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

Anonymous Author(s)

Abstract

This paper describes the theory and practice of an optimization technique that eliminates dynamic occurrences of expressions in the format $a = a \oplus b \otimes c$. The operation $\oplus$ must admit an identity element z, such that $a \oplus z = z \oplus a = a$. Additinally, z must be the absorbing element of the operation $\otimes$, such that $b \otimes z = z \otimes c = z$. In its most basic form, this contract requires that the triple $(S, \oplus, \otimes)$, $\{a, b, c\} \subset S$ forms a ring. This pattern is very common in high-performance benchmarks –its canonical representative being the multiply-add operation $a = a + b \times c$. However, several other expressions involving arithmetic and logic operations can be grouped within the required algebra. We show that the runtime elimination of such assignments can be implemented in a performance-safe way via *in-loco* profiling. The dynamic elimination of redundant expressions involving identity and absorbing elements in the Polybench suite accounts for a speedup of 10%, on average, over LLVM -O3. In three programs from Polybench, we have observed speedups of 1.75x, 1.92x and 2.05x. When added onto `clang`, ring optimizations bring it closer to TACO, a specialized tensor flow compiler.

CCS Concepts • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Symbolic and algebraic manipulation*; • **Computer systems organization** → Serial architectures;

Keywords Compiler, Optimization, Profiling, Ring

ACM Reference Format:
Anonymous Author(s). 2018. Ring Optimization: Dynamic Elision of Expressions with Identity and Absorbing Elements. In *Proceedings of 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3168822

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 $T \times T \rightarrow T$. 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 `m[i, j]+=p[i, k]×q[k, j]`, heart of matrix multiplication, is an example of this family of expressions. In Section 3.1 we show that this family is well-provided with a rich assortment of members. Indeed, any triple $(T, \oplus, \otimes)$ forming an algebraic *ring*[4, Chapter IX] meets the required constraints. In this paper, we show that the pattern $a = a \oplus b \otimes c$ is amenable to a kind of transformation, henceforth called *ring 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-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-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-iii outlines the general transformation.

(i) $a = a + e$

```
1 t0 = ld a
2 t1 = ld e
3 t2 = t0+t1
4 st a t2
```

```
1 t1 = ld e
2 if (t1) {
3   t0 = ld a
4   t2 = t0+t1
5   st a t2
6 }
```

(ii) $a = b \times c$

```
1 t0 = ld b
2 t1 = ld c
3 t2 = t0*t1
4 st a t2
```

```
1 t0 = ld b
2 if (t0) {
3   t1 = ld a
4   if (t1) {
5     t2 = t0*t1
6     st a t2
7   }
8 }
```

(iii) $a = a + b \times c$

```
1 t0 = ld a
2 t1 = ld b
3 t2 = ld c
4 t3 = t1*t2
5 t4 = t3+t0
6 st a t4
```

```
1 t1 = ld b
2 if (t1) {
3   t2 = ld c
4   if (t2) {
5     t3 = t1*t2
6     t0 = ld a
7     t4 = t3+t0
8     st a t4
9   }
10 }
```

Figure 1. The naïve implementation of ring optimizations.

The Issue of Performance Safety. The code transformations seen in Figure 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-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 [5], make register allocation more difficult and, most of all, hinder vectorization. As we show in Section 5, the impossibility to apply vectorization into heavily nested loops might double the runtime of some of the programs in the Polybench suite.

In this paper, we show how to implement ring 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 ring optimizations –two of which resort to on-line profiling:

- Section 4.1 discusses the simplest implementation of ring optimization: the conditional elimination of silent stores due to operations involving the identity element.
- Section 4.2 discusses the conditional elimination of loads whose values are absorbing elements, and of the forward program slice that depends on such loads.
- Section 4.3 presents an on-line profiling technique implemented within loops that avoids checking loads that do not contain absorbing elements.
- Section 4.4 shows how to move the code in charge of on-line profiling outside the loop. This code hoisting harmonizes ring-optimization and vectorization.

Summary of Results. We have implemented the four variations of ring-optimization in LLVM [9]. Even though ring-optimizations may appear, at first, innocuous, their effects are impressive when properly implemented. Section 5 presents empirical evidence supporting this statement. We have tried our implementation onto the 249 programs available in the test suite of LLVM. Ring patterns appear in 128 programs. For concision, we shall present detailed results only for the Polybench suite, although numbers for the other benchmarks shall be mentioned. Polybench consists of 32 programs, out of which 20 contain the ring pattern. The naïve version of ring optimization (Section 4.1) increases performance, with a significance level $\alpha = 0.05$, in nine benchmarks (out of 20). These results use, as baseline, LLVM -O3. We have observed speedups of 2.05x, 1.92x and 1.75x in different PolyBench programs. However, we have also observed slowdowns of 1.35x and 1.25x in two benchmarks. This scenario improves as we move from the naïve elimination of silent stores towards the on-line profiler hoisted outside loops (Section 4.4). In its fourth, and most effective implementation, ring optimization causes a maximum slowdown of 9%, while maintaining all the previous speedups.

2 Overview

In this section, we will explain the optimization that we propose in this paper, and we will introduce three of its variants, which will be further detailed in Section 4. Our exposition will use the naïve matrix multiplication algorithm seen in Figure 2. All the examples in this section 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 5 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,
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         c[i*n + j] += a[i*n + k] * b[k*n + j];
8     }
9 }
```

Figure 2. Naïve matrix multiplication.

Silent Store Elimination. The store operation at line 7 of Figure 2 is *silent* whenever the product `a[i*n+k] * b[k*n+j]` happens to be zero. "Silent Store" is a term coined by Lepak and Lipasti [10] to denote a store operation that writes in memory a value that was already there. As recently demonstrated by Pereira *et al.* [13], expressions involving the value zero are a common source of silent stores. In the context of this paper, 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 shows code that exercises such possibility.

```
1  void mul_SSE(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 aux = a[i*n + k] * b[k*n + j];
8        if (aux) {
9          c[i*n + j] += aux;
10       }
11     }
12 }
```

Figure 3. Naïve matrix multiplication after Silent Store Elimination.

At the first optimization level of `clang`, e.g., `-O0`, the code in Figure 3 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_SSE`, has worse runtime than function `mul_ORG` at that optimization level. Figure 4 shows this comparison, considering input matrices with increasing probability of having cells with the value zero[1]. Function `mul_SSE` has two disadvantages, when

[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.

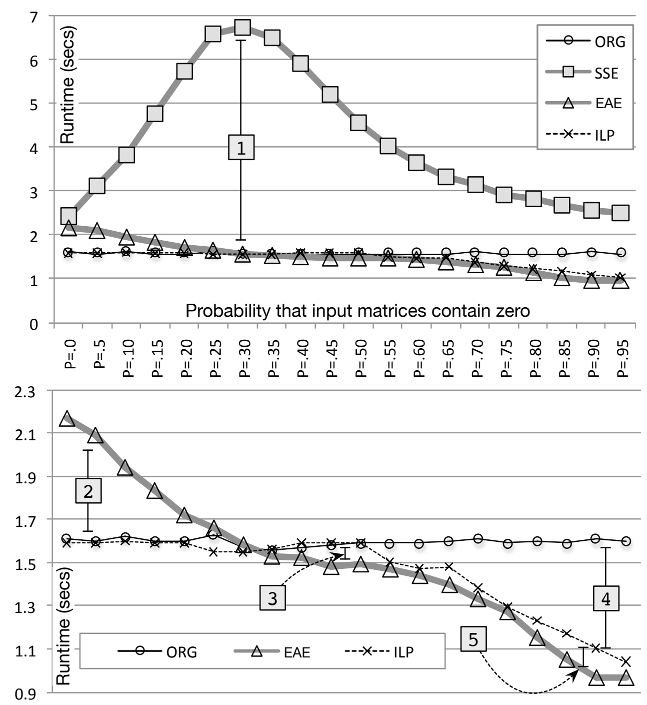


Figure 4. Runtime of different implementations of matrix multiplication ($C = A \times B$), given different probabilities that cells from matrices $A[10^3 \times 10^3]$ and $B[10^3 \times 10^3]$ contain zero. Programs were compiled with `clang -O3`.

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_SSE`. 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 4.

Elision of absorbing elements. The guard of Figure 3 encompasses too narrow a region to be of much benefit. To widen it, we notice that either $\mathtt{a[i*n+k]} = 0$ or $\mathtt{b[k*n+j]} = 0$ is a sufficient condition for a silent store, as zero is the absorbing element of multiplication. Figure 5 uses this observation to optimize the code originally seen in Figure 2. The chart in Figure 4 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 4). 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 5.

```
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 5. Elision of Absorbing Elements applied onto the implementation of naïve matrix multiplication.

Inter-Loop Profiling. Ring 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 ring optimization. Key to this possibility is *profiling*. Figure 6 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 ring pattern, we call this technique *Inter-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 4.3.

```
1 void mul_ILP(float *restrict a, float *restrict b,
2              float *restrict c, int n) {
3   if (sampling(a, n, 0.0F) > 0.50)
4     mul_EAE(a, b, c, n);
5   else
6     mul_ORG(a, b, c, n);
7 }
```

Figure 6. Inter-Loop Profiling applied onto the implementation of naïve matrix multiplication.

The function `sampling` invoked at line 3 of Figure 6 reads a few positions of an array to count occurrences of a value. In Figure 6, 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 ring optimization. Figure 4 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 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 ring 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, Section 5 will show situations in which sampling has a positive effect, due to data prefetching. In Section 4.4 we shall explain how we generate code to carry out sampling for any loop containing the ring pattern.

3 Generalizing Ring Optimizations

This section has two goals. First, in Section 3.1 we present a list of ring patterns. Second, in Section 3.2, we introduce an algorithm to identify load and store instructions that can be eliminated, given the occurrence of identities and absorbing elements in the ring pattern.

3.1 A Family of Ring Expressions

Figure 7 enumerates the patterns that are of interest in this paper. The operations $\oplus$ and $\otimes$ in parts **(i)** and **(ii)** of the figure form true rings whenever the absorbing element of $\otimes$ is the identity of $\oplus$. These rings range on integer and floating-point[2] types. Operations in Figure 7 **(ii)** have only the identity element –the absorbing value is missing. Nevertheless, they form rings when combined with operations that contain the absorbing element. We let LAND stand for logical conjunction, and LOR stand for logical disjunction.

$\oplus$ $\otimes$	Id	Ab	$\oplus$	Id	Ab	$\oplus$(v) $\otimes$(v)	Id	Ab
OR(v_1, v_2)	0	~0	ADD(v_1, v_2)	0	$\perp$	DIV(_, v)	1	$\perp$
MUL(v_1, v_2)	1	0	SUB(v_1, v_2)	0	$\perp$	DIV(v, _)	$\perp$	0
AND(v_1, v_2)	~0	0	XOR(v_1, v_2)	0	$\perp$	MOD(_, v)	$\perp$	1
LOR(v_1, v_2)	F	T	SHL(v_1, v_2)	0	$\perp$	MOD(v, _)	$\perp$	0
LAND(v_1, v_2)	T	F	SHR(v_1, v_2)	0	$\perp$			
(i)			**(ii)**			**(iii)**		

Figure 7. Optimizable patterns that this paper considers.

Example 3.1. The tuples (ADD, MUL, int), (ADD, MUL, float), (AND, OR, int), (OR, AND, int), and (LOR, LAND, bool) are examples of rings produced by combining operations in Figure 7.

Operations DIV and MOD in Figure 7 **(iii)** 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 n $\neq$ 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 3.2 introduces.

Example 3.2. The following identities are examples that follow from applying operations seen in Figure 7 **(iii)**: a == ADD(a, MOD(b, 1)), or a == OR(a, DIV(0, b)).

3.2 Identification of Optimization Points

The function optimize, in Figure 8, identifies optimization points in a program. This function is invoked onto expressions that fit into the pattern $a = a \oplus b$. Figure 8 represents such patterns as st$a(a \oplus b)$. In this case, $\oplus$ is any of the operations that has an identity element as seen in Figure 7, and st is a store operation. Given such a pattern, optimize invokes function pin over b. This procedure will traverse –backwardly– the data-dependence graph of b, looking for operations whose absorbing element is the identity of $\oplus$.

[2]Arithmetic operations ADD, SUB, MUL and DIV accept floating-point types

```
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 8. Identification of optimization points.

The output of function pin is a set of relations s == v, where s is a variable name, and v is the absorbing element of some operation that uses s. These two procedures, optimize and pin, have been designed to operate on programs in the Static Single Assignment format (SSA) [3]. 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 3.3. Figure 9 shows the invocation of optimize on the program St(a, ADD(VAR a, MUL(AND(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) and (d == 0). The point where each equality is created is marked in gray in Figure 9.

There are many ways to use the relations produced by function optimize. In Section 4, we will describe four such techniques. Nevertheless, ring 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 $\oplus$. To keep our report concise and reproducible, in this paper, we focus on the elimination of silent stores only.

Correctness. In this section, we lay out the key invariants of functions optimize and pin. These invariants are stated on top of the semantics of the language of logical and arithmetic expressions that pin traverses. In this context, we define the *store environment* σ : VAR $\longrightarrow$ VALUE as a function that maps

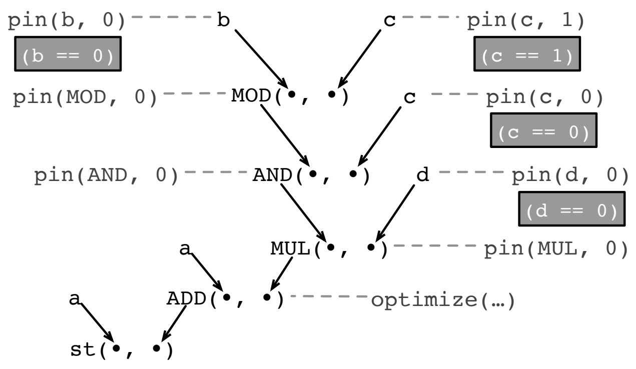


Figure 9. Example of relations produced by `optimize`.

variable names to values. Next, we define an evaluation function `eval` : `st` $\times \sigma \longrightarrow \sigma$, which receives a store instruction, like those seen in Figure 8, 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 an example, for the sake of space:

Example 3.4. If $\sigma = \{a \mapsto 7, b \mapsto 4\}$, then we have that `eval(st` a `ADD(`b, a`)`$, \sigma$`)` $= \sigma \setminus \{a \mapsto 11\}$, and `eval(st` a `ADD` `(DIV(`a, b`)`$, a$`)`$, \sigma$`)` $= \sigma \setminus \{a \mapsto 8\}$. We use $\setminus$ to denote function updating, i.e.: $\sigma \setminus \{s \mapsto v\} = \lambda x.(x = s)?v : \sigma(x)$.

The definition of `eval` lets us state the core invariant of function `pin`, which Theorem 3.5 proves.

Theorem 3.5. *If* `pin`$(b, z) = C$*, then, for any* $(s{=}{=}v) \in C$ *and any store environment* σ, `eval`$(b, \sigma \setminus \{s \mapsto v\}) = z$.

Corollary 3.6. *Let* `optimize(st` $a(\oplus(a, b))) = C$*. If* $(s{=}{=}v) \in C$*, then* `eval`$(\oplus(a, b), \sigma \setminus \{s \mapsto v\} = a$*, whenever* $\sigma(s) = v$

4 Four Variations of Ring Optimization

This section presents four ways to eliminate silent stores related to ring patterns, from the simplest (Section 4.1) towards the most complex (Section 4.4).

4.1 Version 1: Silent Store Elimination (SSE)

The simplest form of ring 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-i. Figure 10 generalizes that example. Notice that the ring pattern makes two trivial optimizations possible. In case variable `t0` is *alive* past the store at line 4 of Figure 10-i, then we must settle for the more conservative transformation seen in Figure 10-ii. Otherwise, we can also avoid the load of `t0`, using the transformation seen in Figure 10-iii. The transformations seen in Figure 10 preserve program semantics. Although trivial, we state this fact formally in Theorem 4.1, for the sake of completeness.

Theorem 4.1. *If* `t0` *is only used at line 3 of Figure 10-i, then Fig. 10-i and Fig. 10-ii are equivalent. Otherwise, Fig. 10-i and Fig. 10-iii are equivalent.*

(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 10. 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.

4.2 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 3.5. Thus, if $(s{=}{=}v) \in$ `pin`(b, z), the evaluation of b yields z whenever the symbol s holds the value v. From Corollary 3.6, $(s{=}{=}v)$ 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 4.2. Fig. 11-i shows the tree in Fig. 9 written in three-address format. Fig. 11-ii shows the guard used to check if the value loaded from `b` is an absorbing element. Fig. 11-iii displays the optimized program, provided that none of the temporary variables is used past the last store instruction.

(i)
```
1  t0 = ld b
2  t1 = ld c
3  t2 = t0 % t1
4  t3 = ld c
5  t4 = t2 & t3
6  t5 = ld d
7  t6 = t4 * t5
8  t7 = ld a
9  t8 = t6 + t7
10 st a t8
```

(ii)
```
1  t0 = ld b
2  t1 = ld c
3  t2 = t0 % t1
4  t3 = ld c
5  t4 = t2 & t3
6  t5 = ld d
7  t6 = t4 * t5
8  t7 = ld a
9  t8 = t6 + t7
10 if (t0 ≠ 0) {
11   st a t8
12 }
```

(iii)
```
1  t0 = ld b
2  if (t0 ≠ 0) {
3    t1 = ld c
4    t2 = t0 % t1
5    t3 = ld c
6    t4 = t2 & t3
7    t5 = ld d
8    t6 = t4 * t5
9    t7 = ld a
10   t8 = t6 + t7
11   st a t8
12 }
```

Figure 11. Checking if `b` is an absorbing element.

In Example 4.2, 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 4.3 clarifies these observations.

Example 4.3. Figure 12 shows three optimized versions of the code snippet from Figure 11. Each version differs on the instructions that can be placed inside the region guarded by the check on `t0`. Notice how, in Figure 12-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 12. The impact of liveness on the elision of instructions that depend on absorbing elements.

4.3 Version 3: Intra-Loop Profiling (ALP)

Ring Optimization is speculative, because its performance depends on program inputs. Thus, the unrestricted application of ring optimization might lead to runtime regression. As explained in Section 2, 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 ring patterns; hence, we call it Intra-Loop Profiling (ALP). It improves branch prediction, but still hinders vectorization. Later, in Section 4.4 we will show how to hoist the profiling code outside the loop; thus, enabling also vectorization.

Figure 13 shows how ALP 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 4.2). 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 ring optimization should take place.

Example 4.4. Figure 14 shows the implementation of ALP on the program used as an example in Section 2. For the sake of clarity, we show code written in C; however, just like all the other optimizations described in this paper, ALP is implemented at the binary level.

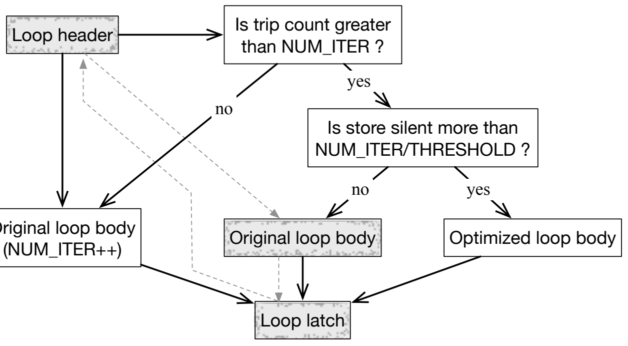

Loop header
Is trip count greater than NUM_ITER ?
yes
no
Is store silent more than NUM_ITER/THRESHOLD ?
no
yes
Original loop body (NUM_ITER++)
Original loop body
Optimized loop body
Loop latch

Figure 13. Control flow graph of loop augmented with code that implements Intra-Loop Profiling. Gray blocks and dashed arrows show code present in the original program. Dashed arrows no longer exist in the profiled program.

```
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 Optimize:
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 == NUM_ITER)
19             target = silent > NUM_ITER/THRESHOLD
20                    ? Optimize : Original;
21           c[i*n + j] += t;
22       }
23     }
24   }
25 }
```

Figure 14. Naïve matrix multiplication augmented with code to implement Intra-Loop Profiling.

4.4 Version 4: Inter-Loop Profiling (ILP)

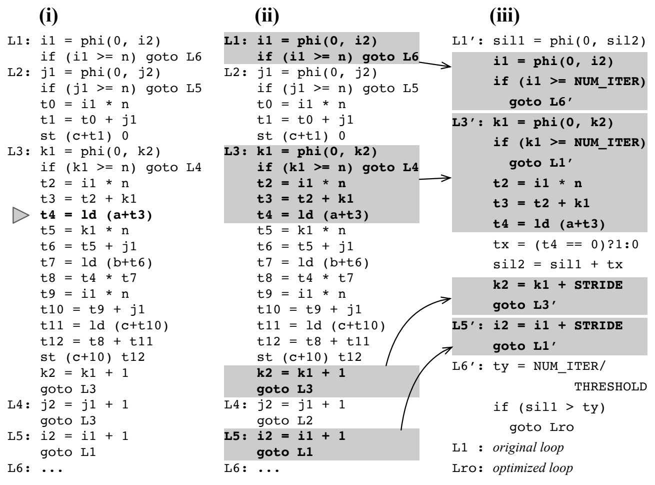
The *in-vivo* profiling technique discussed in the previous section improves the hit rate of the branch predictor, as it tends to reduce the number of branches dynamically executed. However, it is still hard to vectorize the optimized code. As an example, neither LLVM 8.0 nor gcc 6.0 can vectorize the assignment at line 8 of Figure 14. 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.

Program Slicing. We use backward program slicing to implement the Inter-Loop Profiler. A *program slice* with respect to instruction ι in program P is the subset of P that contributes to the execution of ι [17]. To profile the memory location loaded by instruction ι_{ld}, we extract the program slice of ι_{ld}. Because a backward slice considers data and control-dependencies, it contains every loop that contributes to compute the address used in ι_{ld}. In this paper, we have reused a slicing algorithm available for the LLVM compiler [15].

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 4.3 always follows the pattern in which memory is accessed within the original loop. As an example, in Figure 14, sampling happens at line 15. Array `a`'s sampling stride is 1, and array `b`'s is `n`.

Sampling indexed by the loop's trip count might lead to bad decisions. For instance, one of the benchmarks that we analyze in Section 5 is Cholesky decomposition. This benchmark receives a triangular matrix, in which the elements below the main diagonal are zeros. However, sampling based on the trip count, even with a large number of profiling iterations, touches only a few cells under that diagonal. Consequently, profiling misses a substantial region that contains only zeros –multiplication's absorbing element.

Example 4.5. Figure 15 shows the backward slice of the access `a[i*n + j]` at line 7 of Figure 2. The third part of Figure 15 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 6.



```
(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 15. (i) Three-address code version of the naïve matrix multiplication algorithm seen in Fig-2. 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.

5 Evaluation

Research Questions. In this section, we shall investigate the following research questions:

- **RQ1** How often does the ring pattern appear in typical benchmarks?
- **RQ2** What is the runtime benefit of the different versions of Ring Optimization?
- **RQ3** What is the overhead of the different versions of profiling-based Ring Optimizations?
- **RQ4** What is the overhead of Ring Optimization on compilation time?
- **RQ5** How does Ring Optimization compare with a specialized tensor compiler?

Runtime Setup. We have implemented Ring 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 ring optimization on the programs available in the LLVM test suite. In this section, we shall restrict our presentation to PolyBench [14], for concision. However, we shall mention results from other benchmarks when we discuss **RQ1**, e.g., prevalence. In Section 7 we allude to speedups in programs outside the Polybench suite. 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.

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), 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 ring 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.

5.1 RQ1: Prevalence

We have measured the prevalence ring patterns in 259 programs from 36 benchmark suites. Out of the 259 programs, 126 (49%) contain the ring pattern. Figure 16 shows the frequency of ring 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 ring 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 (9% ± 12%) involved in a ring 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.

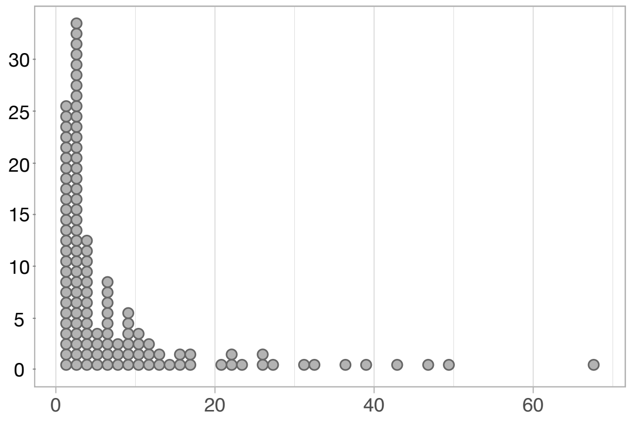


Figure 16. Histogram with the number of static store instances marked for the set of benchmarks used.

Figure 17 shows the prevalence of the ring pattern on the PolyBench test suite. The transparent bar is the ratio $\frac{\text{M-Dyn}}{\text{Dyn}}$ while the gray bar is $\frac{\text{MS-Dyn}}{\text{Dyn}}$. 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 $\frac{\text{M-Dyn}}{\text{Dyn}} \approx 1.0$ and three benchmarks (`cholesky, lu, ludcmp`) have $\frac{\text{MS-Dyn}}{\text{Dyn}} \geq 0.5$. On `ludcmp`, one static store was responsible for almost all 10^9 dynamic instances. Approximately 60% of these instances were silent.

5.2 RQ2: Speedup

Figure 18 shows the absolute runtime of the 20 programs present in the Polybench collection that present any ring 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 3. Notice that these four optimizations are applied independently and exclusively. The category called

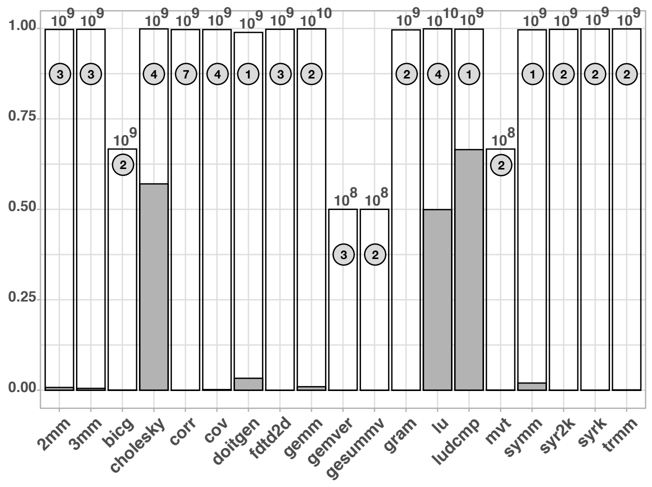


Figure 17. Prevalence of the ring pattern on the Polybench suite. Powers on top of bars denote absolute number of dynamic instances of stores (order of magnitude).

ORG (short for original) represents the Polybench programs compiled with clang -O3. The other categories include one of the ring optimizations discussed in this paper, 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.

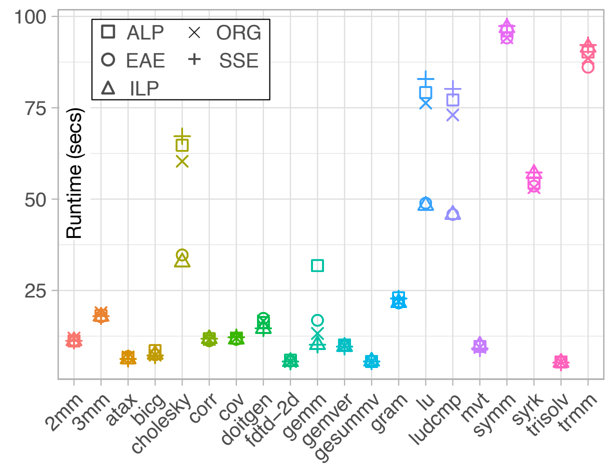


Figure 18. Mean of five executions of the programs in the Polybench collection that presented ring patterns.

Figure 19 shows the speedup of Ring Optimization when compared to the original programs (ORG). Considering significant the experiments with a p-scope under 0.05, we observed statistically significant speedups in 9 out of 13 benchmarks. The largest speedups were observed in the three programs with the highest number of silent stores, as reported in Figure 17: 2.05x on `Cholesky`, 1.72x on `Lu` and 1.75x on `Ludcmp`. All these speedups were produced using the inter-loop profiling technique discussed in Section 4.4. The simple elimination of stores discussed in Section 4.1 is substantially less effective. For the same benchmarks, this version of ring 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 4.3 gives us even smaller speedups: 1.04x, 1.05x and 1.04x.

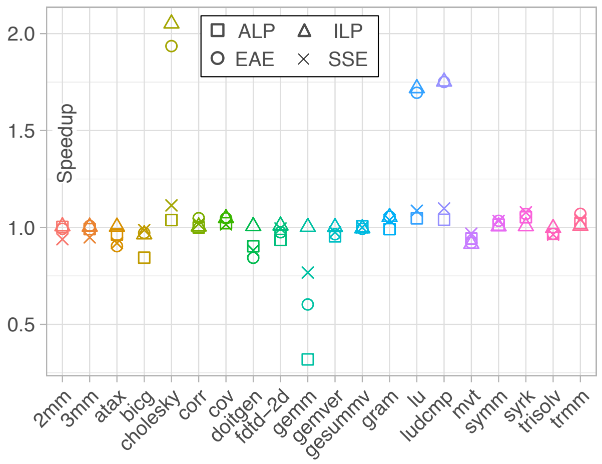


Figure 19. Speedup of optimizations over baseline (ORG)

The benefit of profiling over the irrestrict elision of code (Section 4.2) 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. Inter-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.

5.3 RQ3: Overhead

Figure 20 shows the overhead of the profiling techniques proposed in Sections 4.3 and 4.4. 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 4.3), the execution time of the Polybench programs has varied within the range [−6%, +27%] – that's a considerable impact of runtime variation ILP 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 ILP profiling is negligible.

5.4 RQ4: Impact in compilation time

Figure 21 shows the total compilation time that clang -O3 plus ring 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 implement

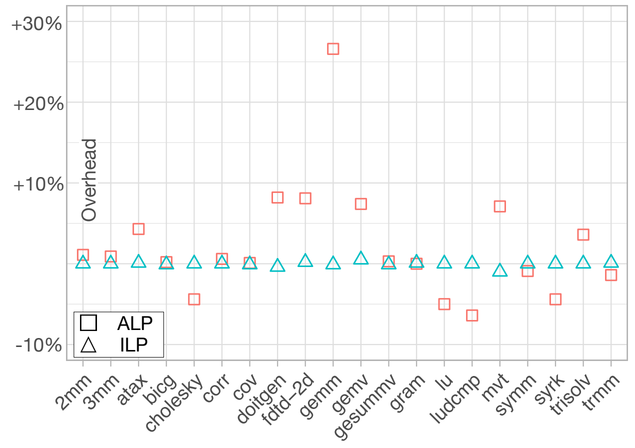


Figure 20. Overhead of profiling (both intra and inter loop). The Y-axis indicate percentages.

ring optimization and (iii) the time taken by the standard LLVM analyses and transformations necessary to enable ring 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 ring optimization that we use. Variation happens mostly between the approaches that use profiling (ALP and ILP) and those that do not (SSE 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.

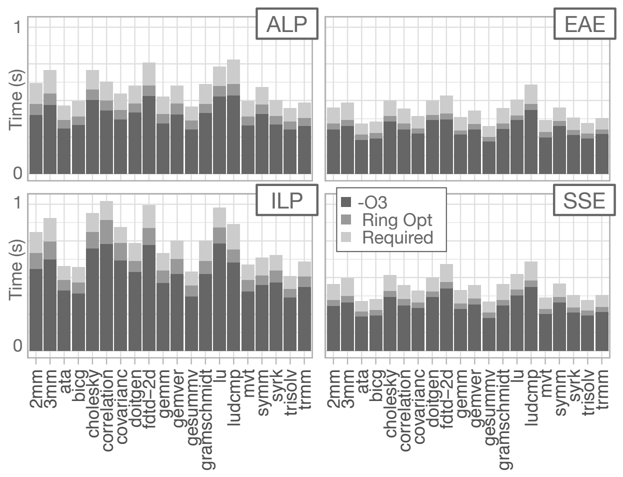


Figure 21. The overhead of Ring Optimization on compilation time over the baseline (ORG). Dark gray is the time to compile with -O3. In gray is the time for Ring Optimization and light gray the time for optimizations required by RO.

Inspection of Figure 21 reveals that ring optimization is practical. In absolute terms, clang -O3 takes approximately 0.30 seconds on average to compile the 20 programs in Figure 21. This number increases by 0.01, 0.01, 0.10 and 0.23

seconds when considering, respectively, SSE, EAE, ALP and ILP. The time taken by ring optimization itself is usually shorter than the time taken by its supporting optimizations.

5.5 RQ5: Comparison with a specialized tensor compiler (TACO)

Ring 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*) [7, 8]. Figure 22 compares TACO with clang (plus our Inter Loop Profiler). We emphasize that this figure relates two different compilers.

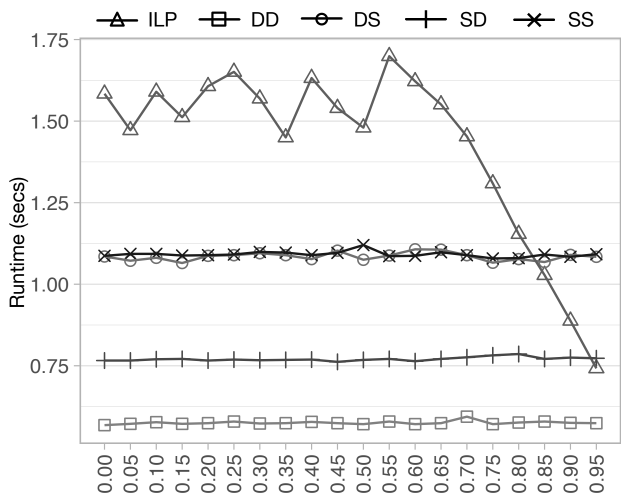


Figure 22. Comparison between TACO and clang plus Inter-Loop Profiling on matrix multiplication, considering different densities of zeros in the input matrices.D stands for *Dense* and S for *Sparse* data structures used to represent the matrices.

When restricted to matrix multiplication, TACO generates code that is almost three times faster than clang -O3 using a sparse representation for the two input matrices. When using a dense representation, this speedup is smaller: approximately 40%. Once we add the profiling-based version of ring optimization to clang, we can reduce this gap. The reduction factor is larger as the density of zeros increases. We notice in Figure 22 that once we achieve 50% of zeros in the sampling rate, the program flow is diverted to optimized code. Gradually, the dynamic elision of loads brings the speed of clang's binary closer to TACO's. Ring optimization can never beat TACO's best code, which uses two sparse matrices. However, this result is already expected: ring optimization is a general compiler optimization; TACO, in turn, uses different data structures to represent the matrices.

6 Related Work

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 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, 11, 12]. More recently, different research groups showed that it is possible to use profiling techniques to help developers to uncover and remove them [6, 18, 19]. Finally, in 2018 Pereira et al. [13] showed how to predict store operations that are likely to be silent statically.

The present paper 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 ring patterns that can lead to silent stores, the theoretical framework that we propose in this paper goes beyond that. As an example, Figure 1(b) shows an example of ring 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 [7, 8], Sparso [16] and TVM [2]. In Section 5 we showed that the optimizations that we introduce in this paper 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 ring optimizations as a competing approach. On the contrary, we believe that this technique could be used to enhance even further these specialized tensor compilers.

7 Conclusion

This paper has described a code optimization technique that avoids certain operations that, depending on the input values, are redundant. We call these operations *ring patterns*. These so called ring 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. For the sake of space, we have restricted our report to Polybench; however, 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 ring optimizations are a viable and effective way to improve the quality of the code generated by mainstream compilers.

A Proofs of Lemmas and Theorems

Theorem 3.5. If $\mathtt{pin}(b, z) = C$, then, for any $(s{=}{=}v) \in C$, and any store environment σ, we have that $\mathtt{eval}(b, \sigma \setminus \{s \mapsto v\}) = z$.

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

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

Corollary 3.6. Let $\mathtt{optimize}(\mathtt{st}\ a(\oplus(a, b))) = C$. If $(s{=}{=}v) \in C$, then $\mathtt{eval}(\oplus(a, b), \sigma \setminus \{s \mapsto v\} = a$, whenever $\sigma(s) = v$

We have that $\mathtt{optimize}(\mathtt{st}\ a(\oplus(a, b))) = \mathtt{pin}(b, z) = C$, where z is the absorbing element of $\oplus$. From Theorem 3.5, if $b{=}{=}z \in C$, then $\mathtt{eval}(b, \sigma \setminus \{s \mapsto z\}) = z$. Thus, $\mathtt{eval}(\oplus(a, b), \sigma \setminus \{s \mapsto v\}) = \mathtt{eval}(\oplus(a, b), \sigma \setminus \{s \mapsto v, b \mapsto z\}) = a$.

Theorem 4.1. If `t0` is only used at line 3 of Figure 10-i, then Fig. 10-i and Fig. 10-ii are equivalent. Otherwise, Fig. 10-i and Fig. 10-iii are equivalent.

We show equivalence from Fig 10-i and Fig 10-iii. The second part of the theorem follows from similar reasoning. If $\mathtt{t1} \neq z$, then both programs execute the same set of assignments. Otherwise, we have that $\mathtt{t2} = \mathtt{t0} \oplus z = \mathtt{t0} = \mathtt{a}$, and the store is silent.

References

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

[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. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *OSDI*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *POPL*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

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

[5] 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. 1993. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *J. Supercomput.* 7, 1-2 (May 1993), 229–248. https://doi.org/10.1007/BF01205185

[6] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. 2017. Detecting and Mitigating Data-dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*. ACM, New York, NY, USA, 27–40.

[7] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A Tool to Generate Tensor Algebra Kernels. In *ASE*. IEEE Press, Piscataway, NJ, USA, 943–948. http://dl.acm.org/citation.cfm?id=3155562.3155683

[8] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (2017), 29 pages. https://doi.org/10.1145/3133901

[9] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[10] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the Value Locality of Store Instructions. In *ISCA*. ACM, New York, NY, USA, 182–191. https://doi.org/10.1145/339647.339678

[11] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent Stores for Free. In *MICRO*. ACM, New York, NY, USA, 22–31.

[12] Kevin M. Lepak and Mikko H. Lipasti. 2002. Temporally Silent Stores. In *ASPLOS*. ACM, New York, NY, USA, 30–41.

[13] Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. 2018. Static Prediction of Silent Stores. *ACM Trans. Archit. Code Optim.* 15, 4, Article 44 (2018), 26 pages. https://doi.org/10.1145/3280848

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

[15] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-channel Detection. In *CC*. ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/2892208.2892230

[16] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *PACT*. ACM, New York, NY, USA, 247–259. https://doi.org/10.1145/2967938.2967943

[17] Mark Weiser. 1981. Program Slicing. In *ICSE*. IEEE, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[18] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *ASPLOS*. ACM, New York, NY,USA, 47–61.

[19] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *ASPLOS*. ACM, New York, NY, USA, 332–347.