THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Investigating A Linear Approach to Arithmetic Re-Association within GraalVM

by
Nathan Corcoran

School of Information Technology and Electrical Engineering,
The University of Queensland.

Submitted for the degree of
Bachelor of Engineering
in the field of Software Engineering

March 2024.

Nathan Corcoran
n.corcoran@uqconnect.edu.au

April 7, 2024

Prof Michael Bruenig
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Q 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Software Engineering, I present the following thesis entitled "Investigating A Linear Approach to Arithmetic Re-Association in GraalVM". This work was performed under the supervision of A/Prof. Mark Utting, Prof. Ian J. Hayes and, Mr. Brae Webb.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

–INSERT SIGNATURE–

Nathan Corcoran.

Acknowledgments

Acknowledge your supervisor, preferably with a few short and specific statements about his/her contribution to the content and direction of the project. If you collaborated with another student, acknowledge your partner's contribution, including any parts of the thesis of which s/he was the principal author or co-author; this information can be duplicated in footnotes to the chapters or sections to which your partner has contributed. Briefly describe any assistance that you received from technical or administrative staff. Support of family and friends may also be acknowledged, but avoid sentimentality—or hide it in the dedication.

Abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimising compilers are a crucial tool for building efficient modern programs. They allow developers to write highly abstracted code without sacrificing program performance. High-level abstractions are relied upon when programming complex models that many of modern software requires. The GraalVM compiler, released by Oracle, is one such compiler that aims to achieve high-levels of optimisation. Arithmetic expressions allow for many optimisation opportunities. Arithmetic re-association plays a pivotal role in this. The method is not used to optimise the code by itself, but rather to create opportunities for further optimisations. Consequently, to maximise the effectiveness of an optimising compiler, we must ensure effectiveness of re-association methods. That is, expressions need to be represented in such a way that eases the implementation of optimisations. Currently, the GraalVM compiler optimises expressions by way of graph comparisons and manipulations when the `ReassociationPhase` is invoked. For example, the phase would transform an expression such as `x + y + z - x - y` by grouping like sub-expressions, forming possibly `x - x + y - y + z`. This allows the compiler to easily reason that further optimisations can be applied, including algebraic simplification, resulting in the simplified expression `z`. For lengthy or complex expressions GraalVM's graph manipulations can become computationally expensive and the `ReassociationPhase` is only utilised on expressions within loop bodies [14].

The purpose of this report is to investigate the effectiveness of current method of arithmetic re-association within the GraalVM compiler to a list-based implementation. The report will attempt to determine short-comings of the compiler by way of missed further expression optimisations. A alternative, list-based method of re-association will be investigated to maximise the effectiveness of optimisations within the GraalVM compiler. The report will have the following structure: Section 2.1 will discuss arithmetic re-association following on with Section 2.2 covering some of the further optimisations it aids, Section 2.3 will cover how a compiler can represent source code using graphs, Section 2.4 will discuss the GraalVM compiler, Section 3

will review some previous work on optimising compilers, and finally, Section 4 will present a plan to investigate shortcomings of current methods within the GraalVM compiler as well as a plan to rectify them.

Chapter 2

Background

2.1 Arithmetic Re-Association

Arithmetic re-association utilises the associative and commutative properties of certain arithmetic operators to re-arrange expressions [13]. This may allow simplifications that were not easily identified in the original expression to be made evident. Re-association methods use associative, distributive, and commutative properties of some operators to re-order arithmetic expressions. Using re-association, a compiler would transform the expression `x = 23 + i + y + 43 + i` into the form `x = i + i + y + 23 + 43`. It is now easier for the compiler to reason that the sub-expression `23 + 43` can be optimised to a single constant. Arithmetic re-association only alters the representation of a given expression, and not its resulting value. Hence, it can only be applied to operators that are associative. Due to hardware limitations, the additive operator is not associative across floating-point values, therefore, re-association methods are generally not applied [12].

2.2 Optimisations

2.2.1 Algebraic Simplification

Algebraic simplification can allow for large performance optimisations in programs. It is the process of simplifying expressions which can produce smaller but equivalent expressions, or replace operators with which are faster to compute. This process utilises mathematical properties such as associativity; commutativity; distributivity; and operator identities. Some simple examples of algebraic simplification can be seen in Fig. 2.1.

```
-(-a) --> a
a - a --> 0
a * 2 --> a << 1
2 * a + 3 * a --> (2 + 3) * a
```

Figure 2.1: Examples of algebraic simplifications

```
for i = 1..10 {
    a = i + b * 3
}
```

```
b0 = b * 3
for i = 1..10 {
    a = i + b0
}
```

Figure 2.2: Example of before and after hoisting respectively

2.2.2 Constant Folding

Constant folding is the process of trivially merging sequences of sub-expressions containing only constant values. Simply, the expression $2+3-1$ would be transformed to be 4.

2.2.3 Loop Invariant Code Motion

Loop invariant code motion, also referred to as *hoisting*, aims to reduce the number of computations required within loop bodies. If a sub- expression is reasoned to be constant, that is invariant, across all loop iterations, this value can be stored elsewhere and its resulting value can be referenced inside of the loop. A trivial example of this can be seen in Fig. 2.2. More complex scenarios arise when function calls are hoisted, yeilding larger performance optimisations.

2.3 Intermediate Representation

Generally, compilers do not perform optimisations directly on high-level code. Instead, an *intermediate representation* (IR) is used to represent the program in such a way that optimisations are easily implemented. Intermediate representations are split into three levels: high, medium, and low. Each level generally represents the source code in different ways, these differences are defined by the optimisations expecting to be performed. Common forms of IR are graph-, tree-, and stack-based structures. These are usually selected depending on how much information is needed from the source code in order to apply an optimisation. This report will focus on graph-based intermediate representation.

2.4 GraalVM

The GraalVM compiler, released by Oracle Labs, is a polyglot optimising compiler for languages that run on the Java Virtual Machine (JVM). It focuses on aggresively optimising programs, with around 62 possible methods [7] of doing so.
The GraalVM compiler uses a graph-based structure as an intermediate representation. Graphs are an efficient way to store program flow contexts. Nodes represent basic code blocks or data values, and directed edges represent dependencies between nodes. Lacking circular dependencies [3], the graph is more accurately a directed acyclic graph (DAG). DAGs minimise redundancy by sharing common sub-graphs. The GraalVM IR also uses a single graph to model both control- and data-flow, creating a *program-graph* [11]. Traversing control edges outlines the order in which a program must be run, whilst traversing data edges outlines how data is passed and what instructions are manipulating it throughout the program's run. This benefits the implementation of optimisations as both contexts are easily accessed. Though, this makes graphs with nested or complex control flows messy and difficult to follow. Hence, it is often stated that GraalVM uses a *sea of nodes* to represent program behaviour.
Within GraalVM optimisations are referred to as *phases*. Execution order of phases are defined as `PhaseSuite`s, an ordered list of optimisations to be applied in sequence. `PhaseSuite`s can be scheduled within other `PhaseSuite`s. The compiler uses three tiers to run phases; low, mid, and high. Progressing from the high to the low tier, lower and lower level optimisations are applied to the graph. After each tier has completed, the graph is prepared for the next, called *lowering* [9]

Chapter 3

Literature Review

3.1 Effective Partial Redundancy Elimination

This report addresses methods to increase the effectiveness of partial redundancy elimination within an optimising compiler. Cooper and Briggs [8] propose a method of *global reassociation* to re-arrange expressions within a program.
The report uses a system to sort sub-expressions by ranking the operators within them. It uses these ranks as a hueristic to determine a sub-expressions new location within an expression and to determine which sub-expressions should be distributed. This report will investigate the methods proposed to determine any benefit in a the GraalVM compiler, though some concerns are immediately apparent. The report proposes the use of *expression trees* to store sub-expressions and their associated ranks, and are forward-propagated. This is generally a space-expensive operation. This report will investigate the pay off from gained optimisations, if any, to the excessive memory usage that may present itself when re-associating very large and complex expressions. The report also explicitly states that what it proposes is not an optimal solution, but rather one that produces good results. This report will investigate extensions to the method presented for optimal arithmetic re-association.

3.2 Finding Missed Compiler Optimizations by Differential Testing

This report investigates the usage of differential testing to detect missed compiler optimisations. Barany [10] generates random programs with well defined source code constraints and compares generated binaries across 3 C compilers; GCC, Clang, and CompCert. The binaries are compared with a Python tool called `optdiff`. The randomly generated code was restricted by omitting any code that would (A) invoke undefined behaviour from the language, and (B) rely on compiler-defined

implementations of certain features, order of evaluation of expressions is presented as an example. The report correctly identifies some difficulties in using differential analysis on generated binaries, however, reducers are presented as a sound solution to these issues.
The report scope is extremely broad in that it attempts to catch missed opportunities from a large range of expected optimisations, some of which are explicitly stated as begin architecture-specific. The report also states that some initial results were found to be false-positive.
This report will investigate the usage of differential analysis to find missed optimisations within the GraalVM compiler. In an attempt to maximise the validity of any findings, this report will narrow the scope of missed optimisations to be searched for, and methods data used when comparing. This report will investigate this testing method focusing on finding missed optimisations relating to arithmetic expressions, and the usage of generated IR graphs for comparison. A method to randomly generate large complex expressions with corresponding simplified equivalents will be investigated. In order to uphold correctness, a verification method will also be investigated.

3.3 Redundancy Elimination Revisited

Cooper, *et al.* [13] present improvements to the method of redundancy elimination through arithmetic re-association. The re-association methods proposed utilise hash-based value numbering to identify equivalent sub-expressions. A method of scalar- replacement is also given, however, this review will focus solely on re-association.
The report expresses concerns with previous methods of ranking expressions by operators or variable names, and proposes the use of a frequency-based affinity instead. Sub-expressions are ranked by their frequencies within a program source and stored in an undirected affinity graph, where redundant subexpressions are found as *maximal-cliques*. The report expresses that finding this is not entirely easy, though many studies are available on the relevant algorithms. The method is finalised with a method to identify the found redundant subexpressions.
This report will investigate the practicality of utilising such a method in a compiler such as GraalVM, including its value-numbering extension to handle function calls within expressions. The method will be investigate by the number of newly found optimisations, especially when dealing with large complex expressions. The methods proposed by the report are predicted to not scale very well having an $O(n^2)$ time complexity. This report will investigate the effect of using a list-based implementation in the hopes of achieving an $O(n)$ time complexity.

Chapter 4

Methodology

In order to maintain a sound workflow and maximise findings, this project will adhere to the following milestones:

Milestone	**Proposed Deadline**
Research	Week 7 Semester 1
Implementation	Week 12 Semester 1
Optimisation	Week 7 Semester 2
Report	Week 13 Semester 2
Testing	Throughout

Table 4.1: Proposed Milestones

Assessment Task	**Due Date**
Academic Integrity Quiz	14 March
Proposal Draft	21 March
Project Proposal	18 April
Progress Seminar	10 May
Poster & Demonstration	18 October
Thesis Report	04 November

Table 4.2: Assessment Due Dates

4.1 Research

Research for the project will consist of the following:

- Further investigate expression re-association techniques
- Further investigate methods of testing compiler optimisations

- Develop implementation plan for alternate expression re-association techniques
- Develop implementation plan for testing alternate methods
- Become familiar with the GraalVM compiler structure

4.2 Implementation

Implementation for the project will consist of the following:

- Develop expression re-association algorithm
- Integrate expression re-association algorithm in to GraalVM compiler
- Consistent meetings with supervisors to review implementation

4.3 Optimisation

Optimisation for the project will be completed if deemed necessary and will consist of the following:

- Profile implementation to identify potential performance bottlenecks
- Plan implementation performance fixes
- Consistent meetings with supervisors to review code fixes

4.4 Report

Report writing for the project will consist of the following:

- Consistent upkeep of progress journal
- Consistent logging of findings
- Determining future improvements of findings
- Determining further usages of re-association algorithm
- Consistent meetings with supervisors to discuss potential edits

4.5 Testing

Testing within the project will be continuous and will ultimately shape the final implementation. Testing for the project will consist of the following:

- Develop testing of compiler optimisations that may be missed
- Develop complex expressions to test compiler optimisation
- Investigate method of automatic running of tests
- Consistent meetings with supervisors and peers for recommendations of testing methods

4.6 Risks

The following risks have been determined for the project along with associated mitigation and prevention strategies:

Risk	**Likelihood**	**Consequence**	**Mitigation**	**Prevention**
Loss of work	Unlikely	Severe	Manage multiple instances of progress	Store progress on UQ server and save and push regularly
Milestone delays	Unlikely	Moderate	Create time management plan	Consistent checking of time management plan and meeting with supervisors
Loss of project direction	Unlikely	Low	Create clear task plan	Consistent meeting with supervisors to review work

Table 4.3: Project Risks

Risk	**Likelihood**	**Consequence**	**Mitigation**	**Prevention**
Unsafe use of Computer	Unlikely	Moderate	Comfortable workspace	Ensure regular breaks during computer usage

Table 4.4: Safety Risks

Chapter 5

Results and discussion ...

Chapter 6

Conclusions

Bibliography

[1] Oracle. 2024. *GraalVM: An advanced JDK with ahead-of-time Native Image compilation*. https://github.com/oracle/graal (accessed Mar. 9, 2024)

[2] Brae J. Webb, Ian J. Hayes, and Mark Utting. 2023. *Verifying Term Graph Optimizations using Isabelle/HOL*. In Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3573105.3575673

[3] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, Hanspeter Moessenboeck. 2013. *Graal IR: An Extensible Declarative Intermediate Representation*. https://api.semanticscholar.org/CorpusID:52231504

[4] OpenJDK Community. 2012. *Graal Project*. https://openjdk.org/projects/graal (accessed Mar. 18, 2024)

[5] Cliff Click, Michael Paleczny. 1995. *A Simple Graph-Based Intermediate Representation*. https://www.oracle.com/technetwork/java/javase/tech/c2-ir95-150110.pdf

[6] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Chapter 12. ISBN 1-55860-320-4

[7] Oracle. 2024. *Oracle GraalVM Enterprise Edition 21*. https://docs.oracle.com/en/graalvm/enterprise/21/docs/reference-manual/java/compiler/#graal-compiler (accessed Mar. 17, 2024)

[8] Preston Briggs and Keith D. Cooper. 1994. *Effective partial redundancy elimination*. SIGPLAN Not. 29, 6 (June 1994), 159–170. https://doi.org/10.1145/773473.178257

[9] M. Sipek, B. Mihaljevic, A. Radovan. 2021. *Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM*. https://doi.org/10.23919/MIPRO.2019.8756917

[10] Gergoe Barany. 2018. *Finding Missed Compiler Optimizations by Differential Testing*. Compiler Construction (CC'18). ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179521

[11] Chris Seaton. 2017. *Understanding How Graal Works - a Java JIT Compiler Written in Java*. https://chrisseaton.com/truffleruby/jokerconf17/ (accessed Mar. 18, 2024)

[12] Martyn J. Corden, David Kreitzer. 2018. *Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?*. Software Services Group, Intel Corporation. https://www.intel.com/content/dam/develop/external/us/en/documents/pdf/fp-consistency-121918.pdf

[13] Keith Cooper, Jason Eckhardt, Ken Kennedy. 2008. *Redundancy Elimination Revisited*. PACT'08, October 25–29, 2008, Toronto, Ontario, Canada. https://cscads.rice.edu/p12-cooper.pdf

[14] Oracle. 2015-2022. *GraalVM Reassociation Phase Source*. https://github.com/oracle/graal/blob/master/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/phases/common/ReassociationPhase.java (accessed Mar. 15, 2024)

[15] David Monniaux, Cyril Six. 2022. *Formally Verified Loop-Invariant Code Motion and Assorted Optimizations*. ACM Transactions on Embedded Computing Systems (TECS). ff10.114/3529507ff.hal03628646