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

Guilherme Vieira Leobas

Semiring Optimization: Dynamic Elision of Expressions with Identity and Absorbing Elements

Belo Horizonte
2019

Guilherme Vieira Leobas

Semiring Optimization: Dynamic Elision of Expressions with Identity and Absorbing Elements

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

Belo Horizonte
2019

Leobas, Guilherme Vieira.

L576s Semiring optimization:[recurso eletrônico] dynamic elision of expressions with Identity and absorbing elements / Guilherme Vieira Leobas – 2019.
1 recurso online (62 f. il, color.) : pdf.

Orientador: Fernando Magno Quintão Pereira.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f. 60-62

1. Computação – Teses. 2. Otimização – Teses. 3. Compiladores (Computadores) - Teses. 4.– Anéis semilocais – Teses. I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III. Título.

CDU 519.6*21(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

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

FOLHA DE APROVAÇÃO

RING OPTIMIZATION: DYNAMIC ELISION OF EXPRESSIONS WITH IDENTITY AND ABSORBING ELEMENTS

GUILHERME VIEIRA LEOBAS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

Edson Borin
PROF. EDSON BORIN
Instituto de Computação - UNICAMP

[illegible]
PROF. GEORGE LUIZ MEDEIROS TEODORO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 3 de Outubro de 2019.

Acknowledgments

O mestrado foi um momento muito importante na minha vida. Concluir essa etapa demonstrou que eu era capaz de ir muito além do que o Guilherme, recém ingresso do curso de Ciência da Computação, em 2012, imaginou.

Agradeço aos meus pais, Antônio Leobas dos Santos e Angela Marisa Vieira Leobas, que me apoiaram incondicionalmente durante esse período. Agradeço também ao meu orientador, Fernando Magno, por acreditar na minha capacidade e pelo suporte e ensinamento durante esses anos de mestrado.

Gostaria também de agradecer a Abdoulaye Gamatiê por fornecer orientação e feedback ao longo do projeto. Além disso, os meses que passei no *Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier* (LIRMM) sendo orientado por Abdoulaye foram extremamente valiosos para o desenvolvimento deste projeto.

Aos meus amigos de laboratório: Breno, Bruno, Carina, Caio, Junio, Marcelo, Marcus, Pedro Caldeira, Pedro Ramos, Roberto, Tarsila e Yukio, por toda a ajuda e companhia desde 2017. Agradeço também aos meus colegas de graduação: Amanda, Francisco, Gabriel, Guilherme, Henrique, Lucas, Marcelo, Mariana, Marina e Thiago, que dividiram grande parte dessa jornada desde 2012.

"Hilbert said once: "We must know. We will know". The truth is: We don't know. Sometimes, we can't know. But in trying to find out, we can discover new things, things that change the world"

(Derek Muller)

Resumo

Essa dissertação descreve uma técnica de otimização que elimina ocorrências dinâmicas de expressões no formato $a = a \oplus b \otimes c$. A operação $\oplus$ deve admitir um elemento identidade z, de forma que $a \oplus z = z \oplus a = a$. Além disso, z deve ser o elemento anulador da operação $\otimes$, de forma que $b \otimes z = z \otimes c = z$. Semi aneis onde $\oplus$ é a operação de adição e $\otimes$ é o operador de multiplicação cumprem esse contrato. Esse padrão é muito comum em benchmarks de alta-performance – seu representante canônico é a operação de adição e multiplicação $a = a + b \times c$. No entanto, varías outras expressões envolvendo operações aritméticas e lógicas podem ser agrupadas dentro da álgebra necessária. Nós mostramos que a eliminação em tempo de execução de tais atribuições podem ser implementadas em uma maneira segura do ponto de vista de performance utilizando perfilamento *in-loco*. A eliminação dinâmica de expressões redundantes envolvendo identidade e elemento anulador em 35 programas da suíte de testes do LLVM é responsável por um ganho de velocidade de 1.19x (tempo total otimizado sobre tempo total não otimizado) quando se comparado com o `clang` -O3. Quando aplicado a toda a suíte de testes (259 programas), a otimização leva a um ganho de 1.025x.

Quando adicionadas ao `clang`, a otimização de semi anel aproxima aquele sistema de TACO, um compilador especializado de álgebra tensorial.

Palavras-chave: compiladores; otimização de código; geração de código; semi-anel.

Abstract

This dissertation describes a compiler optimization to eliminates dynamic occurrences of expressions in the format $a \leftarrow a \oplus b \otimes c$. The operation $\oplus$ must admit an identity element z, such that $a \oplus z = a$. Also, z must be the absorbing element of $\otimes$, such that $b \otimes z = z \otimes c = z$. Semirings where $\oplus$ is the additive operator and $\otimes$ is the multiplicative operator meet this contract. This pattern is common in high-performance benchmarks—its canonical representative being the multiply-add operation $a \leftarrow a + b \times c$. However, several other expressions involving arithmetic and logic operations satisfy the required algebra. We show that the runtime elimination of such assignments can be implemented in a performance-safe way via online profiling. The elimination of dynamic redundancies involving identity and absorbing elements in 35 programs of the LLVM test suite that present semiring patterns brings an average speedup of 1.19x (total optimized time over total unoptimized time) on top of clang -O3. When projected onto the entire test suite (259 programs) the optimization leads to a speedup of 1.025x. Once added onto `clang`, semiring optimizations approximates it to TACO, a specialized tensor compiler.

Keywords: compilers; code optimization; code generation; semiring.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Expressions that fit into the pattern $a = a \oplus b \otimes c$ are common in programs. In this pattern, $\oplus$ and $\otimes$ are binary operations of type $T \times T \to T$. Operator $\oplus$ has an *identity* element z, which is the absorbing element of operator $\otimes$. Therefore, $z \oplus x = x \oplus z = x$ for any $x \in T$, and $z \otimes x = x \otimes z = z$. The multiply-add pattern $\mathtt{m}[\mathtt{i},\mathtt{j}] {+}{=} \mathtt{p}[\mathtt{i},\mathtt{k}] \times \mathtt{q}[\mathtt{k},\mathtt{j}]$, heart of matrix multiplication, is an example of this family of expressions. In Section 4.1 we show that this family is well-provided with a rich assortment of members. Indeed, any triple $(T, \oplus, \otimes)$ forming an algebraic *semiring* [9, Chapter IX] meets the required constraints. In this work, we show that the pattern $a = a \oplus b \otimes c$ is amenable to a kind of transformation, henceforth called *semiring optimization*, that can lead to great profit.

Because z is the identity of $\oplus$, the operation $a = a \oplus e$ is *silent* whenever $e = z$. Thus, this assignment can be replaced by a conditional statement that only allows its execution when $e \neq z$. Figure 1.1-i illustrates this code transformation. Similarly, the operation $a = b \otimes c$ can be reduced to $a = z$ whenever $b = z$ or $c = z$. Figure 1.1-ii shows an example of the latter simplification. In its most general form, the expression $a = a \oplus b \otimes c$ is silent whenever b or c are the absorbing element. Figure 1.1-iii outlines the general transformation.

(i) $\boxed{a \leftarrow a + e}$

```
t0 = ld a
t1 = ld e
t2 = t0+t1
st a t2
```

```
t1 = ld e
if (t1) {
  t0 = ld a
  t2 = t0+t1
  st a t2
}
```

(ii) $\boxed{a \leftarrow b \times c}$

```
t0 = ld b
t1 = ld c
t2 = t0*t1
st a t2
```

```
t0 = ld b
if (t0) {
  t1 = ld c
  if (t1) {
    t2 = t0*t1
    st a t2
  }
}
```

(iii) $\boxed{a \leftarrow a + b \times c}$

```
t0 = ld a
t1 = ld b
t2 = ld c
t3 = t1*t2
t4 = t3+t0
st a t4
```

```
t1 = ld b
if (t1) {
  t2 = ld c
  if (t2) {
    t3 = t1*t2
    t0 = ld a
    t4 = t3+t0
    st a t4
  }
}
```

Figure 1.1: The naïve implementation of semiring optimizations.

The Issue of Performance Safety. The code transformations seen in Figure 1.1 try to eliminate the execution of some instructions by guarding them with conditional checks. Depending on the values loaded from memory, they can save several operations. For instance, whenever variable b in Figure 1.1-iii is zero, none of the instructions between lines 3 and 9 of the transformed program executes. However, the unrestricted application of such transformations might downgrade performance instead of improving it. Regressions are due to the insertion of branches into otherwise straight-line code. Branches add an extra burden onto the branch predictor, compromise the formation of super-blocks [11], make register allocation more difficult and, most of all, hinder vectorization. As we show in Section 6, the impossibility to apply vectorization into heavily nested loops might double the runtime of some of the programs in the Polybench suite.

In this dissertation, we show how to implement semiring optimizations in a *performance safe way*. By performance safety, we mean that if a program runs for a sufficiently long time, then either the code transformation improves its speed, or does not change it in any statistically significant way. The key to achieve performance safety is profiling. Profiling, in this case, is applied *on-line*, that is, while the optimized program runs. We evaluate four different ways to carry out semiring optimizations –two of which resort to on-line profiling:

- Section 5.1 discusses the simplest implementation of semiring optimization: the conditional elimination of silent stores due to operations involving the identity element.
- Section 5.2 discusses the conditional elimination of loads whose values are absorbing elements, and of the forward program slice that depends on such loads.
- Section 5.3 presents an on-line profiling technique implemented within loops that avoids checking loads that do not contain absorbing elements.
- Chapter 5.4 shows how to move the code in charge of on-line profiling outside the loop. This code hoisting harmonizes semiring-optimization and vectorization.

Summary of Results. We have implemented the four variations of the proposed optimization in LLVM 6.0.1 [18]. Even though this optimization may appear, at first, innocuous, its effects are impressive when properly implemented. Chapter 6 supports this statement with experiments performed onto the LLVM test suite. Optimizable patterns appear within loops of 126 programs, 35 of which run for above one second when given their standard inputs. Our of this lot, 20 programs belong into PolyBench, and 13 to TSVC. The naïve optimization (Section 5.1) improves performance, with a significance level $\alpha = 0.05$, in 12 benchmarks (out of 35). These results use, as baseline, LLVM -O3. We have observed speedups of 1.48x, 1.45x and 1.11x in different LLVM benchmarks, for instance. However, we have also observed slowdowns of 1.26x and 1.25x in two benchmarks. This scenario improves as we move from the naïve elimination of silent stores towards the online profiler hoisted outside loops (Section 5.4). In its fourth, and most effective implementation, our optimization causes a maximum slowdown of 1.13x in one benchmark, while maintaining all the previous speedups. Overall, it delivers a performance improvement of 1.19x on the 35 benchmarks, when compared to LLVM -O3 `-ffast-math`[1]. Additionally, we show that when equipped with semiring optimization, vanilla LLVM can output code for matrix multiplication on par with the code produced by TACO [14], a code generation engine specialized in the compilation of linear algebra applications.

1.1 Tools and Publications

The contributions of this dissertation are the result of four articles published during two years of research.

- *Leobas, G. V., and Pereira, F. M. Q. a. (2020). Semiring Optimizations: Dynamic Elision of Expressions with Identity and Absorbing Elements (OOPSLA '20)*

  This article is the direct result of two years of research. In this article, we show how one can leverage semiring optimizations to eliminate dynamic occurrences of expressions in the format $a \leftarrow a \oplus b \otimes c$. We show that the runtime elimination of such assignments can be implemented in a performance-safe way via online profiling. The elimination of dynamic redundancies involving identity and absorbing elements in 35 programs of the LLVM test suite that present semiring patterns brings an average speedup of 1.19x (total optimized time over total unoptimized time) on top of clang -O3.

- *Pereira, F. M. Q. a., Leobas, G. V., and Gamatie, A. (2018). Static prediction of silent stores. ACM Transactions on Architecture and Code Optimization (TACO)*

  In this article, we tried to answer the following question: is it possible to predict the silentness of a store instruction by analyzing the source code? To answer this question, we have combined static analysis techniques and Machine Learning to classify store operations in terms of syntactic features of programs. The conclusions of this dissertation led to the foundations of this research with Semiring Optimizations.

[1]This number is the result of dividing the total running time of programs compiled without semiring optimizations by the running time of the programs compiled with it. The geometric mean of speedups is 1.06x.

- *Leobas, G. V., Guimarães, B. C. F., and Pereira, F. M. Q. (2018). More than meets the eye: Invisible instructions. In Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP '18*

  In this research, we introduce the notion of invisible instructions, which are instructions present in the binary but are not visible in the program's compiler-generated intermediate representation. We use static analysis and profiling techniques to measure the prevalence of these instructions for a wide variety of programs in several benchmark suites, and show that for some instruction types, up to 36% of its occurrences on average are invisible.

- *Leobas, G. V. And Pereira, F. M. Q. Semiring Optimizations: Dynamic Elision of Expressions with Identity and Absorbing Elements, OOPSLA '20,*

  A small testing framework built with composability in mind. TF is capable of many things, including but not limited to:

  i Parallel compilation and execution using *gnu-parallel* [31]

  ii Run programs with a time limit

  iii Easily collect statistics

  iv Instrument programs using Intel PIN [22], Perf [5] or Valgrind [25]

- *Leobas, G. V.*, ***LLVM Test-Suite Benchmarks***, GitHub repository, `https://github.com/lac-dcc/Benchmarks`

  A collection of benchmarks available in the LLVM test-suite composed of 260 benchmarks from 36 test-suites.

Chapter 2

Background Information

In this chapter, we will cover the mathematical definitions of algebraic structures on section 2.1. Additionally, we will present the LLVM Compiler Infrastructure on section 2.2. The last section (sec. 2.3) will introduce concepts and data structures used for program analysis.

2.1 Mathematical Definitions

In this section we focus our attention on the mathematical definitions of Semiring Optimization. We will cover some of the fundamental algebraic structures used in abstract algebra as well as the characterization of operations.

Definition 2.1.1. Set and Element
In mathematics, a set R is a collection of distinct objects. The different objects that compose the set are called elements. For example, the collection of letters of the latin alphabet represents a set while the letters symbolize the elements.

Definition 2.1.2. Binary Operations $\odot$
A binary operation is an operation of arity two that combines two elements to produce another element. Operations may have properties. For instance, an operation can be commutative (2.1), distributive (2.2) and associative (2.3)

$$a \oplus b = b \oplus a \tag{2.1}$$

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \tag{2.2}$$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{2.3}$$

Definition 2.1.3. Identity, Absorbing and Invertible elements
With respect to a binary operation $\odot$, an identity element I is a type of element which when combined with another element leaves the set R unchanged: $a \odot I = I \odot a = a$.

On the other hand, the absorbing element z when combined with other elements of R produces the absorbing element itself: $a \odot z = z \odot a = z$. Moreover, the invertible element a^{-1} is defined as: $a \odot a^{-1} = a^{-1} \odot a = I$.

Definition 2.1.4. Closure
A set R is said to be closed under an operation $\odot$ if the application of that operation on members of the set always produces a member $\in R$. For example, the set of positive integers are closed under multiplication.

Definition 2.1.5. Algebraic structures
Semigroup is an algebraic structure consisting of a set of elements R and an associative operation $\odot$.
Monoid is a semigroup combined with an identity element I.
Group is a set R with a binary operation $\odot$. The set R must be closed under $\odot$ and have identity and invertible elements. The binary operation $\odot$ must be associative.
Abelian group is a commutative group. That is, the result of applying the group operation over two elements does not depend on their order.
Semiring is a set R with two binary operations. These operations are usually called addition ($\oplus$) and multiplication ($\otimes$). In order to be a semiring, the set R must satisfy the following conditions:

(i) $a \oplus b \in R$

(ii) Commutative under $\oplus$

(iii) Associative under $\oplus$

(iv) For every $a \in R$, there should be an identity element I such that $a \oplus I = I \oplus a = a$

(v) There should be a 0 element such that $a \oplus (-a) = 0$

(vi) $a \otimes b \in R$

(vii) Commutative under $\otimes$

(viii) $\otimes$ must be distributive with respect to $\oplus$

The previous conditions stated that: (i) R must be an Abelian group under addition operation; (ii) R is a Monoid under multiplication; and (iii) multiplication should be distributed over addition.

Lemma 2.1.6. The additive identity is unique in a group;

Proof. Let G be a group under addition and let e and f be two distinct identities. So, for any $g \in G$ and by the definition of identity, the following conditions must hold:

$$\begin{aligned} e + g = g + e = g \\ f + g = g + f = g \end{aligned}$$

Let $g = f$ and $g = e$ on the two equations below

$$\begin{aligned} &e + f = f + e = f && (g = f) \\ &f + e = e + f = e && (g = e) \end{aligned}$$

On the first equation, we have that $e + f = f$ and on the second equation, we have that $e + f = e$. Thus, this implies that $f = e$ and therefore both identities are the same element.

□

Lemma 2.1.7. The additive identity is always the multiplicative absorbing element;

Proof. Let 0 be the additive identity in a semiring R. For any element $x \in R$, the condition must hold:

$$\begin{aligned} &x + 0 = 0 + x = 0 && \text{(identity definition)} \\ &x \cdot 0 = 0 \cdot x = 0 && \text{(absorbing definition)} \\ &\Downarrow && \\ &x \cdot 0 = x \cdot 0 && \\ &x \cdot 0 = x \cdot (0 + 0) && \\ &x \cdot 0 = (x \cdot 0) + (x \cdot 0) && \text{(distributive rule)} \\ &x \cdot 0 = 0 + 0 && \\ &x \cdot 0 = 0 && \end{aligned}$$

Thus, for any value of $x \in R$, $x \cdot 0 = 0$. Therefore, 0 is the additive identity and the multiplicative absorbing element. □

2.2 The LLVM Compiler Infrastructure

The LLVM Compiler Infrastructure [18] is an umbrella project that hosts and develops a set of modular tools with well-defined interfaces used to compile, optimize and debug programs. The name "LLVM" was once an acronym for *Low Level Virtual Machine* but it was removed to avoid confusion with Virtual Machines. LLVM is designed around a language-independent intermediate representation called LLVM IR. This representation serves as a portable high-level assembly language that can be optimized for a diversity of architectures, such as Intel x86, IBM Power PC and ARM.

Figure 2.1 shows the (i) C source code and (ii) its LLVM IR representation for a program that computes the factorial of a given number. Notice that the intermediate representation is a typed, 3-address, statement based intermediate representation.

(i) C source code

```c
int fact(int n){
  if (n <= 1)
    return 1;
  return n * fact(n-1);
}
```

(ii) LLVM IR

```llvm
define i32 @fact(i32 %n){
entry:
  %cmp = icmp slt i32 %n, 2
  br i1 %cmp, label %return, label %if.end

if.end: ; preds = %entry
  %sub = add nsw i32 %mul, -1
  %call = call i32 @fact(i32 %sub)
  %mul = mul nsw i32 %call, %n
  ret i32 %mul

return:  ; preds = %entry
  ret i32 1
}
```

Figure 2.1: (i) C source code and (ii) its LLVM IR counterpart.

The project was originally implemented to compile C and C++ using the Clang front-end, but because of its language-agnostic design, LLVM is now the target of many front-ends (i.e. Ada, C#, C, C++, Lisp, D, Rust, Fortran, Python, Java, Kotlin, ...). Figure 2.2 shows the three phrase design adopted by LLVM. In this scheme, creating a compiler for a new language only requires the development of the front-end and LLVM will take care of the optimization and code generation steps.

2.3 Static Program Analysis

Definition 2.3.1. Control Flow Graph (CFG)

In a control flow graph each node in the graph represents a basic block, i.e., a straight line sequence of code with no jumps except at the end of the block Jump targets start a basic block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through

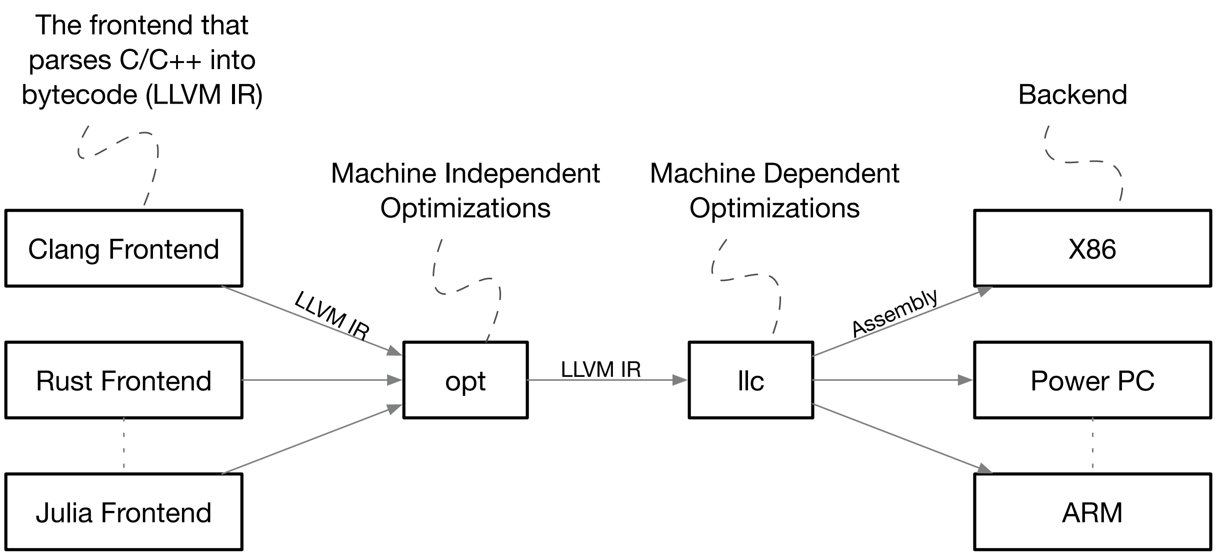

The frontend that parses C/C++ into bytecode (LLVM IR)
Machine Independent Optimizations
Machine Dependent Optimizations
Backend
Clang Frontend
Rust Frontend
Julia Frontend
LLVM IR
opt
LLVM IR
llc
Assembly
X86
Power PC
ARM

Figure 2.2: High level representation of LLVM Architecture. LLVM is language-agnostic, supporting multiple front-ends and back-ends.

which all control flow leaves. Figure 2.3 illustrates a CFG for a program that computes the factorial of `n`.

Definition 2.3.2. Dominance A node $X \in$ CFG dominates another node $Y \in$ CFG if every path from the start node to Y goes through X. On Figure 2.3, node 2 dominates all subsequent nodes but node 4 does not dominate 5. There is a direct path from 1 to 5

```
int fact(int n){
  int p = 1;
  int i = 0;
  while(i < n){
    i = i+1;
    p *= i;
  }
  return p;
}
```

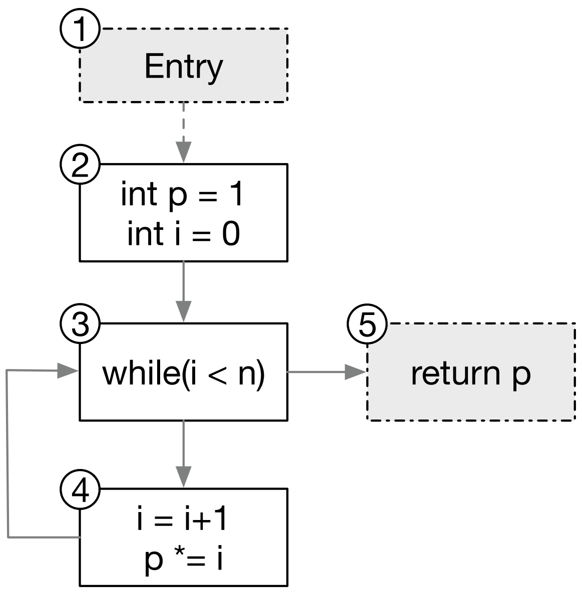
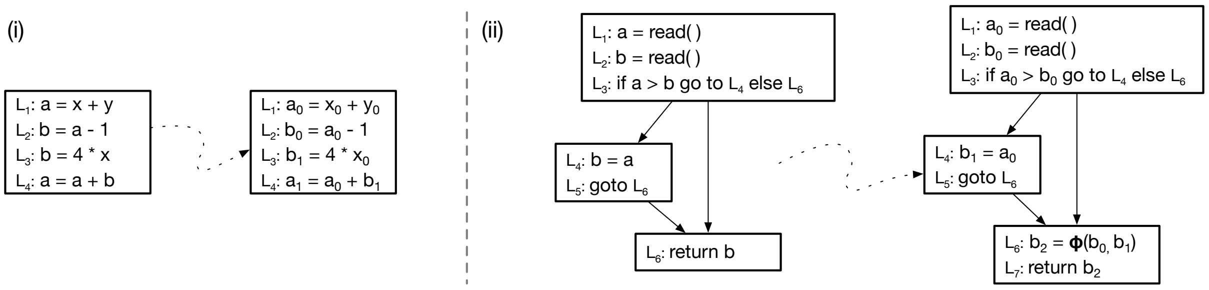


Figure 2.3: Control Flow Graph for the **fact** function. Nodes are basic blocks and edges are used to represents jumps between two blocks.

that does not go through 4. Similarly, a node Y post-dominates X if every path from X to end goes through Y. On Figure 2.3, node 3 post-dominates all previous nodes.

Definition 2.3.3. Static Single Assignment form (SSA)
The Static Single Assignment form (SSA) [4] is an intermediate representation with two fundamental properties: (i) Every variable is assigned exactly once and (ii) every definition dominates its uses. SSA is so important because it simplifies considerably the design and implementation of many compiler analyses and transformations. Any major compiler nowadays implements SSA in its internal representation, i.e., LLVM [18], GCC [7], Soot [30], Erlang OTP [8], WebKit JavaScriptCore [32].

Figure 2.4 shows the conversion of two different programs in the three-address code to SSA form. In a straight-line source code (fig. 2.4(i)), the conversion to SSA is straightforward: every assignment creates a new variable. However, in a program with branches (fig. 2.4(ii)), converting the program requires the usage of a special type of node: PHI nodes (ϕ). A PHI function works as a multiplexer, selecting the value depending on the path taken.



```
(i)
L1: a = x + y          L1: a0 = x0 + y0
L2: b = a - 1          L2: b0 = a0 - 1
L3: b = 4 * x          L3: b1 = 4 * x0
L4: a = a + b          L4: a1 = a0 + b1

(ii)
L1: a = read( )                   L1: a0 = read( )
L2: b = read( )                   L2: b0 = read( )
L3: if a > b go to L4 else L6     L3: if a0 > b0 go to L4 else L6

L4: b = a                         L4: b1 = a0
L5: goto L6                       L5: goto L6

L6: return b                      L6: b2 = Φ(b0, b1)
                                  L7: return b2
```

Figure 2.4: SSA transformation for programs with and without branches

Definition 2.3.4. Program Dependence Graph (PDG)
A *Program Dependence Graph (PDG)* is a graph representation introduced by [16] and later by [6] that makes data dependencies and control dependencies explicit. The *dependence graph* is a digraph $G(V, A)$ whose vertices are program statements and whose arcs are one of the two dependence relations.
Data Dependency: A variable `v` is said to be data dependent on `u` if `u` is used to compute `v`. For instance, the statement in line 5 is data dependent on the statement of line 1.
Control Dependency: A variable `v` is control dependent on `p` if `p` is used as the predicate of a branch that determines the value that `v` is assigned. For example, the statement in line 6 is controlled by the predicate on line 4.

Figure 2.5 illustrates a PDG for a program that computes the factorial of `n`.

```
int fact(int n){
  int p = 1;
  int i = 0;
  while(i < n){
    i = i+1;
    p *= i;
  }
  return p;
}
```

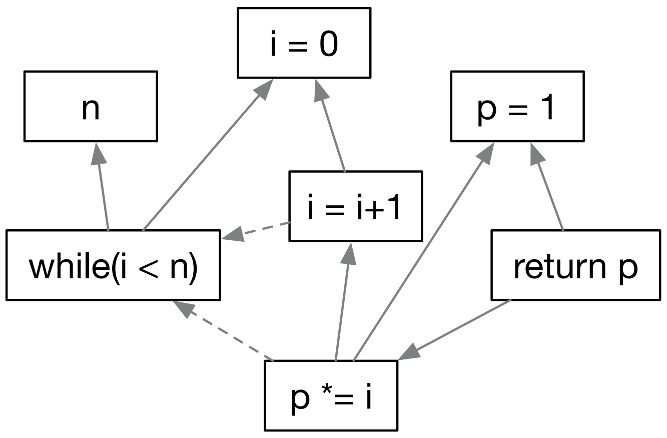


Figure 2.5: Program Dependence Graph for the **fact** function. Solid edges represent data dependences and dashed edges represent control dependencies.

Definition 2.3.5. Program Slicing
A *program slice* [33] with respect to instruction ι in a program is the subset of statements that are necessary to compute ι. A program dependency graph is used to compute the set of data and control dependencies among variables. A program slicing can be used in debugging to find errors more easily or in our case, to automatically generate a sampling function.

2.4 Literature Review

Much of the inspiration behind this work came from the recent developments in the investigation of silent stores. The term *silent store* was coined by *Lepak and Lipasti* [19] in the early 2000's. It denotes a store operation that deposits in memory a value that was already there. Together with his collaborators, Kevin Lepak showed that silent stores are prevalent among well-known benchmarks, and that it is possible to build hardware that mitigates their overhead [1, 20, 21]. More recently, different research groups showed that it is possible to use profiling techniques to help developers to uncover and remove them [12, 34, 35]. Finally, in 2018 [26] showed how to predict store operations that are likely to be silent statically.

The present dissertation differs from this foregoing literature in two ways. First, none of these previous works attempt to remove silent stores automatically via code generation techniques. In contrast, we propose a compiler optimization that affects the target program without any intervention from users. Second, although we chose to eliminate only semiring patterns that can lead to silent stores, the theoretical framework that we propose in this work goes beyond that. As an example, Figure 1.1(b) shows an example of semiring optimization that is not associated with any silent store.

The kind of patterns that we optimize are common in tensor algebra, such as the ever-present multiply-add operation. Incidentally, this kind of algebra has become very fashionable in recent years. Today, there are specialized compilers that generate high-quality code for tensor products, such as TACO [14, 13], Sparso [29] and TVM [2]. In Section 6 we show that the optimizations that we introduce in this work can bring a general compiler closer to TACO, which is probably the state-of-the-art tool in terms of tensor compilation. However, we do not see semiring optimizations as a competing approach. On the contrary, we believe that this technique could be used to enhance even further these specialized tensor compilers.

Chapter 3

Overview

In this chapter, we will explain the optimization that we propose in this dissertation, and we will introduce three of its variants, which will be further detailed in Section 5. Our exposition will use the naïve matrix multiplication algorithm seen in Figure 3.1. All the examples in this chapter are written in C, for the sake of readability; however, our optimization is meant to be implemented in the back-end of a compiler. Indeed, the implementation that we shall evaluate in Section 6 was implemented in the LLVM code generator, and requires no intervention from users –it is fully automatic.

```
1 void mul_ORG(float *restrict a, float *restrict b, float *restrict c, int n)
2 {
3   for (int i=0; i<n; i++)
4     for (int j=0; j<n; j++) {
5       c[i*n + j] = 0;
6       for (int k=0; k<n; k++)
7         c[i*n + j] += a[i*n + k] * b[k*n + j];
8     }
9 }
```

Figure 3.1: Matrix multiplication taken from [3] [page 332]. We have added the `restrict` keyword to run the experiments in this section.

Elimination of Silent Stores. The store operation at line 7 of Figure 3.1 is *silent* whenever the product `a[i*n+k] * b[k*n+j]` happens to be zero. "Silent Store" is a term coined by [19] to denote a store operation that writes in memory a value that was already there. As recently demonstrated by [26], expressions involving the value zero are a common source of silent stores. In the context of this dissertation, silentness happens because zero is the identity element of the addition operation. Thus, we can avoid the store operation by checking if `a[i*n+k] * b[k*n+j]` is zero. Figure 3.2 shows code that exercises such possibility.

At the first optimization level of `clang`, e.g., `-O0`, the code in Figure 3.2 saves one addition, plus one load and one store of `c[i*n+j]`. At `clang -O3`, it saves only one addition, due to the scalarization of `c[i*n+j]`. Non-surprisingly, the new version of matrix multiplication, i.e., function `mul_ESS`, has worse runtime than function `mul_ORG` at that optimization level. Figure 3.3 shows this comparison, considering input matrices

```
1  void mul_ESS(float *restrict a, float *restrict b, float *restrict c, int n)
2  {
3  for (int i=0; i<n; i++)
4    for (int j=0; j<n; j++) {
5      c[i*n + j] = 0;
6      for (int k=0; k<n; k++) {
7        float aux = a[i*n + k] * b[k*n + j];
8        if (aux)
9          c[i*n + j] += aux;
10     }
11 }
```

Figure 3.2: Naïve mat-mul after Elimination of Silent Stores.

with increasing probability of having cells with the value zero[1]. Function `mul_ESS` has two disadvantages, when compared with `mul_ORG`. First, the conditional at line eight downgrades the performance of the branch predictor, as the density of zeros in the input matrices increases. Second, `clang` does not vectorize the innermost loop of `mul_ESS`. In contrast, it unrolls the innermost loop of `mul_ORG` 40x, and parallelizes it using 8-word vectors. These shortcomings lead to the gap marked as region “1” in Figure 3.3.

[1]Data produced with LLVM 10.0.0, in an Intel Core i5 at 1.4GHz, running OSX 10.14.4, and 1000×1000 matrices.

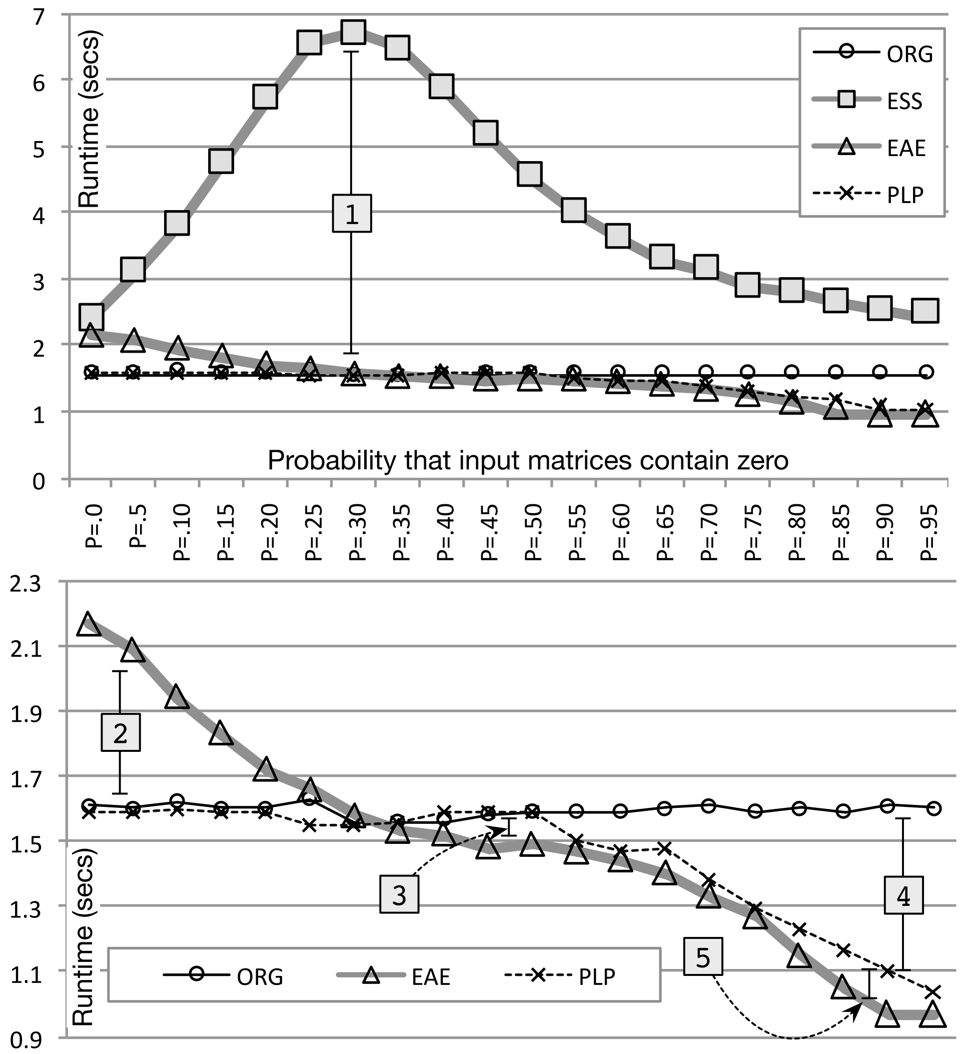


Figure 3.3: Runtime of different implementations of matrix multiplication ($C = A \times B$), given different probabilities that cells from matrices $A[10^3 \times 10^3]$ and $B[10^3 \times 10^3]$ contain zero. Programs were compiled with `clang -O3`. The bottom figure shows the same information as the upper figure, albeit with a reduced scaled along the Y-axis. The runtime of the original implementation of matrix multiplication has been removed from the bottom figure, to improve its readability. We use the following keys: **ORG**: the original program, without any form of semiring optimization, compiled with `clang -O3 -ffast-math`. **EAE**: program optimized with elision of absorbing elements (for details, see Section 5.2). **PLP**: program optimized with a pre-loop profiling that guards the elision of absorbing elements against unprofitable inputs (for details, see Section 5.4).

Elision of absorbing elements. The guard of Figure 3.2 encompasses too narrow a region to be of much benefit. To widen it, we notice that either $\texttt{a}[\texttt{i} * \texttt{n} + \texttt{k}] = 0$ or $\texttt{b}[\texttt{k} * \texttt{n} + \texttt{j}] = 0$ is a sufficient condition for a silent store, as zero is the absorbing element of multiplication. Figure 3.4 uses this observation to optimize the code originally seen in Figure 3.1. The chart in Figure 3.3 reveals that this optimization starts paying off when about 35% of the elements of matrix A are the value zero. Its benefit increases noticeably with the density of zeros. Once over 90% of the elements of A are zeros, we observe a performance boost over the original matrix multiplication (compiled with `clang -O3`) of almost 70% (Gap "4" in lower chart of Figure 3.3). On the other hand, at lower densities we observe important slowdowns, which gap "2" highlights. This slowdown is due, again, to the lack of vectorization, and to poor branch prediction –both negative consequences of the conditional at line 8 of Figure 3.4.

```
 1 void mul_EAE(float *restrict a, float *restrict b,
 2              float *restrict c, int n) {
 3   for (int i=0; i<n; i++)
 4     for (int j=0; j<n; j++) {
 5       c[i*n + j] = 0;
 6       for (int k=0; k<n; k++) {
 7         float t0 = a[i*n + k];
 8         if (t0 != 0.0) {
 9           c[i*n + j] += t0 * b[k*n + j];
10         }
11       }
12     }
13 }
```

Figure 3.4: Elision of Absorbing Elements applied onto the implementation of naïve matrix multiplication. In this example, we check if `a` is non-zero; however, it would also be possible to check if `b` is non-zero, or if their product is non-zero. More details are given in Section 4.3.

Pre-Loop Profiling. Semiring optimization hinders vectorization because its implementation requires inserting conditional tests into straight line code that, if left untouched, would be easy to vectorize. Nevertheless, it is still possible to benefit from vectorization and from semiring optimization. Key to this possibility is *profiling*. Figure 3.5 shows a possible way to apply profiling in this scenario. Because the profiler runs online, immediately before the program flows into the loop that contains the semiring pattern, we call this technique *Pre-Loop Profiling*. We adopt this name to contrast this technique with *Intra-Loop Profiling*, a similar –albeit simpler– methodology that applies profiling within the loop of interest. Intra-loop profiling shall be discussed in Section 5.3.

The function `sampling` invoked at line 3 of Figure 3.5 reads a few positions of an array to count occurrences of a value. In Figure 3.5, the array is matrix A, and the value of interest is zero. If the ratio of zeros exceeds a threshold –in this case, 0.5, then we suppress computation dependent on absorbing elements; otherwise, we run the code without any semiring optimization. Figure 3.3 shows that this approach recovers the performance of the original code when the density of zeros is low. Additionally, it matches the performance of elision of absorbing values when this density is high. The choice of an

```
void mul_PLP(float *restrict a, float *restrict b, float *restrict c, int n)
{
  if (sampling(a, n, 0.0F) > 0.50)
    mul_EAE(a, b, c, n);
  else
    mul_ORG(a, b, c, n);
}
```

Figure 3.5: Pre-Loop Profiling applied onto the implementation of naïve matrix multiplication.

adequate threshold is important. In this example, 0.35 would be a better threshold than 0.5. The gap labeled “3” highlights a region where semiring optimization is profitable, but the high threshold prevents it from happening. Profiling imposes a small overhead onto the program, which the gap labelled “5” outlines. In this example, we sample 1,000 cells of matrix A. Nevertheless, Chapter 6 will show situations in which sampling has a positive effect, due to data prefetching. In Section 5.4 we shall explain how we generate code to carry out sampling for any loop containing the semiring pattern.

Chapter 4

Generalizing Semiring Optimizations

This chapter has two goals. First, in Chapter 4.1 we present a list of semiring patterns. Second, in Chapter 4.3, we introduce an algorithm to identify load and store instructions that can be eliminated, given the occurrence of identities and absorbing elements in the semiring pattern.

4.1 A Family of Semiring Expressions

Figure 4.1 shows examples of operators and values that enable the proposed optimization. Some of these operators, when combined with the proper type, form true algebraic rings. Examples include bitwise operations such as ($\texttt{int}$, $\texttt{OR}_B$, $\texttt{AND}_B$); and logical operations such as ($\texttt{bool}$, $\texttt{OR}_L$, $\texttt{AND}_L$). Other combinations do not yield even semirings, such as ($\texttt{float}$, $\texttt{ADD}$, $\texttt{MUL}$), as floating-point arithmetic lacks associativity [15, Sec.4.2.2]. Nevertheless, these patterns are optimizable, because they present identity and absorbing values: the pattern $\texttt{OP}_1(a, \texttt{OP}_2(b, c))$ is optimizable whenever the absorbing value of $\texttt{OP}_2$ is the identity of $\texttt{OP}_1$. $\texttt{SHR}_L$ stand for logical shift right, and $\texttt{SHR}_A$ denotes arithmetic shift right; $\texttt{SHL}_L$ and $\texttt{SHL}_A$ denote the logical and arithmetic shift left equivalents. We let `sz` be the size of the type, in bits.

Example 4.1.1. The tuples $(\texttt{ADD}, \texttt{MUL}, \texttt{int})$, $(\texttt{ADD}, \texttt{MUL}, \texttt{float})$, $(\texttt{AND}_B, \texttt{OR}_B, \texttt{int})$, $(\texttt{OR}_B, \texttt{AND}_B, \texttt{int})$, $(\texttt{OR}_L, \texttt{AND}_L, \texttt{bool})$, and $(\texttt{ADD}, \texttt{SHL}_A, \texttt{int8})$, are examples of optimizable patterns.

Operations `DIV` and `MOD` in Figure 4.1 do not form true rings. However, we include them, because they contain identity (Id) and absorbing (Ab) elements, albeit position dependently. Thus, `0` is an absorbing element in `DIV(0, n)`, as long as $\texttt{n} \neq \texttt{0}$. Similarly, `1` is an identity in `DIV(n, 1)`, regardless of the value of `n`. We can fit these two operations into the same optimization algorithm that Section 4.3 introduces.

	Id	Ab
$\texttt{OR}_B$	0	~0
$\texttt{AND}_B$	~0	0
$\texttt{OR}_L$	F	T
$\texttt{AND}_L$	T	F
`MUL`	1	0

	Id	Ab
`MAX`	$-\infty$	$+\infty$
`MIN`	$+\infty$	$-\infty$
$\texttt{ADD}_s$	0	$+\infty$
$\texttt{SHR}_L$	0	sz
$\texttt{SHL}_A$	0	sz
$\texttt{SHL}_L$	0	sz

	Id
`ADD`	0
`XOR`	0
$\texttt{SHR}_A$	0

	$\text{Ab}_{r\text{ only}}$
`MOD`(l, r)	1

	$\text{Id}_{r\text{ only}}$
`DIV`(l, r)	1
`SUB`(l, r)	0

$r \neq 0$	$\text{Ab}_{l\text{ only}}$
`MOD`(l, r)	0
`DIV`(l, r)	0

Figure 4.1: Optimizable patterns that this work considers.

Example 4.1.2. The following identities are examples that follow from applying operations seen in Figure 4.1: `a == ADD(a, MOD(b, 1))`, or `a == OR(a, DIV(0, b))`.

4.2 Safety of Semiring Optimizations in the Floating Point Domain

Semiring optimizations must be implemented with care, when applied onto floating-point types. To be safe, semiring optimizations must be restricted to finite arithmetics. Finite arithmetics is assumed in mainstream compilers such as `gcc` and `clang` if the flag `-ffinite-math-only`[1] is enabled. The implementation in this paper guards the optimization with `-ffast-math`, which enables `-ffinite-math-only`. There are four issues that must be considered to understand the limits of this optimization when applied onto the floating-point domain:

- Zero is not the absorbing element in IEEE 754 arithmetics, due to two special values, `NaN` and `INFINITY` [10]. The former propagates through almost every operation involving floating-point numbers. The latter also propagates throughout these operations, except when they involve `NaN`. So, if `c` is `NaN` or `INFINITY` and `b` is zero, then the multiplications in Figure 1.1-ii and Figure 1.1-iii will not be zero.

- Again, considering Figure 1.1, if `a` is -0, `b` is $+0$, and `c` is $+0$, then $\texttt{a} + \texttt{b} \times \texttt{c}$ will return $+0$. However, the optimization will keep `a` as -0. Therefore, semiring optimization on the floating-point domain requires the flag `-fno-signed-zeros`. This flag is enabled by default with `-ffast-math`.

[1]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- the comparison at Line 8 in Figure 3.4 will be skipped when `t0` is either minus or plus zero. If `t0` is a very small nonzero value, e.g. a subnormal number, then the computation will still run although it might not change the value of `c[i*n+j]`. In this case the optimization is innocuous; albeit safe.

- Associativity is missing in the IEEE 754 standard. However, in this case semiring optimizations are safe, because they do not change the order in which operations happen.

4.3 Identification of Optimization Points

The function `optimize`, in Figure 4.2, identifies optimization points in a program for some operations seen in Figure 4.1. This function is invoked onto expressions that fit into the pattern $a = a \oplus b$. Figure 4.2 represents such patterns as `st` a $(a \oplus b)$. In this case, $\oplus$ is any operation that has an identity element and `st` is a store. Given this pattern, `optimize` invokes function `fix` over b. This procedure will traverse –backwardly– the data-dependence graph of b, looking for operations whose absorbing element is the identity of $\oplus$.

The output of function `fix` is a set of relations in the form {`s == v`}, where `s` is a variable name, and `v` is the absorbing element of some operation that uses `s`. This relation means that whenever `s` has the value `v`, then the entire expression that uses `s` can be replaced by a known constant. These two procedures, `optimize` and `fix`, have been designed to operate on programs in the Static Single Assignment format (SSA) [4]. Therefore, every relation `s == v` is unambiguous, because every program variable `s` has just one definition point. The next example illustrates how these two functions work.

Example 4.3.1. Figure 4.3 shows the invocation of `optimize` on the program `St(a, ADD(VAR a, MUL(AND`$_B$`(MOD(VAR b, VAR c), VAR c), VAR d)))`. Function `optimize` produces four equalities for this code snippet, namely: { `(b == 0)`, `(c == 1)`, `(c == 0)`, `(d == 0)` }. The point where each equality is created is marked in gray in Figure 4.3. If any of these equalities hold at run time, then the store to variable `a` will be silent.

To keep our report concise and reproducible, in this paper, we focus on the elimination of silent stores only. Nevertheless, identity patterns enable many other optimizations that we will not explore. For instance, instead of removing silent stores, we can simply remove operations like $a \oplus b$, whenever either a or b are the identity element of $\oplus$.

```
fun pin (VAR s, v)        ⇒ (s == v)
  | pin (OR (a, b), ~0)   ⇒ pin(a, ~0) ∪ pin(b, ~0)
  | pin (MUL (a, b), 0)   ⇒ pin(a, 0) ∪ pin(b, 0)
  | pin (AND (a, b), 0)   ⇒ pin(a, 0) ∪ pin(b, 0)
  | pin (LOR (a, b), T)   ⇒ pin(a, T) ∪ pin(b, T)
  | pin (LAND (a, b), F) ⇒ pin(a, F) ∪ pin(b, F)
  | pin (DIV (a, b), 0)   ⇒ pin(a, 0)
  | pin (MOD (a, b), 0)   ⇒ pin(b, 1) ∪ pin(a, 0)
  | pin _ ⇒ ∅

fun optimize (st a OR(a, b))   ⇒ pin(b, 0)
  | optimize (st a AND(a, b))  ⇒ pin(b, ~0)
  | optimize (st a MUL(a, b))  ⇒ pin(b, 1)
  | optimize (st a LOR(a, b))  ⇒ pin(b, F)
  | optimize (st a LAND(a, b)) ⇒ pin(b, T)
  | optimize (st a ADD(a, b))  ⇒ pin(b, 0)
  | optimize (st a SUB(a, b))  ⇒ pin(b, 0)
  | optimize (st a XOR(a, b))  ⇒ pin(b, 0)
  | optimize (st a SHL(a, b))  ⇒ pin(b, 0)
  | optimize (st a SHR(a, b))  ⇒ pin(b, 0)
  | optimize (st a DIV(a, b))  ⇒ pin(b, 1)
  | optimize (st a MOD(a, b))  ⇒ pin(b, 1)
```

Figure 4.2: Identification of optimization points.

Correctness. In this section, we lay out the key invariants of functions `optimize` and `fix`. These invariants are stated on top of the semantics of the language of logical and arithmetic expressions that `fix` traverses. In this context, we define the *store environment* $\sigma : \texttt{VAR} \rightarrow \texttt{VALUE}$ as a function that maps variable names to values. Next, we define an evaluation function $\texttt{eval} : \texttt{st} \times \sigma \rightarrow \sigma$, which receives a store instruction, like those seen in Figure 4.2, plus an environment σ, and produces a new environment σ'. The implementation of `eval` is standard; hence, instead of defining it formally, we will only illustrate it with Example 4.3.2. Function `eval` lets us state the core invariant of function `fix`, which Theorem 4.3.3 proves.

Example 4.3.2. If $\sigma = \{a \mapsto 7, b \mapsto 4\}$, then we have that $\texttt{eval}(\texttt{st}\ a\ \texttt{ADD}(b, a), \sigma) = \sigma[a \mapsto 11]$, and $\texttt{eval}(\texttt{st}\ a\ \texttt{ADD}\ (\texttt{DIV}(a, b), a), \sigma) = \sigma[a \mapsto 8]$. We use [] to denote function updating, i.e.: $\sigma[s \mapsto v] = \lambda x.(x = s)?v : \sigma(x)$.

Theorem 4.3.3. If $\texttt{fix}(b, z) = C$, then, for any $(s{==}v) \in C$ and any store environment σ, $\texttt{eval}(b, \sigma[s \mapsto v]) = z$.

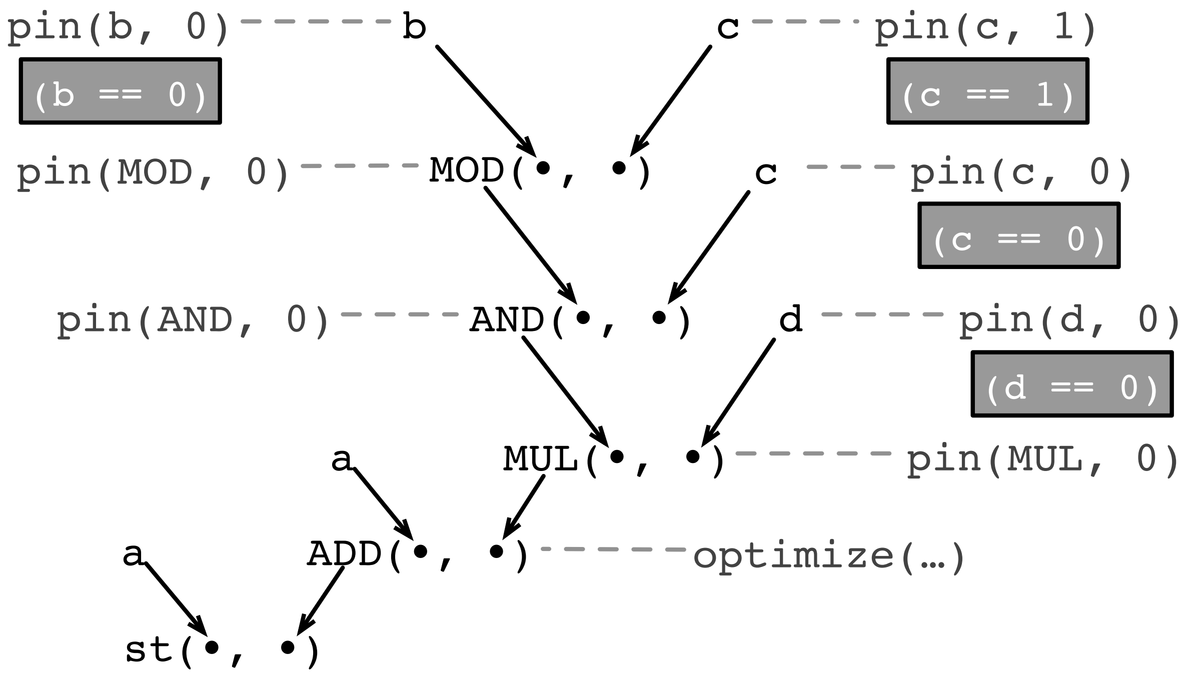


Figure 4.3: Example of relations produced by `optimize`.

The proof is by induction on the derivation tree of `fix`. We shall consider a few cases:

- if $\texttt{fix}(\texttt{VAR}s, v)$, then $C = \{s{==}v\}$. We have that $\texttt{eval}(\texttt{VAR}s, \sigma[s \mapsto v]) = v$;
- if $\texttt{fix}(\texttt{OR}(a,b), \tilde{}0)$, then $C = C_1 \cup C_2$, where $C_1 = \texttt{fix}(a, \tilde{}0)$ and $C_2 = \texttt{fix}(b, \tilde{}0)$. If $s{==}v \in C_1$ (the case for C_2 is analogous), then, by induction, $\texttt{eval}(b, \sigma[s \mapsto v]) = \tilde{}0$. Because $\tilde{}0$ is the destructor of `OR`, we have that $\texttt{eval}(\texttt{OR}(a,b), \sigma[b \mapsto \tilde{}0]) = \tilde{}0$.

Corollary 4.3.4. Let $\texttt{optimize}(\texttt{st}\ a(\oplus(a,b))) = C$.
If $(s{==}v) \in C$, then $\texttt{eval}(\oplus(a,b), \sigma[s \mapsto v] = a$, whenever $\sigma(s) = v$

We have that $\texttt{optimize}(\texttt{st}\ a(\oplus(a,b))) = \texttt{fix}(b,z) = C$, where z is the absorbing element of $\oplus$. From Theorem 4.3.3, if $b{==}z \in C$, then $\texttt{eval}(b, \sigma[s \mapsto z]) = z$. Thus, $\texttt{eval}(\oplus(a,b), \sigma[s \mapsto v]) = \texttt{eval}(\oplus(a,b), \sigma[s \mapsto v, b \mapsto z]) = a$.

Chapter 5

Four Variations of Semiring Optimization

This chapter presents four ways to eliminate silent stores related to semiring patterns, from the simplest (Section 5.1) towards the most complex (Section 5.4).

5.1 Version 1: Elimination of Silent Stores (ESS)

The simplest form of semiring optimization guards a store with a conditional test. We have implemented this transformation as the exhaustive application of the rewriting rule earlier seen in Figure 1.1-i. Figure 5.1 generalizes that example. Notice that the semiring pattern makes two trivial optimizations possible. In case variable `t0` is *alive* past the store at line 4 of Figure 5.1-i, then we must settle for the more conservative transformation seen in Figure 5.1-ii. Otherwise, we can also avoid the load of `t0`, using the transformation seen in Figure 5.1-iii. The transformations seen in Figure 5.1 preserve program semantics. Although trivial, we state this fact formally in Theorem 5.1.1, for the sake of completeness.

Theorem 5.1.1. If `t0` is only used at line 3 of Figure 5.1-i, then Fig. 5.1-i and Fig. 5.1-ii are equivalent. Otherwise, Fig. 5.1-i and Fig. 5.1-iii are equivalent.

Proof. We show equivalence from Fig 5.1-i and Fig 5.1-iii. The second part of the theorem follows from similar reasoning. If $\texttt{t1} \neq z$, then both programs execute the same set of assignments. Otherwise, we have that $\texttt{t2} = \texttt{t0} \oplus z = \texttt{t0} = \texttt{a}$, and the store is silent. $\square$

(i)
```
1 t0 = ld a
2 t1 = ld e
3 t2 = t0⊕t1
4 st a t2
```

(ii)
```
1 t1 = ld e
2 t0 = ld a
3 if (t1≠z) {
4   t2 = t0⊕t1
5   st a t2
6 }
7 is_alive(t0)
```

(iii)
```
1 t1 = ld e
2 if (t1≠z) {
3   t0 = ld a
4   t2 = t0⊕t1
5   st a t2
6 }
7 is_dead(t0)
```

Figure 5.1: Implementation of silent store elimination when z is the identity of $\oplus$. (i) Original program. (ii) Program optimized when `t0` is used in instructions other than the store. (iii) Program optimized when `t0` is used only once. Annotations at Line 7 are not part of the language used to write the computations—rather, they are pseudo-code indicating that variables are either alive or dead past the point where they appear.

5.2 Version 2: Elision of Absorbing Elements (EAE)

The conditional elision of expressions that depend on absorbing elements is based on the invariant stated by Theorem 4.3.3. Thus, if $(s{==}v) \in \texttt{pin}(b, z)$, the evaluation of b yields z whenever the symbol s holds the value v. From Corollary 4.3.4, $(s{==}v)$ is enough to yield the store `st` $a \;\; \oplus(a, b)$ silent. To capitalize on these observations, we proceed in the three steps below, where the pattern `st` $a \;\; \oplus(a, b)$ is called ι_s:

1. We decorate the load of s with a guard g that checks if s receives the value v.
2. We move ι_s to the false branch of g. Thus, ι_s will happen only when g is false.
3. We move to inside the false branch of g any other instruction ι that is only used to compute ι_s, or some other instruction ι' already inside the false branch.

Example 5.2.1. Fig. 5.2-i shows the tree in Fig. 4.3 written in three-address format. Fig. 5.2-ii shows the guard used to check if the value loaded from `b` is an absorbing element. Fig. 5.2-iii displays the optimized program, provided that none of the temporary variables is used past the last store instruction.

In Example 5.2.1, we emphasize that instructions can only be moved into the guarded region if they are not used in expressions other than the silent store. In other words, the region that guards the execution of a potentially silent store ι_s will contain every instruction ι that is part of the backward slice of ι_s –except if ι is used to compute values that do not belong into this slice. Example 5.2.2 clarifies these observations.

Example 5.2.2. Figure 5.3 shows three optimized versions of the code snippet from Figure 5.2. Each version differs on the instructions that can be placed inside the region

(i)
```
t0 = ld b
t1 = ld c
t2 = t0 % t1
t3 = ld c
t4 = t2 & t3
t5 = ld d
t6 = t4 * t5
t7 = ld a
t8 = t6 + t7
st a t8
```

(ii)
```
t0 = ld b
t1 = ld c
t2 = t0 % t1
t3 = ld c
t4 = t2 & t3
t5 = ld d
t6 = t4 * t5
t7 = ld a
t8 = t6 + t7
if (t0 ≠ 0) {
  st a t8
}
```

(iii)
```
t0 = ld b
if (t0 ≠ 0) {
  t1 = ld c
  t2 = t0 % t1
  t3 = ld c
  t4 = t2 & t3
  t5 = ld d
  t6 = t4 * t5
  t7 = ld a
  t8 = t6 + t7
  st a t8
}
```

Figure 5.2: Checking if `b` is an absorbing element.

guarded by the check on `t0`. Notice how, in Figure 5.3-iii, the liveness of `t6` past the store prevents several other instructions from being moved within the guarded region.

(i)
```
t0 = ld b
t1 = ld c
if (t0 ≠ 0) {
  t2 = t0 % t1
  t3 = ld c
  t4 = t2 & t3
  t5 = ld d
  t6 = t4 * t5
  t7 = ld a
  t8 = t6 + t7
  st a t8
}
is_alive (t1)
```

(ii)
```
t0 = ld b
t5 = ld d
if (t0 ≠ 0) {
  t1 = ld c
  t2 = t0 % t1
  t3 = ld c
  t4 = t2 & t3
  t6 = t4 * t5
  t7 = ld a
  t8 = t6 + t7
  st a t8
}
is_alive (t5)
```

(iii)
```
t0 = ld b
t1 = ld c
t2 = t0 % t1
t3 = ld c
t4 = t2 & t3
t5 = ld d
t6 = t4 * t5
if (t0 ≠ 0) {
  t7 = ld a
  t8 = t6 + t7
  st a t8
}
is_alive (t6)
```

Figure 5.3: The impact of liveness on the elision of instructions that depend on absorbing elements.

5.3 Version 3: Intra-Loop Profiling (ALP)

Semiring Optimization is speculative, because its performance depends on program inputs. Thus, the unrestricted application of semiring optimization might lead to runtime regression. As explained in Section 3, regression is due to the fact that the guards that implement the optimization hinder vectorization and complicate branch prediction. It is possible to circumvent these shortcomings via profiling techniques.

In this section, we introduce a form of profiling of easy implementation: its deployment does not require any static program analysis. Profiling happens *in-vivo*, that is to say, during the execution of the profiled program already in production mode. The version of profiling that we describe in this section inserts code within the loops that contain semiring patterns; hence, we call it Intra-Loop Profiling (ALP). It improves branch prediction, but still hinders vectorization. Later, in Section 5.4 we will show how to hoist the profiling code outside the loop; thus, enabling also vectorization.

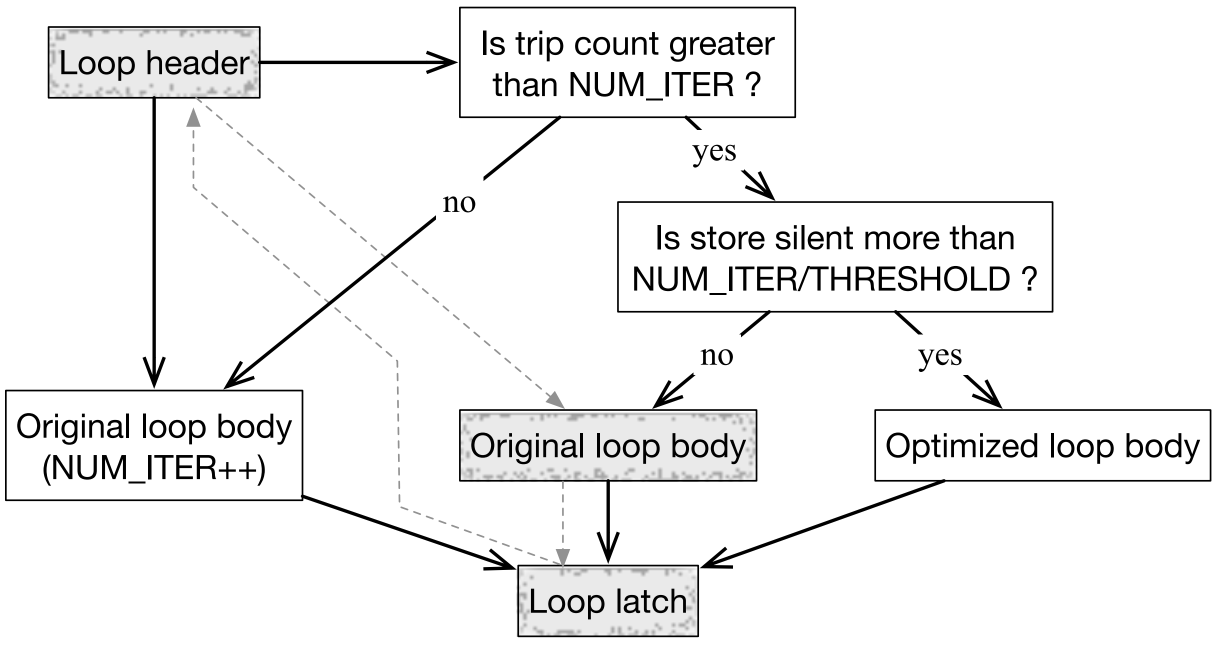

Loop header
Is trip count greater than NUM_ITER ?
no
yes
Is store silent more than NUM_ITER/THRESHOLD ?
no
yes
Original loop body (NUM_ITER++)
Original loop body
Optimized loop body
Loop latch

Figure 5.4: Naïve matrix multiplication augmented with code to implement Intra-Loop Profiling. The `PROFILE` constant that initializes `target` sends the execution flow to the `default` clause of the `switch`.

Figure 5.4 shows how Intra-Loop Profiling augments a loop with code to carry out profiling. As we see in the figure, the body of the loop is cloned twice. Thus, in addition to the original loop, ALP creates a body with an iteration counter, that performs profiling, and another body optimized with the elision of absorbing elements (see Section 5.2). Profiling is guided by two constants. The first, `NUM_ITER`, determines the number of times that stores are inspected, to check if they are silent or not. The second, `THRESHOLD`, determines when semiring optimization should take place.

Example 5.3.1. Figure 5.5 shows the implementation of ALP on the program used as an example in Section 3. For the sake of clarity, we show code written in C; however, just like all the other optimizations described in this dissertation, ALP is implemented at the binary level.

```
 1 void mul_ALP(float *restrict a, float *restrict b,
 2              float *restrict c, int n){
 3 for (int i=0; i<n; i++){
 4   for (int j=0; j<n; j++){
 5     for (int k=0; k<n; k++){
 6       switch (target){
 7         case Original:
 8           c[i*n + j] += a[i*n + k] * b[k*n + j];
 9           break;
10         case Optimise:
11           float t0 = a[i*n + k];
12           if (t0 != 0.0) c[i*n + j] += t0 * b[k*n + j];
13           break;
14         default:
15           float t = a[i*n + k] * b[k*n + j];
16           silent += (t == 0.0) ? 1 : 0;
17           iter += 1;
18           if (iter == PROFILING_THRESHOLD)
19             target = silent > PROFILING_THRESHOLD/THRESHOLD
20                      ? Optimize : Original;
21           c[i*n + j] += t;
22       }
23     }
24   }
25 }
```

Figure 5.5: Naïve matrix multiplication augmented with code to implement Intra-Loop Profiling.

5.4 Version 4: Pre-Loop Profiling (PLP)

The online profiling technique discussed in the previous section improves the hit rate of the branch predictor, as it tends to reduce the number of branches dynamically executed. However, it is still hard to vectorize the optimized code. As an example, neither LLVM 8.0 nor gcc 6.0 can vectorize the assignment at line 8 of Figure 5.5. The culprit is the switch statement at line 6, which leads to three very different variations of the original loop body. To enable vectorization, we must hoist the profiling code outside the loop. This new version of *in-vivo* profiling shall be called Inter-Loop Profiling.

Different hoisting strategies. While designing PLP, we considered three different approaches. Although we have considered every one of them, only the last technique became fully functional. Nevertheless, we discuss them all, to avoid those who intend to expand our ideas start implementations that are difficult to conclude successfully:

- **Loop peeling:** we can split the target loop into two iterators: the first iterates `NUM_ITER` times (the number of samplings performed by the profiler); and the second completes the rest of the loop. In this form of loop peeling [23], both loops do the useful work present in the original program, but only the first samples memory. This strategy was our first approach to hoist the profiler outside the loop. However, although LLVM provides support to peel the innermost loop, no such support exists for the whole loop nest, and we found it difficult to craft a correct implementation.

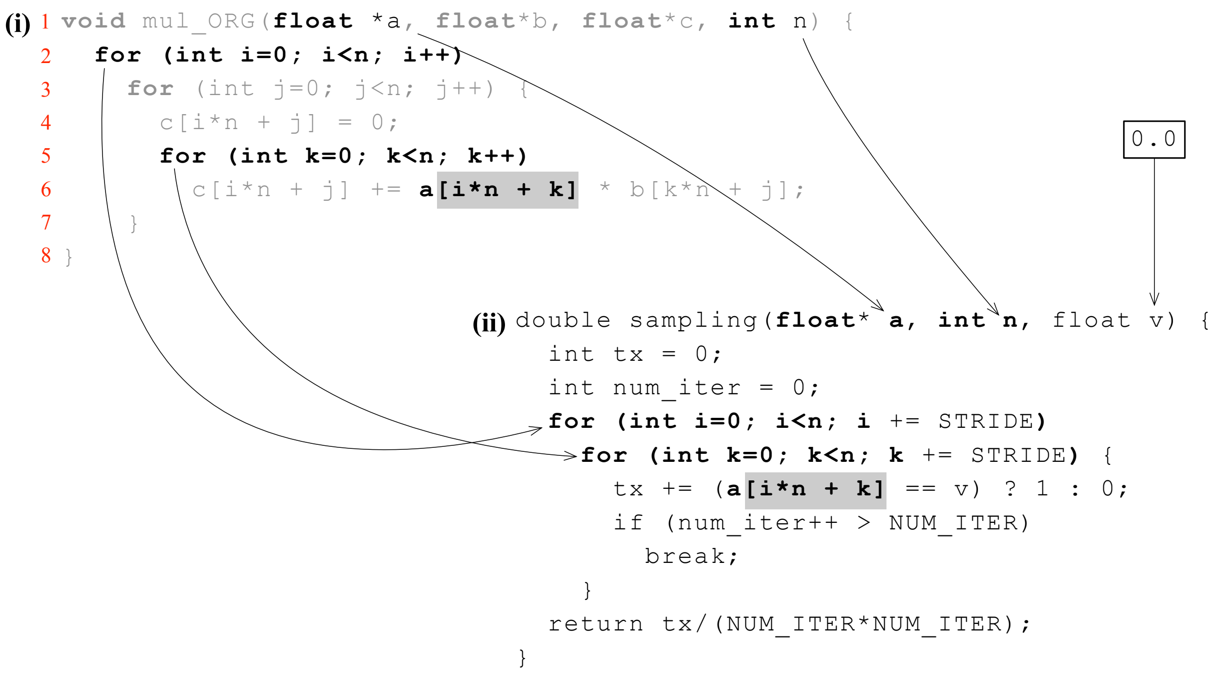
- **Symbolic range analysis:** we can use symbolic range analysis to obtain bounds to the arrays present in LLVM's intermediate representation. To this end, we could reuse DAWNCC's parametric range analysis [24], which is publicly available for LLVM. Unfortunately, DAWNCC's implementation targets a lower version of LLVM than the one we use in this work. Thus, we were not able to build a prototype on top of DAWNCC.
- **Program slice:** a *program slice* with respect to a statement ι in a program P is the subset of P that contributes to the execution of ι [33]. To profile a memory location s, loaded by an instruction ι_{ld}, we extract the program slice of ι_{ld}. Because a backward slice considers data and control-dependences, it gives us every loop nest that contributes to compute the address used in ι_{ld}. In this work, we have implemented a slicing algorithm available for the LLVM compiler [28]. Hence, this was our approach of choice.

Example 5.4.1. Figure 5.6(i) shows the backward slice of the access `a[i*n + j]`, originally at line 7 of Figure 3.1. This memory access depends on the indexing variables, e.g., `i`, `n` and `k`. Control dependences add to the slice the outermost loop (which controls variable `i`), and the innermost (which controls variable `k`). Notice that the middle loop is left out of the slice, as variable `j` bears no influence on the memory access.

Sampling stride. In addition to enabling vectorization, hoisting the profiling code outside the loop of interest via backward slicing brings another advantage: we are free to choose different strides to sample memory locations. The *sampling stride* is the spatial distance between successive addresses inspected via profiling. The stride used in Section 5.3 always follows the pattern in which memory is accessed within the original loop. As an example, in Figure 5.5, sampling happens at line 15. Array `a`'s sampling stride is 1, and array `b`'s is `n`.

Sampling based on the loop trip count might lead to bad decisions. For instance, one of the benchmarks that we analyze in Section 6 is Cholesky's decomposition. This benchmark receives a diagonal matrix, in which the elements below the main diagonal are all zeros. However, sampling based on the trip count, even with a large number of profiling iterations, touches only a handful of memory positions under that diagonal. Consequently, profiling misses a substantial region that contains only zeros –multiplication's absorbing element.

Example 5.4.2. Figure 5.6(ii) shows the `sampling` function that we built after the slice in Figure 5.6(i). The induction variables within the two loops that constitute the slice is incremented by a parameterized interval `STRIDE`. The implementation evaluated in



```
void mul_ORG(float *a, float*b, float*c, int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
      c[i*n + j] = 0;
      for (int k=0; k<n; k++)
        c[i*n + j] += a[i*n + k] * b[k*n + j];
    }
}
```

```
double sampling(float* a, int n, float v) {
  int tx = 0;
  int num_iter = 0;
  for (int i=0; i<n; i += STRIDE)
    for (int k=0; k<n; k += STRIDE) {
      tx += (a[i*n + k] == v) ? 1 : 0;
      if (num_iter++ > NUM_ITER)
        break;
    }
  return tx/(NUM_ITER*NUM_ITER);
}
```

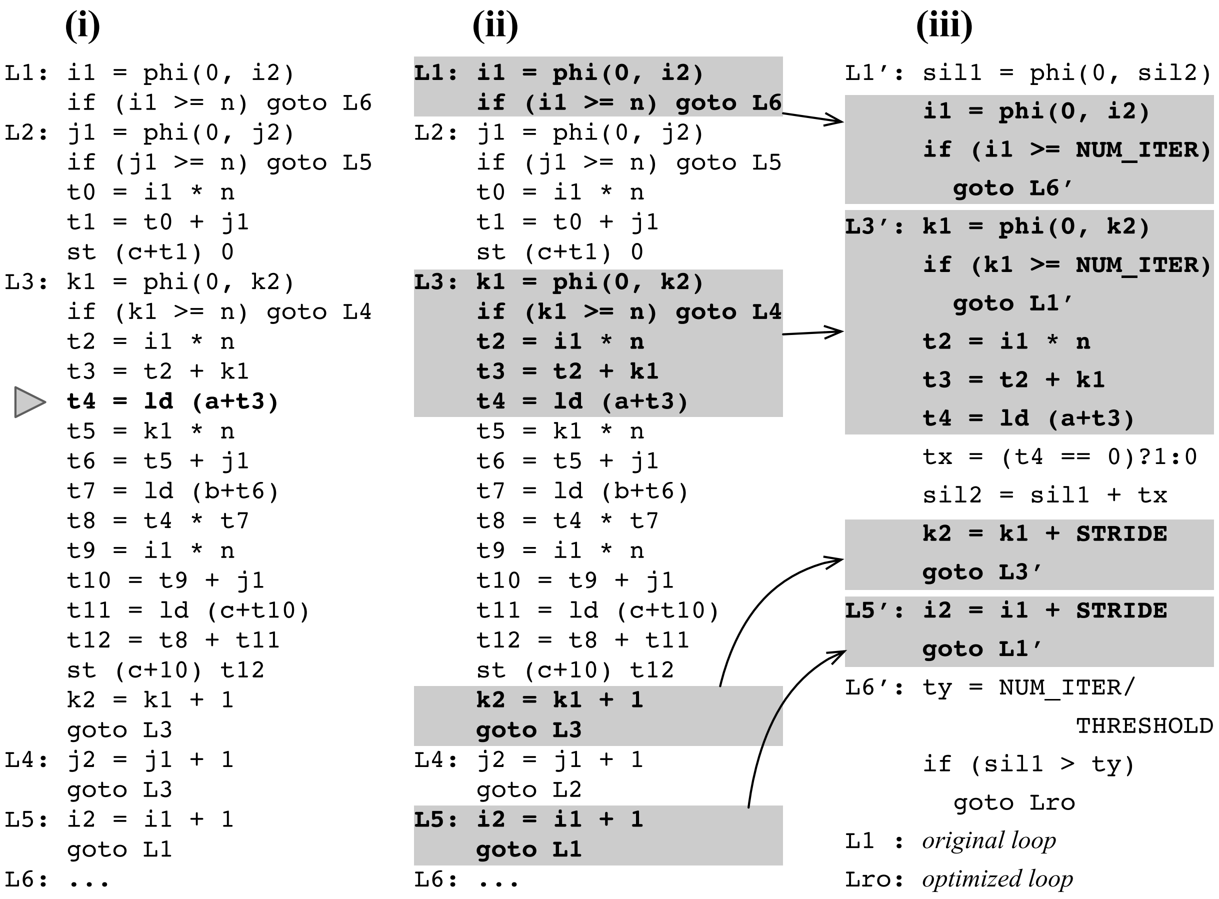
Figure 5.6: (i) Backward slice that determines the memory access `a[i*n+k]`. (ii) The `sampling` function that is derived from this slice. Figure 3.5 shows the final, optimized code, with a call to function `sampling`. Figure 5.7 will show how this code is implemented in practice in the low-level representation of a program.

Section 6 sets the value of this variable at compilation time. However, nothing hinders another implementation from having this value defined per program, for instance.

On the number of profiling iterations. There exists a tradeoff between the number of samples collected by the pre-loop profiler, and the accuracy of the information that it reports. The larger the number of samples, the better this accuracy. However, more samples contribute towards a heavier overhead that the profiler imposes onto the programs that it aims to optimize. Our implementation of of the pre-loop profiler limits the number of samples in 1,000 per loop. This value, like the sampling stride, is defined at compilation time, although users have the option to define it in a per-program basis. In Section 6 we shall consider the same limit of 1,000 samples for all the programs that we analyze.

Low-level implementation. Example shown thus far in this paper are written in C; however, the implementation of all the techniques that it introduces happens at the level of LLVM's intermediate representation. In the case of the pre-loop profiling, there is one further difference between our presentation and our implementation: the profiler, which is seen in Figures 3.5 and 5.6(ii) as a separate function, is, in practice, inlined in the optimized code. Example 5.4.3 provides a more faithful view of this implementation.

Example 5.4.3. Figure 5.7 shows the backward slice of the access `a[i*n + j]` at line 7 of Figure 3.1. The third part of Figure 5.7 shows the code of the sampling function built out of that slice, augmented with profiling. This sampling function is the same that is invoked at line 3 of Figure 3.5.



(i)

```
L1: i1 = phi(0, i2)
    if (i1 >= n) goto L6
L2: j1 = phi(0, j2)
    if (j1 >= n) goto L5
    t0 = i1 * n
    t1 = t0 + j1
    st (c+t1) 0
L3: k1 = phi(0, k2)
    if (k1 >= n) goto L4
    t2 = i1 * n
    t3 = t2 + k1
    t4 = ld (a+t3)
    t5 = k1 * n
    t6 = t5 + j1
    t7 = ld (b+t6)
    t8 = t4 * t7
    t9 = i1 * n
    t10 = t9 + j1
    t11 = ld (c+t10)
    t12 = t8 + t11
    st (c+10) t12
    k2 = k1 + 1
    goto L3
L4: j2 = j1 + 1
    goto L3
L5: i2 = i1 + 1
    goto L1
L6: ...
```

(ii)

```
L1: i1 = phi(0, i2)
    if (i1 >= n) goto L6
L2: j1 = phi(0, j2)
    if (j1 >= n) goto L5
    t0 = i1 * n
    t1 = t0 + j1
    st (c+t1) 0
L3: k1 = phi(0, k2)
    if (k1 >= n) goto L4
    t2 = i1 * n
    t3 = t2 + k1
    t4 = ld (a+t3)
    t5 = k1 * n
    t6 = t5 + j1
    t7 = ld (b+t6)
    t8 = t4 * t7
    t9 = i1 * n
    t10 = t9 + j1
    t11 = ld (c+t10)
    t12 = t8 + t11
    st (c+10) t12
    k2 = k1 + 1
    goto L3
L4: j2 = j1 + 1
    goto L2
L5: i2 = i1 + 1
    goto L1
L6: ...
```

(iii)

```
L1': sil1 = phi(0, sil2)
     i1 = phi(0, i2)
     if (i1 >= NUM_ITER)
       goto L6'
L3': k1 = phi(0, k2)
     if (k1 >= NUM_ITER)
       goto L1'
     t2 = i1 * n
     t3 = t2 + k1
     t4 = ld (a+t3)
     tx = (t4 == 0)?1:0
     sil2 = sil1 + tx
     k2 = k1 + STRIDE
     goto L3'
L5': i2 = i1 + STRIDE
     goto L1'
L6': ty = NUM_ITER/
                 THRESHOLD
     if (sil1 > ty)
       goto Lro
L1  : original loop
Lro: optimized loop
```

Figure 5.7: (i) Three-address code version of the naïve matrix multiplication algorithm seen in Fig-3.1. The grey triangle shows the instruction that we will slice out from the loop. (ii) The grey boxes mark the backward slice of the load of `a[i*n+k]`. (iii) The sampling function built out of the slice. Grey boxes show code present in the original loop.

Chapter 6

Evaluation

In this chapter, we shall evaluate Semiring Optimization aiming to answer the following research questions:

RQ1 How often does the semiring pattern appear in typical benchmarks?

RQ2 What is the runtime benefit of the different versions of Semiring Optimization?

RQ3 What is the overhead of the different versions of profiling-based Semiring Optimizations?

RQ4 What is the overhead of Semiring Optimization on Compilation Time?

RQ5 How does Semiring Optimization compare with a specialized tensor compiler?

RQ6 What is the effect of input variation on semiring-optimizations?

RQ7 What is the effect of semiring-optimizations on performance couters?

Runtime Setup. We have implemented Semiring Optimization onto LLVM 6.0.1. Results reported in this section were produced on an 8-core Intel(R) Core(TM) i7-3770 at 3.40GHz, with 16GB of RAM running Ubuntu 16.04.
Benchmarks. We have applied semiring optimization on the programs available in the LLVM test suite. In this chapter, we shall restrict our presentation to PolyBench [27], for concision. However, we shall present results from other benchmarks when we discuss **RQ1** and **RQ2**, e.g., Prevalence and Speedup. We do not run the **RQ4** experiments for all the program because its execution time would be prohibitive and we think the evidence gathered from this experiment is likely to be indicative of the overall behavior. PolyBench consists of 30 programs written in C. Out of this lot, 20 benchmarks contain the pattern $a = a \oplus b$. Hence, we restrict our evaluation to this subset of PolyBench. Table 6 shows the PolyBench benchmarks we use alongside a small description.
Measurement methodology. We report 5 runs for each version of each program. We adopt a significance level $\alpha = 0.05$. Thus, if the results reported by original and optimized programs cannot be distinguished with a confidence of more than 95% (via Student's Test),

Table 6.1: List of PolyBench benchmarks used

Benchmark	**LoC**	**# Semiring Patterns**	**Description**
2mm	252	3	2 Matrix Multiplication
3mm	267	3	3 Matrix Multiplication
bicg	227	2	BiCG Sub Kernel
cholesky	211	4	Cholesky Decomposition
correlation	248	7	Correlation Computation
covariance	218	4	Covariance Computation
doitgen	214	1	Multiresolution analysis kernel
fdtd-2d	256	3	2-D finite Different Time Domain Kernel
gemm	232	2	Matrix-multiply $C = \alpha \cdot A \cdot B + \beta \cdot C$
gemver	261	3	Vector Multiplication and Matrix Addition
gesummv	222	2	Scalar, Vector and Matrix Multiplication
gramschmidt	231	2	Gram-Schmidt decomposition
lu	210	4	LU decomposition
ludcmp	258	1	LU decomposition
mvt	222	2	Matrix Vector product and Transpose
symm	231	1	Symmetric matrix-multiply
syr2k	225	2	Symmetric rank-k operations
syrk	210	2	Symmetric rank-2k operations
trmm	210	2	Triangular matrix-multiply

then we consider them as originating from the same population. Our baseline is LLVM -O3. At this level, LLVM performs vectorization, unrolling and inlining, for instance.

A note on static and dynamic instances. A *static* instance of a store instruction is the syntactic occurrence of it. A *dynamic* instance, in turn, is its execution. Hence, a static instance can have many corresponding dynamic instances. We call instructions involved in a semiring pattern as *"marked"* instructions. Given these definitions, we shall use `Dyn` to refer to the number of dynamic instances of store instructions in a benchmark, `M-Dyn` for the number of dynamic stores marked and `MS-Dyn` for the number of times a store marked was silent. The proportion of marked instances is the ratio $\frac{\texttt{M-Dyn}}{\texttt{Dyn}}$. Similarly, the ratio $\frac{\texttt{MS-Dyn}}{\texttt{Dyn}}$ gives the percentage of silent instances. Finally, the ratio $\frac{\texttt{MS-Dyn}}{\texttt{M-Dyn}}$ measures the silentness level of marked instructions.

6.1 RQ1: Prevalence

We have measured the prevalence semiring patterns in 259 programs from 36 benchmark suites. Out of the 259 programs, 126 (49%) contain the semiring pattern. Figure 6.1 shows the frequency of semiring patterns among the 126 benchmarks. These benchmarks, when running with their standard inputs, gave us a total of 101,107 static instances of store operations, out of which 1,569 (1%) belong into a semiring pattern. Henceforth, we shall call these instructions *marked*. A large portion of programs (41%) contains one or two marked stores, but this number varies significantly across benchmarks. If we average the number of marked store instructions per program, then we obtain 12 with a standard deviation of 39. In other words, programs tend to have approximately 12 stores involved in a semiring pattern. This high standard deviation is due to the variability in the workload of the test suites we use. Some programs are designed for computer-intensive tasks, whereas others are not. In total, we observed 988,223,728,822 dynamic instances of store instructions (`Dyn`), out of which 276,072,024,712, i.e., 27%, were from marked stores (`M-Dyn`). Out of this lot, 29,765,251,126, i.e., 3%, were silent (`MS-Dyn`). Thus, 1% of all static instances contributed to 27% of all execution of store operations of which 11% were silent.

Figure 6.2 shows the prevalence of the semiring pattern on the PolyBench test suite. The transparent bar is the ratio $\frac{\texttt{M-Dyn}}{\texttt{Dyn}}$ while the gray bar is $\frac{\texttt{MS-Dyn}}{\texttt{Dyn}}$. The numbers on top of each bar are the order of magnitude of `M-Dyn`. Numbers inside circles report the number of static instances. Most of the benchmarks have $\frac{\texttt{M-Dyn}}{\texttt{Dyn}} \approx 1.0$ and three benchmarks (`cholesky, lu, ludcmp`) have $\frac{\texttt{MS-Dyn}}{\texttt{Dyn}} \geq 0.5$. On `ludcmp`, one static store was responsible for almost all 10^9 dynamic instances. Approximately 60% of these instances were silent.

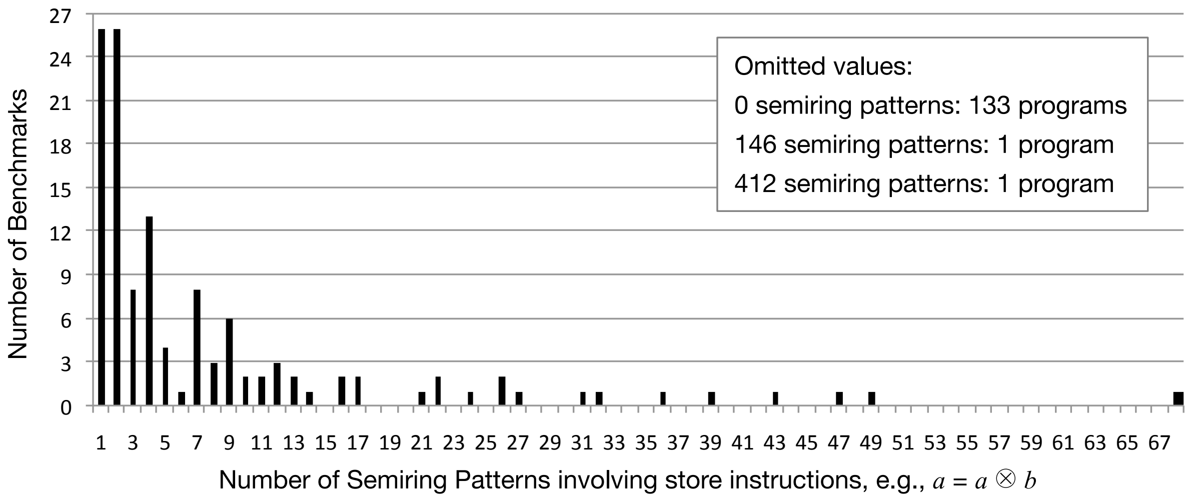


Figure 6.1: Histogram with the number of static store instances marked for the set of benchmarks used.

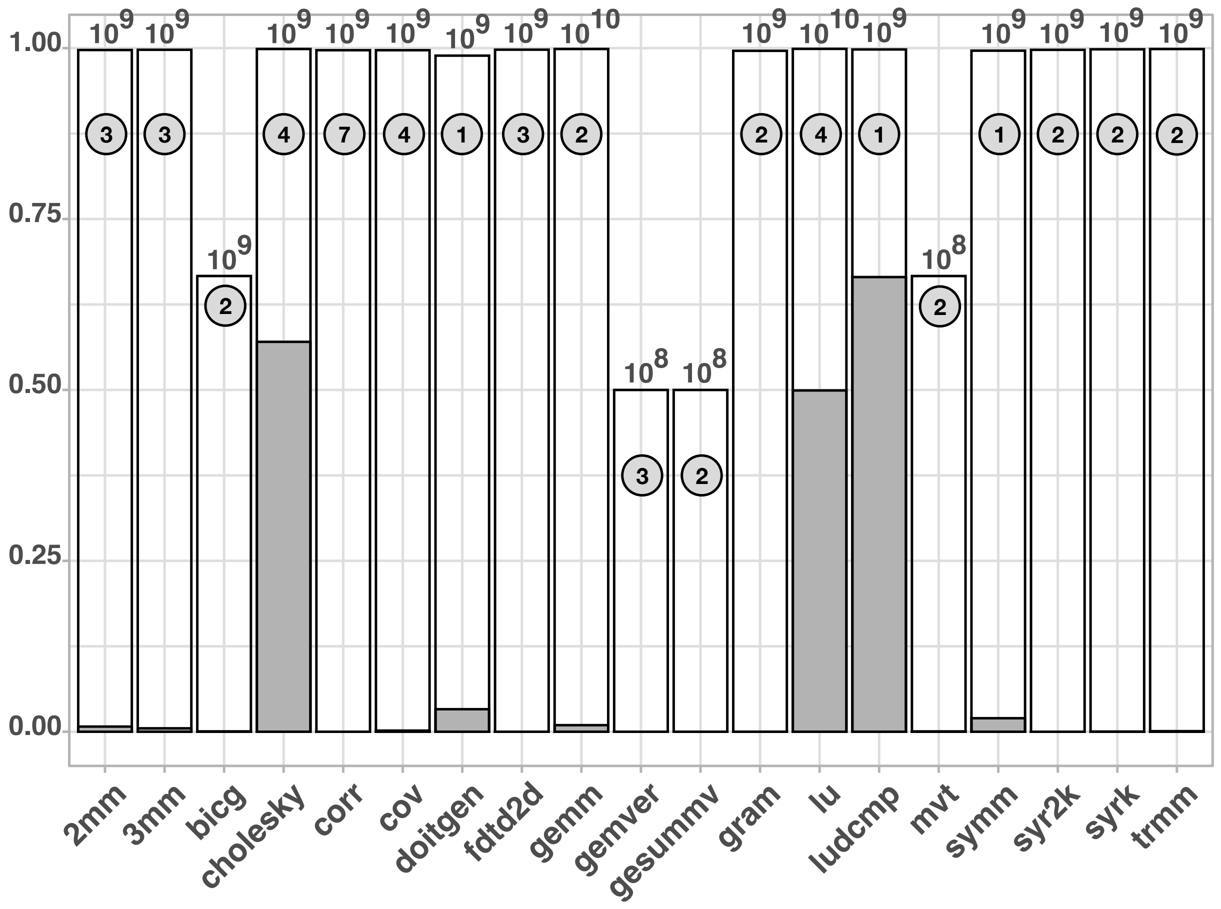


Figure 6.2: Prevalence of the semiring pattern on the Polybench suite. Powers on top of bars denote absolute number of dynamic instances of stores (order of magnitude).

6.2 RQ2: Speedup

Figure 6.3 shows the absolute runtime of the 20 programs present in the Polybench collection that present any semiring pattern. The figure shows the mean of five executions for each one of the five different optimization modes that we consider: the original program, plus the four optimizations described in Section 4. Notice that these four optimizations are applied independently and exclusively. The category called `ORG` (short for original) represents the Polybench programs compiled with clang -O3. The other categories include one of the semiring optimizations discussed in this work, in addition to the other optimizations available in clang -O3. All the original programs run for at least 5 seconds. The longest runtime belongs to `simm`: 97.34 seconds, in the original program.

Figure 6.4 shows the speedup of Semiring Optimization when compared to the original programs (`ORG`). Considering significant the experiments with a p-score under 0.05, we observed statistically significant speedups in 9 out of 13 benchmarks. The largest

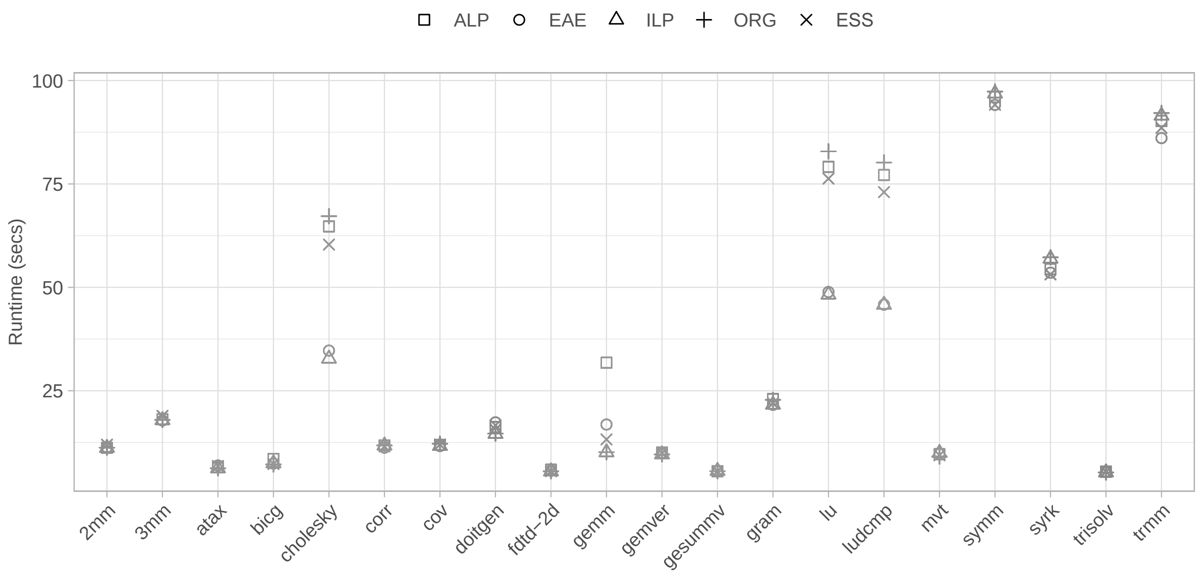


Figure 6.3: Mean of five executions of the programs in the Polybench collection that presented semiring patterns.

speedups were observed in the three programs with the highest number of silent stores, as reported in Figure 6.2: 2.05x on `Cholesky`, 1.72x on `Lu` and 1.75x on `Ludcmp`. All these speedups were produced using the pre-loop profiling technique discussed in Section 5.4. The simple elimination of stores discussed in Section 5.1 is substantially less effective. For the same benchmarks, this version of semiring optimization gives us speedups of 1.11x, 1.09x and 1.10x. Hoisting the profiling code outside the loop is essential for performance. The intra-loop profiler of Section 5.3 gives us even smaller speedups: 1.04x, 1.05x and 1.04x.

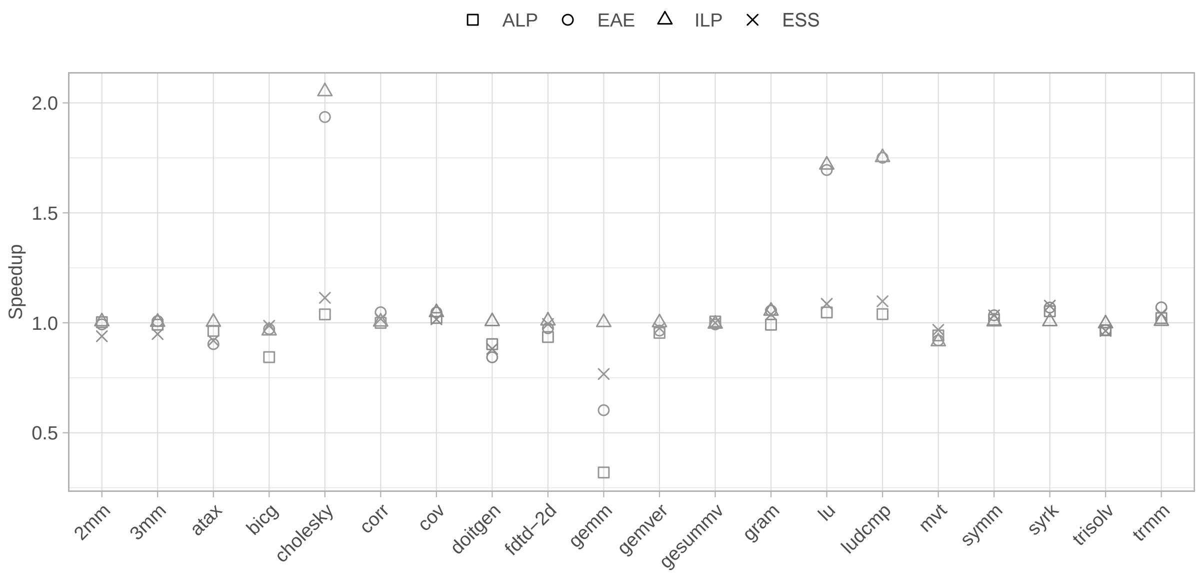


Figure 6.4: Speedup of optimizations over baseline (`ORG`)

The benefit of profiling over the irrestrict elision of code (Section 5.2) is clear once we consider vectorization. Although EAE gives us speedups in some benchmarks, it also yields large slowdowns whenever it disables vectorization. As an example, the original version of **`gemm`** is 1.66x faster than the version optimized with EAE. Pre-loop profiling recovers this slowdown in its totality: there is no statistically significant difference between the original version of **`gemm`** and the version optimized with Intra-Loop Profiling.

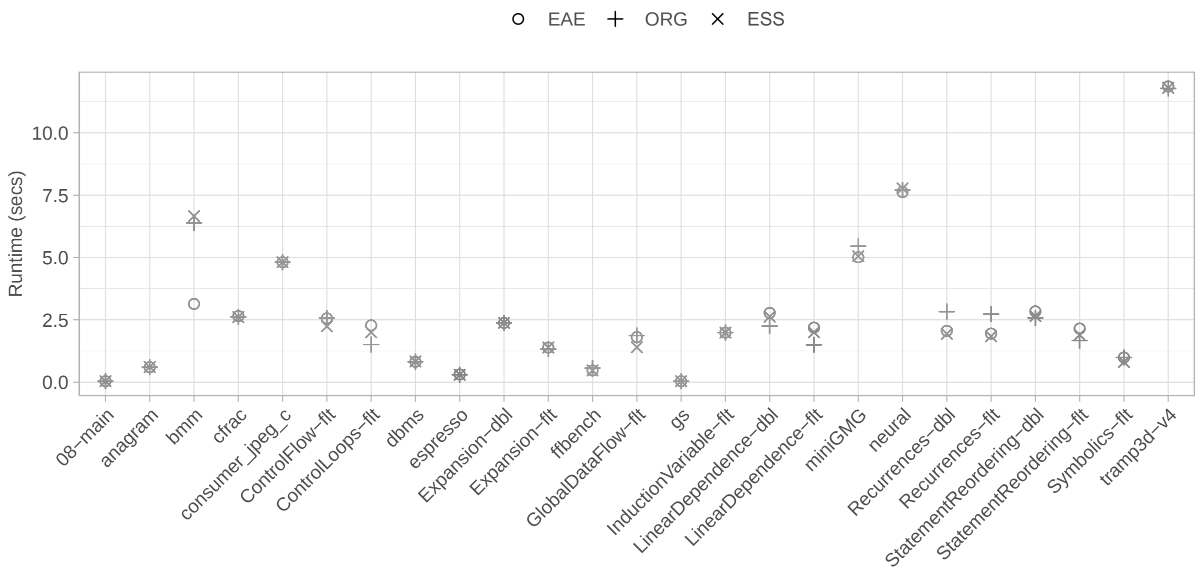


Figure 6.5: Speedup of optimizations over baseline (`ORG`) for other benchmarks

Table 6.2: List of benchmarks used from other suites.

Suite	**Benchmark**	**Semiring Patterns**	$\frac{\texttt{MS-Dyn}}{\texttt{Dyn}}$	**M-Dyn**
cBench	consumer_jpeg_c	11	0.74	10^9
DOE_ProxyApps_C	miniGMG	24	0.79	10^9
FreeBench	neural	2	0.75	10^9
MallocBench	cfrac	3	0.85	10^9
MallocBench	espresso	39	0.91	10^8
MallocBench	gs	26	0.68	10^7
MallocBench	make	2	0.55	10^4
McCat	08-main	1	0.92	10^7
mediabench	jpeg-6a	11	0.72	10^6
MiBench	consumer-jpeg	12	0.51	10^6
Misc	ffbench	2	0.99	10^8
Ptrdist	anagram	4	0.85	10^9
tramp3d-v4	tramp3d-v4	13	0.73	10^9
TSVC	ControlFlow-flt	22	0.79	10^9
TSVC	ControlLoops-flt	5	0.63	10^9
TSVC	Expansion-dbl	2	0.50	10^{10}
TSVC	Expansion-flt	2	0.50	10^{10}
TSVC	GlobalDataFlow-flt	4	0.83	10^{10}
TSVC	InductionVariable-flt	1	0.87	10^{10}
TSVC	LinearDependence-dbl	7	0.69	10^{10}
TSVC	LinearDependence-flt	7	0.69	10^{10}
TSVC	Recurrences-dbl	1	1.00	10^9
TSVC	Recurrences-flt	1	1.00	10^9
TSVC	StatementReordering-dbl	2	0.50	10^{10}
TSVC	StatementReordering-flt	2	0.94	10^{10}
TSVC	Symbolics-flt	2	0.87	10^{10}
VersaBench	bmm	1	0.96	10^9
VersaBench	dbms	7	1.00	10^8

In figures 6.5 and 6.6 we present the results for 25 programs outside the PolyBench test-suite. These programs were chosen from the list of 126 benchmarks mentioned in Section 6.1 and were selected due to their *high proportion* of silent stores. We classify the proportion of silent stores as high if $\frac{\texttt{MS-Dyn}}{\texttt{Dyn}} \geq 5\%$, i.e., the coefficient of dynamic marked store instructions must represent at least 5% of all dynamic instances. Table 6.2 shows the list of benchmarks we use for this experiment. The last column `M-Dyn` is the order of magnitude for the number of dynamic stores marked.

Note that we do not report the runtime and speedup for profiling variants of Semiring Optimization. As a current limitation of our implementation, the program slicing step requires loops to have *Single-Entry Single-Exit* property, which is not the case for most benchmarks. Thus, we choose to not present profiling variants results for those suites.

We have observed 6 statistically significant speedups with EAE. The largest speedups were observed on VersaBench/Bmm (2.03x), TSVC/Recurrences-flt (1.4x), TSVC/Recurrences-dbl (1.37x) and Misc/ffbench (1.22x).

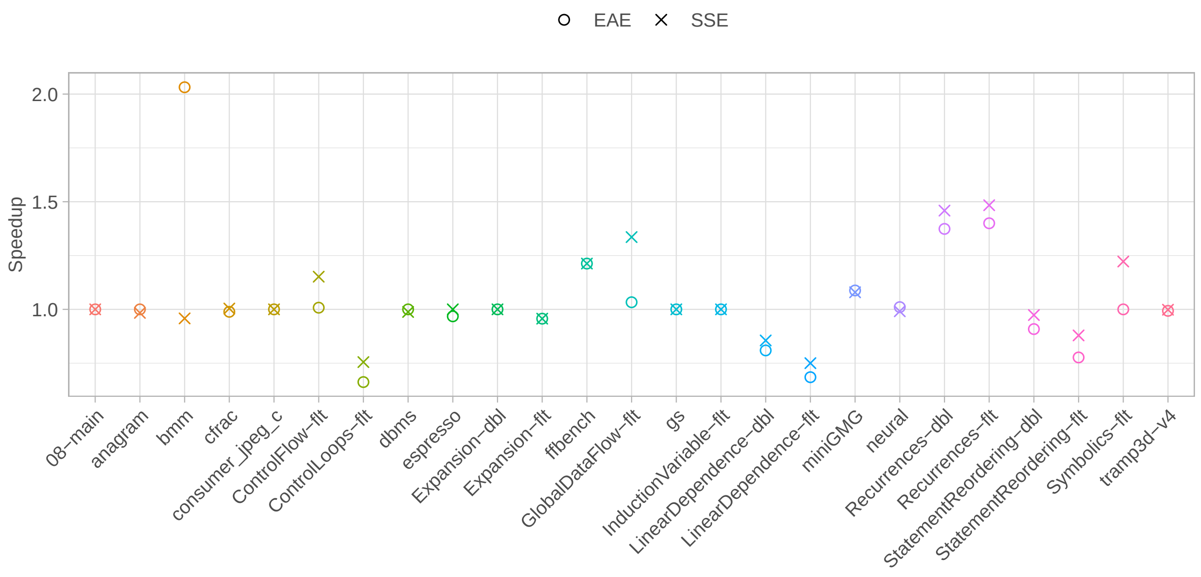


Figure 6.6: Speedup of optimizations over baseline (`ORG`) for other benchmarks

The simplest elimination (ESS) was more conservative and yielded speedups in 4 benchmarks. However, for two of these, the improvements in runtime were slightly better. We have also observed slowdowns on EAE and ESS. The largest slowdowns were 0.66x (EAE) observed on TSVC/ControlLoops-flt and 0.75 (ESS) for both TSVC/LinearDependence-flt and ControlLoops-flt.

6.3 RQ3: Overhead

Figure 6.7 shows the overhead of the profiling techniques proposed in Sections 5.3 and 5.4. Results are given in terms of percentage of the original runtime. To build the figure, we made the "optimized code section" the same as the unoptimized code; hence, any change in runtime is due to profiling. When augmented with the ALP profiler (Section 5.3), the execution time of the Polybench programs has varied within the range $[-6\%, +27\%]$. The few speedups produced by ALP are due to prefetching: `perf` reveals that cache misses decreased in `cholesky`, `lu` and `ludcmp`. PLP was more stable: we could not consistently measure any runtime variation outside the interval $[-0.4\%, +0.5\%]$. Notice that each program runs for at least 5s, whereas profiling accounts for milliseconds of execution. Therefore, we conclude that for long-running applications, the overhead of PLP profiling is negligible.

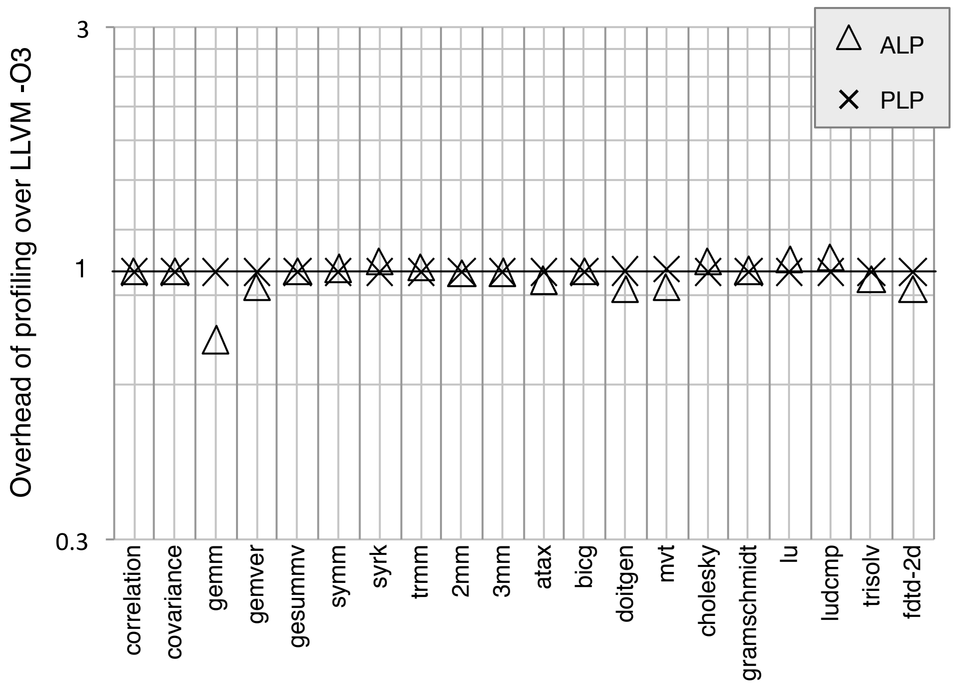


Figure 6.7: Overhead of profiling (both pre and intra loop). The Y-axis indicate percentages.

6.4 RQ4: Impact in Compilation Time

Figure 6.8 shows the total compilation time that clang -O3 plus semiring optimization spends on Polybench. For each benchmark, we show: (i) the total time taken by the optimization passes in clang -O3; (ii) the total time taken to execute the semiring optimization pass and (iii) the time taken by the standard LLVM analyses and transformations necessary to enable semiring optimization. In this last category, we count the following LLVM passes: `instcombine, early-cse, indvars and loop-simplify`. These passes are necessary to simplify the control flow graph. Notice that the time taken by clang -O3 varies according to the version of semiring optimization that we use. Variation happens mostly between the approaches that use profiling (ALP and PLP) and those that do not (ESS and EAE). This difference is due to the fact that the profiling code is inserted before LLVM -O3 runs; thus, there will be more code to be optimized.

Inspection of Figure 6.8 reveals that semiring optimization is practical. In absolute terms, clang -O3 takes approximately 0.30 seconds on average to compile the 20 programs in Figure 6.8. This number increases by 0.01, 0.01, 0.10 and 0.23 seconds when considering, respectively, ESS, EAE, ALP and PLP. The time taken by semiring optimization itself is usually shorter than the time taken by its supporting optimizations.

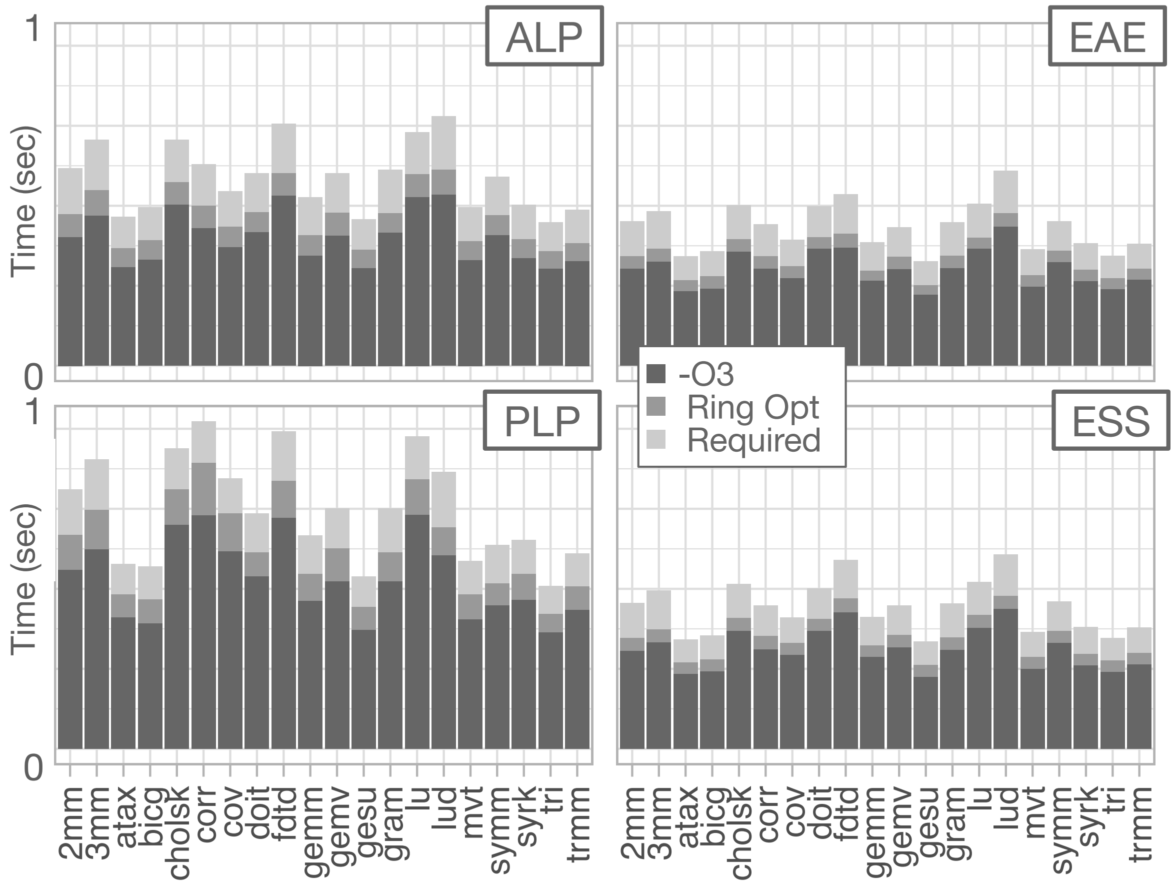
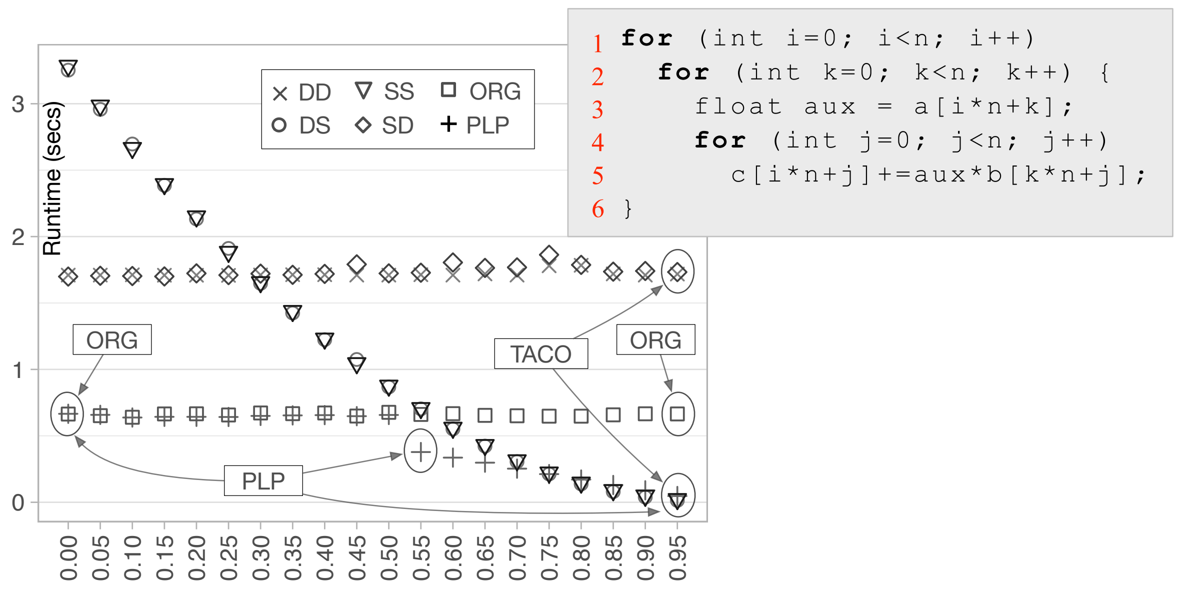


Figure 6.8: The overhead of Semiring Optimization on compilation time over the baseline (ORG). Dark gray is the time to compile with -O3. In gray is the time for Semiring Optimization and light gray the time for optimizations required by RO.

6.5 RQ5: Comparison with a Specialized Tensor Compiler (TACO)

Semiring patterns are common in linear algebra, as our evaluation on Polybench indicates. There are compilers specialized in the generation of code for this kind of applications. We believe that the current state-of-the-art approach in the field is TACO (short for *Tensor Algebra Compiler*) [14, 13]. Figure 6.9 compares TACO with clang (plus our Pre-Loop Profiler). We emphasize that this figure relates two different compilers.

TACO provides different data-structures to represent matrices. We have experimented with dense and sparse representations. In Figure 6.9, DD means that the two input matrices, e.g., A and B in the product $C = A \times B$, are dense; SD means that A is sparse, and B is dense. The other representations, DS and SS, follow similar nomenclature. ORG (short for ORiGinal) is clang -O3, and PLP is Pre-Loop Profiling, with a



```
for (int i=0; i<n; i++)
  for (int k=0; k<n; k++) {
    float aux = a[i*n+k];
    for (int j=0; j<n; j++)
      c[i*n+j]+=aux*b[k*n+j];
}
```

Figure 6.9: Comparison between TACO and clang + Pre-Loop Profiling on matrix multiplication with the `j` and `k` loops manually inverted, using $[1.5k \times 1.5k]$ matrices.

threshold of 50% of absorbing elements. PLP brings clang -O3 very close to TACO as the ratio of zeros mounts, whenever matrix B has a dense representation; hence, reducing a 5x gap to a runtime difference of less than 40%. Nevertheless, clang -O3 plus PLP can never beat TACO. This result is expected, for we are comparing different programs: TACO uses special matrix representations, whereas semiring optimization is a general optimization that avoids unnecessary computations. These techniques are complementary, as semiring optimizations could also be implemented in TACO.

6.6 RQ6: The Impact of Program Inputs

The optimizations proposed in this paper are dependent on input values—a fact already observed in Figures 3.3 and 6.9. Figure 6.10 provides more insight on this behavior. To prepare this experiment, we have changed the `init_array` routine used in three PolyBench kernels to insert zeros into the input matrices with the probabilities seen in the X-axis of Figure 6.10. We show results for three programs: `Cholesky`, `Gemm` and `Gramschmidt`. We chose these three programs because they are representative of the kinds of behaviors that our optimization tends to produce.

`Cholesky`, for instance, already manipulates diagonal matrices. Any flavor of semiring-optimization is already advantageous in this case, as more than half of every

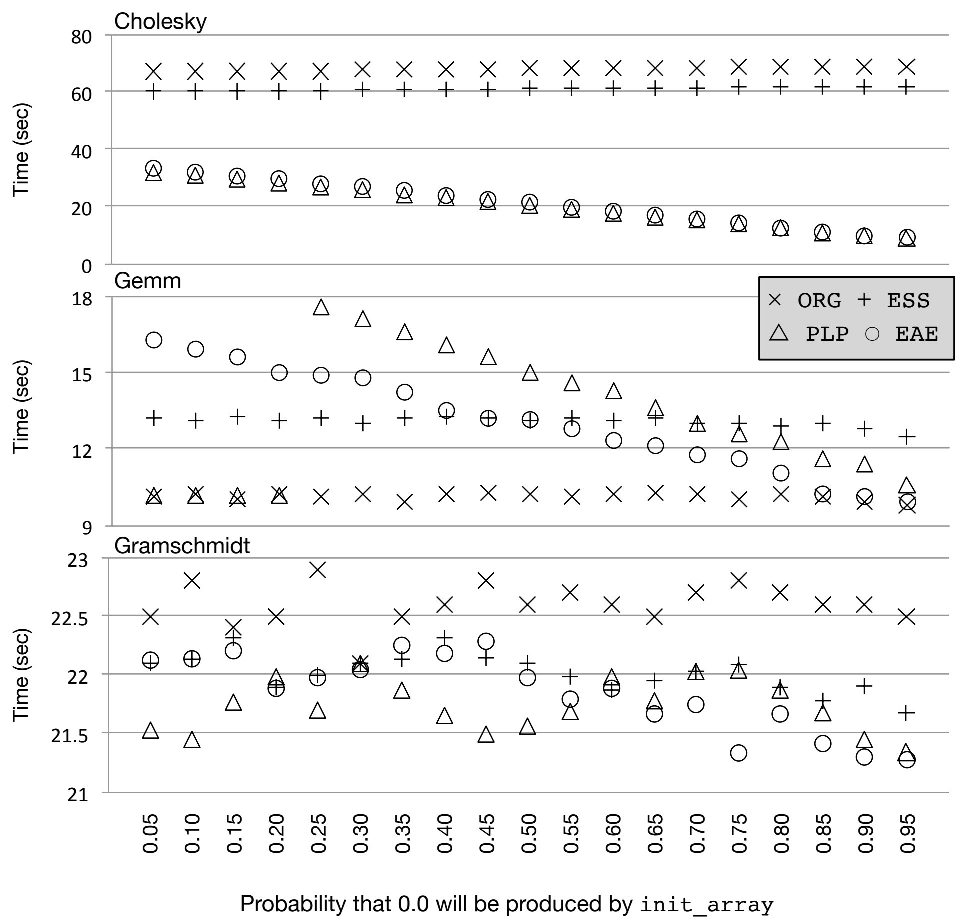


Figure 6.10: Impact of inputs onto three PolyBench programs. X-axis shows probability of a cell being initialized with a zero, in case its value can vary. Cells are independently set to zero, and matrices are produced independently.

matrix is set to zero. Nevertheless, Figure 6.10 shows that the code produced by EAE and PLP runs faster as the density of zeros increases. ESS also enhances performance, but, this improvement tends to remain constant, regardless of the input. PLP, once past the optimization threshold, is not strictly equivalent to EAE. For instance, in `Gramschmidt`, it yields faster code, because it reduces misses in the data cache (as observed via `perf`). In `Gemm`, the profiling overhead leads to slower code, compared to EAE. Finally, the effect of ESS: saving one addition and one store is small compared to the elimination of loads that we obtain with PLP and EAE.

6.7 RQ7: Performance Counters

Figure 6.11 uses performance counters to measure the effect of the different flavors of semiring-optimizations onto the i-k-j version of matrix multiplication that was evaluated in Section 6.4. In this case, the expression `c[i*n+j]+=a[i*n+k]*b[k*n+j]` admits vectorization over all the memory accesses. Figure 6.11 clarifies why simple elimination of silent stores (ESS, Section 5.1) is not an effective optimization for this particular benchmark, whereas EAE (Section 5.2) and PLP (Section 5.4) are. The conditional that ESS inserts within the innermost loop hinders vectorization completely—LLVM -O3 is not able to vectorize none of the memory accesses due to that branch.

Both EAE and PLP still support vectorization in this benchmark. To understand why, we refer the reader to the code in Figure 6.9. When applied onto this code, EAE will insert a conditional test guarding the load at line 3. The second part of this optimization moves the forward slice of that load to within the guard. This transformation will ensure that the innermost loop will be entirely moved inside this guard. Therefore, the innermost loop can still be vectorized.

Figure 6.11 evidences the point in which the online profiler activates semiring-optimization. In this experiment, we adopt a threshold of 50%; that is, one the probability of finding a zero in the input matrix reaches this point, the `sampling` routine seen in Figures 3.5 and 5.6 activates the optimized version of matrix multiplication. It is possible to observe an abrupt change in PLP's behavior in Figure 6.11 at this moment: PLP stops following ORG's pattern, and starts following EAE's. Figure 6.11 also provides some idea on the overhead of the profiler, once we observe the number of last-level-cache accesses: the sampling compromises locality during profiling; hence, more misses in the lower levels of the cache are, indeed, expected.

6.8 Discussion

The experiments described in this section show that when semiring patterns dominate computations, their elision leads to statistically significant speedups (Section 6.2). As Figure 6.4 indicates, gains can be dramatic, reaching three-fold levels, as observed in VersaBench's `bmm`. Furthermore, online profiling tends to preserve performance-safety when applying the optimization: the overhead of the pre-loop profiler is insignificant (Section 6.5), and its benefits noticeable (Figure 6.4). Nevertheless, it tends to increase

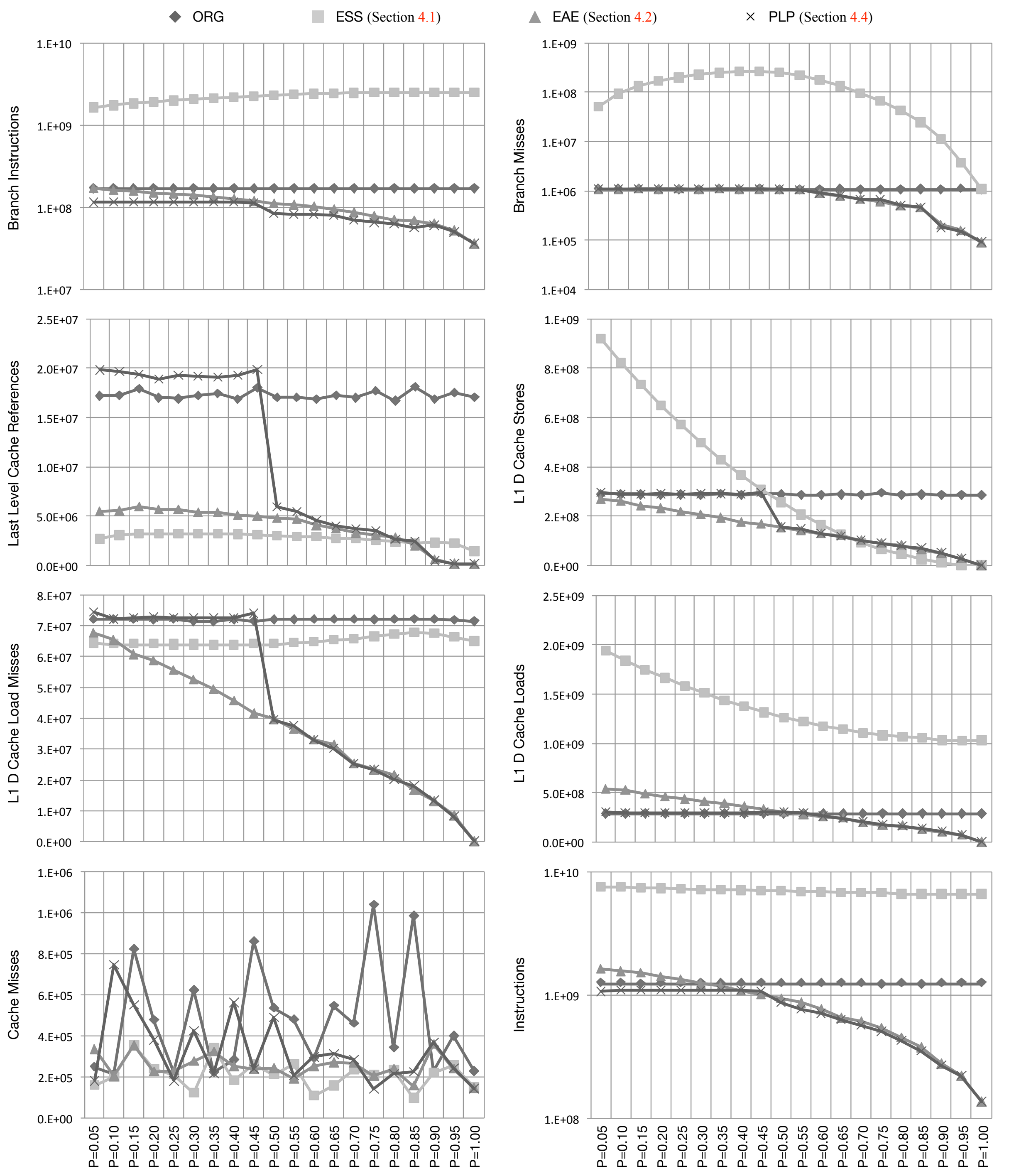


Figure 6.11: The effect of semiring-optimizations on the ikj version of matrix multiplication, seen in Figure 6.9. The X-axis shows the probability that input.

compilation time, although less than two-fold, when compared to the other variants of semiring optimization discussed in this paper (Section 6.3).

We emphasize that silent semiring expressions are relatively uncommon, at least in the benchmarks that we have evaluated. They appeared in 126, out of 259 benchmarks; however, most dynamic occurrences of semiring patterns were not silent—hence, nonoptimizable. As a consequence, the optimization that we advocate in this paper is not as general as classic compiler techniques such as constant propagation or global-value numbering. Our experience indicates that semiring patterns are most common in linear-algebra applications. Yet, there are programs outside this domain that also tend to present semiring patterns. Examples include implementations of SAT solvers over linked lists of booleans or computation of transitive closures via boolean matrix multiplication.

Because semiring patterns are relatively uncommon, when applied onto large programs, our optimizations are unlikely to produce the large gains observed in Section 6.2. We have applied it onto the integer benchmarks in SPEC CPU2006, using the same experimental setup of that section. The elision of absorbing elements led to statistically significant gains only in `perlbench`. When using the reference input, the optimized version of `perlbench` runs in 49.80 seconds. Without the elision of absorbing elements, but still at the -O3 optimization level, `perlbench` runs in 51.50 seconds. No statistically significant running time difference was observed in the other 11 benchmarks, except in `namd`. In that case, without the guard inserted by the pre-loop profiler, we perceived a slowdown (262.9 seconds without the optimization vs 293.80 with it).

Chapter 7

Conclusion

This dissertation has described a code optimization technique that avoids certain operations that, depending on the input values, are redundant. We call these operations *semiring patterns*. These so called semiring optimizations can be implemented in any classic compiler, and work in any commodity hardware. Therefore, it was a pleasant surprise that we could observe speedups of almost 2x over clang -O3 in benchmarks such as Polybench's CHOLESKY, which has been used for years in the gcc and LLVM communities.

We have also observed large speedups in other programs available in the LLVM test suite, such as VERSABENCH/BMM (2.03x), TSVC/RECURRENCES-FLT (1.40x) and MISC/FFBENCH (1.22x), all using simple elision of absorbing elements, without the support of profiling. Thus, we believe that semiring optimizations are a viable and effective way to improve the quality of the code generated by mainstream compilers.

7.1 Future Work

There are small improvements that can be done to fine tuning the optimization. First, our implementation of the Program Slicing algorithm requires loops to have the Single-Entry Single-Exit (SESE) property. This limits us to a subset of programs we can optimize. The work of [24] could be used to infer array bounds to reconstruct loops. Second, the framework we propose goes beyond the elimination of semiring patterns that lead to silent stores (i.e. Figure 1.1(b)). Finally, we think that a just-in-time compiler would benefit more of this optimization. One compiler that we considered implementing this optimization was Numba [17]. Numba is a JIT compiler that translates a subset of Python and NumPy into fast-machine code using LLVM.

References

[1] Gordon B. Bell, Kevin M. Lepak, and Mikko H. Lipasti. Characterization of silent stores. In *PACT*, pages 133–, Washington, DC, USA, 2000. IEEE.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, Cambridge, MA, US, 3rd edition, 2009.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, New York, NY, USA, 1989. ACM.

[5] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

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

[7] Brian Gough and Richard Stallman. An introduction to gcc. *Network Theory, Ltd*, 2004.

[8] Björn Gustavsson. Introduction to ssa. http://blog.erlang.org/introducing-ssa/, 2018.

[9] David Hilbert. *Die Theorie der algebraischen Zahlkörper*. Jahresbericht der Deutschen Mathematiker-Vereinigung, Germany, 1904.

[10] David G. Hough and Mike Cowlishaw. IEEE standard for floating-point arithmetic, 2019.

[11] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An

effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[12] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and mitigating data-dependent dram failures by exploiting current memory content. In *MICRO*, pages 27–40, New York, NY, USA, 2017. ACM.

[13] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *ASE*, pages 943–948, Piscataway, NJ, USA, 2017. IEEE Press.

[14] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, 2017.

[15] Donald Knuth. *Arithmetic*, chapter 4, pages 194–525. Addison-Wesley, Boston, MA, USA, 1998.

[16] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM, 1981.

[17] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[18] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *ISCA*, pages 182–191, New York, NY, USA, 2000. ACM.

[20] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In *MICRO*, pages 22–31, New York, NY, USA, 2000. ACM.

[21] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. In *ASPLOS*, pages 30–41, New York, NY, USA, 2002. ACM.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[23] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE.

[24] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2):13:1–13:25, 2017.

[25] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[26] Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. Static prediction of silent stores. *ACM Trans. Archit. Code Optim.*, 15(4):44:1–44:26, November 2018.

[27] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012. http://www.cs.ucla.edu/pouchet/software/polybench.

[28] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CC*, pages 110–120, New York, NY, USA, 2016. ACM.

[29] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. Sparso: Context-driven optimizations of sparse linear algebra. In *PACT*, pages 247–259, New York, NY, USA, 2016. ACM.

[30] Johannes Späth. A brief overview of shimple. https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple, 2014.

[31] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[32] JavaScriptCore Team. Bare bones backend - webkit. https://webkit.org/docs/b3/, 2015.

[33] Mark Weiser. Program slicing. In *ICSE*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE.

[34] Shasha Wen, Milind Chabbi, and Xu Liu. REDSPY: Exploring value locality in software. In *ASPLOS*, pages 47–61, New York, NY,USA, 2017. ACM.

[35] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *ASPLOS*, pages 332–347, New York, NY, USA, 2018. ACM.