THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Prolog Optimizations in GraalVM

Project Proposal

by

Thi Thuy Tien Nguyen

Supervisors

Brae J. Webb , Mark Utting , Ian J. Hayes

School of Electrical Engineering and Computer Science,
The University of Queensland.

12 September 2024

TABLE OF CONTENTS

1 Abstract

Compiler optimizations are critical for enhancing program execution speed and overall performance. This thesis proposes integrating Prolog-based optimization techniques into the GraalVM compiler framework to potentially enhance compiler performance and optimization expressibility. The approach involves developing a Prolog-based optimizer that translates GraalVM's intermediate representation (IR) of programs into Prolog facts and defines optimization rules using Prolog's declarative syntax. By querying these rules within the GraalVM compiler, the proposal aims to explore how Prolog's logic programming capabilities can improve the expressiveness and effectiveness of compiler optimizations. The study seeks to assess the feasibility of this integration, examining how Prolog's declarative nature might contribute to more sophisticated and potentially more efficient optimization strategies compared to traditional imperative methods. This approach could lead to advancements in compiler technology and open new avenues for research in the optimization phase of compilation.

2 Introduction

Compiler optimizations improve program execution speed and overall performance, which can result in substantial cost savings for large-scale organizations, potentially reducing infrastructure expenses by millions annually. This project examines the integration of GraalVM Ahead-of-Time (AOT) compiler optimizations with rule-based declarative logic programming languages. Logic programming languages allow programs to be analyzed efficiently through queries, enabling the implementation of code optimization that would otherwise be difficult to achieve.

Previous research has extensively utilized Datalog, a prominent logic programming language, for different levels of program analysis [1–4]. However, there is a notable lack of evidence regarding the application of logic programming languages for the optimization and transformation of code. Spinellis' work in 1999 explored an alternative method for expressing optimizations through declarative specifications, as opposed to traditional imperative code [5]. However, Spinellis's work was limited to a rudimentary prototype of an optimizer with a "single optimization specification" and "limited flow-of-control optimizations" [5].

This research aims to assess the feasibility of implementing a Prolog-based optimizer within the GraalVM compiler framework. Prolog is among the earliest formal logic programming languages and continues to be one of the most widely used today. This study seeks to compare the expressiveness and efficacy of Prolog-based optimizations against the existing optimization techniques employed by GraalVM, and to investigate potential new optimizations enabled by Prolog specifications. The proposed approach involves expressing optimization rules using Prolog specifications and subsequently modifying the GraalVM compiler to query and utilize the Prolog database for optimization opportunities.

3 Background

3.1 Background Materials

3.1.1 Ahead-of-time (AOT) Compilation

In the Java Virtual Machine (JVM), byte code compilation can occur either at build time or runtime. Ahead-of-Time (AOT) compilation involves translating the byte code into machine code before execution, resulting in a fully compiled binary that is immediately executable [6]. In contrast, Just-In-Time (JIT) Compilation defers the translation until runtime if at all, dynamically converting Java bytecodes into machine code within the JVM and optimizing frequently executed code paths to enhance performance [6]. AOT compilation generally requires longer build times but offers rapid startup and predictable performance, making it suitable for applications where quick initialization is critical. JIT compilation, while benefiting from shorter build times, incurs longer startup periods but allows for more complex optimizations based on runtime profiling data.

JIT compilation enables techniques such as speculative optimizations, which make assumptions about a program's behavior to apply performance-enhancing transformations based on runtime profiling. Although these optimizations can significantly improve performance by optimizing frequently executed code paths, incorrect assumptions may necessitate deoptimizations, which revert the code to a less optimized but more reliable version [7]. This can complicate the program's IR by adding additional nodes and edges to accommodate these transformations [7]. In this project, the emphasis will be on AOT compilation, where speculative optimizations are excluded [8], resulting in a simpler IR without the need for deoptimization.

3.1.2 GraalVM Compiler Optimizations

In GraalVM, the compilation process is divided into two main phases. The first phase, involving the Graal IR, handles most of the high-level optimizations. This phase is further organized into three tiers: high-tier for high-level optimizations, mid-tier for memory-focused enhancements, and low-tier for low-level IR (LIR) conversion [9]. This project will primarily concentrate on high-tier optimizations within the GraalVM, beginning with the canonicalization phase.

Canonicalization, an essential early phase in the optimization process, focuses on transforming code into a standardized format and eliminating redundancies. This transformation simplifies and facilitates the application of subsequent optimizations. Examples of canonicalization include:

Constant Folding: Replace constant expressions with their computed values, such as simplifying $3+4$ to 7.

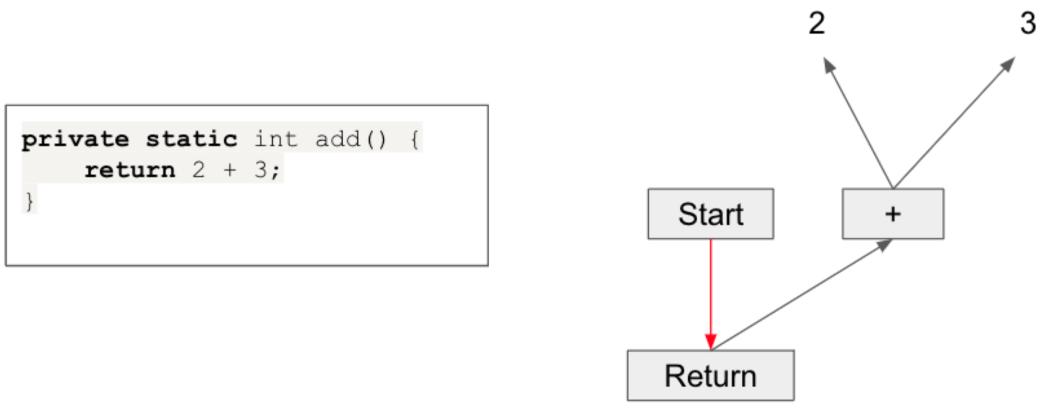
Simplify Redundant Multiplication: Eliminate unnecessary multiplication operations, such as converting $x*0$ to 0, and $x*1$ to x.

Simplifying Conditional Statements: Reduce complexity in conditional logic by removing redundant or always false branches. For example, an if statement with a condition that can never be true, such as $if(false)$, can be simplified by removing the entire branch.

Global Value Numbering: Eliminate redundant computations by assigning unique identifiers to equivalent expressions [10]. For instance, if the expression $a+b$ appears multiple times in the code, global value numbering ensures that it is computed only once and reused, thereby reducing unnecessary recalculations.

3.1.3 Graal Intermediate Representation (IR)

The Graal IR [11] models a program's structure and operations using a directed graph that illustrates both data and control flow between nodes. Each node in this graph is designed to produce a single value and follows the Static Single Assignment (SSA) [12] form. Data flow is represented by input edges pointing upward to the operand nodes, control flow is depicted by successor edges pointing downwards to the successor nodes. This IR framework provides a robust and efficient structure for code analysis and transformation, where optimization processes involve modifying the graph to enhance overall performance. A critical aspect of this project involves converting the Graal IR into Prolog facts to enable the optimizer to query these facts for potential optimizations. Therefore, it is essential to thoroughly understand the IR's structure and components to translate and use it within the Prolog-based optimization framework effectively.



```
private static int add() {
    return 2 + 3;
}
```

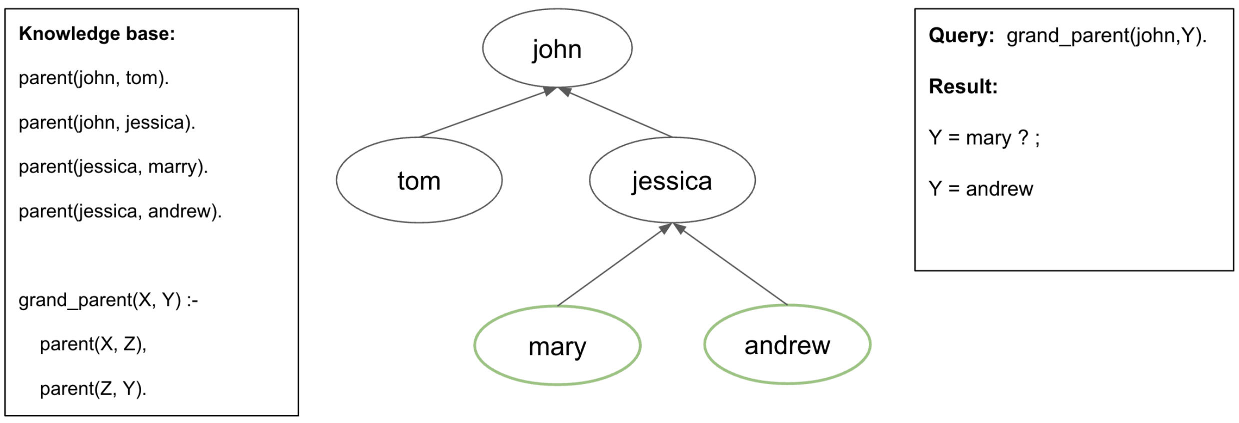
Figure 3.1: Simple IR Graph

Figure 3.1 illustrates a simple IR graph, with control flow edges highlighted in red. The graph begins at the Start node, which connects to the Return node via a successor edge. Upon reaching the Return node, the graph traverses upward through data flow edges to compute the returned expression's value. This traversal demonstrates how control and data flows interact to process and complete the function's execution.

3.1.4 Prolog Programming Language

In traditional imperative languages, a program consists of a sequence of instructions. This approach emphasizes a step-by-step procedure where each instruction modifies the state of the machine to solve a given problem. In contrast, logic programming languages, such as Prolog, operate on a fundamentally different paradigm. Instead of prescribing a sequence of operations, logic programming focuses on defining a knowledge base composed of facts and rules [13]. After that, users can query the knowledge base to search for objects and relations.

In Prolog, facts represent objects and their relationships, while rules imply the relationship between objects given it satisfies all the conditions. Once the knowledge base is established, users can formulate queries to extract information or solve problems by leveraging the logical relationships defined in the base using the depth-first search algorithm [14]. There may be several ways to achieve a given goal. The system initially selects the first available option. If Prolog fails to resolve a specific subgoal, it will backtrack to explore these previously noted alternatives. This mechanism, referred to as backtracking [14], enables Prolog to systematically search for different solutions by revisiting and trying alternative paths.



Knowledge base:

```
parent(john, tom).
parent(john, jessica).
parent(jessica, marry).
parent(jessica, andrew).

grand_parent(X, Y) :-
   parent(X, Z),
   parent(Z, Y).
```

Query: grand_parent(john,Y).

Result:

```
Y = mary ? ;
Y = andrew
```

Figure 3.2: Example of Prolog specification

Figure 3.2 illustrates a Prolog program. The first three clauses are facts: john is a parent of tom and jessica, and jessica is a parent of both mary and andrew. The final clause is a rule defining the conditions for a grandparent relationship. When a query is made to determine the nodes for which john is a grandparent, Prolog performs a search from the first to the last clause, showcasing its backtracking behavior. Initially, Prolog identifies that john is a parent of tom, but since tom is not a parent of anyone, the search backtracks. It then considers the next option: jessica. Prolog then finds that jessica is a parent of mary. After exploring this path, Prolog backtracks again and continues to check the next possibility: jessica is also a parent of andrew, thereby completing the query.

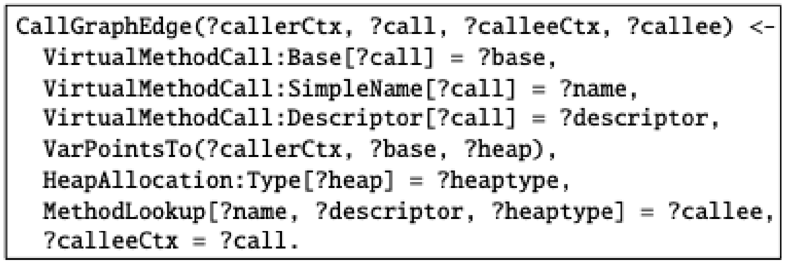
3.2 Literature Review

3.2.1 Advancements Through Domain-Specific Languages (DSLs)

Declarative domain-specific languages (DSLs) have made a big impact on compiler optimization by offering targeted solutions for specific optimization tasks. For example, Halide [15] is a domain-specific language and compiler framework designed to optimize image processing and computational photography. Writing efficient code for image processing often requires complex optimizations to exploit parallelism and memory locality. In Halide, the algorithm specifies the computational logic for these tasks, detailing what needs to be done, such as resizing, sharpening, or blurring an image. The schedule defines the execution strategy, including how computations are ordered, parallelized, and managed in memory. This separation of concerns allows Halide to generate highly optimized code by focusing on both the functional description and the efficient execution of tasks. Meanwhile, Spinellis created a peephole optimizer that uses a declarative DSL to specify optimizations [5]. Spinellis transformed specifications into string regular expressions, which are then applied to the target code, allowing for adaptive and efficient refinements. Although this method has only been tested on smaller programs and specific types of optimizations so far, it shows great potential for quickly experimenting with new techniques and architectures. Elevate [16] is another functional language designed to let programmers define custom optimization strategies. It allows for creating composable optimizations rather than sticking to predefined APIs. Its success in case studies and practical applications highlights its ability to handle complex optimization tasks and deliver strong performance.

3.2.2 Advancements Through Logic Programming Languages

DataLog has established itself as a powerful tool in compiler implementation through its applications in complex program analysis and transformation. For instance, Lam et al. developed a framework using Datalog queries and a specialized language called PQL, which employs deductive databases and binary decision diagrams (BDDs) to tackle complex issues like pointer aliasing and heap object management in a context-sensitive manner [3]. This approach has simplified the creation of sophisticated analyses and has been crucial in identifying security vulnerabilities in C and Java applications. Similarly, the Soufflé Datalog engine [17] introduces a new method for control-flow analysis in Scheme, making it possible to perform scalable and complex analyses of functional programming constructs, and extending the benefits of Datalog-based static analysis to Scheme-like languages. Furthermore, DIMPLE [4], a framework for Java bytecode static analyses, is implemented in the Yap Prolog system. DIMPLE leverages logic programming capabilities to provide a declarative language for specifying analyses and representing Java bytecodes. The framework facilitates iterative experimentation and delivers efficient implementations that are on par with specialized tools. Finally, the Doop framework [1] utilizes Datalog to offer a declarative solution for points-to analysis in Java, providing impressive performance gains and scalability for precise context-sensitive analyses that were previously challenging to achieve.



```
CallGraphEdge(?callerCtx, ?call, ?calleeCtx, ?callee) <-
  VirtualMethodCall:Base[?call] = ?base,
  VirtualMethodCall:SimpleName[?call] = ?name,
  VirtualMethodCall:Descriptor[?call] = ?descriptor,
  VarPointsTo(?callerCtx, ?base, ?heap),
  HeapAllocation:Type[?heap] = ?heaptype,
  MethodLookup[?name, ?descriptor, ?heaptype] = ?callee,
  ?calleeCtx = ?call.
```

Figure 3.3: DOOP's rule for virtual method invocations [1]

Figure 3.3 illustrates a Datadog rule that Doop uses to represent a call graph. The rule holds if there is a virtual method call where the receiver object is accessed through (?base) that points to (?heap) which contains (?callee) method in its type (?heaptype) [1].

3.2.3 Transformation, Rewriting Techniques, and Verification

Transformation and rewriting techniques are essential to modern compiler optimization, enabling sophisticated processes like grammar specification, pattern matching, and program rewriting. Stratego [18] is a language designed for specifying program transformation systems using rewriting strategies, which allow for precise control over rule application. The compiler transforms these rules into C code and leverages the ATerm library for effective term representation. It has been used in several applications, including program transformation tools like XT, and domain-specific optimization frameworks like CodeBoost [18]. On the other hand, Veriopt [19] offers a comprehensive framework for formal verification of Graal compiler optimizations. This framework employs "term rewriting rules on the abstract term representation" [19] and then maps these rules to term graph transformations. In the Veriopt project, a term rewriting system, also known as an optimization phase, comprises a set of rules that can be applied repeatedly to optimize expressions. Each phase is associated with different types of root nodes in patterns, such as AddNode, AbsNode, and MulNode.

```
phase Conditional
 terminating trm
begin
optimization NegateCond: $((!c)\ ?\ t : f) \longmapsto (c\ ?\ f : t)$
optimization TrueCond: $(true\ ?\ t : f) \longmapsto t$
optimization FalseCond: $(false\ ?\ t : f) \longmapsto f$
optimization BranchEqual: $(c\ ?\ x : x) \longmapsto x$
optimization LessCond: $((u < v)\ ?\ t : f) \longmapsto t$
            when (stamp-under (stamp-expr u) (stamp-expr v)
                 $\wedge$ wf-stamp u $\wedge$ wf-stamp v)
end
```

Figure 3.4: Conditional canonicalization rules [19]

Figure 3.4 provides an example of such an optimization phase tailored for conditional expressions. NegateCond transforms negated conditions using logical equivalences, while TrueCond simplifies expressions where the condition is always true by directly applying the true outcome. Conversely, FalseCond addresses conditions that are always false by eliminating irrelevant branches. BranchEqual simplifies expressions with equality checks by optimizing or removing redundant comparisons.

3.2.4 Identified Research Gaps

Despite the established effectiveness of logic programming languages like Datalog in program analysis, there is limited research on integrating these techniques within modern compiler frameworks such as GraalVM. While significant advancements have been made in transformation and rewriting techniques using domain-specific languages (DSLs) and other paradigms, there is a notable absence of comparative studies on incorporating logic programming-based optimizations into GraalVM. This gap extends to a lack of empirical evidence assessing the impact of such techniques on optimization time and efficiency. Integrating Prolog-based rules with GraalVM's graph-based IR is particularly challenging due to insufficient research on mapping Graal IR to Prolog and ensuring that Prolog-based optimizers work effectively within the Java environment. Addressing these challenges could enhance our understanding of how to apply logic programming in modern compilers and improve compiler optimization practices by providing valuable insights into the performance and effectiveness of these techniques.

4 Project Plan

4.1 Project Aims

This project aims to evaluate the potential of integrating a Prolog-based optimization framework within the GraalVM compiler. Logic programming languages have demonstrated significant advantages in the program analysis phase of compilers. This project seeks to explore their application in the optimization phase—a complex and critical component of the compiler. Enhancing the optimization phase could lead to substantial improvements in optimization specification and performance. Specifically, the project strives to accomplish the following items:

Develop Prolog-Based Optimization Rules: Implement Prolog rules for various optimization techniques within the Ahead-of-Time (AOT) compiler.

Integrate Prolog-Based Optimization: Assess the feasibility of incorporating a Prolog-based optimization framework within the GraalVM compiler.

Enhance Optimization Expressiveness: Evaluate how Prolog can improve the expressiveness and effectiveness of compiler optimizations compared to existing GraalVM techniques.

Evaluate New Optimizations: Investigate potential new optimizations enabled by Prolog specifications and compare their performance against existing GraalVM optimization methods.

4.2 Methodology

4.2.1 Prolog-Based Optimization Framework

The Prolog-based optimization framework is composed of three key components.

4.2.1.1 Prolog fact generator

Generate Prolog facts that accurately represent the IR graph of the program. This component will parse the IR and convert it into a Prolog-compatible format, allowing Prolog rules to operate on the program's structure. The approach involves traversing the IR graph and generating Prolog facts based on the graph's nodes and edges.

4.2.1.2 Prolog optimization rules

Express optimization rules as Prolog facts and predicates. This involves translating optimization concepts into Prolog syntax, ensuring that the rules are both expressive and applicable to the GraalVM compiler's IR. This translation process is critical for leveraging Prolog's capabilities in optimizing compiler IR.

Canonicalization Phase: Begin with the canonicalization phase, which aims to standardize the IR code. This phase reduces the variety of IR forms by transforming them into a canonical format, thereby simplifying the optimization process for subsequent phases. By establishing a uniform representation, canonicalization facilitates more effective and efficient optimization in later stages.

Conditional Elimination Phase: Following canonicalization, the Conditional Elimination phase should be addressed. This phase focuses on removing redundant or unnecessary conditional expressions. This phase is a logical next step due to its relative simplicity and straightforward nature. It can effectively utilize the standardized forms resulting from the canonicalization phase.

These two phases—canonicalization and conditional elimination—represent the initial focus of this project. Further optimization phases will be identified and analyzed as the project progresses over the course of the year.

4.2.1.3 Prolog-based optimizer

Recursively query Prolog optimization rules, execute Prolog queries, and apply transformations to the IR. This system should iteratively refine the program based on the query results. The component will be developed in Java and could potentially leverage the GNU Prolog for Java library to facilitate the execution of Prolog queries within the Java environment.

To ensure compatibility between Prolog facts and rules, the development of the first two components will occur in parallel.

4.2.2 Integration Into GraalVM Compiler

After developing the standalone components of the Prolog-based optimizer, the next step is to integrate it with the GraalVM compiler. GraalVM is a substantially large and very complex project. Therefore, this integration may encounter potential compatibility issues. It is necessary to treat the integration phase as a distinct part of the development process. This involves modifying the GraalVM compiler to interface effectively with the Prolog-based optimizer and ensuring that the compiler can pass IR data to the Prolog-based optimizer and receive optimized IR data back for further processing.

4.2.3 Testing

Since the optimization phases involve significant transformations to the original code, ensuring the soundness and correctness of the optimizer is crucial. To achieve this, a comprehensive test suite is developed to:

Verify Prolog Rules: Test the Prolog rules to ensure they function correctly and identify all valid optimization opportunities without introducing errors.

Validate Fact Generation: Confirm that the generated Prolog facts accurately represent the original IR graph, ensuring consistency and correctness.

Assess Integration: Evaluate the integration of the Prolog-based optimizer with the GraalVM compiler, ensuring that the optimized code accurately reflects the original code and that no errors are introduced during the optimization process.

Additionally, GraalVM provides a robust suite of unit tests that will be leveraged during this testing phase. These existing tests will aid in validating the correctness of the optimization process and ensure that the Prolog-based optimizer integrates smoothly with GraalVM.

4.2.4 Evaluation

After validating the optimizer's correctness, performance must be evaluated and compared with existing methods to determine any improvements offered by Prolog-based optimizations. Additionally, it is important to assess whether the declarative syntax enhances documentation, comprehension, and maintenance of the optimizer.

Performance Metrics: Measure various performance metrics, including execution speed, resource utilization (e.g., memory and CPU), and optimization throughput. Compare these metrics with those of existing GraalVM optimization techniques to measure improvements or regressions. This comparison will help identify whether the Prolog-based optimizer provides tangible performance benefits or if it introduces any new challenges.

Potential New Optimizations: Explore and document potential new optimization techniques that Prolog's declarative syntax might enable. This involves identifying unique optimization opportunities that may not be feasible with traditional imperative methods. By leveraging Prolog's rule-based approach, we can uncover novel optimization strategies that could enhance the efficiency and effectiveness of the optimization process.

Strengths and Weaknesses: Evaluate the Prolog-based optimizer and compare it to traditional approaches in terms of usability, including documentation clarity and the ease of adding new rules; maintenance, focusing on the ease of updates and code clarity; debugging and testing, assessing how effectively errors are identified and resolved; and scalability, ensuring the optimizer remains effective as codebases and optimization demands grow. This assessment will reveal the optimizer's strengths and limitations relative to conventional methods.

4.3 Timeline

The project timeline is divided into two parts: the first half (Semester 2, 2024) and the second half (Semester 1, 2025). The timeline is further divided into 4 phases: Project Proposal, Proof of Concept, Extension and Evaluation, and Report and Demonstration. The total duration of the project is 30 weeks, with a commitment of 10 working hours per week. The first half involves conducting research, writing the project proposal, and developing a proof of concept. The second half focuses on refining and expanding the project to encompass a comprehensive set of optimization rules, as well as preparing the final paper, report, and presentation. The timeline may be adjusted based on the results of the proof of concept phase to accommodate unforeseen challenges or to incorporate additional development needs. The distribution of weeks per task is illustrated in the timeline graph below.

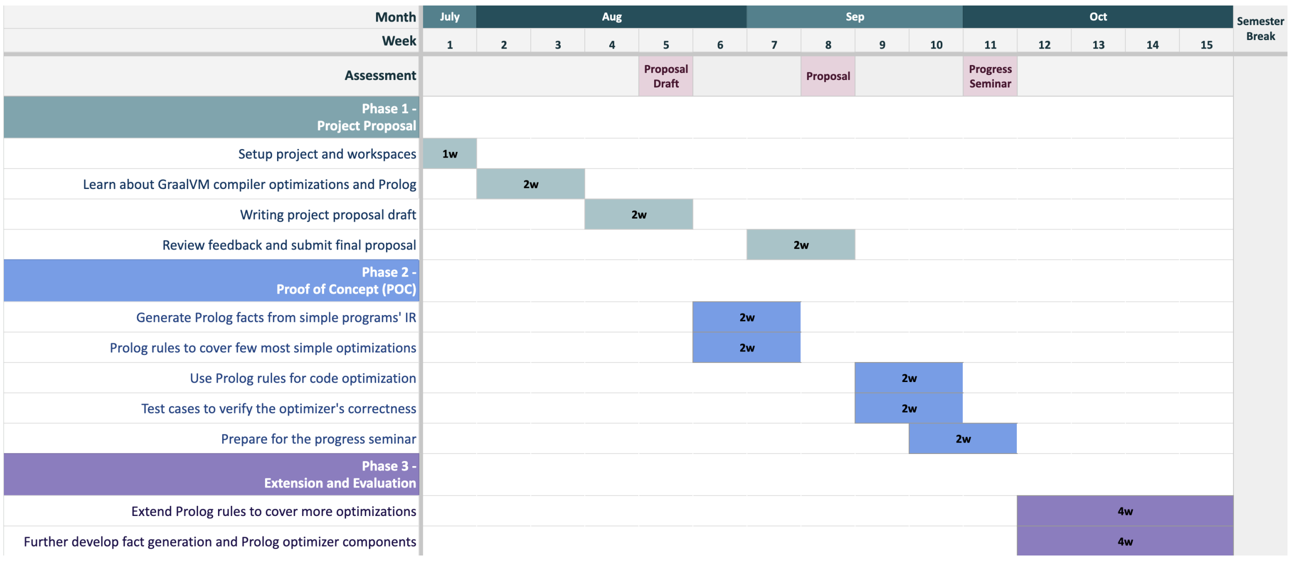


Figure 4.1: Project Timeline (first half)

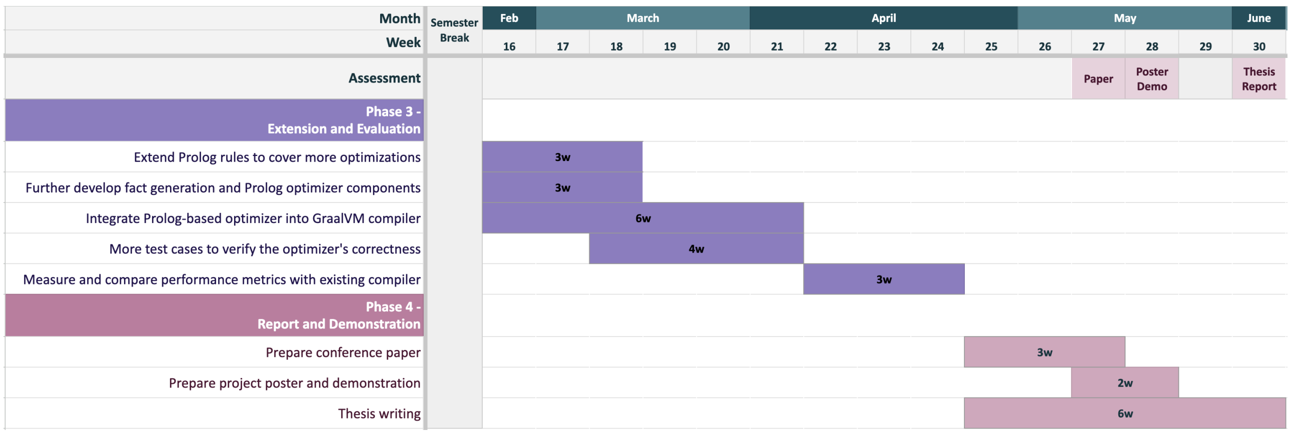


Figure 4.2: Project Timeline (second half)

Each project phase concludes with a milestone, except for the Extension and Evaluation phase, which includes two milestones due to a three-month semester break dividing this phase. The first three milestones occur during the first part of the project involve planning, validating the concept, and the initial development cycle. The final two milestones, scheduled for the second part, focus on advancing development and completing the final thesis.

4.3.1 Milestone 1: Project Proposal

Date: 19/09/2024 (Week 8)
Deliverables: Project proposal

To deliver the project proposal, several tasks need to be completed. The first week of the project involved meeting with the supervisors to gain a better understanding of the project and setting up the necessary tools and workspaces. The following two weeks focused on studying GraalVM compiler optimizations, Graal IR, and Prolog. The subsequent two weeks were dedicated to conducting additional research and a literature review in preparation for the draft proposal. Based on the feedback received, the final project proposal is expected to be delivered by the eighth week as an assessment item.

4.3.2 Milestone 2: Proof of Concept (PoC) and Progress Seminar

Date: 08/10/2024 (Week 11)
Deliverables: PoC and progress update presentation

In the second phase of the project, the focus will be on developing a proof of concept (PoC) for the Prolog-based optimizer. This PoC will employ a limited set of simple optimization techniques to assess the feasibility of the proposed approach. The PoC will also identify any additional development efforts required that may have been overlooked in the initial phase. Finally, this phase will conclude with a progress update seminar which is the second assessment item.

4.3.3 Milestone 3: Initial Extension and Improvement Cycle

Date: 01/11/2024 (Week 15)
Deliverables: More comprehensive set of optimization rules and a better Prolog-based optimizer

In the third phase, the focus will shift to extending and refining the proof of concept (PoC). Following the initial verification of optimization rules, efforts will be directed toward expanding the Prolog optimization rule sets to support more complex and advanced optimizations. Additionally, components such as the Prolog fact generator and the Prolog-based optimizer, will be revised and enhanced for increased robustness and efficiency, incorporating insights gained from the PoC results. The phase will conclude at the end of the semester, with a more comprehensive set of optimization rules and an improved optimizer from the previous phase.

4.3.4 Milestone 4: Completion of Development and Testing

Date: 25/04/2025 (Week 24)

Deliverables: Final version of the Prolog-based optimizer

The second half of the project will focus on continuing and enhancing the Prolog-based optimizer based on insights from the previous phase. A key objective will be to integrate the Prolog-based optimizer with the GraalVM compiler. This phase will also involve developing comprehensive test cases to ensure the optimizer's correctness and robustness. Performance metrics will be assessed and compared with those of the existing GraalVM compiler to identify strengths and weaknesses. The phase will conclude with the finalization of the Prolog-based optimizer and a detailed review of its capabilities, performance, and potential for future enhancements.

4.3.5 Milestone 5: Thesis Writing and Demonstration

Date: 09/06/2025 (Week 30)

Deliverables: Final thesis report

With all development and testing completed in the previous phase, the final phase of the project will focus on preparing the conference paper, project poster, project presentation, and final thesis report, which are the remaining assessment items. The submission of the final thesis report will mark the project's completion.

4.4 Ethics & Risk Assessment

4.4.1 Ethics Assessment

This project exclusively engages with software and does not involve the collection or use of data from human or animal subjects. As a result, it does not require any substantial ethical considerations.

4.4.2 Risk Assessment

The table below outlines potential risks that could impact project success, along with corresponding mitigation strategies to minimize their likelihood and severity.

Risk	**Likelihood**	**Severity**	**Risk Level**	**Mitigation Strategy**
Unable to meet key requirements and lagging behind scheduled milestones.	Possible	Significant	High	Conduct regular meetings with supervisors and team members to assess project progress, identify and resolve any obstacles or difficulties, and ensure efficient time management.
Lose development progress due to computer problem.	Unlikely	Significant	Medium	Backup all work progress to multiple places, e.g. code to GitHub and reports to drive.
Insufficient expertise in logical programming languages and the GraalVM compiler.	Likely	Minor	Medium	Dedicate time to acquiring domain knowledge and studying Prolog, GraalVM optimizations, and IR constructs. Consult supervisors regularly for guidance and support.
The Prolog-based optimizer may be less accurate and add performance overhead.	Possible	Moderate	Medium	Develop various test cases, conduct benchmarking to assess optimizer performance, and analyze the accuracy of the results.
Health issues while working with the project.	Possible	Moderate	Medium	Ensure that the project is systematically organized and well paced to prevent overwork and mitigate stress.
Safety issues in the workplace.	Unlikely	Minor	Low	Safety concerns are limited, given that the work is conducted in a controlled, in-house environment and involves only computer systems.

Table 4.1: Risk Assessment.

References

[1] M. Bravenboer and Y. Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses,” *ACM SIGPLAN Notices*, vol. 44, pp. 243–262, 10 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1639949.1640108

[2] R. van Tonder, “Towards fully declarative program analysis via source code transformation,” *CoRR*, vol. abs/2112.12398, 2021. [Online]. Available: https://arxiv.org/abs/2112.12398

[3] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, “Context-sensitive program analysis as database queries,” in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 6 2005, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/1065167.1065169

[4] W. C. Benton and C. N. Fischer, “Interactive, scalable, declarative program analysis,” in *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, 7 2007, pp. 13–24. [Online]. Available: https://dl.acm.org/doi/10.1145/1273920.1273923

[5] D. Spinellis, “Declarative peephole optimization using string pattern matching,” *ACM SIGPLAN Notices*, vol. 34, pp. 47–50, 2 1999. [Online]. Available: https://dl.acm.org/doi/10.1145/307903.307921

[6] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, “AOT vs. JIT: impact of profile data on code quality,” in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, vol. 52. ACM, 6 2017, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3078633.3081037

[7] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck, “An intermediate representation for speculative optimizations in a dynamic compiler,” in *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*. ACM, 10 2013, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2542142.2542143

[8] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, “Initialize once, start fast: application initialization at build time,” *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, 10 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3360610

[9] M. Sipek, B. Mihaljevic, and A. Radovan, “Exploring aspects of polyglot high-performance virtual machine GraalVM,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 5 2019, pp. 1671–1676. [Online]. Available: https://ieeexplore.ieee.org/document/8756917/

[10] C. Click, “Global code motion/global value numbering,” in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM, 6 1995, pp. 246–257. [Online]. Available: https://dl.acm.org/doi/10.1145/207110.207154

[11] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, “Graal IR: An extensible declarative intermediate representation,” in *Conference: 2nd Asia-Pacific Programming Languages and Compilers WorkshopAt: Shenzhen, China*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:52231504

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, 10 1991. [Online]. Available: https://dl.acm.org/doi/10.1145/115372.115320

[13] M. Bramer, “Clauses and predicates,” in *Logic Programming with Prolog*. London: Springer London, 2013, pp. 13–27. [Online]. Available: https://doi.org/10.1007/978-1-4471-5487-7_2

[14] K. R. Chowdhary, “Logic programming and prolog,” in *Fundamentals of Artificial Intelligence*. New Delhi: Springer India, 2020, ch. 5, pp. 111–141. [Online]. Available: https://doi.org/10.1007/978-81-322-3972-7_5

[15] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Halide: decoupling algorithms from schedules for high-performance image processing,” *Communications of the ACM*, vol. 61, pp. 106–115, 12 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3150211

[16] B. Hagedorn, J. Lenfers, T. Koehler, S. Gorlatch, and M. Steuwer, “A language for describing optimization strategies,” *CoRR*, vol. abs/2002.02268, 2020. [Online]. Available: https://arxiv.org/abs/2002.02268

[17] D. R. Silverman, Y. Sun, K. K. Micinski, and T. Gilray, “So you want to analyze scheme programs with datalog?” *CoRR*, vol. abs/2107.12909, 2021. [Online]. Available: https://arxiv.org/abs/2107.12909

[18] E. Visser, “Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5,” in *Rewriting Techniques and Applications*, A. Middeldorp, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 357–361. [Online]. Available: https://doi.org/10.1007/3-540-45127-7_27

[19] B. J. Webb, I. J. Hayes, and M. Utting, “Verifying term graph optimizations using Isabelle/HOL,” in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 1 2023, pp. 320–333. [Online]. Available: https://dl.acm.org/doi/10.1145/3573105.3575673