4 + 1
u0 = s3 + t1
return u0
```

Source: The author.

The utility of program slicing extends far beyond code-size reduction. The ability to isolate the logic relevant to a specific computation is valuable in numerous domains:

- **Debugging:** Slicing is used to identify the sources of errors. A backward slice from a variable with an incorrect value can show all the code that might have influenced it, dramatically narrowing the search space for the bug.

Figure 1.2: Instructions selected for the new slice.



```
BB0:
N0 = ...
x0 = ...
s0 = ...
t0 = ...
br BB1
```

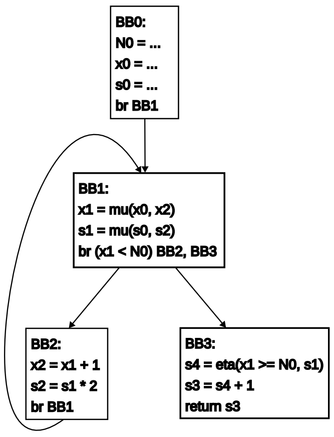
```
BB1:
x1 = mu(x0, x2)
s1 = mu(s0, s2)
t1 = mu(t0, t2)
br (x1 < N0) BB2, BB3
```

```
BB2:
x2 = x1 + 1
s2 = s1 * 2
t2 = t1 + 3
br BB1
```

```
BB3:
s4 = eta(x1 >= N0, s1)
s3 = s4 + 1
u0 = s3 + t1
return u0
```

Source: The author.

Figure 1.3: An idempotent backward slice.



```
BB0:
N0 = ...
x0 = ...
s0 = ...
br BB1
```

```
BB1:
x1 = mu(x0, x2)
s1 = mu(s0, s2)
br (x1 < N0) BB2, BB3
```

```
BB2:
x2 = x1 + 1
s2 = s1 * 2
br BB1
```

```
BB3:
s4 = eta(x1 >= N0, s1)
s3 = s4 + 1
return s3
```

Source: The author.

- **Parallelism:** The independent nature of program slices makes them well-suited for parallel execution on multiprocessor systems, as they can often operate without requiring shared memory or synchronization.
- **Test Case Generation:** By analyzing the slice for a given program feature, one can generate more effective and targeted test cases [24].
- **Lazyfication:** Slices can be used to transform function arguments into expressions that are evaluated lazily, only when needed by the callee [6].

Our primary goal is to develop a practical and robust method for generating executable code slices for code-size reduction. The main contributions of this thesis are:

1. The implementation of an algorithm to construct the GSA form for programs represented in LLVM Intermediate Representation (LLVM IR) in Section 3.1.1.
2. A novel program slicing algorithm that operates on the GSA form to extract Idempotent Backward Slices in Section 3.1.4.
3. The delivery of our implementation as an open-source, *out-of-tree* LLVM pass, facilitating its use and extension by the research community. The implementation is available at `https://github.com/lac-dcc/Daedalus`.
4. A patch for the LLVM Test Suite, enabling rigorous and reproducible evaluation of the pass's effectiveness and correctness, which is available at `https://tinyurl.com/ye2a9ypt`.

In short, the central contribution of this work is a complete, functional, and open-source program slicer for LLVM, based on the Gated Single Assignment form, designed for the purpose of code-size reduction.

The remainder of this thesis is organized as follows. Chapter 2 surveys foundational literature. Chapter 3 introduces our GSA construction and slicing algorithms. Chapter 4 formally argues for the soundness of our approach. Chapter 5 presents our empirical evaluation, including experimental setup and benchmark results. Finally, Chapter 6 summarizes our contributions and discusses future work.

Chapter 2

Literature Review

This chapter begins by presenting fundamental definitions from graph theory and program analysis that are essential for understanding our approach. First, definitions of program analysis terms are given, then we describe the classic definition of a *Program Slice* and the traditional method for its computation, contrasting it with our proposed approach. Following this, we survey recent computational approaches and present an algorithm with a near-linear time complexity for constructing the Gated Single Assignment form for our program slicer. Also, we portray two relevant compiler optimizations: *Function Outlining* and *Function Merging*. The chapter concludes by identifying key gaps in the existing literature that our research addresses.

2.1 Definitions

This section introduces the fundamental concepts of program analysis that underpin our work. First, we define key data structures and concepts that model its implementation, such as *Control Flow Graphs*, *Dominance*, and *Data and Control Dependencies.* Next, we define what a *Data-flow Analysis* is. Finally, we depict two program intermediate representations: the *Static Single Assignment form*, and the *Gated Single Assignment form.* Thus, establishing the theoretical foundation for our work.

2.1.1 Control Flow Graph

A program is represented as a directed graph, called a *Control Flow Graph.* Formally, a CFG is:

- A **digraph** (N, E, n_0), where n_0 is the entry node.

  - N: set of program statements (nodes).
  - E: directed edges representing control flow between statements.
  - **Start**: n_0, or a special node that can reach every other node, but no other node can reach it [19].

For slicing purposes, we also define a *hammock graph*, which extends the concept of a control flow graph as follows.

- A **hammock graph** (N, E, n_0, n_e) is a control flow graph with a unique exit node n_e [19].
  - **Exit**: n_e, or a special node that is reachable from every other node, but does not reach any other node.

Example 2.1.1. Consider the C program from Listing 2.1. Its corresponding Control Flow Graph (CFG) is illustrated in Figure 2.1. In this graph, each node represents a *basic block*, a sequence of instructions that executes linearly and ends with a single *terminator instruction*. The directed edges indicate the flow of control between these blocks. A key convention in our CFG representation is that every basic block must end with an explicit terminator to define its successor(s), except the exit one. Also, most of the time we will omit the Start and Exit blocks.

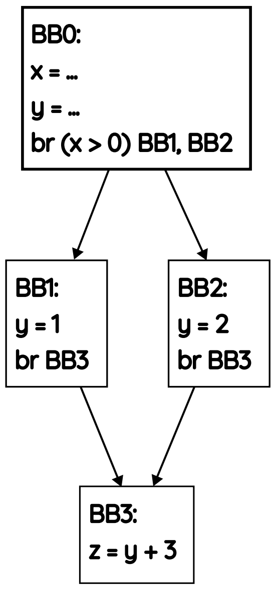
2.1.2 Data-flow Analysis

In compiler design, the backend is responsible for translating an intermediate representation (IR) of a program into an equivalent set of instructions for a target architecture. Before generating the final machine code, compilers perform numerous optimizations to improve performance, so as to reduce code size.

A cornerstone of these optimizations is *Data-flow Analysis*, a technique used to statically approximate the dynamic (runtime) behavior of a program. While precisely determining a program's runtime behavior is an undecidable problem, *Data-flow Analysis* offers a sound approximation by statically modeling the program as a graph [10]. To facilitate such modeling, the program is first transformed into a *Control Flow Graph* (CFG). The choice of a high-quality, machine-independent IR is crucial for performing effective optimizations on this graph [2].

For instance, Example 2.1.2 describes a type of data-flow analysis called *Liveness Analysis*.

Figure 2.1: A simple CFG.



Source: The author.

Example 2.1.2. A common optimization enabled by data-flow analysis is *Liveness Analysis*. This analysis is a classic backward data-flow problem, where information about variable usage propagates from a point of use through the paths of the CFG *backwardly* [2]. The analysis is formalized using a system of data-flow equations defined for each node n in the CFG. These equations rely on the following sets:

- $use[n]$: The set of variables used in node n before any definition.
- $def[n]$: The set of variables defined (assigned a value) in node n.
- $in[n]$: The set of variables that are live at the entry point of node n.
- $out[n]$: The set of variables that are live at the exit point of node n.
- $succ[n]$: The set of successor nodes to node n in the CFG.

The $use[n]$ and $def[n]$, are also known as the *def-use chain* [21]. The relationships between these sets are captured by the following data-flow equations:

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{2.1}$$

$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{2.2}$$

Equation 2.1 states that a variable is live at the *entry* of a block if it is either used within that block or if it is live at the *exit* and not redefined by the block. Equation 2.2 states that a variable is live at the *exit* of a block if it is live at the entry of *any* of its successors. The use of the union operator means a variable is considered live as long as there is at least one future path where it might be used.

These equations are solved iteratively. The *in* and *out* sets for all nodes are initialized as empty and are repeatedly computed until they reach a fixed point, where an iteration causes no further changes to any set [2]. This iterative process is demonstrated in Algorithm 1.

Algorithm 1 Iterative solution for data-flow equations

```
// Initialize in and out sets for all nodes
for each node $n$ in CFG do
    $in[n] \leftarrow \emptyset$
    $out[n] \leftarrow \emptyset$
end for
// Iterate until a fixed point is reached
repeat
    for each node $n$ in CFG do
        $in'[n] \leftarrow in[n]$
        $out'[n] \leftarrow out[n]$
        $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
        $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
    end for
until $in'[n] = in[n] \land out'[n] = out[n]$ for all n
```

Finally, it is important to emphasize that the analyses presented here are *intraprocedural*, meaning they take into account instructions only within a single procedure or function. This contrasts with *interprocedural* analysis, which considers multiple procedures simultaneously to achieve higher precision. Accordingly, our slicing approach is performed intraprocedurally, operating over the program's *def-use chain*.

2.1.3 Dominance

The concepts of dominance and post-dominance are fundamental to control flow analysis in compilers. They provide a structured way to understand the mandatory paths of execution within a program's CFG.

Let's consider a control flow graph $G = (N, E, \text{Start}, \text{Exit})$, where N is the set of basic blocks, E is the set of directed edges representing the flow of control, and $\text{Start}, \text{Exit} \in N$ are the unique entry and exit blocks of the graph, respectively.

A node $d \in N$ **dominates** a node $n \in N$, denoted as $d \text{ dom } n$, if every path from the entry block, Start, to n must pass through d. A node d **strictly dominates** n, denoted $d \text{ sdom } n$, if $d \text{ dom } n$ and $d \neq n$.

For any node n other than the entry block, there are one or more strict dominators. The **immediate dominator** of a node n, denoted $\text{idom}(n)$, is the unique strict dominator of n that is closest to n on any path from the entry. That is, $\text{idom}(n)$ is the strict dominator d of n such that any other strict dominator of n also dominates d. The existence and uniqueness of the immediate dominator for every node (except the entry block) is a foundational property [2].

This immediate dominance relationship allows us to construct the **dominator tree**. This tree is a data structure where:

- The set of nodes is the same as the set of basic blocks N in the CFG.
- The parent of any node n is its immediate dominator, $\text{idom}(n)$.
- An edge exists from node d to node n if and only if $d = \text{idom}(n)$.

The root of the dominator tree is the CFG's entry block, Start. This tree provides a concise representation of the dominance relationships within the procedure.

Furthermore, post-dominance is the dual concept to dominance. It analyzes control flow relative to the exit points of the graph. A node $p \in N$ **post-dominates** a node $n \in N$, denoted as $p \text{ pdom } n$, if every path from n to the Exit block must pass through p. Similarly, a node p **strictly post-dominates** n, denoted $p \text{ spdom } n$, if $p \text{ pdom } n$ and $p \neq n$.

For any node n that is not an exit block, there exists a unique **immediate post-dominator**, denoted $\text{ipdom}(n)$. This is the strict post-dominator of n that is closest to n on any path towards the exit. The uniqueness of the immediate post-dominator is guaranteed [2].

The immediate post-dominance relationship allows for the construction of the **post-dominator tree**. In this tree:

- The set of nodes is the set of basic blocks N.
- The parent of any node n is its immediate post-dominator, $\text{ipdom}(n)$.
- An edge exists from node p to node n if and only if $p = \text{ipdom}(n)$.

The root of the post-dominator tree is the exit block of the CFG. This structure is crucial for analyses such as control-dependence analysis.

Example 2.1.3 shows how dominance trees are built for a given program.

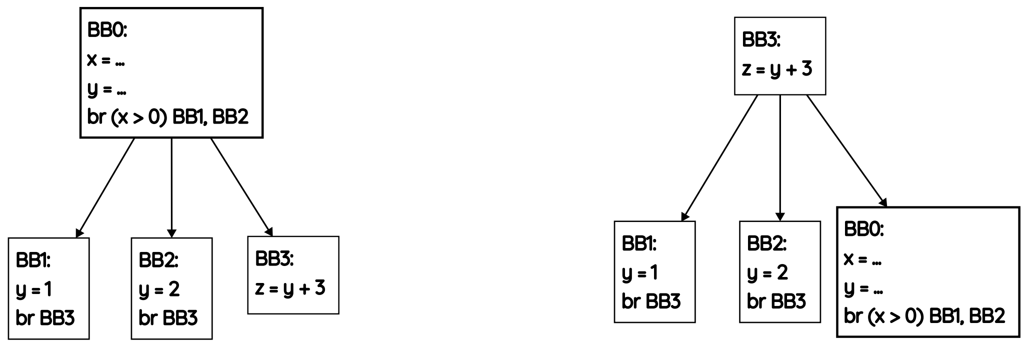
Example 2.1.3. Consider the C program from Listing 2.1. The dominator tree from Figure 2.2a, is induced by the immediate dominance relationships between blocks `BB0`, `BB1`, `BB2`, and `BB3`. Analogously, the immediate post-dominance relation produces the post-dominator tree (Figure 2.2b).

```c
if (x > 0) {
  y = 1;
} else {
  y = 2;
}
z = y + 3;

```

Listing 2.1: Example C program.

Figure 2.2: Simple Dominator trees.

(a) A simple dominator tree. (b) A simple post-dominator tree.



Source: The author.

Therefore, the dominance tree allows the compiler to infer whether the CFG of a program contains loops, while the post dominance tree permits the analyzer to tell if the execution of an instruction depends on another [1].

2.1.4 Data and Control Dependencies

When performing program analysis and transformations, we commonly use the CFG representation of a program. However, this representation contains unnecessary relationships computed, when we want to optimize a program with respect to the sequence between some of its operations. For slicing purposes, we want to know which program statements or variables *depend* on each other. Thus, following the definition of Ferrante et al. [5], let the two notions of dependencies be:

Definition 1 (Data dependency)**.** A statement j is *data dependent* on statement i if a value computed at i is used at j in some program execution.

Definition 2 (Control dependency)**.** A node j is *control dependent* on a node i if:

1. there exists a directed path P from i to j such that j post-dominates every node in P, excluding i and j, and
2. i is not post-dominated by j.

Both definitions are used to construct a program dependency graph, which captures the data and control dependencies between the slice criterion and all program elements on which it depends.

Example 2.1.4. Consider the C program in Listing 2.1. Following Definition 1, the variable `z` is *data dependent* on the variable `y`. To analyze control dependencies, we compute the program's post-dominator tree, shown in Figure 2.2b. According to Definition 2, the assignment to `y` is *control dependent* on the branch instruction `br (x > 0) BB1, BB2`. This dependency exists because `y`'s assignment execution is determined by the path taken at this conditional branch.

Program Dependency Graph

Following Cytron et al. [4], we define a *Program Dependency Graph* (PDG) in the context of SSA form. A PDG is a directed graph $G = (V, E)$, where V represents the set of vertices, with one vertex corresponding to each variable in the program. The set E consists of directed edges, where an edge $(u, v) \in E$ exists if and only if v appears on the left-hand side of an instruction in which u appears on the right-hand side. Example 2.1.5 illustrates a PDG for a simple program.

Example 2.1.5. Figure 2.4b shows an example of a program dependency graph. An edge in this graph can be either solid or dotted. The former represents data dependencies, while the latter represents control dependencies. For instance, in the figure, `t0` is data dependent on `b2`, and control dependent on `w0`.

2.1.5 Static Single Assignment Form

The static single assignment (SSA) form is a program representation in which each variable is assigned exactly once [4]. Therefore, the **Single Information Property** holds when the information associated with a variable v during data-flow analysis remains invariant at every program point where it is *live* [17]. This property simplifies data-flow analysis and facilitates reasoning about the behavior of a program.

In SSA form:

1. Each variable is assigned a unique name whenever it is defined.
2. If a variable is defined in multiple control flow paths, a special ϕ-function is introduced to merge the values from these paths.

Example 2.1.6 shows how these properties emerge in SSA-form programs.

Example 2.1.6. Consider the C code from Listing 2.1. After transforming it into SSA form, we have the program on Listing 2.2.

```c
if (x1 > 0) {
  y1 = 1;
} else {
  y2 = 2;
}
y3 = phi(y1, y2);
z1 = y3 + 3;

```

Listing 2.2: Example in SSA form.

Here, the ϕ-function combines the values of `y1` and `y2` based on the control flow.

2.1.6 Gated Single Assignment Form

The *Gated Single Assignment* (GSA) form is an extension of the widely-used SSA form that makes control dependencies explicit within the intermediate representation.

In a standard SSA form, distinct values for the same variable are assigned unique names (e.g., $x_1, x_2, \dots$). At points where control flow paths merge, a conceptual ϕ-function is used to select the appropriate value. A typical ϕ-assignment is:

$$x_3 \leftarrow \phi(x_1, x_2)$$

This assignment indicates that x_3 will take the value of x_1 if control arrives from one predecessor block, and x_2 if it arrives from another. While SSA form excels at representing data flow, the ϕ-function abstracts away the crucial control-flow information, that is, *which predicate* caused a particular path to be taken.

GSA remedies this by replacing the ϕ-functions with explicit *gating functions*, integrating control-flow predicates directly into the data-flow graph [23]. As Rastello [17] defines, there are three *gating functions*:

1. **The γ (gamma) function**: This function acts as a guarded conditional assignment, explicitly controlled by a predicate. It is typically placed where control flow joins (e.g., at an `if-then-else` statement). Its form is:

   $$v_{out} \leftarrow \gamma(p, v_{true}, v_{false})$$

   Here, if the predicate p evaluates to true, v_{out} is assigned the value of v_{true}. Otherwise, it is assigned the value of v_{false}. The γ function effectively embeds the branch condition into the value selection process.

2. **The μ (mu) function**: This function selects the initial and loop-carried values, and only appears at loop headers. Its form is:

   $$v_{out} \leftarrow \mu(v_1, v_2)$$

   In this case, when the loop execution starts, v_1 is assigned to v_{out}, then, for next iterations, v_2 is used instead.

3. **The η (eta) function**: This function determines the value of a variable at the end of a loop. It has the following form:

   $$v_{out} \leftarrow \eta(p, v_3)$$

   Where v_3 stores the definition reaching a point after a loop exits, and p is the predicate that controls its execution.

To address the limitations of prior approaches that map control dependencies using GSA concepts, as highlighted in Section 2.5, we introduce a formal definition for the transitive control dependency between basic blocks. Definition 3 formalizes this concept by leveraging the explicit, predicate-driven nature of gating functions to capture control relationships that may span multiple blocks.

Definition 3 (Transitive Control Dependency)**.** A basic block B_k is **transitively control dependent** on a predicate p_i originating from a block B_i if either:

1. The execution of B_k is conditioned by a gating function (e.g., γ or η) that is directly controlled by the predicate p_i.
2. There exists an intermediate block B_j such that:
   a) The execution of B_j is conditioned by a gating function controlled by p_i.
   b) The control flow path from B_j to B_k is unconditional, meaning B_k is executed under the same gating condition as B_j without the influence of any intermediate gating function.

Example 2.1.7. By using the gating functions, GSA makes both data and control dependencies explicit. For instance, a traditional ϕ-function at a post-dominator of an `if-then-else` can be rewritten into a γ function at the join point. This explicit representation provides a richer and more precise substrate for advanced program analysis and transformation. An illustrative example is given by extending the code in SSA form from Listing 2.2, to GSA on Listing 2.3.

```
if (x1 > 0) {
  y1 = 1;
} else {
  y2 = 2;
}
y3 = gamma(x1 > 0, y1, y2);
z1 = y3 + 3;

```

Listing 2.3: Program in GSA form.

2.2 Program Slice

Weiser's classic algorithm provides a method for approximating a program slice by analyzing both data and control dependencies, even though finding a perfectly minimal slice is computationally unfeasible [24]. The process works iteratively by first tracing dependencies backward from a specific point of interest, known as the *slicing criterion*. It begins by identifying the variables directly used at that point and then recursively includes all statements throughout the program that could have influenced their values.

Alongside this data flow analysis, the algorithm also identifies control dependencies, that is, any conditional branches or loops that determine whether a relevant statement gets executed. The final program slice is constructed by combining these two sets: it consists of all statements that either perform a relevant calculation or control the execution of another statement already included in the slice.

Before computing all statements that affect a given program point, a **slicing criterion** $C = (i, V)$ is defined by:

- i: a statement index where slicing occurs.
- V: a subset of variables observed at i.

Thus, a backward traversal on the def-use chain of variables in V starts the process of slicing. At the end of the traversal, statements from P that does not affect the computation of variables from the slicing criterion can be deleted.

The sliced program must have the same behavior as the original one, except for the deleted statements. Hence, they define that a **state trajectory** of length k is a sequence:

$$(n_1, s_1), (n_2, s_2), \ldots, (n_k, s_k)$$

where each n_i is a statement and each s_i is a mapping of variables to values. Also, they define that a **projection function** extracts only relevant information from a state trajectory:

$$\text{Proj}_{(i,V)}(n, s) = \begin{cases} (n, s \mid V), & \text{if } n = i \\ X, & \text{otherwise} \end{cases}$$

where $s \mid V$ restricts s to the variables in V.

For a trajectory T, they apply:

$$\text{Proj}_{(i,V)}(T) = \text{Proj}_{(i,V)}(t_1) \ldots \text{Proj}_{(i,V)}(t_n).$$

Formally, Weiser [24] defines that a program slice S of a program P satisfies:

1. S is obtained by deleting zero or more statements from P.

2. S produces the same projection function as P for all inputs where P terminates.

Now, following the generalization of these definitions into an algorithm, Weiser [24] computes a *program slice* by executing the following steps:

Step 1 - Compute Directly Relevant Variables: Define $R_C(n)$, the set of **relevant variables** at statement n:

$$R_C(n) = \begin{cases} V, & \text{if } n = i \\ \{v \mid v \in \text{USE}(n) \text{ and } w \in \text{DEF}(n) \cap R_C(m)\}, & \text{if } m \text{ is a successor of } n \end{cases}$$

This ensures that variables affecting V at i are traced back through the program.

Step 2 - Identify Control Dependencies: A statement b influences a statement s if it controls whether s executes. Define **INFL** as:

$$\text{INFL}(b) = \{n \mid n \text{ is on a path from } b \text{ to its nearest post-dominator}\}.$$

All statements affecting any $n \in S_C$ are included in the slice.

Step 3 - Construct the Slice: The final slice S_C consists of all statements where:

$$R_C(n+1) \cap \text{DEF}(n) \neq \emptyset.$$

The primary objective of this thesis is to present an algorithm that, for a given slicing criterion, computes an *Idempotent Backward Slice*. Our approach is consistent with Weiser's definition but extends it with a fundamental guarantee: the resulting slice constitutes the maximal subset of executable instructions that satisfies the idempotent property. This guarantee ensures that when the slice is executed with a given set of inputs, it produces the exact value that the criterion's variable would have held at the corresponding program point in an execution of the original program with the same inputs.

Consequently, our analysis focuses on dependencies between variables, rather than statements. The final slice is synthesized into a self-contained function whose sole purpose is to compute the value of the variable from the slicing criterion.

2.3 Algorithms

Weiser's algorithm performs a dense analysis by associating relevant information with pairs of variables and program points. However, his approach to identifying data and control dependencies, which relies on computing consecutive sets of transitively relevant statements, can be made more efficient through the use of alternative data structures [21].

To address the efficiency problem of a dense analysis, an algorithm was proposed by Rodrigues et al. [19], that computes program slices using a sparse analysis, which handles data and control dependencies more effectively. Subsequently, another slicing approach was presented by Guimarães and Pereira [6], that uses gates on ϕ-functions to determine control dependencies.

Nevertheless, their notion of gating is insufficient for our purposes. Instead, our approach relies on path expressions to compute predicates. These predicates are then used to construct the GSA form, which assists our program slicer on finding control dependencies.

2.3.1 Sparse Slicing

A dense analysis computes data and control dependencies by associating information with pairs of variables and program points. A program point represents a region between two instructions in a control flow graph. Weiser's approach has a worst-case complexity of $\mathcal{O}(V^2)$, where V is the number of variables. However, by leveraging the SSA form, this quadratic complexity can be reduced to $\mathcal{O}(V)$ [19].

To achieve this reduction, the program must first be transformed into SSA form. In this format, each variable is assigned a value exactly once. Consequently, all tracking information is bound directly to the variable's name, enabling a more efficient computation of dependencies.

Consider the C code presented on Listing 2.4. Its SSA form is visualized within the CFG in Figure 2.3.

To compute data and control dependencies between variables, Rodrigues et al. [19] use the *Program Dependency Graph* (PDG) and the *Dominator Tree* (DT) data structures. The PDG represents dependencies between program variables, while the DT captures dominance relationships between nodes in the CFG.

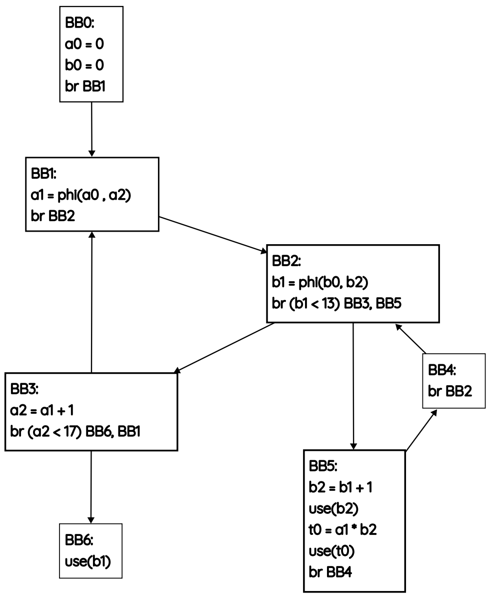
1. **Building the PDG**: After selecting a slice criterion (e.g., a variable or program point of interest), a traversal is performed over the PDG to identify all relevant dependencies. The PDG is constructed by analyzing the program's data and control flow, linking variables and instructions based on their dependencies.

2. **Building the Dominator Tree**: To construct the DT, a dominance analysis over the CFG is performed. This analysis identifies the dominance relationships between nodes, in such a way that the resulting tree maps each of them to its immediate dominator, enabling efficient traversal and dependency analysis.

```c
a = 0;
b = 0;
do {
    while (b < 13) {
        b = b + 1;
        x = x + a * b;
    }
    a = a + 1;
} while (a < 17);
use(b);
```

Listing 2.4: Example C program.

Figure 2.3: The program's CFG in SSA form.



Source: The author.

3. **Influence Region**: The influence region of a block B is the set of blocks dominated by B but not post-dominated by it. This region determines which variables are controlled by a predicate.

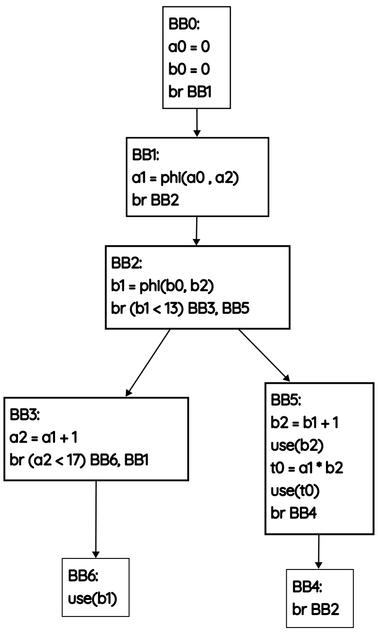
The program's DT and PDG are shown in Figure 2.4a and Figure 2.4b, respectively. These structures are used to compute backward dependencies based on the slice criterion `use(b2)` (Figures 2.5c and 2.5b), and the resulting backward slice is shown in Figure 2.5a. Finally, the algorithm is generalized as follows:

1. **Input**: A program in SSA form, represented as a dominance tree.
2. **Output**: A slice containing only relevant instructions.
3. **Steps**:
   - Traverse the dominance tree to identify the influence region of each predicate.
   - Link predicates to variables defined within their influence region.
   - Use the PDG to compute transitive dependencies efficiently.

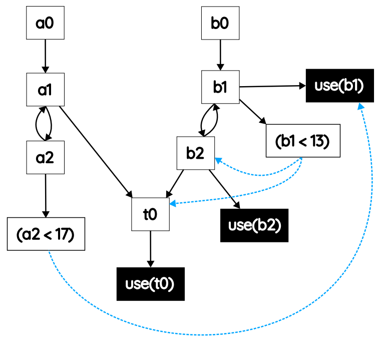
This algorithm improves upon Weiser's dense analysis by first transforming the program into SSA form. This transformation enables a sparse analysis, reducing the complexity of computing program slices from $\mathcal{O}(V^2)$ to $\mathcal{O}(V)$ [19]. It is also worth noting that, unlike Ferrante et al. [5], the authors focus on tracking dependencies between variables rather than between program statements.

Figure 2.4: The Dominator Tree and the Program Dependency Graph.

(a) The Dominator Tree.



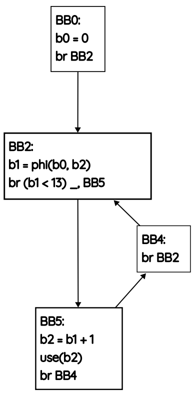
(b) The Program Dependency Graph (PDG).



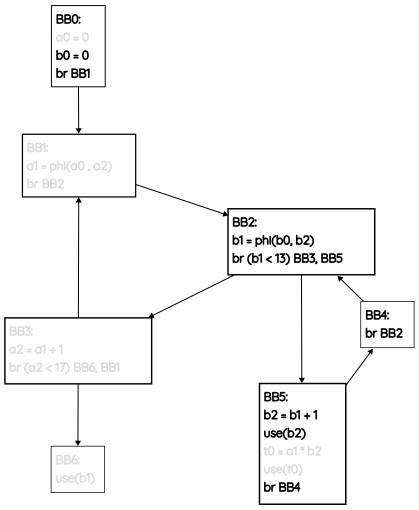
Source: The author.

Figure 2.5: Comparison of program slice, CFG, and PDG fragments.

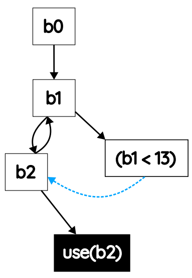
(a) The outlined function.



(b) The CFG fragment.



(c) The PDG fragment.



Source: The author.

2.3.2 Gating Phi-functions

Another approach to extract program slices was proposed by Guimarães and Pereira [6]. Their work is not focused on program slicing, but they worked on a single-pass algorithm to outline instructions into delegate functions, that would benefit from having its arguments lazily evaluated. To accomplish the slicing task, they define the following concepts: *Backward Slice*, *Dependencies*, *Gates*, and *First Dominator*. Consequently, they illustrate how a *sliced program* is derived.

Similar to Rodrigues et al. [19], the authors focus on dependencies between program variables rather than program statements. Accordingly, we restate their program slicer definitions.

Definition 4 (Backward Slice)**.** Given a program P, and a variable v defined at a program point $p \in P$, the *backward slice* of v at p is a subset P_s of P's program points containing p, such that if P computes v with value n given input I, then P_s also computes v with value n given input I. The pair (p, v) is called a *slice criterion.*

Definition 5 (Dependencies)**.** Variable's dependencies are defined as follows:

1. A variable u is *data dependent* on a variable v if u is defined by an instruction that uses v.

2. A variable u is *control dependent* on a variable v if the assignment of u depends on a terminator (e.g., a conditional branch) controlled by v.

3. A variable u *depends* on a variable v if it is either data dependent or control dependent on v, or if it depends on a variable w that depends on v.

Definition 6 (Gates)**.** Let $G = (V, E \cup \{b_{start}, b_{end}\})$ be a control flow graph, and let $b_0, b_1 \in V$ be two basic blocks.

1. We say that b_1 *post-dominates* b_0 if every path from b_0 to b_{end} goes through b_1.

2. We say that b_1 is the *immediate post-dominator* of b_0 if b_1 post-dominates b_0, and for any other node b_p that post-dominates b_0, either $b_p = b_1$ or b_p post-dominates b_1.

Given that a predicate p is a terminator at b_0, we say that p *gates* every ϕ-function in the basic block b_1 that immediately post-dominates b_0.

Definition 7 (First Dominator)**.** Given a set of nodes in a slice B, and a basic block $b \notin B$, the *first dominator* of b (within B) is a node $b_n \in B$ (with $b_n \neq b$) such that b_n dominates b and b_n is the closest parent of b in the dominance tree.

Definition 8 (Sliced Program)**.** Let a program P be represented by a CFG (V_p, E_p). Let $V_s \subseteq V_p$ be the set of basic blocks from P that belong to a backward slice created from some slice criterion. From V_s we derive a new program $P_s = (V_s, E_s \cup \{b_{start}, b_{end}\})$, where E_s is defined as follows:

1. If $b_0 \to b_1 \in E_p$ for some $b_0 \in V_s$ and $b_1 \in V_s$, then $b_0 \to b_1 \in E_s$.
2. If $b_0 \to b_1 \in E_p$ for some $b_0 \notin V_s$ and $b_1 \in V_s$, and b_1 contains a use of a variable v not defined in b_1, then $b_f \to b_1 \in E_s$, where b_f is the first dominator of b_1 in V_s.
3. If a block $b \in V_s$ contains a definition of every variable used in it, then E_s contains the edge $b_{start} \to b$.
4. If b contains the slice criterion, then E_s contains the edge $b \to b_{end}$.

In summary, the slicing procedure employed in their implementation integrates two fundamental concepts: **purity** and **gating of ϕ-functions**.

Purity ensures that extracted slices are free from side effects, excluding instructions that store to memory, invoke impure functions, or may raise exceptions, thereby preserving program semantics even when the execution order changes.

Gating ϕ-functions, in turn, tries to augment the SSA representation with explicit predicate dependencies, to transform control dependencies into data dependencies. This approach constitutes an initial step toward implementing a program slicer based on the GSA form.

Starting from a selected slice criterion, the algorithm traverses the control flow graph backward, collecting all data and control dependencies, and outline the visited instructions into a new function. The resulting sliced function is guaranteed to compute the target value exactly once, without side effects, and with all dependencies explicitly represented.

2.3.3 Gating by Path Expressions

Unlike Guimarães and Pereira [6], which computes ϕ-function gates using post-dominance relationships, Tu and Padua [23] propose the usage of *path expressions* to compute gating functions and transform a program from SSA to GSA form. A *path expression* is a regular expression that represents the set of paths taken in a program's CFG. The foundational algorithm for computing these path expressions was originally introduced by Tarjan [20]. In their method, new symbols are introduced to construct the path expressions that represent gating functions. This approach simplifies the identification of which edges are traversed along a path reaching a ϕ-function, which predicate controls it, and which gating function should replace it.

1. **Path Expression**: Given a CFG (N, E), any path in the CFG can be treated as a string of edges in E, but not all such strings are valid paths in CFG. A *path expression* P of type (u, v) is a simple regular expression over E such that every string in $\sigma(P)$ is a path from node u to node v, where $\sigma(P)$ denotes the set of strings generated by the regular expression P. Every sub-expression of a path expression is itself a path expression whose type can be determined as follows:

   - If $P = P_1 \cup P_2$, then P_1 and P_2 are path expressions of type (u, v).
   - If $P = P_1 \cdot P_2$, then there exists a unique node w such that P_1 is a path expression of type (u, w) and P_2 is a path expression of type (w, v).
   - If $P = P_1^*$, then $u = v$ and P_1 is a path expression of type $(u, v) = (u, u)$.

2. **Path Expressions as Gating Functions**: Different paths reaching a ϕ-function node are represented by path expressions. To define the symbols for edges such that a path expression also takes the form of a *gating function*, only outgoing edges from conditional statements are needed to unambiguously represent a path. The following conventions are adopted:

   - A white space symbol Λ represents an *unconditional* edge.
   - A white space symbol $\varnothing$ represents an edge *not taken* at a branch node.
   - A γ expression $\gamma(p, e_1, e_2, \ldots, e_n)$, where exactly one e_i is Λ and all others are $\varnothing$, represents the ith edge from an n-way branch statement with condition p.

   Using these symbols, a path expression can be represented as a gating function $R(u, v)$ for a path from node u to node v. The following simplification rules apply:

$$R_1 \cup R_2 = \begin{cases} R_2, & \text{if } R_1 = \varnothing, \\ R_1, & \text{if } R_2 = \varnothing, \\ \gamma(p, R_{1t}, R_{1f}) \cup \gamma(p, R_{2t}, R_{2f}), \\ \text{otherwise, yielding } \gamma\big(p,\ R_{1t} \cup R_{2t},\ R_{1f} \cup R_{2f}\big) \end{cases}$$

$$R_1 \cdot R_2 = \begin{cases} \varnothing, & \text{if } R_1 = \varnothing \text{ or } R_2 = \varnothing, \\ R_2, & \text{if } R_1 = \Lambda, \\ R_1, & \text{if } R_2 = \Lambda, \\ \gamma(p, R_{1t}, R_{1f}) \cdot R_2, & \text{if } R_2 \neq \varnothing \ \Rightarrow\ \gamma\big(p,\ R_{1t} \cdot R_2,\ R_{1f} \cdot R_2\big) \end{cases}$$

This formalism allows the computation of gating functions for ϕ-functions, where the gating function encodes the concatenation of conditional branches along a gating path [23].

2.4 Optimizations

There are two compiler optimizations particularly relevant to our approach: **function outlining** and **function merging**. The former reduces code size by extracting code fragments into new functions, while the latter combines identical or similar functions into a single one. In our use case, we aim to capture both behaviors so that slicing can contribute to reducing overall code size.

Function outlining has long been used to compact programs by isolating repeated or structurally similar fragments into separately callable functions. Function merging, in turn, is a complementary optimization that eliminates redundancy by consolidating identical or near-identical functions, as performed by LLVM's default function merging pass [13]. Both techniques are widely studied in the literature. For instance, Lee et al. [9] generalize these ideas to a global scope, enabling the linker to optimize functions across different modules and even separate builds.

Example 2.4.1 demonstrates, in a high level view, the changes of a couple of given functions, before and after the application of a function merging strategy.

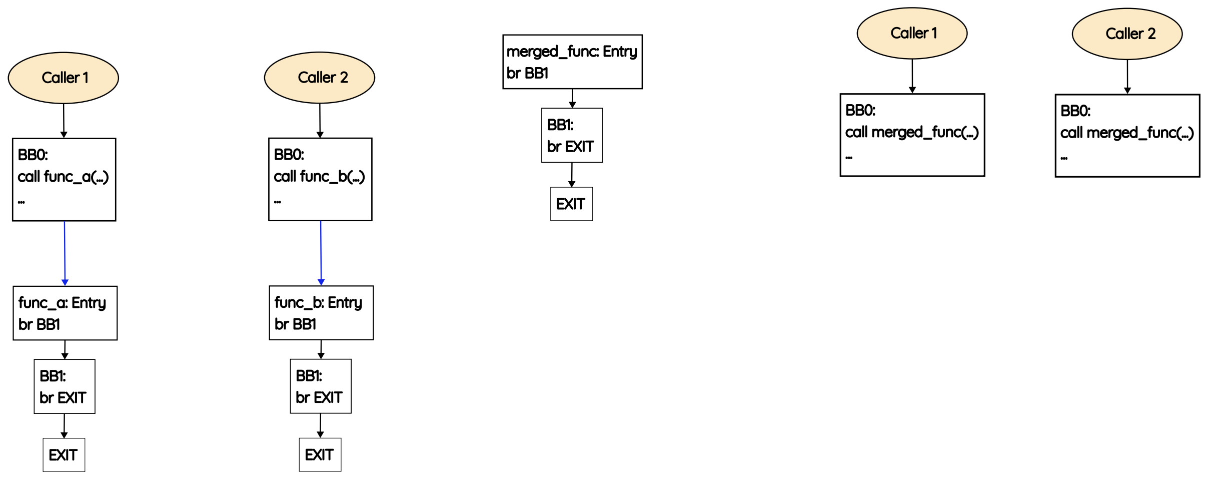
Example 2.4.1. Consider LLVM's default pass to merge similar functions `mergefunc`. Their approach consists on identifying and merging functions that are semantically identical to reduce the final code size. The process begins by calculating a hash of each function's structure to quickly filter out dissimilar candidates. For functions with matching hashes, a more detailed, instruction-by-instruction comparison is performed to confirm they are exact equivalents. When two identical functions, say F and G, are found, one is chosen as the canonical version, and all calls to the other function are redirected to it, often by replacing the duplicate function with an alias or a simple stub that calls the canonical one [13].

For instance, consider two identical functions, `func_a` and `func_b`, each invoked by a different caller (Figure 2.6a). A function merging pass first identifies that these functions are equivalent. It then combines them into a single, new function named `merged_func`, as depicted in Figure 2.6b. As a final step, the pass updates the original call sites, redirecting them to the new, unified function (Figure 2.6c) and thus reducing code size.

Another way to implement the outlining optimization for code-size reduction is de-

Figure 2.6: The traditional function merging process.

(a) Before merging. (b) Merged functions. (c) Call sites after merging.



Source: The author.

scribed by Tomašević et al. [22]. Their method searches for the longest repeated sequence of instructions within a function. As they explain, this problem is analogous to finding the longest common substring, where basic blocks correspond to strings and instructions to characters, and can be efficiently solved using suffix trees [8, 22].

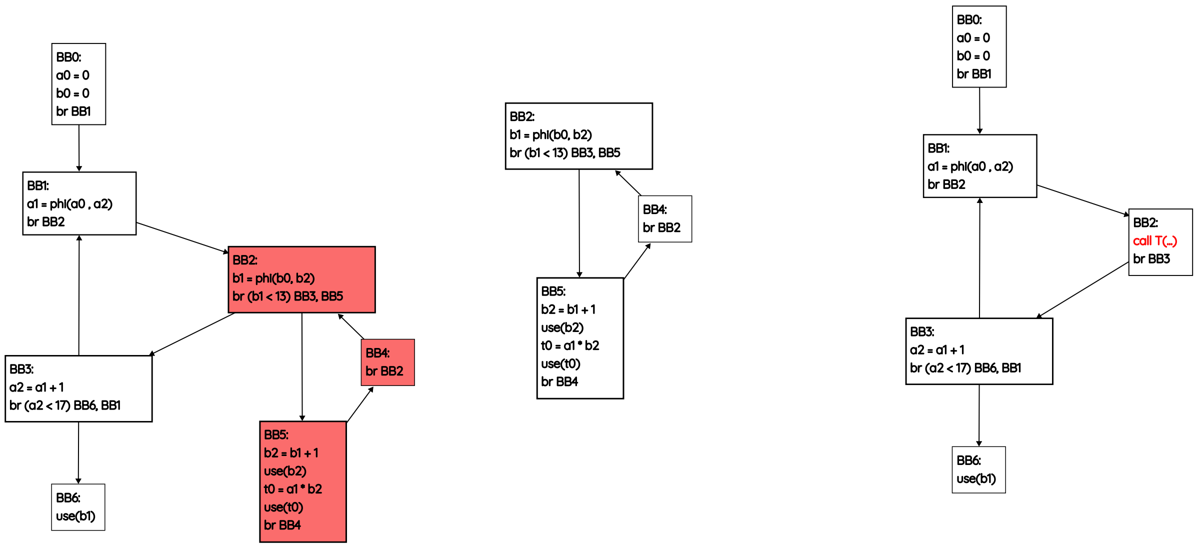
Similarly to the function merging example, a high level view of the function outlining process is given on Example 2.4.2.

Example 2.4.2. The common outline strategy relies on the selection of a specific region of code, usually an infrequently executed *cold* segment, is identified and moved out of its original *host* function. This extracted code is placed into a new, separate function, and the original code block is replaced with a function call to this new routine [11]. Thus, given a host function F and its CFG on Figure 2.7a, the region highlighted in red was identified by the compiler as a *cold* segment, and further extracted into function T (Figure 2.7b). Finally, the host function contains a call to the extracted function, as pictured on Figure 2.7c.

Our approach, however, differs fundamentally. Instead of searching for repeated instruction patterns, we focus on the instructions that are relevant for computing a variable at a given program point. Consequently, our solution requires a full traversal of both the data and control dependencies of that variable, ensuring that the resulting slice preserves the program's semantics.

Figure 2.7: The traditional function outlining process.

(a) The host F function, with a highlighted *cold* segment. (b) The extracted T function. (c) The new host function.



Source: The author.

2.5 Motivating Example

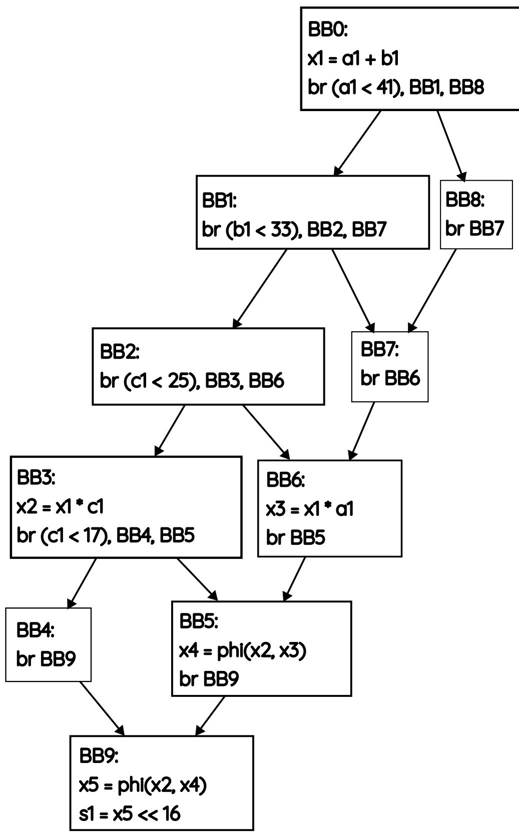
While Guimarães and Pereira [6] implements a program slicer based on gated ϕ-functions, we extend their implementation, *Wyvern*, to handle programs with intricate control flow, such as the ladder graph. The primary challenge in this setting is to extract a slice when the slice criterion resides in the exit block (block BB9 in Figure 2.8).

During dependency traversal, *Wyvern* cannot reliably determine which predicate controls implicit transitive dependencies by considering only the post-dominators of ϕ-functions. We address this limitation by encoding control dependencies as regular expressions, thereby making implicit transitive dependencies explicit to our one-pass traversal algorithm.

Let `s1 = x5 << 16` be our slice criterion. The algorithm starts by traversing backwardly `s1`'s def-use chain. Therefore, it will identify the instructions on Figure 2.9a as data dependencies.

Values `a1`, `b1`, and `c1` are function arguments. After traversing the data dependencies, the algorithm also considers the control dependencies associated with each ϕ-function. These correspond to the terminator instructions of the basic blocks that control the execution of the incoming values to the ϕ-functions. Consequently, the instructions in Figure 2.9b determine the execution of each ϕ-function in the sliced program. Specifically, instructions `1` and `3` from Figure 2.9b control instruction `1` from Figure 2.9a, while

Figure 2.8: The ladder control flow graph.



Source: The author.

Figure 2.9: Slice criterion's dependencies.

(a) Data dependencies.

```
1. x5 = phi(x2, x4)
2. x2 = x1 * c1
3. x4 = phi(x2, x3)
4. x1 = a1 + b1
5. x3 = x1 * a1
```

(b) Control dependencies.

```
1. br (c1 < 17), BB4, BB5
2. br (c1 < 25), BB3, BB6
3. br (a1 < 41), BB1, BB8
```

Source: The author.

instructions 2 and 3 from Figure 2.9b control instruction 3 from Figure 2.9a.

Although their method for identifying which predicates control each ϕ-function is functional, it remains incomplete. When their outlining procedure constructs the slice function with the selected instructions, it omits the branch from the `BB1` block that governs execution of `BB2`. This omission occurs because their gating computation does not account for predicates that control the execution of other predicates, that is, they miss the notion of transitive control dependencies between predicates (Definition 3). Consequently, this omission breaks their `first dominator` (Section 2.3.2) computation implementation, and makes the creation of an invalid branch from `BB0` to `BB9`.

Unlike *Wyvern*, our approach captures transitive control dependencies by the use of *path expressions* to represent the edges that govern the execution of ϕ-functions. Our choice is motivated by the explicit representation of edges that regular expressions provide, while also enabling the concatenation of branches along a gating path. As a result, the missing terminator in the given example is successfully captured by our program slicer. A detailed example is given on section 3.3.

Chapter 3

Algorithms

This chapter details the core algorithms of our methodology. First, we present an algorithm to convert a program from Static Single Assignment form into the Gated Single Assignment form required by our slicer. Following this, we formalize our novel algorithm for extracting program slices from GSA-based programs, which extends the work of Tu and Padua [23] and Guimarães and Pereira [6] to support our slicing objectives.

3.1 Program Slicer

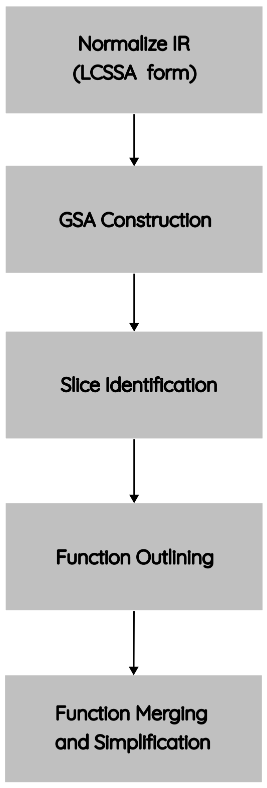
Our program slicer is implemented as an *out-of-source* LLVM pass called *Daedalus*. The source code is publicly available at `https://github.com/lac-dcc/Daedalus`.

To enable program slicing, we implemented the algorithm of Tu and Padua [23] on top of LLVM's data structures. Our pass extends LLVM's SSA representation by modeling gating functions as regular expressions. Given that the η function does not contribute to the predicate computation, we restrict the mapping of regular expressions to γ and μ gating functions. The transformation begins by normalizing programs into Loop-Closed SSA form (LCSSA), where every value defined inside a loop is exclusively used within that loop [15].

The slicing procedure begins with a single-step traversal of data dependencies to collect all dependencies of the chosen slice criterion. If the outlined instructions do not produce side effects in the original function, they are extracted into a new function.

After outlining all possible slices, the pass attempts to merge similar ones, removing redundant instructions from the original function. Finally, the *simplifyCFG* pass is applied to the merged functions, producing the smallest possible code. An overview of the program slicer is presented in Figure 3.1.

Figure 3.1: Program slicer overview.



Source: The author.

3.1.1 GSA Construction

This subsection formalizes our implementation of GSA construction and the extraction of controlling predicates for ϕ-functions. The implementation is divided into two stages. First, Algorithm 6 computes, for each basic block, symbolic *path expressions* that summarize how control can reach that block. Each path expression is decomposed into a non-loop-carried component γ and a loop-carried component μ. Second, Algorithm 7 traverses these expressions to collect the concrete LLVM predicate instructions (branch or switch terminators) that gate each ϕ-function.

To set the notation, let $G = (V, E)$ be the CFG with a distinguished entry block, and let `DT` be the dominator tree over V. For any $v \in V$, $\text{pred}(v)$ denotes the set of CFG predecessors of v. Symbolic path expressions are drawn from the algebra P:

$$P ::= \varnothing \mid \lambda(b) \mid \text{edge}(b, i) \mid p_1 \cup p_2 \mid p_1 \cdot p_2,$$

where $\varnothing$ represents an unsatisfiable path, $\lambda(b)$ denotes an unconditional branch to block

b, and edge(b, i) denotes following successor i of block b. Union ($\cup$) and concatenation ($\cdot$) follow the simplification rules $\varnothing \cup p = p$, $\varnothing \cdot p = \varnothing$, and $\lambda(b) \cdot p = p$.

Algorithm 6 operates by performing a single, bottom-up traversal of the dominator tree (`DT`), processing nodes in reverse post-order. It employs a union-find data structure to efficiently manage and merge path information as it moves up the tree. The core of the algorithm is a three-phase process executed for each node u in `DT`:

DERIVE Phase For each child v of u, the algorithm inspects all incoming control-flow edges $w \to v$. Using the dominance relation, it classifies each edge:

- If v dominates w, the edge is a **back-edge**. Its path expression is immediately accumulated into the loop-carried component, μ_v.
- Otherwise, the edge is a **forward or cross-edge**. A direct merge is insufficient here, as the full path from the current subtree root is not yet known. The path is therefore temporarily stored in a per-child list for later processing.

MERGE Phase This phase resolves the paths from forward and cross-edges using a fixed-point iteration. It repeatedly traverses the temporarily stored paths, prepending the path expressions from the roots of their respective sibling subtrees. This iterative process continues until a stable state is reached, ensuring that all contributing control-flow paths from different dominator subtrees are correctly composed into the non-loop-carried component, γ_v. This step is critical for handling complex, non-nested control flow.

LINK Phase Once the path expressions for a child v are finalized, v is linked to its parent u in the union-find structure. Its now-complete γ_v expression becomes the path from u to v, enabling the next level of the bottom-up traversal.

Edge expressions are constructed by EDGEEXPR (Algorithm 4): if w has only one successor, the expression is $\lambda(w)$, whereas conditional or switch terminators yield edge(w, i). To avoid exponential growth in structured control flow, the MERGE function (Algorithm 5) eagerly simplifies unions and concatenations.

The complexity of Algorithm 6 is bounded by $O(|E|\,\alpha(|V|))$, where α is the inverse Ackermann function [20]. Each CFG predecessor edge is processed once, and union-find operations are amortized. Simplifications occur in linear time relative to the expression size. The invariant maintained throughout the traversal is that, after processing a child v of u in `DT`, the mapping $\mathsf{R}[x]$ for every x in the subtree of v represents the simplified symbolic path from the union-find root to x [20].

Algorithm 2 FIND operation (union-find with path compression)

```
1: function FIND($u$)
2:     $p \leftarrow \mathsf{Parent}[u]$
3:     if $p = u$ then                          ▷ $u$ is the root of the current partition
4:         return $u$
5:     end if
6:     $r \leftarrow \mathrm{FIND}(p)$                        ▷ Recursively find the root of the parent
7:     $\mathsf{R}[u] \leftarrow \mathsf{R}[p] \cdot \mathsf{R}[u]$                         ▷ Update path from root to $u$
8:     $\mathsf{Parent}[u] \leftarrow r$                                 ▷ Path compression
9:     return $r$
10: end function
```

Algorithm 3 EVAL operation (compute root-to-x path)

```
1: function EVAL($u$)
2:     $p \leftarrow \mathsf{Parent}[u]$
3:     if $p = u$ then                               ▷ $u$ is the root of its partition
4:         return $(\mathsf{NeedsPhi}[u] \lor \mathsf{IsInitialDef}[u], \mathsf{R}[u])$
5:     end if
6:     $(\phi_{\text{parent}}, p_{\text{parent}}) \leftarrow \mathrm{EVAL}(p)$                          ▷ Recurse on parent
7:     $\mathsf{Parent}[u] \leftarrow \mathsf{Parent}[p]$            ▷ Path compression (point to eventual root)
8:     $\mathsf{R}[u] \leftarrow p_{\text{parent}} \cdot \mathsf{R}[u]$                   ▷ Accumulate path from root to $u$
9:     $\phi_{\text{needed}} \leftarrow \phi_{\text{parent}} \lor \mathsf{NeedsPhi}[u] \lor \mathsf{IsInitialDef}[u]$
10:    return $(\phi_{\text{needed}}, \mathsf{R}[u])$
11: end function
```

Algorithm 4 EDGEEXPR constructor

```
1: function CreateEdgeExpr($w \to v$)
2:     $T_w \leftarrow$ terminator instruction of $w$
3:     if $T_w$ has one successor then
4:         return $\lambda(w)$
5:     else
6:         $i \leftarrow$ index of successor $v$ in $T_w$
7:         return edge$(T_w, i)$
8:     end if
9: end function
```

Algorithm 5 MERGE operation (path expression simplification)

```
1: function MERGE($p_1, p_2$)
2:     if $p_1 = \varnothing$ then return $p_2$
3:     end if
4:     if $p_2 = \varnothing$ then return $p_1$
5:     end if
6:     return $p_1 \cup p_2$
7: end function
```

Algorithm 6 Gated SSA path construction

```
Input: CFG $G = (V, E)$, dominator tree DT, initial-def set $S \subseteq V$
Output: For each $v \in V$: path expressions $\gamma_v, \mu_v$ and boolean $\mathsf{NeedsPhi}_v$
 1: Initialization:
 2: for all $u \in V$ do
 3:     $\mathsf{Parent}[u] \leftarrow u$;  $\mathsf{R}[u] \leftarrow \varnothing$;  $\gamma_u \leftarrow \varnothing$;  $\mu_u \leftarrow \varnothing$;  $\mathsf{NeedsPhi}[u] \leftarrow$ false
 4:     $\mathsf{IsInitialDef}[u] \leftarrow (u \in S)$
 5: end for
 6: for each node $u$ in reverse post-order of DT do
 7:                                                             ▷ DERIVE PHASE
 8:     Let ListP be a map from nodes to lists of (subroot, path) pairs
 9:     for each child $v$ of $u$ in DT do
10:         for each predecessor $w$ of $v$ do
11:             if dominator tree parent of $v$ is $w$ then
12:                 $\gamma_v \leftarrow \text{MERGE}(\gamma_v), \text{CreateEdgeExpr}(w \to v)$
13:             else
14:                 $(\phi, p_w) \leftarrow \text{EVAL}(w)$
15:                 $\mathsf{NeedsPhi}[v] \leftarrow \mathsf{NeedsPhi}[v] \vee \phi$
16:                 $r_w \leftarrow \text{FIND}(w)$
17:                 $P \leftarrow p_w \cdot \text{CreateEdgeExpr}(w \to v)$
18:                 if $v$ dominates $w$ then                     ▷ Back-edge $w \to v$
19:                     $\mu_v \leftarrow \text{MERGE}(\mu_v, P)$
20:                 else                                      ▷ Cross or forward edge
21:                     Add (block of $r_w, P$) to $\mathsf{ListP}[v]$
22:                 end if
23:             end if
24:         end for
25:     end for
26:                                                              ▷ MERGE PHASE
27:     $changed \leftarrow$ true
28:     while $changed$ do
29:         $changed \leftarrow$ false
30:         for each child $v$ of $u$ in DT do
31:             for each $(b_r, P)$ in $\mathsf{ListP}[v]$ do          ▷ Iterate over cross-paths
32:                 $P' \leftarrow \gamma_{b_r} \cdot P$                 ▷ Prepend path from subroot's block
33:                 $\gamma'_v \leftarrow \gamma_v$
34:                 $\gamma_v \leftarrow \text{MERGE}(\gamma_v, P')$
35:                 if $\gamma_v \neq \gamma'_v$ then $changed \leftarrow$ true
36:                 end if
37:             end for
38:         end for
39:     end while
40:                                                               ▷ LINK PHASE
41:     for each child $v$ of $u$ in DT do
42:         $\mathsf{R}[v] \leftarrow \gamma_v$
43:         $\mathsf{Parent}[v] \leftarrow u$
44:     end for
45: end for
46: return $\{\ (\gamma_v, \mu_v)\ \mid\ v \in V\ \}$
```

Once path expressions have been computed, Algorithm 7 extracts the concrete LLVM predicates. For $\lambda(b)$, the gates are inherited from the unique predecessor of b. For edge(b, i), b's terminator is conditional, and is recorded as a controlling predicate. Union and concatenation of path expressions correspond to set union of gates. The result is that, for any block v with ϕ-functions, the final predicate set is the union of the gates extracted from both γ_v and μ_v.

Algorithm 7 Gate extraction for ϕ-functions

```
1:  function CollectGates($p$, Gates, VisitedExprs)
2:      if $p = \varnothing$ or $p \in$ VisitedExprs then return
3:      end if
4:      Add $p$ to VisitedExprs
5:      switch type of $p$:
6:      case edge($T_b, i$):
7:      if $T_b$ is a conditional branch or switch instruction then
8:          Add $T_b$ to Gates (as a set)
9:      end if
10:     case $p_1 \cup p_2$ or $p_1 \cdot p_2$:
11:     CollectGates($p_1$, Gates, VisitedExprs)
12:     CollectGates($p_2$, Gates, VisitedExprs)
13:     case $\lambda(b)$:
14:     CollectGates($\gamma_b$, Gates, VisitedExprs)      ▷ Collect from predecessor's path
    expression
15:     Remove $p$ from VisitedExprs
16: end function
17:
18: function GetGatesForAllBlocks
19:     Let $AllGates$ be a map from blocks to sets of gates
20:     for all $v \in V$ with predecessors do
21:         $G_v \leftarrow \varnothing$;    VisitedExprs $\leftarrow \varnothing$
22:         CollectGates($\gamma_v$, $G_v$, VisitedExprs)
23:         CollectGates($\mu_v$, $G_v$, VisitedExprs)
24:         $AllGates[v] \leftarrow G_v$
25:     end for
26:     return $AllGates$
27: end function
```

In practice, these gates serve as the link between symbolic path summaries and the actual instructions in the IR. The program slicer queries the GSA mappings to retrieve the per-block predicate sets, which are then attached to ϕ-functions. This ensures that the slice preserves the exact control predicates determining which incoming value is selected at each ϕ-function.

From a correctness perspective, any feasible CFG path π into v without a back-edge contributes a conjunct in γ_v symbolizing π, while paths re-entering v from a dominated region contribute to μ_v. Because the CollectGates procedure distributes over union and concatenation and records the precise conditional terminators along these paths, the resulting set $AllGates[v]$ conservatively over-approximates all predicates that can influence the incoming edge choice of a ϕ-function in v.

3.1.2 Slice Identification

This section specifies the properties of program slices that are candidates for outlining. We then present a formalization of the single-step traversal algorithm used for their identification and detail the precise conditions under which a slice can be extracted into a new function.

Our implementation operates as a pass over an entire LLVM module. It iterates through all functions, and upon visiting a binary instruction, it designates that instruction as the slicing criterion. This criterion sources the single-step dependency traversal algorithm to identify an extractable code region. If outlining succeeds, the parent function is modified to call the new slice, and the pass then recursively analyzes the body of the new function for further outlining opportunities, making the creation of recursive slices possible.

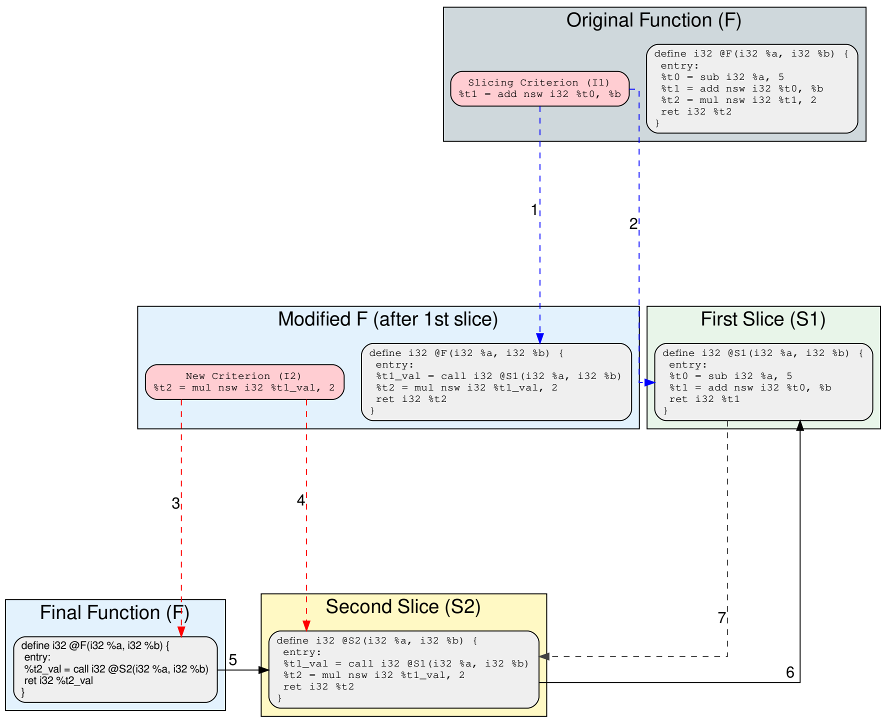
Example 3.1.1 shows how our approach create slices that have calls to other slices.

Example 3.1.1. Figure 3.2 illustrates the process of creating **recursive slices**, with its labeled edges highlighting each step, and instructions described in LLVM IR syntax. The process begins with the *Original Function (F)*, where the instruction that defines variable `%t1` is selected as the first **slicing criterion**. **Step 1** marks this instruction, and **Step 2** outlines its backward slice into a new function, `S1`, while rewriting `F` to call it.

The process then repeats on the modified function. In **Step 3**, a new criterion, `%t2`, is identified within the updated `F`. **Step 4** extracts its corresponding slice into a second function, `S2`. The resulting control flow is shown in **Steps 5 and 6**: the original function `F` now calls `S2`, which in turn calls `S1`.

Finally, **Step 7** indicates the data dependency of the multiplication inside `S2` on the value returned by `S1`, which completes the chain of a slice being derived from another slice.

Figure 3.2: The creation process of recursive slices.



```
%t1 = add nsw i32 %t0, %b
```

```
define i32 @F(i32 %a, i32 %b) {
 entry:
 %t0 = sub i32 %a, 5
 %t1 = add nsw i32 %t0, %b
 %t2 = mul nsw i32 %t1, 2
 ret i32 %t2
}
```

```
%t2 = mul nsw i32 %t1_val, 2
```

```
define i32 @F(i32 %a, i32 %b) {
 entry:
 %t1_val = call i32 @S1(i32 %a, i32 %b)
 %t2 = mul nsw i32 %t1_val, 2
 ret i32 %t2
}
```

```
define i32 @S1(i32 %a, i32 %b) {
 entry:
 %t0 = sub i32 %a, 5
 %t1 = add nsw i32 %t0, %b
 ret i32 %t1
}
```

```
define i32 @F(i32 %a, i32 %b) {
 entry:
 %t2_val = call i32 @S2(i32 %a, i32 %b)
 ret i32 %t2_val
}
```

```
define i32 @S2(i32 %a, i32 %b) {
 entry:
 %t1_val = call i32 @S1(i32 %a, i32 %b)
 %t2 = mul nsw i32 %t1_val, 2
 ret i32 %t2
}
```

Source: The author.

Single-step Traversal Algorithm

This subsection formalizes the single-step data dependence traversal algorithm implemented in our pass. Given a slice criterion instruction I, the procedure performs a bounded def-use expansion. The expansion collects immediate operand dependencies of I and transitively those of newly discovered dependencies, records basic blocks that can influence I either by defining those dependencies or by controlling ϕ-functions, and classifies out-of-scope operands as effective *function arguments* of the slice. In addition, control dependencies at ϕ-functions are injected from the per-block predicate map computed by the GSA mappings.

To establish the setting, let I be the slice criterion instruction in basic block $\mathsf{bb}(I)$, and let $\mathcal{BB}$ be the set of basic blocks and V be the set of program variables. A mapping is defined as $\mathsf{predicates} : \mathcal{BB} \to V^*$, where V^* denotes the set of all finite sequences (lists) of elements from V (Algorithm 6). Let $\mathcal{L}$ denote the loop containing I (if any) with header H. The algorithm returns three sets: deps, the visited data dependencies; BBs, the basic blocks that define those dependencies or serve as incoming predecessors for encountered ϕ-functions; and funcArgs, the operands are treated as formal parameters of the slice. The classification of operands into expansion targets or function arguments is handled by a scope guard. The predicate $\textsc{OutsideOrHeader}(v, \mathcal{L}, H)$ (Algorithm 8) decides whether an operand v should terminate expansion and instead be recorded as a boundary argument: values defined outside $\mathcal{L}$ or ϕ-functions in the loop header are not expanded further.

The traversal proceeds breadth-first using a FIFO queue, with a Visited set preventing redundant expansions. Each dequeued value x is added to deps. If x is an instruction, its defining block bb(x) is added to BBs, which therefore conservatively accumulates blocks that may need to be retained or cloned in the slice. Operand processing (Algorithm 9) is governed by three filters: (i) globals are rejected, since slices do not track global memory; (ii) type and visitation checks skip non-instruction values and duplicates; and (iii) the scope guard prevents traversal beyond loop boundaries. The first and second filters are omitted from the pseudocode in Algorithm 10 to improve readability.

When the dequeued value x is a `PHINode`, the algorithm records all incoming blocks of x in BBs and injects the associated gating predicates from predicates. Each predicate is considered for expansion unless it lies outside the current loop or in the header, or if it resides in the same block as the slice criterion. This constraints ensures that data selection at ϕ-functions is coupled with the control conditions selecting the incoming edge, thereby preserving the logical predicate instructions of the original program.

By construction, deps collects every value reachable by the one-step traversal under these rules, BBs conservatively accumulates defining and incoming blocks, and funcArgs identifies the loop boundary cut set. Because memory beyond direct operands is not chased, alias analysis and deeper memory reasoning remain out of scope for this phase. The injected control dependencies guarantee soundness with respect to the GSA-based predicate over-approximation.

Finally, the traversal algorithm performs a search analogous to a Breadth-First Search over the dependency subgraph of the slice criterion. Let this subgraph be $G_S = (V_S, E_S)$, where V_S is the set of variables in the slice (deps), and E_S is the set of directed edges where the target node is a variable that uses the source node variable.

Due to the Visited set, each vertex in V_S is enqueued and processed exactly once. The work performed at each vertex involves iterating over its outgoing edges (operands for instructions, predicates for ϕ-functions). Therefore, the time complexity is linear in the size of the subgraph, i.e., $O(|V_S| + |E_S|)$. The space required is dominated by the Worklist and Visited sets, leading to a space complexity of $O(|V_S|)$. In practice, the bounded nature of the slice yields a runtime that is nearly linear in the number of instructions within the slice.

Algorithm 8 OutsideOrHeader predicate

```
1: function OutsideOrHeader($v, \mathcal{L}, H$)
2:     if $\mathcal{L} = \varnothing$ or $\mathcal{L}$ is invalid then
3:         return false
4:     end if
5:     if $v$ is a $\phi$-function then
6:         if bb($v$) $= H$ then
7:             return true                          ▷ $\phi$-function in loop header
8:         end if
9:         if bb($v$) $\notin \mathcal{L}$ then
10:            return true                          ▷ $\phi$-function outside loop
11:        end if
12:    else if $v$ is an Instruction then
13:        if bb($v$) $\notin \mathcal{L}$ then
14:            return true                          ▷ instruction outside loop
15:        end if
16:    end if
17:    return false
18: end function
```

Algorithm 9 ProcessOperand helper

```
1: function ProcessOperand($v$)
2:     if $v \in$ Visited then
3:         return true
4:     end if
5:     insert $v$ into Visited
6:     if OutsideOrHeader($v, \mathcal{L}, H$) then
7:         insert $v$ into funcArgs
8:         return true
9:     end if
10:    enqueue $v$ into Worklist
11:    return true
12: end function
```

Algorithm 10 Single-step data-dependence traversal

```
Input: slice criterion instruction $I$; predicate map $\mathsf{predicates}$; loop $\mathcal{L}$ (possibly $\varnothing$) with
    header $H$
Output: triplet $(\mathsf{BBs}, \mathsf{deps}, \mathsf{funcArgs})$
 1: $\mathsf{deps} \leftarrow \emptyset$; $\mathsf{BBs} \leftarrow \emptyset$; $\mathsf{funcArgs} \leftarrow \emptyset$
 2: $\mathsf{Visited} \leftarrow \{I\}$; $\mathsf{Worklist} \leftarrow \langle I \rangle$          ▷ FIFO worklist
 3: while Worklist not empty do
 4:     $x \leftarrow$ pop front of Worklist
 5:     insert $x$ into deps
 6:     if $x$ is an Instruction then
 7:         insert $\mathsf{bb}(x)$ into BBs
 8:         for all operands $u$ of $x$ do
 9:             if not ProcessOperand($u$) then
10:                 break
11:             end if
12:         end for
13:     end if
14:     if $x$ is a $\phi$-function then
15:         for all incoming blocks $B$ of $x$ do
16:             insert $B$ into BBs
17:         end for
18:         for all $p \in \mathsf{predicates}[\mathsf{bb}(x)]$ do
19:             if $p \notin \mathsf{Visited}$ and $p$ is Instruction then
20:                 if OutsideOrHeader($p, \mathcal{L}, H$) then
21:                     continue
22:                 end if
23:                 insert $p$ into Visited
24:                 enqueue $p$ into Worklist
25:             end if
26:         end for
27:     end if
28: end while
29: return $(\mathsf{BBs}, \mathsf{deps}, \mathsf{funcArgs})$
```

Outline Restrictions

A slice is deemed unsuitable for outlining if it fails preliminary legality checks, contains complex control flow such as `try-catch` blocks, or violates specific heuristic thresholds. We establish constraints on the minimum and maximum number of instructions in the slice, as well as a limit on the number of parameters that would be required for the new function, ensuring that the overhead of the function call does not outweigh the benefits of outlining.

Furthermore, a slice cannot be outlined if it introduces memory-related inconsistencies or undefined behavior. This is determined by several critical conditions:

1. The slice contains stack allocations (`alloca` instructions) that are modified elsewhere in the parent function, creating potential for memory corruption.

2. Any load instructions within the slice target memory locations that could be clobbered by other instructions in the parent function, thus violating data dependency rules.

3. The slice includes calls to functions with unknown side effects, such as indirect calls or external functions that are not known to be read-only. However, calls to outlined functions created by *Daedalus* are allowed.

4. Any instruction within the slice cannot guarantee its return, which would disrupt the control flow of the parent function. This includes instructions that might throw exceptions, trap, or lead to abnormal program termination.

3.1.3 Idempotent Backward Slice

Definition 9 (Idempotent Backward Slice)**.** Given a program P and a slicing criterion c specified by an instruction I_c (which may be located within a loop $\mathcal{L}$ with header H), an **Idempotent Backward Slice** is an executable subprogram composed of input parameters and all instructions required to compute the value of c.

This computation follows two primary modes of interaction with the loop $\mathcal{L}$: It can be restricted to a single conceptual iteration of $\mathcal{L}$, which computes c directly (in this case, the loop header acts to limit c's computation); or, alternatively, c may depend on data produced by a variable updated within the loop (in which case the loop itself, or relevant portions thereof, is considered a dependency necessary for c's computation).

The components of the slice are determined by the single-step traversal (Algorithm 10) of P's CFG. First, the set I_S is the set of instructions transitively required to compute the value defined by I_c, within the possible bounds of the loop $\mathcal{L}$. This set corresponds to the instructions found in the deps output of the traversal algorithm:

$$I_S = \{i \in \mathsf{deps} \mid i \text{ is an instruction}\}$$

Second, the input parameters V_{in} are variables used by instructions in I_S and by definitions that lie outside $\mathcal{L}$'s boundary. This set corresponds to the funcArgs output of the algorithm:

$$V_{in} = \{v \in \mathsf{funcArgs} \mid v \text{ is a variable}\}$$

The resulting slice S is a subprogram P_S whose body contains the instructions I_S and whose parameters are the variables in V_{in}. This program computes the value of c and is called as a function inside the original slice criterion instruction I_c in P.

3.1.4 Function Outlining

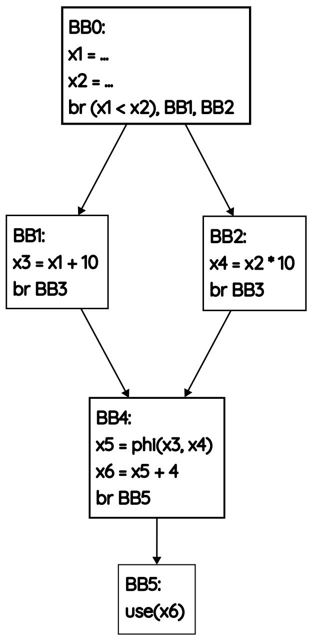
This subsection formalizes the function outlining phase, a powerful optimization technique that automatically extracts a region of code into a new, standalone function. The primary goal is to replace a segment of computation with a simple, equivalent function call. This process begins with the results of our single-step data-dependence traversal, which provides the necessary basic blocks (BBs), value dependencies (deps), and the set of required function arguments (funcArgs).

The entire outlining procedure can be understood as a sequence of four main steps, which we will illustrate with figures.

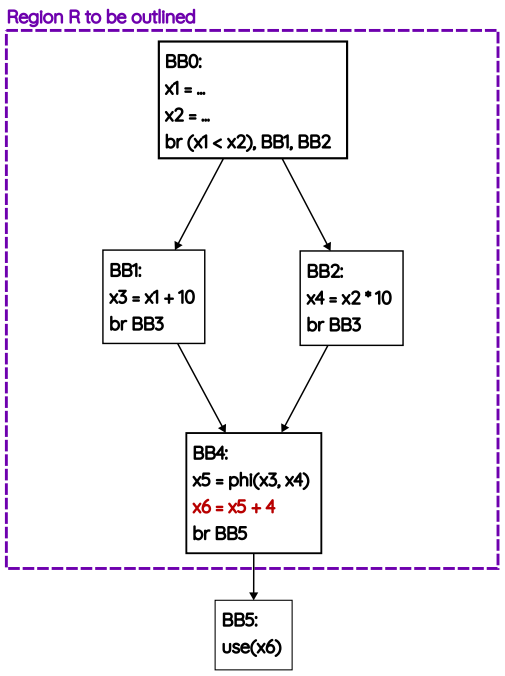
Step 1: Identify the Slice and Region. The process starts by targeting a specific instruction, known as the *slice criterion* `x6`. Figure 3.3 illustrates this initial step. We begin with the original CFG, where the instruction `x6` resides (Figure 3.3a). A dependency analysis is then performed, tracing backwards from `x6` to identify all the basic blocks and values that contribute to its result. This collection of blocks (V_R) and their induced CFG edges (E_R) define the *region* $R = (V_R, E_R)$ to be outlined, as visually isolated in Figure 3.3b.

Figure 3.3: Identification of the slice and the corresponding region to be outlined.

(a) The original function.



(b) The region R on the original function.



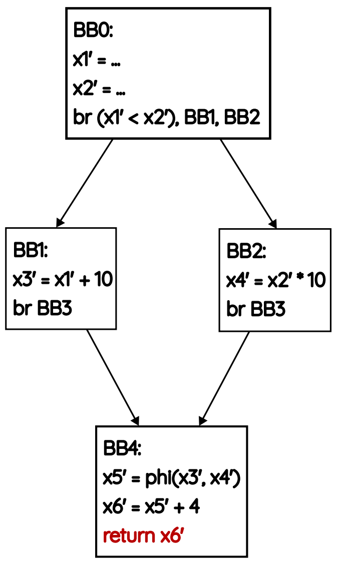
Source: The author.

Step 2: Verify Legality and Define the Interface. Before modifying the code, the pass verifies that the identified region is safe to extract. Outlining is permitted only if the region is *pure*, which imposes three key conditions: the new function must be self-contained, has no external side effects, and its output depends solely on its inputs.

Once these conditions are met, the interface for the new function, F_{slice}, is constructed. Its arguments (funcArgs) are derived from values that are used inside the region R but defined externally. The return type of F_{slice} is set to match the type of the instruction `x6`.

Step 3: Outline the New Function. With the interface defined, the cloning and repair stage begins. A new, empty function F_{slice} is created, as shown in Figure 3.4. The basic blocks from region R are then cloned into this new function. During this process, a value map is used to remap operands: any dependencies on external values are replaced by references to the new function's formal parameters ($\vec{p}$). Internal control flow, including predicates for ϕ-functions, is preserved within the cloned blocks. Finally, a single `ret` instruction is added to return the cloned value `x6'` (the equivalent of the original instruction `x6`).

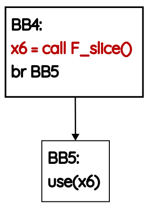
Figure 3.4: The outlined function.



Source: The author.

Step 4: Replace the Region with a Call. In the final step, the pass modifies the original function. The entire region of blocks R is removed and replaced by a single `call` instruction to the newly created F_{slice}. The values that were previously identified as external dependencies ($\vec{p}$) are passed as arguments to this call. As shown in Figure 3.5, this dramatically simplifies the original function's CFG, abstracting away the complex computation into a clean and reusable function.

Figure 3.5: The new original function.



Source: The author.

First Dominator

To correctly reconstruct the control flow graph (CFG) of an outlined slice, it is essential to manage branches inside the slice. Our approach relies on the concept of dominance to identify the correct successor block for such branches. Let a CFG be a directed graph $G = (V, E)$, where V is a set of basic blocks and E is a set of edges representing control flow. Let $S \subseteq V$ be the subset of blocks included in the program slice. First, we define a key concept for our branch rerouting logic.

Definition 10 (First Dominator in Slice)**.** For any block $v \in V$, the *first dominator* in the slice is the closest strict dominator of v that is also a member of the slice's blocks set S. This is found by traversing up the dominator tree from v and selecting the first node encountered that belongs to S.

The branch rerouting procedure is applied to any edge $(b, s) \in E$ where the source block $b \in S$ is part of the slice, but the successor block $s \notin S$ is not. The objective is to find a new target block t for the branch from b within the reduced CFG. The algorithm proceeds as follows:

1. Initiate a forward traversal of the original CFG starting from the successor block s.
2. Avoid revisiting nodes already seen during traversal. The source block b is skipped to prevent traversing loops.
3. If the slice criterion is inside a loop, apply the following restrictions:
    a) Skip traversal from the loop header, to avoid traversing invalid paths.
    b) Skip blocks that are outside the current loop, preserving loop consistency and preventing traversal beyond the loop's scope.
4. Search for a reachable block t such that its first dominator in the slice is the source block b; or the t's first dominator also dominates b.
5. If such a block t is found, reroute the exiting branch from b to target t in the new sliced CFG.
6. If traversal completes and no such block exists, reroute the branch from b to a terminal block representing an unreachable or exit condition, preserving the CFG's integrity.

This method ensures that control flow leaving the slice is redirected to the correct program point that is post-dominated by the exit block of the slice, while also respecting loop structure and maintaining program semantics.

3.2 Function Merging and Simplification

To eliminate redundancy among the newly created slices, we first employ the LLVM *mergefunc* pass [13]. This pass identifies and merges semantically equivalent functions. For each set of identical slices, it establishes a single canonical version and updates all call sites accordingly. This process relies on a mapping from each deleted function to its new, canonical equivalent, ensuring that all references within the module remain valid.

Once the outlined functions have been merged, the transformation is finalized within the original parent functions. The instruction sequences that were extracted are now replaced with *call* instructions targeting the new canonical functions. To conclude the process, we execute the *simplifyCFG* pass [14]. This final step removes any basic blocks or control flow structures that became redundant as a result of the outlining, ensuring the resulting functions are as compact and efficient as possible.

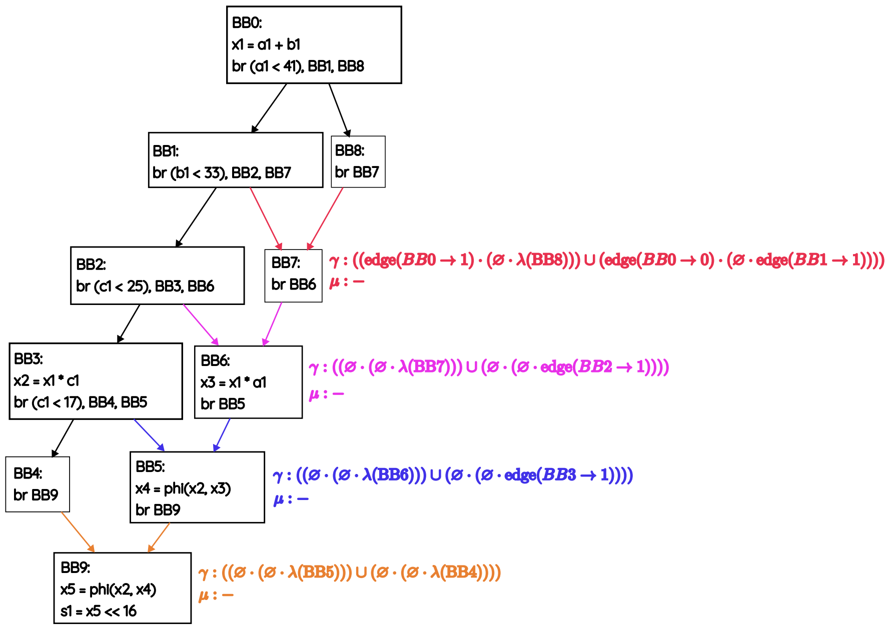
3.3 Slicing Example

In contrast to *Wyvern*, we first map all gating functions as regular expressions, making predicates explicit to our single-step data traversal algorithm. This ensures that no implicit control dependency is overlooked, which makes it possible to outline a slice function for the ladder graph case. Furthermore, we identified key limitations in the `first dominator` implementation from prior work (Section 2.3.2). To address these deficiencies, we propose and show a more robust algorithm working, from Section 3.1.4.

Let the program from Section 2.5 be the input to our algorithm. Following the steps illustrated in Figure 3.1, we first normalize the program into LCSSA form using the *mem2reg* and *lcssa* LLVM passes. In this example, the program is already in SSA.

For every merge point in the program, we apply Algorithm 6 to construct a regular expression of its corresponding gating functions γ and μ. Figure 3.6 shows each basic block with a merge point, annotated with the relative path expression that represents its gating function. Colors indicate which paths are taken to reach each merge point. Finally, a mapping between basic blocks and their controlling predicates (the blocks' conditional terminators) is constructed. This mapping specifies which predicate governs the execution of each block, concluding the GSA Construction step.

Figure 3.6: The ladder graph in GSA form.



Source: The author.

As in *Wyvern*, we assume `s1 = x5 << 16` is the slice criterion and proceed to the Slice Identification step. Our algorithm identifies both the data dependencies listed in 3.7a and, unlike *Wyvern*, the controlling predicates listed in 3.7b. It retrieves these predicates from the mapping generated by Algorithm 7. This complete dependency information is crucial, as it enables our corrected `first dominator` (Section 3.1.4) algorithm to properly reconstruct control flow, ensuring a branch from `BB0` correctly targets `BB1` instead of the erroneous successor `BB9`. A new function is then outlined, represented in Figure 3.8.

Figure 3.7: Slice criterion's dependencies.

(a) Data dependencies.

```
1. x5 = phi(x2, x4)
2. x2 = x1 * c1
3. x4 = phi(x2, x3)
4. x1 = a1 + b1
5. x3 = x1 * a1
```

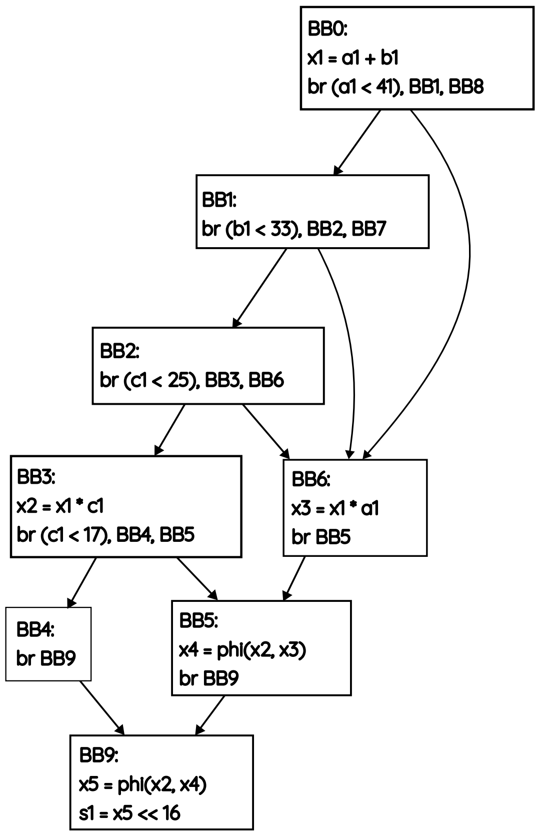
(b) Control dependencies.

```
1. br (c1 < 17), BB4, BB5
2. br (c1 < 25), BB3, BB6
3. br (c1 < 33), BB2, BB7
4. br (a1 < 41), BB1, BB8
```

Source: The author.

Finally, once all outlined functions are identified, the pass attempts to merge similar ones and simplify them. This step is carried out by leveraging the *mergefunc* and *simplifycfg* LLVM passes within our implementation.

Figure 3.8: Final sliced function.



Source: The author.

Chapter 4

Soundness

This chapter provides a formal argument for the soundness of our program slicing algorithm. We aim to prove that an extracted *Idempotent Backward Slice* (Definition 9) is semantics-preserving with respect to its slicing criterion. That is, for any given program input, the value computed for the criterion variable by the slice is identical to the value computed by the original program.

To formalize the semantics-preserving property, we first introduce a minimal intermediate language that explicitly includes the Gated Single-Assignment (GSA) gating functions central to our slicing algorithm. We then define its structural operational semantics. Finally, we proceed to state and prove the soundness theorem by induction.

4.1 MiniGSA: A Minimal GSA Language

This section defines a minimal language called *MiniGSA*, which contains the basic syntax necessary to explain the semantics of the Gated Static Assignment format.

4.1.1 Syntax

A program P is a map from labels L to basic blocks B, and program termination is handled by the stop instruction $\mathcal{I}_{stop}$.

$$
\begin{array}{lccl}
\text{Variables} & v & \in & \text{Var} \\
\text{Labels} & L & \in & \text{Label} \\
\text{Values} & z & \in & \text{Value} \\
\text{Operators} & \oplus & \in & \{+, -, *, /, \dots\} \\
\text{Restart Set} & R & \subseteq & \text{Var} \\
\text{Standard Instr} & \iota & ::= & v_0 = v_1 \oplus v_2 \\
 & & \mid & v_0 = \gamma(p, v_{\text{true}}, v_{\text{false}}) \\
 & & \mid & v_0 = \mu(v_{\text{init}}, v_{\text{loop}}) \\
 & & \mid & v_0 = \eta(p, v_{\text{exit}}, R) \\
\text{Control Flow} & \tau & ::= & \text{br } L \mid \text{br } p, L_{\text{true}}, L_{\text{false}} \mid \mathcal{I}_{stop} \\
\text{Basic Blocks} & B & ::= & \iota; B \mid \tau \\
\text{Programs} & P & ::= & L \mapsto B, \dots
\end{array}
$$

4.1.2 Semantics

We define a structural operational semantics. A program configuration is a tuple $\langle \mathcal{I}; B, \sigma \rangle$, where $\mathcal{I}$ is the sequence of instructions from the current basic block yet to be executed, and σ is the store. The single-step transition relation is $P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}; B, \sigma \rangle \to \langle \mathcal{I}', \sigma' \rangle$. We assume a store σ maps a variable v to a value $\sigma[v]$, or the uninitialized marker $\perp$.

Instruction Semantics. These rules define the execution of standard instructions. They operate within the current sequence of instructions $\mathcal{I}$, until it reaches the stop instruction $\mathcal{I}_{stop}$.

$$\frac{z = \sigma[v_1] \oplus \sigma[v_2]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = v_1 \oplus v_2; B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z]\rangle}$$

$$\frac{\sigma[p] = \text{True} \quad z = \sigma[v_{\text{true}}]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = \gamma(p, v_{\text{true}}, v_{\text{false}}); B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z]\rangle}$$

$$\frac{\sigma[p] = \text{False} \quad z = \sigma[v_{\text{false}}]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = \gamma(p, v_{\text{true}}, v_{\text{false}}); B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z]\rangle}$$

$$\frac{\sigma[v_0] = \bot \quad z = \sigma[v_{\text{init}}]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = \mu(v_{\text{init}}, v_{\text{loop}}); B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z]\rangle}$$

$$\frac{\sigma[v_0] \neq \bot \quad z = \sigma[v_{\text{loop}}]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = \mu(v_{\text{init}}, v_{\text{loop}}); B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z]\rangle}$$

$$\frac{\sigma[p] = \text{False} \quad z = \sigma[v_{\text{exit}}]}{P, \mathcal{I}_{stop} \vdash \langle v_0 = \eta(p, v_{\text{exit}}, R); B, \sigma\rangle \to \langle B, \sigma[v_0 \mapsto z, \forall r \in R.\ r \mapsto \bot]\rangle}$$

The η function is defined with an additional set R of variables, which we denote $R \subseteq \text{Var}$. This set R contains loop-dependent variables (i.e., μ-defined variables) that must be effectively *reset* to $\bot$ upon loop exit. This explicit resetting of μ-defined variables ensures that if the loop is re-entered, the relevant μ nodes will correctly re-select their initial value v_{init}, thereby making the variables reusable and preserving the required single-assignment property for subsequent loop iterations in the overall program flow. Formally, upon $p = \text{False}$, the rule simultaneously assigns the exit value z to v_0 and maps every variable $r \in R$ to the uninitialized value $\bot$ in the store: $\sigma[v_0 \mapsto z, \forall r \in R.\ r \mapsto \bot]$.

Control Flow and Termination Semantics. These rules define control transfer. They update the current sequence of instructions and replace them with the instructions from the target basic block. When the stop instruction is reached, the program state is its output store σ.

$$\frac{P(L_{next}) = B}{P, \mathcal{I}_{stop} \vdash \langle \text{br } L_{next}, \sigma\rangle \to \langle B, \sigma\rangle} \qquad \frac{\sigma[p] = \text{True} \quad P(L_{\text{true}}) = B_{\text{true}}}{P, \mathcal{I}_{stop} \vdash \langle \text{br } p, L_{\text{true}}, L_{\text{false}}, \sigma\rangle \to \langle B_{\text{true}}, \sigma\rangle}$$

$$\frac{\sigma[p] = \text{False} \quad P(L_{\text{false}}) = B_{\text{false}}}{P, \mathcal{I}_{stop} \vdash \langle \text{br } p, L_{\text{true}}, L_{\text{false}}, \sigma\rangle \to \langle B_{\text{false}}, \sigma\rangle} \qquad \frac{}{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_{stop}; B, \sigma\rangle \to \sigma}$$

4.2 Soundness

Definition 11 (Transitive Closure for Transition Relation)**.** The **transitive closure** $\rightarrow^*$ for the transition relation $\rightarrow$ over program configurations $s = \langle \mathcal{I}; B, \sigma \rangle$ is the smallest relation satisfying the following inference rules:

$$\frac{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_{stop}; B, \sigma \rangle \rightarrow \sigma}{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_{stop}; B, \sigma \rangle \rightarrow^* \sigma} \quad \text{(Termination)}$$

$$\frac{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}; B, \sigma \rangle \rightarrow \langle B, \sigma_0 \rangle}{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}; B, \sigma \rangle \rightarrow^* \langle B, \sigma_0 \rangle} \quad \text{(Base Case)}$$

$$\frac{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_0; B, \sigma_0 \rangle \rightarrow^* \langle \mathcal{I}_1; B, \sigma_1 \rangle \qquad P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_1; B, \sigma_1 \rangle \rightarrow^* \langle \mathcal{I}_2; B, \sigma_2 \rangle}{P, \mathcal{I}_{stop} \vdash \langle \mathcal{I}_0; B, \sigma_0 \rangle \rightarrow^* \langle \mathcal{I}_2; B, \sigma_2 \rangle} \quad \text{(Transitivity)}$$

Thus, the full execution of a program P starting from the entry point L_{entry} with initial store σ_{in}, and terminating at instruction $\mathcal{I}_{stop}$ with final store σ_{out}, is formally expressed using the transitive closure as:

$$P, \mathcal{I}_{stop} \vdash \langle L_{entry}, \sigma_{in} \rangle \rightarrow^* \sigma_{out}$$

Definition 12 (CFG of a Slice)**.** Let P be a program with CFG $G_P = (V_P, E_P)$. Let S be an Idempotent Backward Slice of P for a criterion c, as in Definition 9, with instruction set I_S. The control-flow graph of the slice subprogram P_S is $G_S = (V_S, E_S)$, where:

1. V_S is the set of basic blocks containing at least one instruction from I_S, plus any blocks synthesized during the materialization of the subprogram.

2. E_S is the set of edges (u, v) where $u, v \in V_S$ that are either preserved from G_P or are newly introduced to connect synthesized blocks.

Lemma 1 (Slice Entry as Dominator)**.** *Let S be an Idempotent Backward Slice with CFG $G_S = (V_S, E_S)$ (Definition 12), constructed by a backward dependency traversal from a criterion I_c in block B_c. This construction process identifies a unique block, $L^S_{entry} \in V_S$, that dominates every other block $B \in V_S$ within the slice CFG G_S.*

Proof. The slice's instruction set, I_S, is formed by the deps output of Algorithm 10. This algorithm performs a bounded, transitive traversal starting from the criterion I_c. It collects not only data dependencies (operands of instructions) but also control dependencies. This is achieved by querying the GSA predicates map (constructed by Algorithm 6 and 7) whenever a ϕ-function is encountered, adding the gating predicate instructions to the worklist for further traversal.

This process selects a subgraph G_S, from the original program's CFG G_P. This subgraph contains all blocks (BBs) necessary to host the instructions in I_S and all control-flow paths that enable the computation of the criterion. Because both data and control dependencies are transitively included, the resulting set of blocks and the edges between them form the subgraph G_S.

For G_S to constitute a valid computation, all such paths must originate from a common entry context. The backward traversal, by including necessary control-flow predecessors, ensures that all dependency paths eventually trace back to a common ancestral block within the slice. This block, which we denote L^S_{entry}, serves as the nexus for all control flow entering the slice's computation. By construction, every valid execution path from the beginning of the slice's computation to any arbitrary block $B \in V_S$ must necessarily pass through L^S_{entry}. This is the formal definition of dominance (L^S_{entry} dom B). This dominator is identified and used for CFG reconstruction via the *First Dominator in Slice* logic defined in Section 3.1.4.

Therefore, the construction process naturally identifies a unique block, L^S_{entry}, that dominates all other blocks within the slice's CFG. □

Theorem 1 (Every idempotent backward slice has a single entry block). *Let S be an Idempotent Backward Slice (Definition 9). Then the slice CFG $G_S = (V_S, E_S)$ has a single entry block.*

Proof. By Lemma 1, the backward dependency traversal used to construct the slice S yields a CFG G_S, containing a block L^S_{entry} that dominates all other blocks in V_S. In a control-flow graph, a block that dominates all other blocks is, by definition, the unique entry point of that graph. The existence of any other potential entry would imply a path to some block within the slice that does not pass through L^S_{entry}, which would violate its dominance property.

Consequently, L^S_{entry} is the single entry block of the slice.

Furthermore, the construction implies a dual property: the criterion's block, B_c, post-dominates the entry block L^S_{entry} within G_S. This is a direct consequence of the slice's purpose: every valid execution path starting from the entry *must* eventually reach the criterion for the computation to be meaningful. □

Corollary 1 (When does the slice entry equal the program entry?). *If the construction of P_S reuses the original entry block L_{entry} and retains all control required to reach every block in V_S from it, then $L^S_{entry} = L_{entry}$. Otherwise, $L^S_{entry} \neq L_{entry}$ is permitted.*

Theorem 2 (Slicing Soundness). *Let P be a program, I_v be a slicing criterion (the instruction that defines variable v), and $S = \mathrm{Slice}(P, I_v)$ be the extracted slice with single* *entry L_{entry}. The slicing algorithm is sound if for any initial store σ_{in} and entry label L_{entry}, the following holds:*

$$\textit{If } P, I_v \vdash \langle L_{entry}, \sigma_{in}\rangle \rightarrow^* \sigma_P$$
$$\textit{then } S, I_v \vdash \langle L_{entry}, \sigma_{in}\rangle \rightarrow^* \sigma_S \textit{ and } \sigma_S[v] = \sigma_P[v].$$

This states that if the original program P terminates by reaching I_v with a final store σ_P, then the slice S also terminates by reaching I_v with a final store σ_S, and the value of the criterion c is identical in both final stores.

Proof. We prove this theorem by induction on the length of the execution trace of the original program P. Let the execution of P be a sequence of configurations $s_0, s_1, \ldots, s_n$ such that $s_0 = \langle L_{entry}, \sigma_{in}\rangle$ and $P, I_v \vdash s_k \rightarrow s_{k+1}$ for $0 \leq k < n$, resulting in a final store σ_P.

Let S be the slice of P. We define a corresponding execution trace for S where we only consider the instructions present in S. Let σ_P^k and σ_S^k be the stores after k steps of the original program and the corresponding steps in the slice, respectively. Let Defs(S) be the set of all variables appearing on the left-hand side of an assignment instruction within the slice S. This set represents all variables whose values are computed by the slice.

Our inductive hypothesis, $H(k)$, is: For any variable w defined within the slice, its value is the same in both stores after k steps. This includes the uninitialized value $\perp$. Formally:

$$\forall w \in \mathrm{Defs}(S), \quad \sigma_S^k[w] = \sigma_P^k[w]$$

Base Case ($k = 0$): The execution of both P and S starts with the initial store σ_{in}. For any $w \in \mathrm{Defs}(S)$, its initial value is $\sigma_{in}[w]$ (which may be $\perp$) in both configurations. Thus, $\sigma_S^0[w] = \sigma_P^0[w]$, and the hypothesis holds.

Inductive Step: Assume that the hypothesis $H(k)$ holds for some $k \geq 0$. We need to show that $H(k+1)$ also holds after the execution of the next instruction, ι, in the program P. Let this transition be $P, I_v \vdash \langle \iota; \ldots, \sigma_P^k\rangle \rightarrow \langle \ldots, \sigma_P^{k+1}\rangle$. We perform a case analysis on ι.

- **Case 1: $\iota \notin S$.** By definition, ι defines some variable $v' \notin \mathrm{Defs}(S)$. The slice execution does not change, so $\sigma_S^{k+1} = \sigma_S^k$. For any $w \in \mathrm{Defs}(S)$, we know $w \neq v'$, so its value is not affected by ι. Thus, $\sigma_P^{k+1}[w] = \sigma_P^k[w]$. By the inductive hypothesis, $\sigma_P^k[w] = \sigma_S^k[w]$. Combining these, we get $\sigma_P^{k+1}[w] = \sigma_S^{k+1}[w]$. The hypothesis holds.

- **Case 2:** $\iota \in S$**.** The instruction ι is executed in both P and S. By the inductive hypothesis, the values of all input variables to ι are identical in σ_P^k and σ_S^k, since the definitions of those input variables must also be in S and thus their variables are in $\mathrm{Defs}(S)$.
  - $\iota \equiv (v_0 = v_1 \oplus v_2)$: The definitions of operands v_1 and v_2 must be in S. By IH, $\sigma_P^k[v_1] = \sigma_S^k[v_1]$ and $\sigma_P^k[v_2] = \sigma_S^k[v_2]$. Since $\oplus$ is a deterministic operator, the result $z = \sigma_P^k[v_1] \oplus \sigma_P^k[v_2]$ is identical to $z' = \sigma_S^k[v_1] \oplus \sigma_S^k[v_2]$. Both stores are updated with the same value for v_0.
  - $\iota \equiv (v_0 = \gamma(p, v_{\text{true}}, v_{\text{false}}))$: The definition of the predicate p must be in S. By IH, $\sigma_P^k[p] = \sigma_S^k[p]$. If the predicate is true, the definition of v_{true} must be in S, and by IH, its value is the same in both stores. The same holds for v_{false} if the predicate is false. Thus, the γ function selects the same value in both executions.
  - $\iota \equiv (v_0 = \mu(v_{\text{init}}, v_{\text{loop}}))$: The choice between v_{init} and v_{loop} depends on whether $\sigma[v_0] = \bot$. Since v_0's definition (ι) is in S, $v_0 \in \mathrm{Defs}(S)$. By IH, $\sigma_P^k[v_0] = \sigma_S^k[v_0]$. Therefore, both executions make the same choice. The chosen variable (v_{init} or v_{loop}) must also have its definition in S, so by IH its value is also the same. The assignment to v_0 is identical.
  - $\iota \equiv (v_0 = \eta(p, v_{\text{exit}}, R))$: Similar to γ, the predicate p and operand v_{exit} must have their definitions in S. By IH, their values are identical in σ_P^k and σ_S^k. Both executions will assign the same value to v_0. The rule also resets variables in R to $\bot$. For any $r \in R$ that is also in $\mathrm{Defs}(S)$, its value becomes $\bot$ in both σ_P^{k+1} and σ_S^{k+1}, maintaining the equivalence.
  - $\iota \equiv (\text{br } p, L_{\text{true}}, L_{\text{false}})$: A conditional branch is included in S if it provides control dependence. The definition of p must be in S. By IH, $\sigma_P^k[p] = \sigma_S^k[p]$. Therefore, both P and S will branch to the same successor block, ensuring that the sequence of executed blocks from the slice is the same. The same logic applies to an unconditional branch.

  In all sub-cases where $\iota \in S$, the update to the store for any variable in $\mathrm{Defs}(S)$ is identical. The hypothesis $H(k+1)$ holds.

By induction, the hypothesis holds for all steps up to the termination of the program at the stop instruction I_v. Since the definition of v is the slicing criterion itself, its defining instruction is in S and $v \in \mathrm{Defs}(S)$. Therefore, at the final step, the value computed for the criterion variable v is the same in both stores: If $P, I_v \vdash \langle L_{entry}, \sigma_{in} \rangle \to^* \sigma_P$, then $S, I_v \vdash \langle L_{entry}, \sigma_{in} \rangle \to^* \sigma_S$ and $\sigma_S[v] = \sigma_P[v]$. □

Finally, we implemented an interpreter that is derived from the inference rules from Section 4.1.2. The code is presented on Appendix A.

Chapter 5

Evaluation

Having established the theoretical background, this chapter presents the empirical evaluation of our implementation. We begin by specifying our benchmark setup and experimental methodology. To assess the effectiveness of our approach, we compare its results against two key baselines: the `func-merging` pass introduced by Rocha et al. [18] and the standard `IROutliner` pass in LLVM 17. The data gathered from these comparisons is then used to address the following research questions, thereby demonstrating the feasibility of our work:

RQ1: How much code-size reduction can the outlining of idempotent slices achieve, and how does this result compare with techniques of similar goals?

RQ2: What is the impact of slice outlining on the running time of benchmarks, and how does this impact compare with previous work?

RQ3: What is the overhead that slice outlining adds to the compilation pipeline, and how does this overhead compare with previous work?

RQ4: What is the asymptotic behavior of the slice outlining algorithm?

RQ5: What is the time taken by the different phases of the outlining optimization proposed in this paper?

RQ6: Can we observe a cumulative benefit of running the different code-size optimizations in combination?

5.1 Experimental Setup

The experiments consisted of running three transformation passes over the LLVM Test Suite to collect statistics in four categories of metrics: number of LLVM instructions (`Instcount`), size of the `.text` segment of the executable, execution time and compilation time. For this purpose, a dedicated machine was selected, a patch for the test suite was developed, and all baseline configurations were prepared. Once the benchmark environment was ready, we compiled 2007 programs using `func-merging` and `IROutliner`, and compared their results against those obtained with `Daedalus`. Our results were obtained using the parameter settings defined in our cost model experiment.

Hardware

The experiments were conducted on a server provided by the Compilers Laboratory at DCC/UFMG, with the following configuration:

- **CPU:** AMD Ryzen Threadripper 7970X 32-Cores 4GHz
- **Memory:** 128 GiB RAM

This hardware was chosen primarily for its high core count, which allowed the experiments to be executed efficiently in a multi-threaded setting.

Benchmark Environment

The experiments were conducted using baselines that required a different build and configuration of LLVM 17. The choice of this version was motivated by compatibility with `func-merging`, by the stability of `IROutliner`, and by the modifications necessary to support `Daedalus`. The benchmark environment for each baseline is summarized as follows:

- `func-merging` **baseline**
  - Implemented as an LLVM 17 patch used by the original authors of `func-merging`.
- `IROutliner` **baseline**
  - Implemented as an upstream LLVM 17 pass.
  - Stable across LLVM versions, ensuring consistent behavior across builds.
- `Daedalus`

– The `mergefunc` pass in LLVM 17 was extended to support the core merge procedure for arbitrary sets of functions.

– Daedalus source code can be found at: `https://github.com/lac-dcc/Daedalus`.

The specific LLVM 17 build source code is available at `https://tinyurl.com/ye55ax9d`. Also, all experiments were executed on the selected hardware using shell scripts developed for this study, publicly available at `https://tinyurl.com/ye26drz6`.

To reproduce our experiments, one needs to build our artifact image using *docker*, with the *Dockerfile* located at `./artifact/docker/Dockerfile` within `Daedalus`'s repository.

Compilation Pipeline

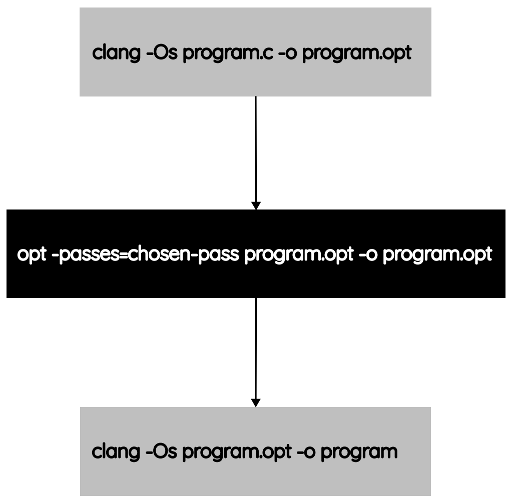
For each program in the test corpus, metrics were collected by compiling it with the `-Os` optimization flag, both before and after applying the selected pass, as illustrated in Figure 5.1. This procedure isolates the effect of the transformation, ensuring that any differences in the reported metrics are attributable solely to the chosen pass, thereby minimizing external noise.

This setup is not configured by default on LLVM Test Suite (version 17). To achieve this pipeline, the build system of the LLVM Test Suite was modified to include a post-build pipeline that extracts, transforms, and recompiles each program in order to evaluate the effect of an arbitrary pass. The sequence of steps integrated into the CMake configuration is illustrated in Figure 5.2.

Programs are first compiled with Link Time Optimization (LTO) [16] flags, which embed their fully linked bitcode into the target executable. The resulting binary is then processed with `objcopy` to extract the embedded `.llvmbc` section as a standalone `.bc` file. This bitcode is normalized using canonicalization passes (`mem2reg,lcssa`) to ensure it is in LCSSA form and suitable for further analysis. The selected pass is subsequently applied with optional arguments, producing a transformed version of the bitcode. The optimized bitcode is recompiled into a native executable with the `-Os` flag, and finally, metrics are collected. Finally, the Test Suite patch is available on `https://tinyurl.com/ye2a9ypt`.

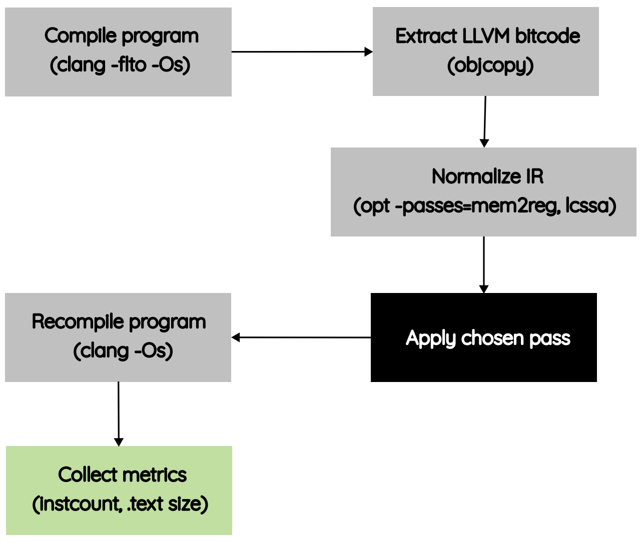
The main motivation for recompiling the programs using this method is to leverage the existing configuration of the test suite subprojects. Each subproject already defines a specific set of compilation flags, which makes individual modifications difficult to apply. By operating at this level, our approach becomes scalable and simplifies the evaluation of arbitrary passes.

Figure 5.1: Compilation pipeline.


clang -Os program.c -o program.opt
opt -passes=chosen-pass program.opt -o program.opt
clang -Os program.opt -o program

Source: The author.

Figure 5.2: Build system modification steps.


Compile program (clang -flto -Os)
Extract LLVM bitcode (objcopy)
Normalize IR (opt -passes=mem2reg, lcssa)
Apply chosen pass
Recompile program (clang -Os)
Collect metrics (instcount, .text size)

Source: The author.

Cost Model Experiment

To achieve the greatest possible code-size reduction, we conducted a cost model experiment in which we limited the number of arguments, instructions, and users of an outlined function. These checks are performed during the Slice Identification step described in Section 3.1.2.

The cost model experiment involved compiling the entire LLVM Test Suite multiple times while varying the three parameters and their combinations. The range for the number of arguments was $[0, 20]$, for the number of instructions $[10, 20, 40, 80, 160]$, and for the number of users $[10, 20, 40, 80, 160, 320, 640]$.

After recompiling and collecting the geometric means of the metrics across 735 runs, we concluded that the greatest reduction in the *Instcount* metric was achieved with outlined functions containing at most one argument, no more than 20 instructions, and at most 10 users.

5.2 Research Questions

The experimental results were analyzed and summarized using tables and graphs. Each table reports the number of programs affected and the corresponding geometric mean, where positive percentages indicate an increase in a given metric and negative percentages indicate a decrease. Table 5.1a details the instances where metrics remained unchanged, while Table 5.1b summarizes the overall geometric mean for each of the selected metrics.

RQ1: Code-Size Reduction

This section evaluates the code-size reduction capabilities of our approach. The effectiveness of `Daedalus` is measured by its impact on the final executable's `.text` section size across the benchmark suite. The analysis reveals that while the technique can yield substantial rewards in specific cases, its overall effect is more complex than a straightforward reduction, highlighting a critical trade-off between analytical precision and general

Table 5.1: Experimental results across different metrics.

(a) Programs with unchanged metrics.

	Daedalus	Function Merging	IROutliner
Metric	**Count**	**Count**	**Count**
Instcount	1865	1824	1800
.text size	1874	1817	1847
Exec. Time	1846	1830	1847
Compile Time	1704	1766	1749

(b) Overall metrics.

	Daedalus		Function Merging		IROutliner	
Metric	**Total**	**Geomean**	**Total**	**Geomean**	**Total**	**Geomean**
Instcount	2007	-0.24%	2007	-0.35%	2007	-0.65%
.text size	2007	0.11%	2007	0.39%	2007	-0.19%
Exec. Time	2007	0.06%	2007	0.25%	2007	-0.09%
Compile Time	2007	4.22%	2007	2.06%	2007	2.48%

Source: The author.

applicability.

The evaluation reveals that `Daedalus` has a nuanced impact on code size. Overall, the technique resulted in a slight geometric mean increase of 0.11% across the 2007 programs in the test suite. This aggregate result, however, masks a significant trade-off visible in the detailed breakdown. Furthermore, an analysis across all metrics reveals that none of the evaluated passes were able to reduce every metric for any single program. Conversely, in a few specific cases, each pass increased all metrics, as outlined in Table 5.2.

Table 5.2: Number of Programs with All Metrics Positive.

Metric Name	Daedalus	func-merging	IROutliner
(+) Instcount (+) .text size (+) Exec. Time (+) Comp. Time	14	26	0

Source: The author.

The optimization is highly targeted, leaving the `.text` section size of 1874 programs entirely unchanged (Table 5.1a). This indicates that the specific, recurrent patterns of *Idempotent Backward Slices* that `Daedalus` identifies are not prevalent in most of the benchmark programs.

When the optimization was applicable, its effects were pronounced but mixed:

- **Effective Reductions:** In 23 programs, `Daedalus` achieved a substantial average `.text size` reduction of -8.39%, and -9.96% in `Instcount`. This demonstrates that for programs with suitable structures, the algorithm can be highly effective at compacting code.
- **Size Increases:** Conversely, the pass led to a size increase in a larger set of 57 programs, with an average code-size increase of 2.09% in `.text` size.

Tables 5.4 and 5.3 detail the number of programs where both the instruction count and `.text` section size metrics concurrently increased or decreased, respectively. A positive value indicates an increase, while a negative value indicates a reduction.

Table 5.3: Number of Programs with Negative Instcount and .text size.

Metric Name	**Daedalus**	**func-merging**	**IROutliner**
(-) Instcount (-) .text size	23	30	105
Diff Geomean	-8.39%	-10.72%	-4.65%

Source: The author.

Table 5.4: Number of Programs with Positive Instcount and .text size.

Metric Name	**Daedalus**	**func-merging**	**IROutliner**
(+) Instcount (+) .text size	57	56	10
Diff Geomean	2.09%	6.02%	3.37%

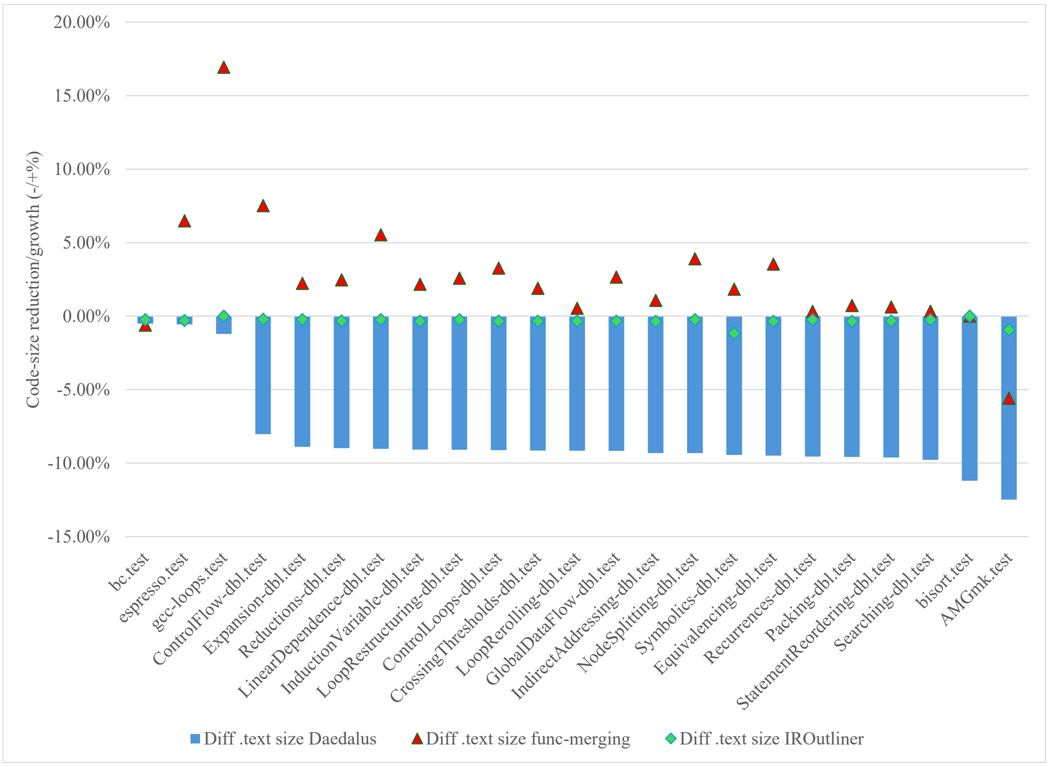
Source: The author.

When analyzing the programs that `Daedalus` successfully reduces, it consistently and significantly outperforms the baseline techniques. Figure 5.3 illustrates this trend across the programs where `Daedalus` achieved a code-size decrease. In nearly all these instances, `Daedalus` (blue bars) provides a substantial reduction, while `IROutliner` (green rhombus) offers only minimal savings, and `func-merging` (red triangles) frequently increases the code size.

A representative example of this performance gap is the `IndirectAddressing-dbl.test` program. For this test case:

- `Daedalus` achieved a remarkable code-size reduction of -9.32%.
- `IROutliner` provided a negligible reduction of only -0.35%.
- `func-merging` actually increased the code size by 1.07%.

Figure 5.3: Code-size reduction between Daedalus, func-merging, and IROutliner.



Source: The author.

This specific case highlights the core strength of our approach. While more broadly applicable algorithms failed to find meaningful savings, the precision of GSA-based slicing allowed `Daedalus` to identify and eliminate a significant redundancy.

This disparity is a direct result of the overhead from a new function's prologue, epilogue, and call-site instructions, which can easily negate the savings from outlining a small code fragment. To address this, our approach is guided by the cost model described in Section 5.1, which we specifically tuned to minimize this overhead.

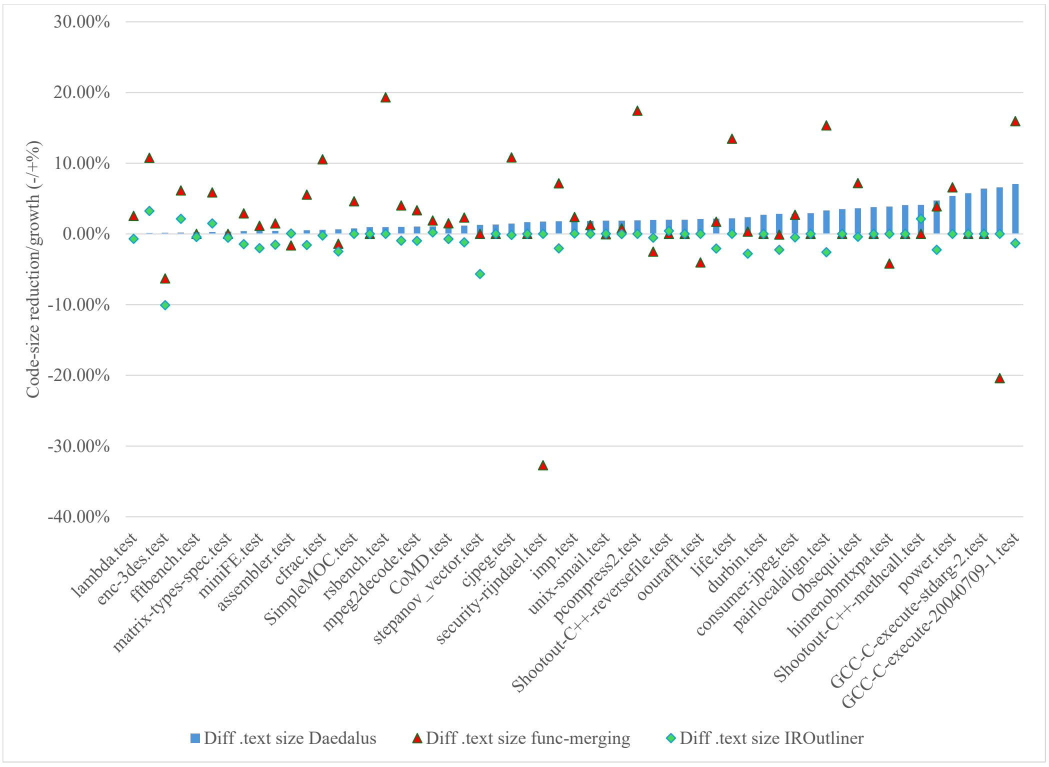
The trade-offs inherent in our approach are particularly evident in programs where `Daedalus` increases code size. Figure 5.4 compares the code-size impact of `Daedalus` against the baselines for these cases.

The `ldecod.test` program serves as a clear example of this trade-off:

- `Daedalus` increased the code size by 1.97%.
- `IROutliner` achieved a small code-size reduction of -0.53%.
- `func-merging` yielded a significant code-size reduction of -2.5%.

Figure 5.4 presents the programs detailed in Table 5.4, where each pass increased both the binary's `.text` section size and the number of LLVM IR instructions.

Figure 5.4: Code-size growth between Daedalus, func-merging, and IROutliner.



Source: The author.

The difference in code-size impact among the passes stems from the trade-off between pattern granularity and recurrence. `Daedalus` operates on highly specific, fine-grained data-flow patterns that may not recur often enough to amortize call overhead. In contrast, `IROutliner` and `func-merging` target larger, more frequently occurring code segments, generally increasing optimization opportunities. This contrast is evident in the analysis of `activate_sps` function from `ldecod.test` described on Table 5.5.

Table 5.5: Comparison of Code-Size Impact for `activate_sps` Across Passes.

Pass	Final Size (B)	Δ (B)	# Fns	New Fn Sizes (B)	Total Calls	Granularity / Recurrence
Original	3007	0	0	–	–	Baseline
Daedalus	3052	3052 − 3007	2	9, 9	2	Fine / Low
IROutliner	3200	3200 − 3007	2	174, 180	9	Medium / Moderate
func-merging	2963	2963 − 3007	6	211, 514, 303, 314, 515, 1145	24	Coarse / High

Source: The author.

`Daedalus` increased the size of `activate_sps` from 3007 to 3052 bytes. It produced two very small slice functions (9 and 9 bytes), each used once within this function (2 total calls), so prologue/epilogue overhead dominated the minimal savings from outlining fine-grained, non-recurrent patterns.

`IROutliner` also led to a local size increase, reaching 3200 bytes. Although it identified two larger, recurrent regions (174 and 180 bytes; 9 total calls), the added call-site complexity in this function prevented a net reduction.

By contrast, `func-merging` reduced the function to 2963 bytes. It captured the broadest and most varied set of optimizable regions, replacing original code with 24 calls to six merged functions (211–1145 bytes). In this case, the coarse-grained, highly recurrent transformations amortized call overhead and delivered the best compaction.

RQ2: Running Time

An essential requirement of any compiler optimization is that it must not degrade the runtime performance of the transformed program. This section examines this aspect by evaluating the execution time of the benchmark suite after being processed by `Daedalus`. The results show that the structural transformations introduced by slice outlining exert opposing effects on performance, which ultimately balance out to a neutral overall impact.

Across all benchmarks, the effect of `Daedalus` on execution time is negligible. The geometric mean of runtime variation shows a change of only 0.06% (Table 5.1b), and the vast majority of programs (1846, see Table 5.1a) exhibited no measurable difference in runtime performance.

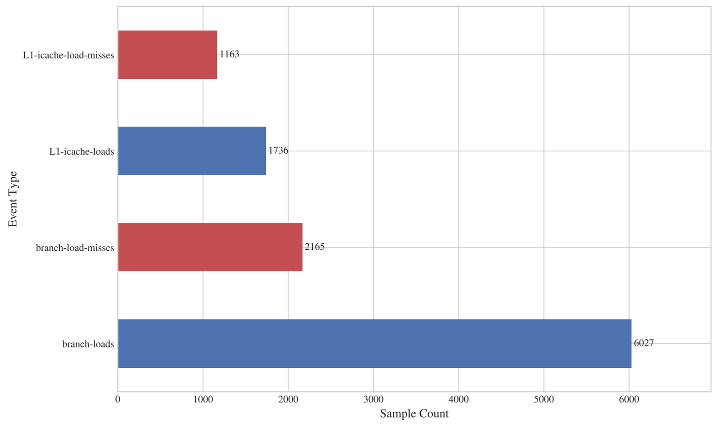
For the smaller subset of programs that were affected, performance variations were nearly symmetrical:

- **Slowdown:** Among programs that experienced a reduction in both instruction count and `.text` section size, 14 exhibited a runtime increase of 4.48% on average (Table 5.6). This behavior is consistent with expectations, as outlining introduces additional function call overhead for logic that was previously inlined.

- **Speedup:** Conversely, 7 benchmarks showed an average runtime decrease of -3.39% (Table 5.7). These cases, though less frequent, likely benefit from improved instruction cache locality: consolidating duplicated code into a single outlined function allows the processor's instruction cache to be used more efficiently.

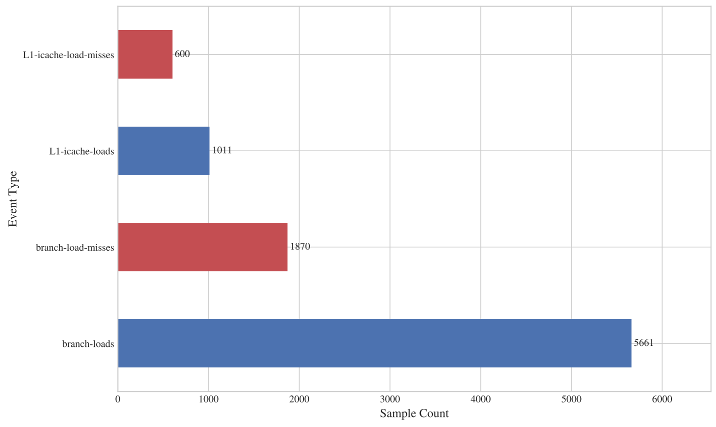
Cache Performance Analysis A comparison of hardware performance counters between the baseline and `Daedalus`-optimized executions reveals a clear shift in cache behavior, suggesting improved instruction locality and more efficient cache utilization, despite a slight increase in control-flow complexity. Figures 5.5 and 5.6 illustrate the relevant hardware event distributions before and after applying the pass.

Figure 5.5: Baseline cache performance profile.



Source: The author.

Figure 5.6: Cache performance profile after applying `Daedalus`.



Source: The author.

In the baseline configuration (Figure 5.5), hardware counters recorded 6027 branch loads and 2165 branch-load misses, alongside 1736 L1 instruction cache loads and 1163 misses. After applying `Daedalus` (Figure 5.6), these values shifted to 5661 branch loads, 1870 branch-load misses, and 1011 L1 instruction cache loads with 600 misses.

The optimization improved microarchitectural performance. The branch miss ratio decreased from 35.92% to 33.03%, indicating more predictable control flow. More substantially, the L1 data cache miss ratio fell from 66.99% to 59.35%. This reduction points to enhanced temporal and spatial data locality, meaning frequently accessed data remained resident in the cache for longer during execution.

These results suggest that the fine-grained outlining and code reorganization performed by `Daedalus` improve the reuse of frequently executed instruction sequences. The restructuring leads to tighter clustering of related code in memory, thereby reducing instruction fetch latency and minimizing cache thrashing. In addition to the modest decrease in branch mispredictions, the overall memory access pattern becomes more cache-efficient. Consequently, `Daedalus` enhances instruction cache locality without introducing significant control-flow penalties.

Table 5.6: Benchmarks with Reduced Instruction Count and `.text` Size but Increased Execution Time.

Metric	**Daedalus**	**func-merging**	**IROutliner**
(-) Instcount (-) `.text` size (+) Exec. Time	14	6	29
Diff Geomean	4.48%	19.96%	6.67%

Source: The author.

Table 5.7: Benchmarks with Reduced Instruction Count, `.text` Size, and Execution Time.

Metric	**Daedalus**	**func-merging**	**IROutliner**
(-) Instcount (-) `.text` size (-) Exec. Time	7	6	27
Diff Geomean	-3.39%	-15.16%	-8.34%

Source: The author.

RQ3: Compilation Overhead

The practicality of a compiler optimization depends not only on its benefits but also on its computational cost, particularly in terms of compilation time. This section quantifies the overhead introduced by `Daedalus`. As shown in Table 5.1b, the advanced analysis required for GSA-based slicing incurs a noticeable compile-time penalty. On average, `Daedalus` introduces a geometric mean compilation time increase of 4.22% across all benchmarks.

For the subset of programs that exhibited reductions in both `Instcount` and `.text` size, `Daedalus` did not yield faster compilation times (Table 5.9). Instead, it added an average overhead of 80.65% for these 23 benchmarks (Table 5.10).

Table 5.8: Benchmarks with Reduced Compilation Time and Corresponding Geometric Mean Differences.

Metric	**Daedalus**	**func-merging**	**IROutliner**
(-) Comp. Time	9	11	5
Diff Geomean	-19.81%	-17.25%	-17.51%

Source: The author.

Table 5.9: Benchmarks with Reduced Instruction Count, `.text` Size, and Compilation Time.

Metric	**Daedalus**	**func-merging**	**IROutliner**
(-) Instcount (-) `.text` size (-) Comp. Time	0	1	1
Diff Geomean	-	-8.57%	-1.61%

Source: The author.

Table 5.10: Benchmarks with Reduced Instruction Count and `.text` Size but Increased Compilation Time.

Metric	**Daedalus**	**func-merging**	**IROutliner**
(-) Instcount (-) `.text` size (+) Comp. Time	23	19	82
Diff Geomean	80.65%	24.31%	22.35%

Source: The author.

This overhead is an inherent consequence of the analyses employed by our technique. The `Daedalus` pass performs several computationally intensive operations not present in traditional outlining approaches:

1. The pass operates *recursively*: after identifying and outlining a slice function for a given criterion, it reanalyzes the original function to detect further slicing opportunities.

2. A custom function outliner was implemented (Section 3.1.4), since existing LLVM utilities such as `CodeExtractor` [12] cannot outline semantically defined regions derived from dependency graphs.

3. The pass explicitly removes original instructions from the parent function after a slice has been outlined and merged, rather than relying on subsequent dead-code elimination passes.

While these steps enable more precise and semantically aware slicing, they also increase compile-time costs relative to standard outlining algorithms.

RQ4: Daedalus Asymptotic Behavior

In this section, we detail the experiment conducted to measure the asymptotic behavior of the `Daedalus` pass. We selected the 100 largest programs from the test suite and measured the Pearson correlation between the number of LLVM IR instructions and the absolute time each transformation took to execute. Finally, we plotted a graph for each pass to visualize its asymptotic behavior.

A complexity analysis of the algorithms implemented by the `Daedalus`, `IROutliner`, and `func-merging` passes suggests linear behavior as the input size increases. To verify this empirically, we selected the 100 largest programs from the 2007 programs in our test suite. As shown in Table 5.11, the compilation time is highly correlated with program size. Furthermore, scatter plots of instruction count (X-axis) versus compilation time (Y-axis) reveal a linear trend line for all three passes, as shown in Figures 5.7, 5.8, and 5.9.

Table 5.11: Pearson correlation between compilation time and instruction count for each pass.

Pass	**Pearson Correlation**
Daedalus	0.837392779
func-merging	0.940710506
IROutliner	0.918238062

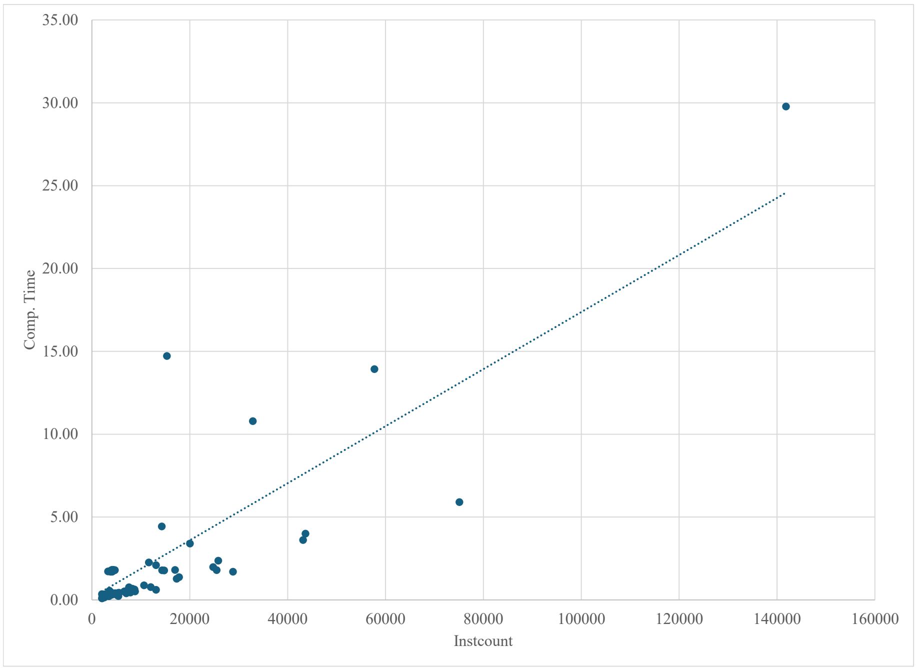
Source: The author.

Therefore, `Daedalus` exhibits linear performance scaling with input size and shows compilation times competitive with `IROutliner` and `func-merging`.

RQ5: Time of Daedalus Phases

In this section, we break down the transformation steps of `Daedalus` and measure the time taken by each phase for a given input program.

Figure 5.7: Daedalus: Compilation Time vs. Instruction Count.



Source: The author.

The `Daedalus` pass is composed of four phases: Outlining, Merging Slices, Removing Instructions, and Simplification. As depicted in Figure 3.1, the *GSA Construction*, *Slice Identification*, and *Function Outlining* steps are encapsulated within the Outlining Phase. The *Function Merging and Simplification* step corresponds to its own dedicated phase. The Remove Instructions Phase is responsible for deleting instructions from the original function that become redundant after being moved into a newly merged function.

To perform this analysis, we ran `Daedalus` on all 2007 tested programs, collecting the percentage of execution time spent in each phase. We then computed the geometric mean of these percentages across all programs. The results are presented in Table 5.13, which summarizes the main phases of `Daedalus`, and Table 5.12, which provides a detailed breakdown of the Outline phase.

Name	**Wall Time**
Slice Identification Phase Timer	42.389%
canOutline Phase Timer	32.581%
Function Outline Phase Timer	11.541%
GSA Construction Phase Timer	34.350%

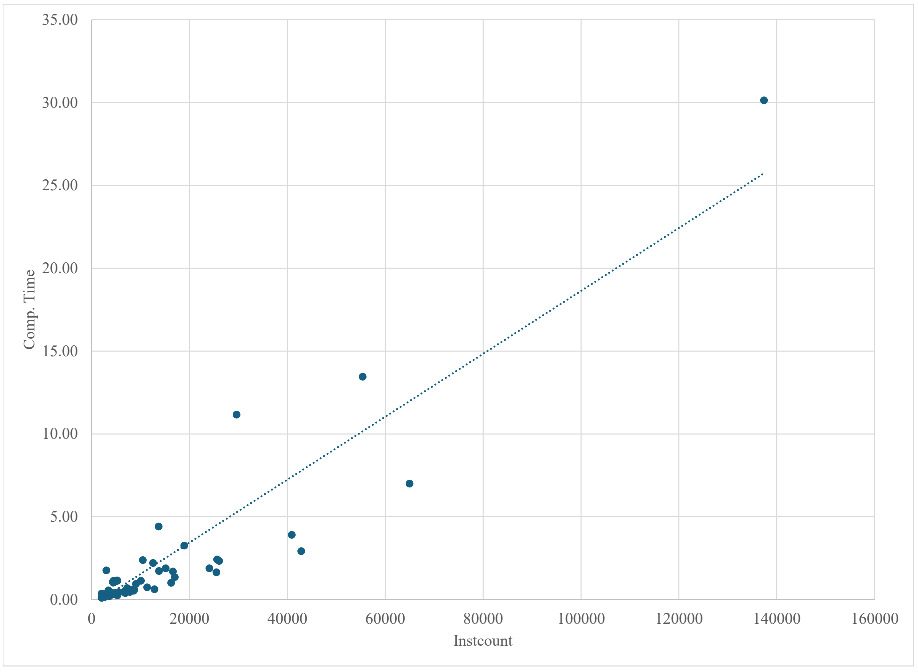
Table 5.12: Timers for Outline Sub-Phases.

Name	**Wall Time**
Outline Phase Timer	48.434%
Merge Phase Timer	22.885%
Remove Instructions Phase Timer	18.744%
Simplify Phase Timer	18.383%

Table 5.13: Timers for `Daedalus` Phases.

The column *Name*, identifies the phase being measured, while column *Wall Time*

Figure 5.8: IROutliner: Compilation Time vs. Instruction Count.



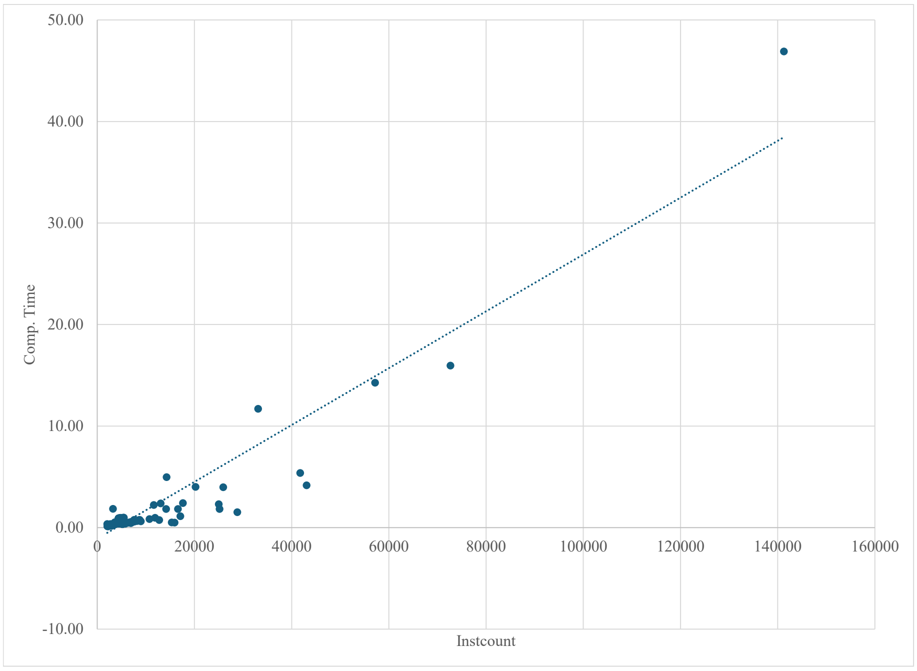
Source: The author.

indicates the real elapsed time, including any waiting or synchronization delays. Percentages denote each phase's relative contribution to the overall compilation time. Given the results in Tables 5.12 and 5.13, we conclude that the *Outline Phase* is the most time-consuming stage. Additionally, breaking down the *Outline Phase*, `Daedalus`'s *Slice Identification* and `canOutline` sub-phases expends significant time analyzing instruction's dependencies, and memory loads and stores. The latter analysis is required to account for potential memory clobbering through various levels of indirection, which is essential for verifying that a candidate function is side-effect-free before it can be outlined.

RQ6: Passes combinations' metrics

We conducted an experiment to determine if combining `Daedalus`, `IROutliner`, and `func-merging` could yield a better code compression ratio. To this end, we applied six different pass sequences to the 2007 programs in the test suite and collected performance metrics.

Figure 5.9: func-merging: Compilation Time vs. Instruction Count.



Source: The author.

Table 5.14 summarizes the overall geometric mean for each metric across the different pass sequences. Each sequence is identified by a compound name indicating the order of pass application. The passes are abbreviated as follows: `Daedalus` (ded), `IROutliner` (iro), and `func-merging` (fum).

Table 5.14: Comparison of metrics across different optimization pass orders.

Metric Name	**ded-iro-fum**	**fum-ded-iro**	**ded-fum-iro**	**fum-iro-ded**	**iro-ded-fum**	**iro-fum-ded**
Instcount	-1.22%	-1.01%	-1.07%	-0.98%	-1.19%	-1.16%
.text size	0.43%	0.33%	0.38%	0.33%	0.41%	0.39%
Exec. Time	0.28%	0.28%	0.37%	0.39%	0.16%	0.29%
Comp. Time	5.57%	5.31%	5.49%	5.20%	5.50%	5.09%

Source: The author.

The compilation time overhead is consistent across all pass sequences. Similarly, program execution time is affected, but the overhead is small. Given the nature of these outlining passes, the `.text` section size also increases consistently. We attribute this to the cumulative function call overhead introduced by the three passes, each identifying different patterns to outline. Finally, the number of LLVM IR instructions is reduced effectively, with the best pass configuration (`ded-iro-fum`) achieving a -1.22% reduction.

Summary

The empirical evaluation demonstrates the specific trade-offs of `Daedalus`. While it can achieve significant code-size reductions in targeted cases (RQ1), its narrow applicability results in a slight average size increase across the benchmark suite. This precision comes at the cost of a noticeable, albeit linear, compilation time overhead (RQ3, RQ4), primarily concentrated in the complex analysis of the Outline Phase (RQ5). Encouragingly, these transformations have a negligible net effect on program runtime, as the costs of function call overhead are balanced by gains in cache locality (RQ2). Furthermore, combining `Daedalus` with other outlining techniques does not yield further size reductions, highlighting its distinct optimization strategy (RQ6). We conclude that `Daedalus` is not a general-purpose size-reduction tool but rather a specialized optimization. Its value is most pronounced in domains where codebases feature the fine-grained, recurrent dataflow patterns that its GSA-based slicing is uniquely capable of identifying and eliminating.

Chapter 6

Conclusion

This thesis addressed the persistent challenge of code-size reduction in compiler optimization, a critical concern for software deployed on resource-constrained systems. We focused on the program slicing paradigm, a powerful but complex technique for isolating relevant program logic. While the concept, introduced by Weiser, has been influential for decades, the efficient generation of precise, executable slices remains an open problem. Our work confronted this challenge by leveraging the Gated Single Assignment (GSA) form to provide a richer semantic foundation for program analysis, enabling a more robust slicing methodology.

The central thesis of this work was that the explicit control-dependency information embedded in the GSA form could be used to generate self-contained, executable *Idempotent Backward Slices*.

To validate this thesis, our contributions progressed from foundational theory to practical implementation and evaluation. We began by successfully designing and implementing a robust algorithm to convert programs from the standard LLVM Intermediate Representation into the GSA form. Upon this foundation, we developed a novel program slicing algorithm that operates on that representation to extract *Idempotent Backward Slices* for code-size reduction. To ensure our work is reproducible and extensible, we delivered this implementation as an open-source *out-of-tree* LLVM pass, complete with a corresponding patch for the LLVM Test Suite to facilitate rigorous, standardized evaluation.

Summary of Results The empirical evaluation in Chapter 5 demonstrates the specific trade-offs of our approach. While `Daedalus` can achieve significant code-size reductions in targeted cases (RQ1), its narrow applicability results in a slight average size increase across the benchmark suite. Encouragingly, these transformations have a negligible net effect on program runtime, as the costs of function call overhead are balanced by gains in cache locality (RQ2). This precision, however, comes at the cost of a noticeable, albeit linear, compilation time overhead (RQ3, RQ4), which is primarily concentrated in the complex analysis of the Outline Phase (RQ5). Furthermore, combining `Daedalus` with other outlining techniques does not yield further size reductions, highlighting its distinct optimization strategy (RQ6). We conclude that `Daedalus` is not a general-purpose size-reduction tool but rather a specialized optimization. Its value is most pronounced in domains where codebases feature the fine-grained, recurrent dataflow patterns that its GSA-based slicing is uniquely capable of identifying and eliminating.

Limitations This study has several limitations. First, the effectiveness of our technique is highly dependent on a program's structure. The empirical data shows that the recurrent patterns it targets are rare in general-purpose software, limiting its impact. Finally, our implementation was developed and tested against LLVM 17, and its compatibility with other versions of the framework is not guaranteed without further engineering effort.

Future Work The contributions and limitations of this thesis open several promising avenues for future research. A primary direction would be to develop an algorithm for identifying promising slice candidates, which could make the pass more practical for production compilers. Another valuable extension would be to adapt the concept of *Idempotent Backward Slices* for other applications beyond code-size reduction, such as targeted debugging, security analysis, or program parallelization. Finally, extending the slicer to handle more complex constructs, like inter-procedural slicing and exception handling, would significantly broaden its applicability.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, & tools.* Pearson Addison-Wesley, Boston Munich, 2. ed., pearson internat. ed edition, 2007. ISBN 9780321486813 9780321491695.

[2] Andrew W. Appel and Maia Ginsburg. *Modern compiler implementation in C.* Cambridge Univ. Press, Cambridge, new, expanded textbook edition, 2004. ISBN 9780521583909 9780521607650.

[3] Sandrine Blazy, Andre Maroneze, and David Pichardie. Verified validation of program slicing. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 109–117, Mumbai India, January 2015. ACM. ISBN 9781450332965. doi: 10.1145/2676724.2693169. URL https://dl.acm.org/doi/10.1145/2676724.2693169.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 451–490, October 1991. ISSN 0164-0925, 1558-4593. doi: 10.1145/115372.115320. URL https://dl.acm.org/doi/10.1145/115372.115320.

[5] 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*, 9(3):319–349, July 1987. ISSN 0164-0925, 1558-4593. doi: 10.1145/24039.24041. URL https://dl.acm.org/doi/10.1145/24039.24041.

[6] Breno Campos Ferreira Guimarães and Fernando Magno Quintão Pereira. Lazy evaluation for the lazy: automatically transforming call-by-value into call-by-need. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 239–249, Montréal QC Canada, February 2023. ACM. ISBN 9798400700880. doi: 10.1145/3578360.3580270. URL https://dl.acm.org/doi/10.1145/3578360.3580270.

[7] Yann Herklotz, Delphine Demange, and Sandrine Blazy. Mechanised semantics for gated static single assignment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 182–196, Boston MA USA, January 2023. ACM. ISBN 9798400700262. doi: 10.1145/3573105.3575681. URL https://dl.acm.org/doi/10.1145/3573105.3575681.

[8] Shuo Jiang, Zhanhao Liang, Hanming Sun, Wenhan Shang, Bifeng Tong, Mengting Yuan, Chun (Jason) Xue, Jiang Ma, and Qingan Li. Lightweight Code Outlining for Android Applications. *ACM Transactions on Architecture and Code Optimization*, page 3776753, November 2025. ISSN 1544-3566, 1544-3973. doi: 10.1145/3776753. URL https://dl.acm.org/doi/10.1145/3776753.

[9] Kyungwoo Lee, Manman Ren, and Ellis Hoag. Optimistic and scalable global function merging. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2024, pages 46–57, New York, NY, USA, June 2024. Association for Computing Machinery. ISBN 979-8-4007-0616-5. doi: 10.1145/3652032.3657575. URL https://dl.acm.org/doi/10.1145/3652032.3657575.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, Berlin, softcover version of original hardcover edition 1999 edition, 2010. ISBN 9783642084744.

[11] Peng Zhao and J.N. Amaral. Function Outlining and Partial Inlining. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pages 101–108, Rio de Janeiro, RJ, Brazil, 2005. IEEE. ISBN 9780769524467. doi: 10.1109/CAHPC.2005.26. URL http://ieeexplore.ieee.org/document/1592562/.

[12] LLVM Project. LLVM: llvm::CodeExtractor Class Reference, 2025. URL https://llvm.org/doxygen/classllvm_1_1CodeExtractor.html.

[13] LLVM Project. MergeFunctions pass, how it works — LLVM 17.0.1 documentation, 2025. URL https://releases.llvm.org/17.0.1/docs/MergeFunctions.html.

[14] LLVM Project. LLVM: llvm::SimplifyCFGPass Class Reference, 2025. URL https://llvm.org/doxygen/classllvm_1_1SimplifyCFGPass.html.

[15] LLVM Project. Loop terminology (and canonical forms) — llvm 22.0.0git documentation, 2025. URL https://llvm.org/docs/LoopTerminology.html.

[16] LLVM Project. Link time optimization: design and implementation — llvm 22.0.0git documentation, 2025. URL https://llvm.org/docs/LinkTimeOptimization.html.

[17] Fabrice Rastello. *SSA-based compiler design*. Springer International Publishing AG, Cham, 1st ed edition, 2022. ISBN 9783030805159.

[18] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 854–868, London UK, June 2020. ACM. ISBN 9781450376136. doi: 10.1145/3385412.3386030. URL https://dl.acm.org/doi/10.1145/3385412.3386030.

[19] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 110–120, Barcelona Spain, March 2016. ACM. ISBN 9781450342414. doi: 10.1145/2892208.2892230. URL https://dl.acm.org/doi/10.1145/2892208.2892230.

[20] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981. ISSN 0004-5411, 1557-735X. doi: 10.1145/322261.322273. URL https://dl.acm.org/doi/10.1145/322261.322273.

[21] Frank Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3, 1994. URL https://api.semanticscholar.org/CorpusID:9882901.

[22] Vojislav Tomašević, Đorđe Todorović, and Maja Vukasović. Implementation of the debugging support for the llvm outlining optimization. In *Proceedings of the International Scientific Conference - Sinteza 2025*, pages 233–240, Beograd, Serbia, 2025. Singidunum University. ISBN 978-86-7912-841-6. doi: 10.15308/Sinteza-2025-233-240. URL http://portal.sinteza.singidunum.ac.rs/paper/1041.

[23] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 47–55, La Jolla California USA, June 1995. ACM. ISBN 9780897916974. doi: 10.1145/207110.207115. URL https://dl.acm.org/doi/10.1145/207110.207115.

[24] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10 (4):352–357, July 1984. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.1984.5010248. URL https://ieeexplore.ieee.org/document/5010248/.

Appendix A

The MiniGSA Interpreter Implementation

This appendix presents the complete Python source code for the interpreter of the *MiniGSA* language, which was formally introduced in Chapter 4. This implementation serves as a concrete, executable counterpart to the Structural Operational Semantics (SOS) defined therein. It is designed to adhere strictly to the semantic rules for each instruction, acting as an executable specification that clarifies the behavior of the GSA gating functions (γ, μ, η) and control flow constructs.

The interpreter was used to verify the behavior of program on Listing 1.1 from Example 1.0.1. Thus ensuring our formal analysis is grounded in a practical and operational model.

The full commented version of the python code can be found at `https://tinyurl.com/ms4epnej`.

```python
import operator
from collections import defaultdict
from dataclasses import dataclass, field
from typing import Dict, List, Any, Union, Set

Value = Union[int, bool, None]
Store = Dict[str, Value]
Program = Dict[str, List['Instruction']]

class Instruction:
    pass

@dataclass
class BinOp(Instruction):
    dest: str
    op1: Union[str, Value]
    op: str
    op2: Union[str, Value]

@dataclass
class Gamma(Instruction):
    dest: str
    pred: str
    v_true: str
    v_false: str

@dataclass
class Mu(Instruction):
    dest: str
    v_init: str
    v_loop: str

@dataclass
class Eta(Instruction):
    dest: str
    pred: str
    v_exit: str
    restart_set: Set[str] = field(default_factory=set)

@dataclass
class Branch(Instruction):
    target: str

@dataclass
class ConditionalBranch(Instruction):
    pred: str
    l_true: str
    l_false: str

@dataclass
class Stop(Instruction):
    pass

class Interpreter:
    def __init__(self, program: Program):
        self.program = program
        self.operators = {
            '+': operator.add,
            '-': operator.sub,
            '*': operator.mul,
            '/': operator.truediv,
            '<': operator.lt,
            '>': operator.gt,
            '<=': operator.le,
            '>=': operator.ge,
            '==': operator.eq,
        }

```

```python
    def _evaluate(self, operand: Any, store: Store) -> Value:
        if isinstance(operand, str):
            return store.get(operand)
        return operand

    def run(self, entry_label: str, initial_store: Store) -> Store:
        instruction_stream = list(self.program[entry_label])
        store = initial_store.copy()
        max_steps = 1000
        for step_count in range(max_steps):
            if not instruction_stream:
                raise RuntimeError("Execution fell off the end of a basic block without a terminator.")
            current_inst = instruction_stream.pop(0)
            if isinstance(current_inst, Stop):
                print(f"--- Program Halted in {step_count+1} steps ---")
                return store
            if isinstance(current_inst, BinOp):
                val1 = self._evaluate(current_inst.op1, store)
                val2 = self._evaluate(current_inst.op2, store)
                if val1 is None or val2 is None:
                    raise ValueError(f"Attempted to use uninitialized variable in BinOp: {current_inst}")
                op_func = self.operators.get(current_inst.op)
                if not op_func:
                    raise ValueError(f"Unknown operator: {current_inst.op}")
                result = op_func(val1, val2)
                store[current_inst.dest] = result
            elif isinstance(current_inst, Gamma):
                predicate_val = self._evaluate(current_inst.pred, store)
                if predicate_val is None:
                    raise ValueError(f"Predicate '{current_inst.pred}' is uninitialized.")
                if predicate_val:
                    value = self._evaluate(current_inst.v_true, store)
                else:
                    value = self._evaluate(current_inst.v_false, store)
                store[current_inst.dest] = value
            elif isinstance(current_inst, Mu):
                if current_inst.dest not in store:
                    value = self._evaluate(current_inst.v_init, store)
                else:
                    value = self._evaluate(current_inst.v_loop, store)
                store[current_inst.dest] = value
            elif isinstance(current_inst, Eta):
                predicate_val = self._evaluate(current_inst.pred, store)
                if predicate_val is None:
                    raise ValueError(f"Predicate '{current_inst.pred}' is uninitialized.")
                if not predicate_val:
                    value = self._evaluate(current_inst.v_exit, store)
                    store[current_inst.dest] = value
                    for r_var in current_inst.restart_set:
                        if r_var in store:
                            del store[r_var]
            elif isinstance(current_inst, Branch):
                instruction_stream = list(self.program[current_inst.target])
            elif isinstance(current_inst, ConditionalBranch):
                predicate_val = self._evaluate(current_inst.pred, store)
                if predicate_val is None:
                    raise ValueError(f"Predicate '{current_inst.pred}' is uninitialized.")
                if predicate_val:
                    target_label = current_inst.l_true
                else:
                    target_label = current_inst.l_false
                instruction_stream = list(self.program[target_label])
            else:
                raise TypeError(f"Unknown instruction type: {type(current_inst)}")
        raise RuntimeError("Maximum execution steps exceeded.")


if __name__ == '__main__':
    example_program: Program = {
        'entry': [
            Branch('BB1')
        ],
        'BB1': [
            Mu('x1', 'x0', 'x2'),
            Mu('s1', 's0', 's2'),
            Mu('t1', 't0', 't2'),
            BinOp('p0', 'x1', '<', 'n0'),
            ConditionalBranch('p0', 'BB2', 'BB3')
        ],
        'BB2': [
            BinOp('x2', 'x1', '+', 1),
            BinOp('s2', 's1', '*', 2),
            BinOp('t2', 't1', '+', 3),
            Branch('BB1')
        ],
        'BB3': [
            Eta('s4', 'p0', 's1', restart_set={'x2','s2','t2','p0'}),
            BinOp('s3', 's4', '+', 1),
            BinOp('u0', 's3', '+', 't1'),
            Stop()
        ]
    }
    initial_values: Store = {
        'n0': 10,
        'x0': 0,
        's0': 1,
        't0': 0
    }
    interpreter = Interpreter(example_program)
    final_store = interpreter.run('entry', initial_values)
    print("\n--- Final Store (Original Program) ---")
    for var, val in sorted(final_store.items()):
        print(f"{var}: {val}")
    expected_var_val = 2 ** final_store['n0'] + 1
    print(f"\nExpected final values: For n0=10 -> s3={expected_var_val},
    u0={expected_var_val + final_store['t1']}")

    example_program_slice: Program = {
        'entry': [
            Branch('BB1')
        ],
        'BB1': [
            Mu('x1', 'x0', 'x2'),
            Mu('s1', 's0', 's2'),
            BinOp('p0', 'x1', '<', 'n0'),
            ConditionalBranch('p0', 'BB2', 'BB3')
        ],
        'BB2': [
            BinOp('x2', 'x1', '+', 1),
            BinOp('s2', 's1', '*', 2),
            Branch('BB1')
        ],
        'BB3': [
            Eta('s4', 'p0', 's1', restart_set={'x2','s2','p0'}),
            BinOp('s3', 's4', '+', 1),
            Stop()
        ]
    }
    initial_values_slice: Store = {
        'n0': 10,
        'x0': 0,
        's0': 1,
    }
```

```python
    interpreter = Interpreter(example_program_slice)
    final_store = interpreter.run('entry', initial_values_slice)
    print("\n--- Final Store (Sliced Program) ---")
    for var, val in sorted(final_store.items()):
        print(f"{var}: {val}")
    out_var_val = 2 ** final_store['n0'] + 1
    print(f"\nExpected final values: For n0=10 -> s3={out_var_val} (is equal to original output? {expected_var_val == out_var_val})")
```

Listing A.1: Python Interpreter for MiniGSA