Enabling loop optimizations through Loop Splitting

Augusto Noronha[1], **Luís Góes**[1], **Rodrigo Rocha**[2]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – Minas Gerais – Brazil

[2]School of Informatics
University of Edinburgh
Edinburgh – United Kingdom

augusto.noronha@sga.pucminas.br, lfwgoes@pucminas.br, r.rocha@ed.ac.uk

Abstract. *Modern optimizing compilers run many different optimization passes in order to generate faster or more memory efficient code. Out of those, loop optimizations are an important category. Many of the current loop optimizations require that the loop's body contain no branching instructions in order to work. In this paper we implement loop splitting for the LLVM compiler infrastructure, an optimization pass that enables many of the currently existing loop optimization passes to work. Loop splitting with constants is a well-known compiler optimization, however, we further it by implementing loop splitting between induction variables that represent linear equations. Our benchmarks show that, when applicable, loop splitting enables optimizations that would otherwise not be applied, and has significant gains over LLVM's current implementation. However, we found that the applicability for the optimization seems to be somewhat rare in real world applications.*

1. Introduction

A compiler is a program that transforms source code in some source language to machine code of a targeted architecture. A compiler's job is to produce correct machine code, according to the specifications of the source language. Another equally important role is optimizing the generated code as much as possible, producing machine code that is faster, smaller, or with less memory consumption.

It is a very common task in computing to evaluate some expression over multiple iterations, making the optimization of loop constructs a particularly important task. Many techniques have been developed in order to make loops faster, or to completely eliminate them. *Cooper et al.* [Cooper et al. 2001] described an *adhoc* implementation of operator strength reduction (also known as loop strength reduction), an optimization that replaces costly operations with cheaper ones. *Haghighat* introduces *symbolic differencing* as a means for identifying generalized induction variables in order to perform loop optimizations [Haghighat and Polychronopoulos 1996]. *Chains of recurrences* are introduced in [Van Engelen 2001] for the identification of generalized induction variables, and demonstrates that symbolic differencing can potentially generate incorrect code. Much work has been done on optimization passes using *chains of recurrences* as it's foundation [Berlin et al. 2004, Sui et al. 2016, Grosser et al. 2011, Sui et al. 2018].

The *Low Level Virtual Machine* (LLVM) [Lattner 2002], is a compiler infrastructure that supports the development of compilers of many source languages to many backends. Many of the currently implemented loop optimizations on LLVM depend on a loop structure without internal branching in order to work. As a result, many loops that could be potentially optimized, are not so. In this paper we develop a loop splitting implementation for LLVM. While loop splitting is already a known optimization in the compiler field in the case of splitting for constants, we further it by presenting loop splitting for induction variables that represent linear equations. This optimization enables many of the currently implemented loop optimizations to work. Furthermore, we show that loop splitting in and on itself in some cases speeds up the optimized program, when compared with other optimizations implemented in the compiler.

Section 2 introduces induction variables and the chains of recurrences algebra, existing loop optimizations implemented in LLVM that rely on it and some of LLVM's terminology for understanding loop optimizations. Section 3 delves into the workings of loop splitting. Section 4 shows our results. Section 5 lists the related work and section 6 contains our conclusion.

2. Background

2.1. Induction Variables

Basic induction variables (BIVs) are variables that changes by the same constant over every iteration of a loop. In listing 1, i and k are basic induction variables, as their value change is constant over multiple iterations.

```
int k = 0;
for (int i = 0; i < n; i++) {
    k = k + 3;
}
```

Listing 1. An example of basic induction variables

Generalized induction variable (GIV) are induction variables whose value changes are not constant, have multiple updates per loop, or are dependent on other induction variables. In listing 2, b is a BIV, a is a GIV (update value is not constant), and c is also a GIV (dependent on another induction variable).

```
int a = 1;
int b = 0;
int c = 1;
for (int i = 0; i < n; i++) {
    a = a * 2 + 3;
    b = b + 1;
    c = c + b;
}
```

Listing 2. An example of basic and generalized induction variables

BIVs can be described by the function $x(n) = x(n-1)+k$, where n is the iteration number and k is some constant. GIVs can be described by the function $x(n) = \phi(n)+ar^n$,

where n is the iteration number, ϕ represents a polynomial, r and a are loop invariant expressions.

Induction variables can also be classified as independent and dependent. Independent variables are those where the updates are not derived from other induction variables, whereas dependent variables are those that are. Basic induction variables are, by definition, always independent.

```
for (int i = 0; i < n; i ++) {
    a = a + 1;
    b = b + a;
    c = c * 2;
}
```

Listing 3. An example of independent and dependent induction variables

In listing 3, a is a basic independent induction variable, b is a generalized dependent induction variable, and c is a generalized independent one.

2.2. Chains of Recurrences

Chains of recurrences are a useful formalism for analyzing BIVs and GIVs and express them as recurrences. They were introduced as an effective method to evaluate functions at regular intervals. In this section we review its properties and notation.

It is a common task in computing to evaluate some closed-form function at various points. More specifically, given some function F, with some increment h, we would like to evaluate $F(x_0 + i * h)$, where $i = 0, 1, ..., n$. This function F can be written as the basic recurrence relation:

$$f(i) = \begin{cases} \iota_0, & \text{if } i = 0 \\ f(i-1) \odot g, & \text{if } i > 0 \end{cases} \tag{1}$$

where $\odot \in \{+, *\}$, g is a function defined over $\mathbb{N}$ and ι_0 is some constant. CRs commonly use an alternative representation for the same equation, as shown below:

$$f(i) = \{\iota_0, \odot, g\}_i \tag{2}$$

This notation can be evaluated as:

$$\begin{cases} f(0) = \iota_0 \\ f(1) = \iota_0 \odot g(0) \\ f(2) = \iota_0 \odot g(0) \odot g(1) \\ \vdots \\ f(i) = f(i-1) \odot g(i-1) \end{cases}$$

The CR notation is also capable of representing more complex functions. A closed-formed function f can be rewritten as a system of recurrence relations $f_0, f_1, f_2, ..., f_k$ where $f_j(i)$ for $j = 0, ..., k-1$ are linear recurrences of the form:

Table 1. Progression for $\phi = \{3, +, 5, *, 2\}$

	3	+	5	*	2
1.	8		10		2
2.	18		20		2
3.	38		40		2
4.	78		80		2
5.	158		160		2

$$f_j(i) = \begin{cases} \iota_0, & \text{if } i = 0 \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1), & \text{if } i > 0 \end{cases} \tag{3}$$

In this CR notation, the same function can be represented as:

$$\phi_i = \{\iota_0, \odot_1, \{\iota_1, \odot_2, \ldots, \{\iota_{k-1}, \odot_k, g\}_i\}_i\}i \tag{4}$$

which can be flattened to:

$$\phi_i = \{\iota_0, \odot_1, \iota_1, \odot_2, \ldots, \iota_{k-1}, \odot_k, g\}i \tag{5}$$

For $\phi = \{3, +, 5, *, 2\}$, Table 1 demonstrates it's values with $x = 1, 2, 3, 4, 5$.

CR's value as a means for optimization comes from it's rewrite rules, with which we can write an equivalent, but simpler CR. Table 2 lists CR simplification rules, given in [Birch 2002].

2.3. Optimizations based on CR

In this section, we explore some of the current optimization techniques enabled by CR.

2.3.1. Loop Strength Reduction

Loop strength reduction is an optimization pass that attempts to replace operations with a higher cost by those those semantically equivalent but with a lower cost. For example, this optimization might replace multiplication by accumulative additions. It is a common optimization pass implemented in multiple compilers, including LLVM [West 2011]. Listing 4 shows a function optimizable by this technique.

```
void f(int *a, int n) {
    int b = 1;
    for (int i = 0; i < n; i++) {
        a[i] = (i + 2) * 3 + b;
        b = b + 3;
    }
}
```

Listing 4. A function optimizable by loop strength reduction

Table 2. Chains of Recurrences simplifications rules

	Expression		Rewrite
1.	$\{\iota_0, +, 0\}_i$	$\Longrightarrow$	ι_0
2.	$\{\iota_0, *, 1\}_i$	$\Longrightarrow$	ι_0
3.	$\{\iota_0, *, 0\}_i$	$\Longrightarrow$	0
4.	$-\{\iota_0, +, f_1\}_i$	$\Longrightarrow$	$\{-\iota_0, +, -f_1\}_i$
5.	$-\{\iota_0, *, f_1\}_i$	$\Longrightarrow$	$\{-\iota_0, *, f_1\}_i$
6.	$\{\iota_0, +, f_1\}_i \pm E$	$\Longrightarrow$	$\{\iota_0 \pm E, +, f_1\}_i$
7.	$\{\iota_0, *, f_1\}_i \pm E$	$\Longrightarrow$	$\{\iota_0 \pm E, +, f_1 * (f_1 - 1), *, f_1\}_i$
8.	$E * \{\iota_0, +, f_1\}_i$	$\Longrightarrow$	$\{E * \iota_0, +, E * f_1\}_i$
9.	$E * \{\iota_0, *, f_1\}_i$	$\Longrightarrow$	$\{E * \iota_0, *, f_1\}_i$
10.	$E/\{\iota_0, +, f_1\}_i$	$\Longrightarrow$	$1/\{\iota_0/E, +, f_1/E\}_i$
11.	$E/\{\iota_0, *, f_1\}_i$	$\Longrightarrow$	$\{E/\iota_0, *, 1/f_1\}_i$
12.	$\{\iota_0, +, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Longrightarrow$	$\{\iota_0 \pm \psi_0, +, f_1 \pm g_1\}_i$
13.	$\{\iota_0, *, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Longrightarrow$	$\{\iota_0 \pm \psi_0, +, \{\iota_0 * (f_1 - 1), *, f_1\}_i \pm g_1\}_i$
14.	$\{\iota_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Longrightarrow$	$\{\iota_0 * \psi_0, +, \{\iota_0, +, f_1\}_1 * g_1 + \{\psi_i, +, g_1\}_i * f_1 + f_1 * g_1\}_i$
15.	$\{\iota_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Longrightarrow$	$\{\iota_0 \pm \psi_0, *, f_1 * g_1\}_i$
16.	$\{\iota_0, *, f_1\}^E$	$\Longrightarrow$	$\{\iota_0^E, *, f_1^E\}_i$
17.	$\{\iota_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Longrightarrow$	$\{\iota_0^{\psi_0}, *, \{\iota_0, *, f_1\}_i^{g_1} * f_1^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$
18.	$E^{\{\psi_0, +, f_1\}_i}$	$\Longrightarrow$	$\{E^{\iota_0}, *, E^{f_1}\}_i$
19.	$\{\iota_0, +, f_1\}_i^n$	$\Longrightarrow$	$\begin{cases} \{\iota_0, +, f_1\}_i * \{\iota_0, +, f_1\}_i^{n-1} & \text{if } n \in, n > 1 \\ \{1/\iota_0, +, f_1\}_i^{-n} & \text{if } n \in, n < 0 \end{cases}$
20.	$\{\iota_0, +, f_1\}_i!$	$\Longrightarrow$	$\begin{cases} \{\iota_0!, *, f_1\}_i, *, (\prod_{j=1}^{f_1} \{\iota_0 + j, +, f_1\}_i)\}_i & \text{if } f_1 \geq 0 \\ \{\iota_0!, *, f_1\}_i, *, (\prod_{j=1}^{\lvert f_1 \rvert} \{\iota_0 + j, +, f_1\}_i)^{-1}\}_i & \text{if } f_1 < 0 \end{cases}$
21.	$\{\iota_0, +, \iota_1, *, f_1\}_i$	$\Longrightarrow$	$\{\iota_0, *, f_2\}_i$ when $\iota_0/\iota_1 = f_1 - 1$

As a first step, we would like to derive the equivalent CR for the loop. In this case, we start by i's CR: $\{0, +, 1\}$ and b's CR: $\{1, +, 3\}$. By using CR's algebraic rules we can derive:

$$
\begin{gathered}
((\{0, +, 1\} + 2) * 3) + \{1, +, 3\} \\
(\{2, +, 1\} * 3) + \{1, +, 3\} \\
\{6, +, 3\} + \{1, +, 3\} \\
\{7, +, 6\}
\end{gathered}
$$

After computing the CR, we can rewrite the function as written in listing 5:

```
int f(int *a, int n) {
    int a[n];
```

```
    int x = 7;
    for (int i = 0; i < n; i++) {
        a[i] = x;
        x = x + 6;
    }
}
```

Listing 5. A function after being optimized by loop strength reduction

By applying this optimization, we have reduced the number of operations from 4 additions and 1 multiplications to 2 additions.

2.3.2. Induction Variable Elimination

Induction variable elimination are optimizations that attempt to remove induction variables [Aho et al. 1986]. In some cases, they are able to solve loops at compile time, replacing it for a closed-formula.

```
int f(int n) {
    int s = 0;
    int j = 3;
    for (int i = 0; i < n; i++) {
        j += 2;
        s += j;
    }
    return s;
}
```

Listing 6. A function optimizable by induction variable elimination

In listing 6, we can derive the following CRs: $i = \{0, +, 1\}$, $j = \{3, +, 2\}$, $s = \{0, +, 3, +, 2\}$. By analyzing these CRs, we can see that i and j are redundant, either one can be re-written as the other. We could thus re-write this function as:

```
int f(int n) {
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += 3 + (i + 1) * 2;
    }
    return s;
}
```

Listing 7. A function after being optimized by induction variable elimination

2.3.3. Auto-vectorization

Auto-vectorization works by converting the operations inside the loop from a scalar to a vectorized implementation, doing multiple iteration's worth of work at a time [Porpodas et al. 2018]. This optimization requires hardware that supports SIMD operations. As they are not expressible in pure C code, we show no examples of this optimization.

2.4. LLVM

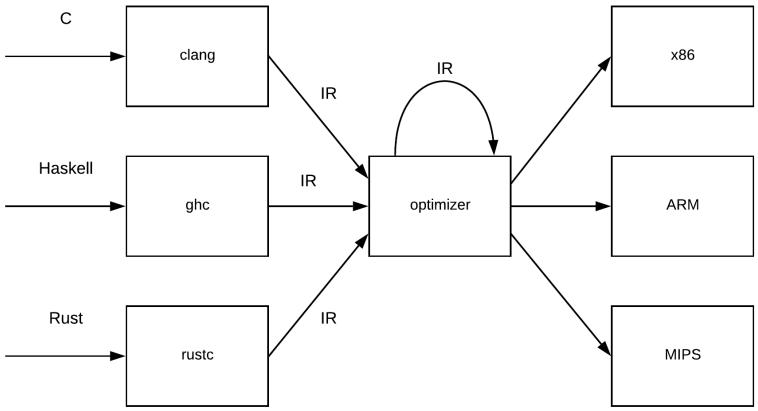

Figure 1. The LLVM compilation architecture

LLVM is a compiler infrastructure that supports compilation from many source languages to many architectures. LLVM's architecture separates the compilation process in three major phases: the front-end, optimization passes and machine code generation. The front-end compiles the source language (eg. *C, Fortran, Swift, Haskell, Rust*) down to a common intermediate representation (simply called *IR*). The optimization passes, which are applied exclusively to the IR, are run one after the other, each taking as input the latter's output, the passes applied depend on the optimization flag passed to the front-end (e.g.: *clang* offers *O0, O1, O2, O3* among other optimization flags, each applying a different set of optimization passes to the generated code). The machine code generation translates the IR to some target architecture (eg. *x86, PowerPC, ARM*).

2.5. LLVM terminology

A *basic block* (or simply *block*) is a series of non-branch instructions, ended by a branch one. LLVM represents programs as *control flow graphs*, which is a directed graph where a block is a node and a branching from one block to another is an edge. A block b_1 is said to *dominate* b_2 if all paths to b_2 must pass through b_1.

A loop is a maximal set of strongly connected blocks where there exists a single block which dominates all others within the loop, this block is called the *header*. The *preheader* is the singular loop predecessor which contains a single edge to the loop header. An *exiting block* is a block that contains at least one edge for a block outside the loop and one for a block inside the loop. Loops may contains any number of exiting blocks. An *exit block* is a block outside the loop that has a predecessor inside the loop (an exiting block). A *latch* is a block that contains an edge to the loop header, this edge is called *backedge*. Loops may either be completely disjoint or a loop may contains sub-loops.

3. Methodology

Given a loop L than contains in its body at least one branch instruction unrelated to the loop control, loop splitting works by first selecting one of these branches and then replacing L by two other loops, the first over the iterations where the selected branching condition is always true and the second over the iterations where the branching condition

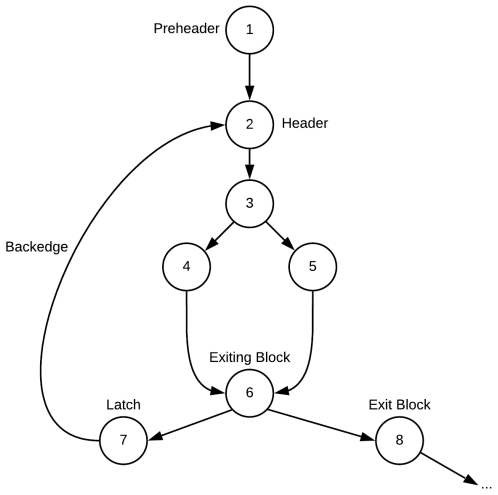

Figure 2. A natural loop example

is always false, or vice-versa. For the rest of this section, we define c as the comparison instruction used in the branch inside the loop, and c_{op}, its comparison operator. The loop's *intersection iteration*, further denoted as x, is the loop's iteration where c changes from false to true or vice-versa. In order for the optimization to work, it requires that one of the arguments for c be a CR that represents a linear equation (which we will furthermore call as *linear CR*) and the other is either a constant value or also a linear CR. This means that c will change values at most once. It also requires that $c_{op} \in \{<, <=, >, >=\}$ and that the linear CR values be integers.

3.1. Intersection iteration calculation

Given some constant p and a linear CR that can be written as $y = ax + b$ which are compared in c, we can find the intersection iteration by substituting y for p, and solving the equation for x.

$$p = ax + b$$
$$x = \left\lfloor \frac{p - b}{a} \right\rfloor$$

Given a function with a comparison between two linear CRs, two possibilities arise:

1. One of the CR's loop is a sub-loop of the other
2. Both CRs belong to the same loop

For the first case, we execute the constant and linear CR loop splitting algorithm, substituting as the constant the value of the outer loop's CR at its current iteration. For the second one, we have two linear CRs, $i = \{b_1, +, a_1\}$ and $j = \{b_2, +, a_2\}$, we can calculate their intersection iteration by using the two line intersection formula and solving for x.

$$a_1 x + b_2 = a_2 x + b_2$$
$$x = \left\lfloor \frac{b_2 - b_1}{a_2 - a_1} \right\rfloor$$

In both cases we use the *floor* function since a loop iteration count must be an integer. The information for calculating the intersection iteration may not be available until run-time (for example, some value may be provided as a function parameter), if that is the case, we insert for the calculation at run-time in the loop's preheader, otherwise, we do the whole calculation at compile time.

3.2. Splitting the loop

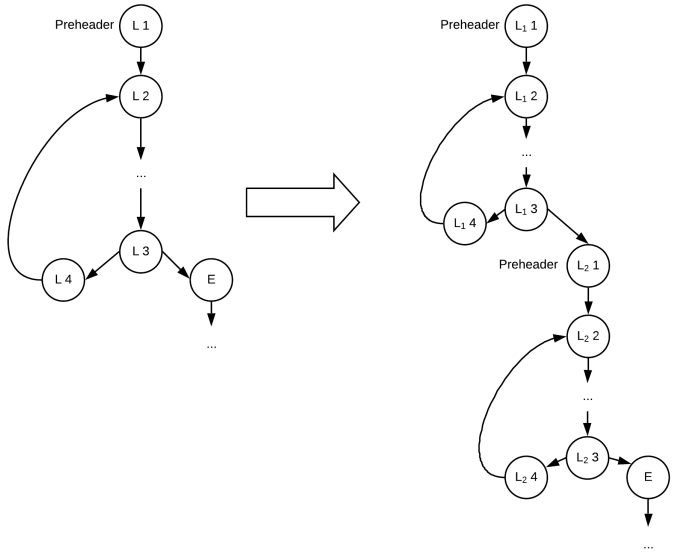

Figure 3. A cloned loop example

We now find out if c is true before or after x. If comparing a linear CR cr and a constant we evaluate cr at iteration $x + 1$, and substitute cr's induction value by the evaluated value in comparison c. If comparing two linear CRs from the same loop, we evaluate both values at iteration $x + 1$, and replace their respective induction variables in comparison c.

We now clone L and place the two new loops, L_1 and L_2, adjacent to each other in place of L. We remap L_1's exiting block such that it now branches to L_2's preheader block. Figure 3 shows the cloned loop structure after the split. We then negate the cloned c in L_1. We set L_1's bound as either its value on x, or the next iteration - denoted as $x + 1$, based on c_{op} and if c is true or false at the intersection iteration, denoted respectively as $true$ and $false$ in table 3. For the case of a constant comparison or different loop CRs we rewrite the inequality so that the induction variable is in the right hand side.

3.3. Multiple branchings

A loop may contain more than one split point. A loop may also contain sub-loops which are themselves splittable in multiple points. In order to split the loop in all its possible

Table 3. L_1**'s** lc **operand value**

	$<$	$<=$	$>$	$>=$
$true$	x	x	$x+1$	$x+1$
$false$	$x+1$	$x+1$	x	x

points, we run the algorithm as described in 1. The algorithm is run in multiple passes so any split points left in cloned loops are split themselves. The current loop splitting algorithm is greedy: it splits any and all loops which are suitable for doing so. This means that for a function F with loops $L_1, L_2, ..., L_n$ with, respectively $s_1, s_2, ..., s_n$ split points, the algorithm with replace the loops with $2^{s_1} + 2^{s_2} + ... + 2^{s_n}$ loops. Future work may be done in order to analyze if a loop should be split or not.

Algorithm 1 The loop splitting general algorithm

```
function LoopSplitting(F: Function)
    while True do
        let S be all the splitting points for all loops in F
        if S = ∅ then
            return F
            for each split point s ∈ S do
                update F splitting in s
```

3.3.1. Logical Operators

The loop splitting algorithm also works for logical operators *and* and *or* without any special treatment. That is possible because LLVM's IR treats branchings from logical operators as separate independent branchings from each other. The code in listing 8 is more similar in LLVM's IR as written in listing 9.

```
int f(int n) {
    int s = 0;
    for (int i = 0; i < n; i ++) {
        if (10 < i || i > 30) {
            s += i;
        }
    }
    return s;
}
```

Listing 8. A function with logical operators

```
int f(int n) {
    int s = 0;
    for (int i = 0; i < n; i ++) {
```

```
        if (10 < i) {
          goto increment;
        }
        if (i > 30) {
          goto increment;
        }
        goto end;
        increment: s += i;
        end: ;
    }
    return s;
}
```

Listing 9. Re-write of function with logical operators

3.4. Examples

We now show examples of loop splitting for the three different cases described above. It is emphasized that this is not how the functions are actually re-written as the optimization works on LLVM's IR level, and the examples show how C code could be rewritten given the explanation presented in subsections 3.1 and 3.2.

3.4.1. Constant and linear CR

```
int f(int n) {
    int s = 0;
    for (int i = 4; i < n; i += 2) {
        if (10 < i) {
            s += i;
        }
    }
    return s;
}
```

Listing 10. A function with constant and linear CR comparison

Given the function in listing 10, we calculate the intersection iteration of the comparison $10 < i$, where i is a linear CR $\{4, +, 2\}$. We solve for x in:

$$10 = 2x + 4$$
$$x = \frac{10 - 4}{2}$$
$$x = 3$$

Since c_{op} is $<$, L_1's induction variable limit will be set for its value at iteration $x + 1 = 4$:

$$y = 2 * 4 + 4$$
$$y = 12$$

We now re-write the function as seen in listing 11.

```
int f(int n) {
    int s = 0;
    int i = 4;
    for (; i < 12; i += 2) {
        if (!(10 < 12)) {
            s += i;
        }
    }
    for (; i < n; i += 2) {
        if (10 < 12) {
            s += i;
        }
    }
    return s;
}
```

Listing 11. A function with constant and linear CR comparison after loop splitting

With the loop in this format, LLVM's *constant folding* and *loop deletion* passes remove the first loop entirely, propagating i's intial value to 48, as well as removing the second loop's always true comparison. Transforming the code as seen in listing 12, before further optimizations.

```
int f(int n) {
    int s = 0;
    for (int i = 48; i < n; i += 2) {
        s += i;
    }
    return s;
}
```

Listing 12. A function with constant and linear CR comparison after loop splitting, *constant folding* and *loop deletion* passes

3.4.2. Different loop linear CRs

Given the function:

```
void function(int n, int* s) {
    for (int i = 0; i < n; i += 3) {
        for (int j = 0; j < n; j++) {
            if (j < i) {
                s[i] += i + j;
            }
        }
    }
}
```

Listing 13. A function with different loop linear CRs comparison

The function in listing 13 has two different CRs. i's CR is associated with the outer loop, and j's the inner loop. We use the constant and linear CR algorithm, using i's current value for calculating the intersection point. Since the outer CR's induction variable changes every loop, we need to insert code in the loop's preheader to calculate x at run-time. We thus rewrite the loop as seen in listing 14.

```
void function(int n, int* s) {
    int s = 0;

    for (int i = 2; i < n; i += 3) {
        int x = (i - 30) / 4;
        int value = x * 4 + 30;
        int value_after = (x + 1) * 4 + 30;
        int true_at_point = i < value;
        int true_after = i < value_after;
        int split_point = true_at_point ? value : value_after;
        int j = 30;
        for (; j < split_point; j += 4) {
            if (!true_after) {
                s[i] += i + j;
            }
        }
        for (; j< n; j += 4) {
            if (true_after) {
                s[i] += i + j;
            }
        }
    }
    return s;
}
```

Listing 14. A function with different loop linear CRs comparison after loop splitting

3.4.3. Same loop linear CRs

```
int f(int n) {
    int s = 0;
    int a = 30;
    for (int i = 0; i < n; i += 3) {
        if (a < i) {
            s += i;
        }
        a++;
    }
    return s;
}
```

Listing 15. A function with same loop linear CRs comparison

The loop in listing 15 has CRs $i = \{0, +, 3\}$ and $a = \{30, +, 1\}$. We calculate x:

$$3x + 0 = 1x + 30$$
$$x = \frac{30}{2}$$
$$x = 15$$

Since they both refer to the same loop and all information is available at compile time, we calculate both variable's value at iteration $x+1$, which are equal to $16*3+0 = 48$ and $16 * 1 + 30 = 46$. The former of those is also L_1's upper bound, according to the explanation given in subsection 3.1. We now rewrite the loop as seen in listing 16.

```
int f(int n) {
    int s = 0;
    int a = 30;
    int i = 0;
    for (; i < 48; i += 3) {
        if (!(46 < 48)) {
            s += i;
        }
        a++;
    }
    for (; i < n; i += 3) {
        if (46 < 48) {
            s += i;
        }
        a++;
    }

    return s;
}
```

Listing 16. A function with same loop linear CRs comparison after loop splitting

4. Results

For testing our results, we use the *clang* compiler version 9.0.0, the C front-end for LLVM. We compare 4 different optimization strategies: *unoptimized*, *loop split* (which consists of loop splitting followed by the loop deletion and simplify control flow graph passes present in LLVM, which are used to delete the dead code generated by the loop splitting pass), *optimized* (which is equivalent to clang's *-O3* flag) and *loop split then optimized*, which applies loop splitting, followed by all optimizations in the *-O3* flag. All the benchmarks are run 20 times and results are averaged. The tests are done in a Macbook Pro Late 2013, with processor Intel(R) Core(TM) i5-4258U CPU @ 2.40GHz, 8GB of DDR3 RAM with frequency of 1600 MHz and operating system macOS Mojave 10.14.6. We tested our optimization in a number of different benchmarks. Unfortunately, the vast majority of them did not contain loops that presented the correct structure to be optimized by our pass (i.e. branchings with comparison between linear CRs or linear CR and constant inside loops). We therefore evaluate 2 synthetic functions and 1 function from an existing benchmark:

1. Code in listing 13.
2. Code in listing 15.
3. Function s276 from the TSVC benchmark [Maleki et al. 2011], given in listing 17.

```
int s276() {
    int mid = (LEN/2);
    for (int nl = 0; nl < 4*200000; nl++) {
        for (int i = 0; i < LEN; i++) {
            if (i+1 < mid) {
                a[i] += b[i] * c[i];
            } else {
                a[i] += b[i] * d[i];
            }
        }
        dummy(a, b, c, d, e, aa, bb, cc, 0.);
    }
    return 0;
}
```

Listing 17. Function s276 extracted from the TSVC benchmark, that shows an example where loop splitting enables vectorization

In figures 4 and 5 we see that the function in listing 15 runs in constant time for the loop splitting and optimized compilation, no matter the input size, while the 3 other optimization strategies run in linear time. This happens because after loop splitting, LLVM's *indvars* pass is able to find a closed formula for the loop. This changes the complexity of the function from $O(n)$ to $O(1)$. We also see that the loop split and simplified version has a lower constant factor when compared with the unoptimized version.

The function in listing 13 runs in quadratic time, with the running time for all tests being very similar, except for the optimized loop split version, which runs in linear time, with a much lower constant factor as shown in figures 6 and 7.

Finally, function s276 present a linear running time for all tests, the loop splitting version does beat the other versions by it's constant factor, as seen in figure 8. It presents a speedup of 3.16 over the optimized version. This happens because after the removal of control flow from inside both loops, LLVM is able to auto-vectorize the loop. Also of note, is the fact that the simplified loop split version, without further optimizations, presents a speedup of 1.19 over the optimized version. This happens because splitting removes the control flow overhead from inside the loop.

5. Related Work

The study on how to treat loops with conditionals is not new. *Bacon et al.* [Bacon et al. 1994] describe a great deal of loop optimizations, including loop splitting for constant values. *Falk et al.* [Falk and Marwedel 2003] present loop nest splitting, which minimizes the number of executed if-statements in loop nests of embedded multimedia applications. Their loop splitting algorithm is focused in enabling paralellization and improving I-cache performance due to smaller loop bodies. *Quillete et al.* [Quilleré et al. 2000] introduce

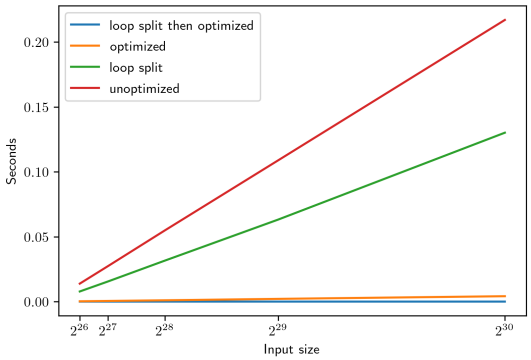

Figure 4. Running time for function in listing 15

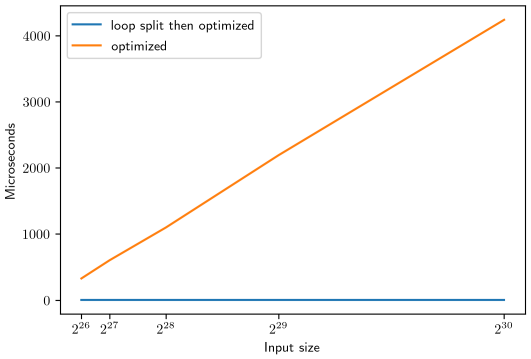

Figure 5. Running time for function in listing 15 optimized only

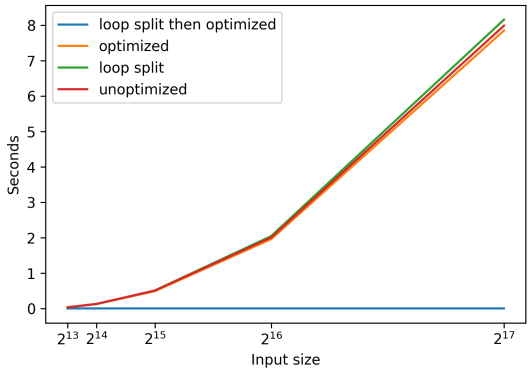

Figure 6. Running time for function in listing 13

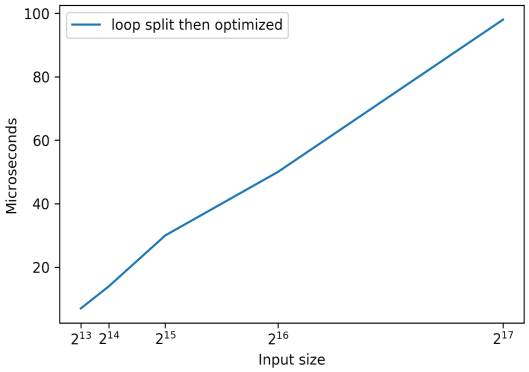

Figure 7. Running time for function in listing 13 optimized only

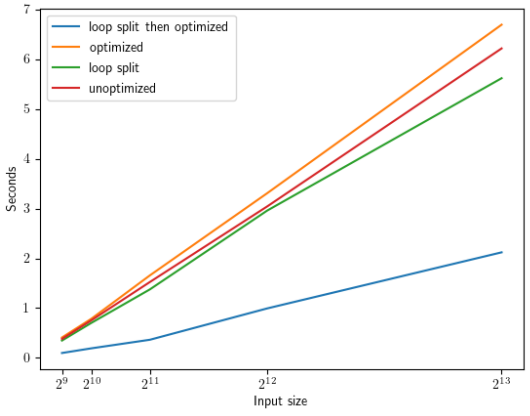

Figure 8. Running time for function s276.

generation of efficient nested loops from polyhedra. Polyhedra is a formalism for reasoning about parallel computations using systems of affine recurrence equations defined over polyhedral shapes. Their optimization focuses in rewriting nesting loop's bounds and steps in order to generate loops better suitable to parallelization. *Tanguay* [Tanguay 1993] describes loop splitting for distributed memory multiprocessors. His work involves splitting loops in multiple sub-loops so that the data needed by each processor executing a different sub-loop is local to it. *Liu et al.* [Liu et al. 2016] describe loop splitting for loops that contain either uncertain or non-uniforms data dependencies, in order to make them more pipeline-friendly. Although loop splitting is common in the case of comparison with constant values, this is the first work that is able to deal with comparisons between variables representing first degree equations, and more complex logical expressions.

6. Conclusion

Modern optimizing compilers work by applying a series of optimization passes in sequence. We show that loop splitting enables other optimization passes to operate on loops, when the original loop would not be optimized as well. We also show that loop splitting in and on itself presents a speedup over the original loop in some cases. Unfortunately we found that the prerequisites for loop splitting are rare in benchmarks, and can be therefore presumed to be rare in real applications as well. For future work, we plan to develop loop splitting for different polynomial functions, as well as to develop a cost analysis for loop splitting, since the current implementation can potentially increase the code size exponentially.

References

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). Compilers, principles, techniques. *Addison wesley*, 7(8):9.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420.

Berlin, D., Edelsohn, D., and Pop, S. (2004). High-level loop optimizations for gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54. Citeseer.

Birch, J. L. (2002). Using the chains of recurrences algebra for data dependence testing and induction variable substitution.

Cooper, K. D., Simpson, L. T., and Vick, C. A. (2001). Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):603–625.

Falk, H. and Marwedel, P. (2003). Control flow driven splitting of loop nests at the source code level. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 10410. IEEE Computer Society.

Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., and Pouchet, L.-N. (2011). Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1.

Haghighat, M. R. and Polychronopoulos, C. D. (1996). Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):477–518.

Lattner, C. A. (2002). *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign.

Liu, J., Wickerson, J., and Constantinides, G. A. (2016). Loop splitting for efficient pipelining in high-level synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 72–79. IEEE.

Maleki, S., Gao, Y., Garzar, M. J., Wong, T., Padua, D. A., et al. (2011). An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382. IEEE.

Porpodas, V., Rocha, R. C., and Góes, L. F. (2018). Vw-slp: auto-vectorization with adaptive vector width. In *PACT*, pages 12–1.

Quilleré, F., Rajopadhye, S., and Wilde, D. (2000). Generation of efficient nested loops from polyhedra. *International journal of parallel programming*, 28(5):469–498.

Sui, Y., Fan, X., Zhou, H., and Xue, J. (2016). Loop-oriented array-and field-sensitive pointer analysis for automatic simd vectorization. In *ACM SIGPLAN Notices*, volume 51, pages 41–51. ACM.

Sui, Y., Fan, X., Zhou, H., and Xue, J. (2018). Loop-oriented pointer analysis for automatic simd vectorization. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):56.

Tanguay, D. O. (1993). Compile-time loop splitting for distributed memory multiprocessors. B.S. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

Van Engelen, R. A. (2001). Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer.

West, B. N. (2011). Adding operator strength reduction to llvm. Technical report.