Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Schleifenausrollen mit nicht konstanten Grenzen in FIRM

Bachelorarbeit von

Adrian E. Lehmann

an der Fakultät für Informatik

Erstgutachter: Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter: Prof. Dr. rer. net. Bernhard Beckert
Betreuende Mitarbeiter: M. Sc. Andreas Fried

Abgabedatum: 30. September 2019

Zusammenfassung

Das Schleifenausrollen ist der Prozess, bei dem der Körper einer Schleife mehrfach dupliziert wird, um die Anzahl der bedingten Sprünge zu reduzieren. Während Schleifen mit konstanten Grenzen leicht ausrollbar sind, erweist sich das Ausrollen von Schleifen mit nicht konstanten Grenzen als eine Herausforderung.

In diesem Fall ist es möglich, spekulativ eine Schleife um einen bestimmten Faktor auszurollen, sofern sichergestellt wird, dass die ausgerollte Schleife so oft wie möglich ausgeführt wird, aber nicht öfter als die ursprüngliche Schleife. Um die daraus resultierende Diskrepanz auszugleichen, wird sogenannter Fixup Code erstellt, der für die Ausführung der restlichen Iterationen verantwortlich ist. Um den oben genannten Fixup Code zu erstellen werden zwei Ansätze verfolgt: Zum einen kann die Originalschleife dupliziert werden und zum anderen, eine generalisierte Form von Duff's Device verwendet wird.

Bei der experimentellen Bewertung des Ansatzes, obwohl die Anzahl der ausrollbaren Schleifen verdoppelt wurde (auf 10% aller Schleifen), können wir keine Beschleunigung außerhalb des Messfehlers feststellen. Die Ergebnisse deuten darauf hin, dass das Schleifenausrollen ohne weitere Optimierung keinen (signifikanten) Nutzen zu haben scheint.

Abstract

Loop unrolling is the process of duplicating a loop's body multiple times to reduce the number of conditional jumps. While we can easily unroll loops with constant bounds, unrolling loops with non-constant bounds proves to be a different challenge.

In this case we can speculatively unroll a loop by a given factor, while making sure that the unrolled loop runs as often as possible, but less than or equal times to the original loop. To compensate for the resulting discrepancy, we create so-called fixup code, which is responsible for running the remaining iterations. We choose two different strategies for creating the aforementioned fixup code: One where we duplicate the original loop and another where we utilize a generalized form of Duff's device.

When experimentally evaluating the approach, even though we double the amount of unrollable loops to now unroll more than 10% of all loops, we cannot note any speed-up outside of the margin of error. The results hence indicate that loop unrolling does not seem to have a (significant) benefit without further optimization.

Contents

1 Introduction

When developers craft code, there is a need to convert it from a human-readable high-level language into a machine-understandable language, called assembly. In order to do this programmers run a *compiler* that checks the code for multiple sources of errors, and, if the code is correct, converts it into an executable file. While converting the program into machine code the compiler optimizes the code. This is only beneficial to the developer, as it ensures that his/her application, in the end, runs faster and/or requires fewer system resources. A simple example of an optimziation is constant folding, where the compiler analyzes code and precalculates all constant values, instead of letting the operations on them waste valuable runtime to calculate each time the application runs. An example can be seen below: Humans immediately see that in figure 1.1a b is always equal to 9 and using constant folding the compiler will be able to also perform this precomputation. The result of this can be seen in figure 1.1b. Therefore, the optimization reduces the runtime of the (admittedly small) program, by virtue of there being one less calculation required. Simplifying just one expression seems (and for a matter of fact is) quite useless, but in real-world code, an optimization like this can be applied on numerous similar expressions and hence noticeably improves the final product.

$$a \leftarrow 7$$
$$b \leftarrow a + 2$$

(a) Sample code snippet for constant folding

$$a \leftarrow 7$$
$$b \leftarrow 9$$

(b) Code with constants folded

Figure 1.1: Example of constant folding optimization

Of course there is a plethora of possibilities to optimize code. Considering that loops make up approximately 10% of code of many real-world applications[1], they are a natural point to focus optimization efforts upon. Loops can be unrolled fairly straight forward if you know how often they are iterated, as is discussed in chapter 2.

For example, figure 1.2a can be easily converted to figure 1.2b, while keeping all semantics intact. In figure 1.3 things get trickier, since the (exact) value of N is unknown, simply unrolling a loop by copying its body a fixed number of times does not preserve the original semantics.

In chapter 2 fundamentals for working with these loops are discussed, which are enhanced, juxtaposed and integrated in chapter 3. Finally, in chapter 4 we evaluate the approach experimentally to see whether it yields a tangible benefit.

[1]Measured using gcc (spec2006): 8.6% of FIRM nodes are in loops

$i \leftarrow 0$
while $i < 5$ **do**
  PRINT(i)
  $i \leftarrow i + 1$
end while

(a) Loop with constant bounds

PRINT(0)
PRINT(1)
PRINT(2)
PRINT(3)
PRINT(4)

(b) Loop with constant bounds unrolled

Figure 1.2: Unrolling a loop with constant bounds

$i \leftarrow 0$
$N \leftarrow$ FAIRDICEROLL() ▷ Random number in $[1, 6]$
while $i < N$ **do**
  PRINT(i)
  $i \leftarrow i + 1$
end while

Figure 1.3: Loop without constant bound

2 Basics and related work

2.1 Compiler

The primary function of a compiler is to automatically convert high-level code created by a developer into (optimized) machine code. As a compiler is an inherently large software project, an architecture needs to be chosen that allows for extensions and modifications. Modern compilers mostly follow a layered architecture style: They are each comprised of a front-, middle-, and back-end. In this architecture, the front-end converts the high-level code into an abstract intermediary representation, which is then used by the middle-end for optimizations and transformations. Lastly, the back-end is responsible for converting the optimized intermediary code into instructions for the target system architecture (e.g., RISC-V, x86, ARM, or similar).

2.2 Basic blocks and control-flow

To better handle code and give it a logical structure, most compilers divide code up into so-called *basic blocks*. Basic blocks are sets of consecutive operations that do not contain jumps or targets thereof, but rather only jumps connecting them. Therefore, a basic block is either executed entirely or not executed at all.

A usual way to represent this in a human-readable form is to output it as a control-flow-graph (CFG). CFGs depict basic blocks as nodes and jumps between basic blocks as edges. Furthermore, it is a convention in these graphs to have precisely one start-node and one end-node.

We note that CFGs, in general, are cyclical graphs. They are non-cyclical graphs if the original code does not contain any jumps going backward in the control-flow.

Another important concept of CFGs is dominance. To explain this concept, we define a starting node S, and assume there are two (not necessarily different) nodes, present in the CFG, N_1, and N_2. With this information, we define dominance as follows:

$$N_1 \text{ dominates } N_2 \iff \forall p \in \mathrm{Paths}(S, N_2): \; N_1 \in p$$

In lucid terms, this means N_1 dominates N_2, iff to get to N_2 from S you have to visit N_1 on the way. It is important to note that a block always dominates itself.

2.3 Loops

We define a loop to be a set of nodes that are all in a cyclical control-flow structure. Formally this can be expressed as:

$$L \text{ is a loop} \iff L \neq \emptyset \wedge \forall n_1, n_2 \in L : \ \exists Path(n_1, n_2) \subseteq L$$

Henceforth let L be a loop.

If a loop is completely contained inside another loop, it is said to be nested.

$$L' \text{ is nested in } L \iff L' \subsetneq L \wedge L' \text{ is a loop}$$

If a loop has no nested loops inside of it, we will henceforth call it the *innermost* loop.

$$L \text{ is innermost loop} \iff \nexists L' : L' \text{ is a loop nested loop in } L$$

Loops can furthermore have a header, which is the sole entry point into a loop [1] and defined as follows:

$$H \text{ is header of } L \iff H \in L \wedge \forall n \in L : H \text{ dominates } n$$

N.B.: Not all loops have to have a header.

If a loop has a header, its body is the set containing all blocks in the loop, except for the header.

$$B \text{ is body of } L \iff B = L \backslash \{H\}, H \text{ is header of } L$$

2.4 Single-Static-Assignment (SSA)

The *single-static-assignment* (*SSA*) form is a property of intermediary representations, that requires each value to only be assigned exactly once. Moreover, every value has to be assigned before it is being used [2]. This property mainly implies that the block, which declares a given value v, has to dominate all blocks that use v. This declaration point will be unambiguous across all possible usages.

The SSA form is used to simplify optimizations in the regard that one can be sure that a set point in the code currently defines a given value in use.

Figure 2.1 shows an example program in SSA form. While in the base code x is assigned twice, in the code transformed into SSA form, simply a new value was defined to make sure that each variable is only defined once.

In a loop or a conditional statement, a scenario might arise where a given value could be assigned at multiple locations. In cases like these we can use a Φ-*function*. A Φ-function is a theoretical construct that returns the correct value depending on the control-flow predecessor.

Figure 2.1 shows an example of the use of a Φ-function, where depending on the control-flow, either m_1 or m_2 are selected.

$x \leftarrow 1$
PRINT(x)
$x \leftarrow 7$
PRINT(x)

(a) An example piece of code not in SSA form

$x_1 \leftarrow 1$
PRINT(x_1)
$x_2 \leftarrow 7$
PRINT(x_2)

(b) Thre same code in SSA form

Figure 2.1: An example program in SSA form

function MAX($a : \mathbb{N}, b : \mathbb{N}$)
    $m : \mathbb{N}$
    if a > b **then**
        $m \leftarrow a$
    else
        $m \leftarrow b$
    end if
    return m
end function

(a) Non-SSA Code for a function that returns the maximum of its parameters

function MAX($a : \mathbb{N}, b : \mathbb{N}$)
    $m : \mathbb{N}$
    if $a > b$ **then**
        $m_1 \leftarrow a$
    else
        $m_2 \leftarrow b$
    end if
    $m \leftarrow \Phi(m_1, m_2)$
    return m
end function

(b) Same Function converted into SSA

Figure 2.2: An example program transformed into SSA form

2.5 Loop-Closed-Single-Static-Assignment (LCSSA)

An extension to the SSA form is the *loop-closed-single-static-assignment* (*LCSSA*) form. A CFG in LCSSA form has all properties that a CFG in SSA form has, and additionally the property that each value assigned in a given loop and used outside of this loop has to be used by a Φ-node in the first block after the loop. This form is used to reduce special casing when transforming loops [1], and is therefore used through all of chapter 3.

To visualize this property, figure 2.3 depicts its effect.

2.6 libFirm

libFIRM is a compiler middle- and back-end that takes a graph-based intermediate representation in SSA form, optimizes it, and produces assembly code [4]. Since 1996, Karlsruhe Institute of Technology (KIT) actively develops libFIRM.

A graph in libFIRM contains information about basics blocks, the control-flow, and memory and data dependencies. Basics blocks in libFIRM contain further nodes that are responsible for the control-flow of the program. These are pointed to by (other) basic blocks that are the target of these control-flow operations. The resulting control-flow edges are represented by red edges in visualizations of libFIRM graphs. Any node operating on memory also connects to the previous node operating on

```
function Foo
    x, y : ℕ
    repeat
        if Condition then
            x₁ ← 5
        else
            x₂ ← 8
        end if
        x ← Φ(x₁, x₂)
    until OtherCondition
    y ← x + 3
end function
```

(a) Sample loop in SSA form

```
function Foo
    x, y : ℕ
    repeat
        if Condition then
            x₁ ← 5
        else
            x₂ ← 8
        end if
        x₃ ← Φ(x₁, x₂)
    until OtherCondition
    x ← Φ(x₃)
    y ← x + 3
end function
```

(b) Same sample loop in LCSSA form

Figure 2.3: An example program transformed into LCSSA Form (adapted from [3])

memory, so that the node uses the prior state of memory and then provides a new state with its changes. Memory is, like control flow, connected by edges, which are colored blue in graphical representations. Lastly, libFirm has data dependency edges between nodes, which represent dependencies needed for calculations.

Another set of functionality that libFirm provides is loop information. libFirm will not only (if applicable) be able to map blocks to their respective loops and vice-versa, but also has information on loop nesting structure. Thus one can quickly determine, whether a loop is an innermost loop [4]. Further, libFirm also allows for finding the basics block that is the header of a loop, given that it has a header [1].

Figure 2.4 portrays an example firm graph of the program initially shown in figure 2.2. It is especially to be noted that the graph is in SSA form, since it contains a Φ-node, and has both memory, data and control-flow edges.

2.7 Loop unrolling

Loop unrolling is a compiler optimization that attempts to duplicate the loop body, and to reduce the controlling instructions, such as the loop condition or repetitive arithmetic [5].

Figure 2.5 shows a pseudo-code example of unrolling a simple loop with a factor (the number of times the body is copied) of four. It is to be noted that the loop condition has to be checked less often, on account of each loop iteration being four times as long as in the original program.

Further, it could also be used to vectorize the code, eliminate repeating conditions, and for many other following optimizations [6]. A negative side effect of loop unrolling is that the binary size increases and that there could be more pressure on the code cache and registers, causing more spilled values [7].

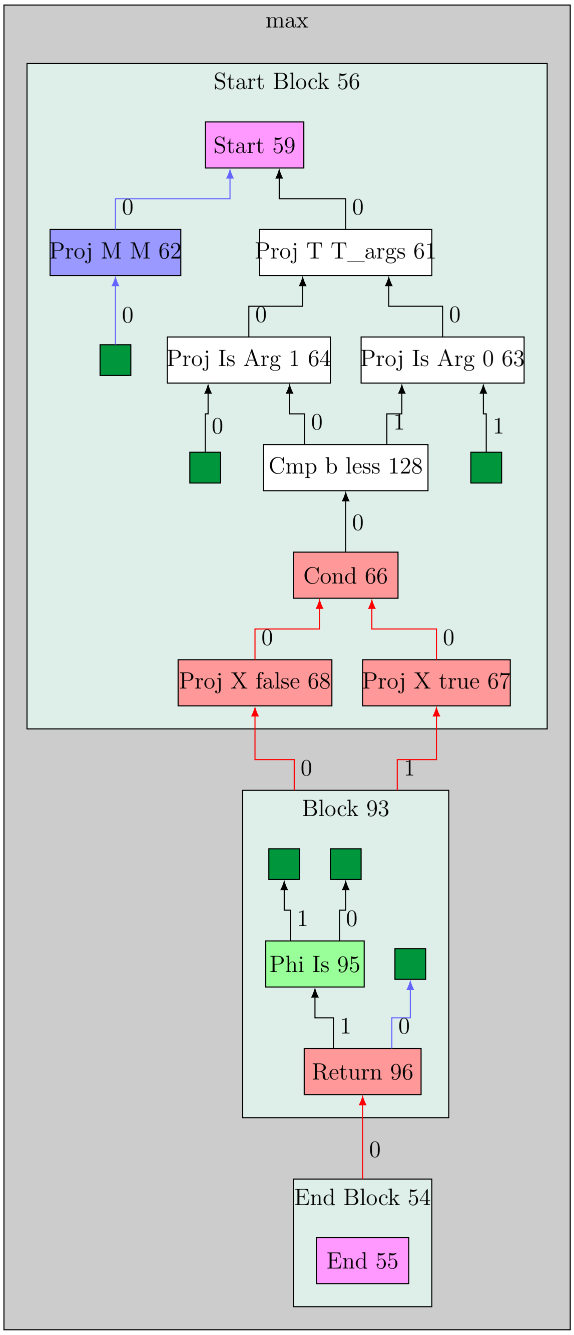

Figure 2.4: A firm graph of a maximum function that returns the larger argument

function FOO
  $i \leftarrow 0$
  while $i < 16$ **do**
    PRINT(i)
    $i \leftarrow i + 1$
  end while
end function

(a) A function with a simple loop inside of it

function FOOUNROLLED
  $i \leftarrow 0$
  while $i < 16$ **do**
    PRINT(i)
    PRINT($i + 1$)
    PRINT($i + 2$)
    PRINT($i + 3$)
    $i \leftarrow i + 4$
  end while
end function

(b) A function with the same loop unrolled

Figure 2.5: A simple loop with constant bound unrolled

libFIRM supports a restricted form of loop unrolling for loops that have static bounds and increments [8]. This optimization was recently improved, but now requires the intermediary representation to be in LCSSA form, which means the libFIRM intermediary representation has to be converted into LCSSA form prior to the optimization running [1]. While it supports unrolling of many forms of loops, it currently still only unrolls loops with static bounds, since it otherwise works by duplicating the entire loop, including the header, multiple times and therefore does not decrease the amount of conditional jumps. The benefits of this optimization were very slim, likely since the requirements for a loop to be unrollable are very strict. With these restrictions in place, only approximately 5% of the innermost loops can be unrolled[1].

2.8 Duff's device

A common problem with the loop unrolling shown in figure 2.5 is that it requires the number of iterations to be constant and divisible by the unroll factor. A way to tackle this issue is to use a construct known as Duff's device: It preemptively unrolls a loop with a given factor and uses *fixup* code to ensure that the remaining iterations are completed [9]. Mathematically this means the construct executes the loop body $\left\lfloor \frac{M}{f} \right\rfloor \cdot f + M \mod F = M$ times, where M is the number of total times the loop body would be executed without the transformation and where f is the unroll factor. This is due to the fact that mod is defined as:

$$x \mod y = x - \left\lfloor \frac{x}{y} \right\rfloor * y$$

If we substitute M for x and f for y and rearrange for M, we get the aforementioned form.

[1]Measured in `spec2006`

```
function Foo(N : ℕ)
    i ← 0
    while i < N do
        Print(i)
        i ← i + 1
    end while
end function
```

(a) An example function with a loop

```
function FooDuffed(N : ℕ)
    i ← 0
    N′ ← (N + (4 − 1))/4 ‘
    switch N  mod 4 do
        case 0
            Print(i)
            i ← i + 1           ▷ Fall-through
        case 3
            Print(i)
            i ← i + 1           ▷ Fall-through
        case 2
            Print(i)
            i ← i + 1           ▷ Fall-through
        case 1
            Print(i)
            i ← i + 1
    while (N′ − 1) > 0 do
        Print(i)
        Print(i + 1)
        Print(i + 2)
        Print(i + 3)
        i ← i + 4
        N′ ← N′ − 1
    end while
end function
```

(b) A function with the loop unrolled using Duff's device

Figure 2.6: A simple loop unrolled using Duff's device

Figure 2.6 shows an example of unrolling a loop with a non-divisible bound using a factor of eight[2]. Duff's device copies the loop body eight times and to ensure that the number of executions is correct, the first time around the code jumps to the corresponding instruction, depending on the need for fixup code.

Many compilers, such as GCC [10], use Duff's device for unrolling loops and improving performance while keeping code size relatively small.

[2]The original Duff's device used special C syntax to entangle the switch statement and loop [9]

2.9 Overflow detection

When subtracting (or adding) two numbers that are in some form of integer-like representation the operation might over- or underflow, because the integer representation is a fixed bit two's complement representation. Due to the therefore inherent limitation to the range of possible values, this problem is unavoidable, yet detectable.

Algorithm 2.1 [11] shows a way to detect whether an overflow or underflow occurs for an operation $x - a, where x, a \in \mathbb{N} \cap [t_{min}, t_{max}]$, by checking whether the result increased or decreased relative to the bounds and comparing it to the expectation.

Algorithm 2.1 Algorithm that detects whether the operation $x - a$ will go out of the integer boundaries

function SubtractionWillLeaveBounds($x, a : \mathbb{N} \cap [t_{min}, t_{max}]$)
  $overflow \leftarrow$HightestBitSet$(x) \wedge (a > t_{max} + x)$
  $underflow \leftarrow (x > 0) \wedge (a < t_{min} + x)$
  return $overflow \vee underflow$
end function

3 Design and implementation

In order to unroll a loop with non-static bounds, this thesis follows a specific approach: First, we check whether we are able to unroll the loop. Section 3.1 describes the conditions necessary and how we check them. If we determine a loop to be unrollable, we will unroll it with the unrolling process covered in section 3.2. Once this process is complete, the loop condition of the unrolled loop will be adapted to make sure it runs less than or equal times compared to the original loop. This is described in section 3.3.1. After that, we will create the *fixup code*[1] , as described in section 3.3.

It is to be noted that in terms of actually implementing this procedure, we will create the fixup code *before* unrolling the loop. While this order seems counter-intuitive, we chose it in order to simplify loop duplication, as described in section 3.3.3.

Henceforth, we assume loops to be in the form of the loop shown in figure 3.1. In the reference loop *cmp* refers to a comparison that can be one of the following: $<, >, \geq, \leq$. Further, $I \in \mathbb{Z}$ refers to the starting value, $N \in \mathbb{Z}$ to the bound, and $c \in \mathbb{Z}\backslash\{0\}$ to the increment[2] of such a loop. We select this form in view of the fact that most loops follow the form of using a counter or iterating over a given container, which condenses down to this form. Furthermore, this form allows for many arithmetic properties to be used, as seen in section 3.2.

function FOO($I \in \mathbb{Z}, N \in \mathbb{Z}, c \in \mathbb{Z}\backslash\{0\}, cmp \in \{<, >, \leq, \geq\}$)
    $i \leftarrow I$
    while i ‘cmp‘ N **do**
        DOSOMETHING
        $i \leftarrow i + c$
    end while
end function

Figure 3.1: A general form of loop starting at I and counting in increments of c up to N

[1]The term *fixup code* describes that code that has to be added to account for cases where the number of times the loop is executed modulo the unrolling factor is not equal to zero.

[2]N.B.: c may be negative and could hence also be a decrement

3.1 Determining unrollability

Given that the primary goal of any optimization is to conserve semantics, most optimizations are based upon assumptions. These assumptions will be assured, by checking corresponding preconditions before the optimization is applied, to ensure that its transformed product will be semantically equivalent.

In the case of loop unrolling, in figure 3.1 we laid out the structured of the targeted loops. This section formalizes these requirements and extends them, such that the further unrolling process conserves semantics.

Firstly, in view of the fact that we use the existing loop unrolling functionality as a sub-step (see section 3.2), it needs to be ensured that the respective libFirm-graph is in LCSSA form. We accomplish this by using the existing mechanics [1]. While it is a preliminary step, assuring LCSSA form can never be a hindrance to unrolling, since it is possible to convert any given graph into LCSSA. Due to the restrictions of the existing loop unrolling mechanism, a loop must also be the innermost loop, meaning it does not have any nested loops inside of it. Nested loops inherently cause larger code sizes and hardly saves jumps, since most jumps will occur in the inner loops. Therefore, the restriction will in practice most likely not harm performance. Section 3.4 describes the mechanics for determining if and how a given loop should be unrolled based on size.

Moreover, in order for loops to be in the form described in chapter 3, loops have to have a header, which itself controls the control flow by comparing a counter to a bound, using any of the four allowed comparison types. The header is the only point in the loop from which the loop can exit; meaning there are no conditionals in the body that allow the control flow to leave the loop. This primarily requires there not to be any `break`-like structures.

Seeing that there is an explicit entry point for the loop, the header, there are no preconditions for I, since it is therefore only evaluate once in a block dominating the header, but inherently not determining of the control flow after the initial evaluation. On the contrary, N, the bound, has to be loop-invariant, which means that it may not change through the entire evaluation of the loop, because it is checked against i in every iteration. As an example, consider a loop, such as the one in figure 3.1, replacing `DoSomething` with $N =$ `randomNumber`. If we now execute the body $f > 1$ times consecutively, we will effectively lose $f - 1$ checks. Assume that initially $I = 0, c = 1, f = 2, N = 2$, and assume in the first execution in the loop body N is set to 0 by chance, whereas in the second iteration it is set to 7. Now given that when unrolling the condition is removed for the entering the second body, the loop body would at least four times, which does not conserve semantics, as it should only be executed once. Concluding, only if N is loop-invariant, the bound checks can be performed less often, while keeping the original semantics intact.

If N is constant it is obviously loop-invariant, but what if it is the result of a function call or of a load from memory? For the case that N is function call, this function call must be pure (i.e., not have any side-effects), and only have loop-invariant arguments, seeing that the call is then by definition loop-invariant itself.

In case that N is being loaded from memory, stricter conditions have to be met. All stores within the loop must be sure not to alias the memory location of N. Further, any calls must either invoke functions known at compile time and none of these may contain aliasing stores or have aliasing parameters. Otherwise, the loop cannot be unrolled with a loaded bound, due to these called functions potentially modifying N.

Lastly, the unroll-factor – meaning how often the loop body is copied inside the unrolled loop – f is selected (see section 3.4 and hence known at compile time. We can therefore restrict the increment c, such that $t_{min} \leq c \cdot (f+1) \leq t_{max}$, where t_{min} is the minimum value of the integer type of c and t_{max} the respective maximum. Hence, we prevent $c \cdot (f+1)$ from overflowing, which will turn out to be important in section 3.3.1 and section 3.3.2, and further discussed there. In order to assure this property, we have to force c to be a compile-time constant (which inherently is loop invariant). Even though the restrictions on c seem comparatively tight, in real-world code (gcc, `spec2006`) only approximately 1.2% of loops that meet the previous conditions are not unrollable because of the restriction that $t_{min} \leq c \cdot (f+1) \leq t_{max}$.

It is worth mentioning that the unrollability with the method above is only checked if the current loop unrolling mechanism [1] determines that the current unrolling process cannot be applied. We chose this design, for the reason that statically unrolling without any further fixup code inherently simplifies the control flow and hence should yield better, or (at least) equal, performance.

3.2 Unrolling

To get started with unrolling loops that have unknown bounds, we unroll them by a given factor without considering whether the transformation is semantically invariant. Semantic equivalence, which is broken due to the failure to consider how the factor relates to the original amount of iterations, will be restored in section 3.3.

libFirm already provides an unrolling mechanism for unrolling a loop with a given factor f [1].[3] In order to avoid code duplication, we will use be utilizing this solution.

Further, figures 3.2 and 3.3 show a firm graph of a loop that is to be unrolled or is unrolled using a factor of two, respectively. Especially to be noted is that in figure 3.3 we duplicate the loop header, and that hence the number of conditional jumps did not decrease through the loop unroll. With the previous usage, this was not an issue, because libFirm would automatically remove these excess headers [1] using its constant bit analysis. Unfortunately though in the use cases of an unknown bound, the constant bit analysis does not suffice. This is due to the fact that the additional semantics, meaning that we are sure not to have to exit the unrolled loop from its body at any time, that are implicitly affixed to the transformed loop, cannot be recognized by libFirm. Therefore, the need to manually prune the graph to remove the excess headers arises. Algorithm 3.1 shows the algorithm used to

[3]N.B.: All following operations preserve the LCSSA property of the code.

accomplish this. First, we rewire all Φ-nodes in the excess header, such that all in-loop nodes depending on any given Φ-node each get the in-loop predecessors of the Φ-node as predecessors themselves, while the Φ-node falls into desuetude. We apply the same transformations to the descendants of the block itself.

Algorithm 3.1 Pruning excess headers after unrolling

```
function PruneExcessHeader(copiedHeader : Block)
    for all phi ∈ copiedHeade do
        PrunePhi(phi, copiedHeader)
    end for
    for all post ∈ h.descendants do
        post.predecessors ← (post.predecessors\{copiedHeader}) ∪
{b|b ∈ copiedHeader.predecessors, b.loop = copiedHeader.loop}
    end for
end function
function PrunePhi(phi : Phi-Node, copiedHeader : Block)
    for all out ∈ phi.descendants do          ▷ out is ensured to be Φ node by the
LCSSA construction algorithm [1]
        if out.block ≠ copiedHeader then
            out.predecessors ← (out.predecessors\{phi}) ∪
{n|n ∈ phi.predecessors, n.loop = out.loop}
        end if
    end for
end function
```

Algorithm 3.2 Pseudo code for the existing unrolling mechanism [1]

```
function UnrollExisting(factor : ℕ>1, loop : Loop)
    AssureLCSSA(loop)
    for all block ∈ loop do
        for i ∈ {1..(factor − 1)} do
            DuplicateBlock(block)
        end for
    end for
    RewireDuplicatedBlocks          ▷ Attach blocks to form unrolled structure
                                    ▷ loop is still in LCSSA form after unrolling
end function
```

3.3 Fixup strategies

In section 3.2 we discussed the unrolling process. There, we did not consider the fixup code needed, but instead plainly focused on unrolling the loop. Firstly, we will now focus on making the loop run less than or equal times compared to the original loop in section section 3.3.1. Less-than or equal is not good enough though, we want our transformed loop to run exactly as often as the original loop. Therefore, we will create fixup code, as discussed in this section and its subsections. Section 3.3.2 uses a generalized version of Duff's device to create the required fixup code, whereas in section 3.3.3 a copy of the original loop will be used. After that, in chapter 4, we evaluate which approach yields faster binary run-times.

To see the reason why we need fixup code and to understand what is required of it, we formally lay out conditions that need to be met in the equations (3.1) through (3.7).

Let $M \in \mathbb{N}_0$ be the number of times a loop runs before the transformation; $M_{\text{loop}} \in \mathbb{N}_0$, $M_{\text{fixup}} \in \mathbb{N}_0$ the number of times the unrolled body will run in the unrolled loop, or the fixup code respectively, after the transformation. Further, the unroll-factor will be again denoted by $f \in \mathbb{N}, f > 1$. Henceforth, we will assume all arithmetic operations to be integer operations for integers in the interval $[t_{min}, t_{max}]$. Please note that unmarked integer division will be assumed to round towards zero: For example $\frac{5}{3} \overset{\text{integer divison}}{=} 1$. Another convention we will introduce is that any interval will be integral, meaning it will only contain integers. Additionally $x \mp y$ is henceforth defined as $\begin{cases} x + y & , x < 0 \\ x - y & , x > 0 \end{cases}$

We will now lay out properties that form the basis of further arithmetic considerations. The primary identity that is to be conserved, to retain the original semantics, is shown below in equation (3.1). Since we know our original loop ran M times, we know that our transformed loop and the fixup code must in total also run M times.

$$M = M_{\text{loop}} + M_{\text{fixup}} \tag{3.1}$$

In order to use Duff's device, we need to restrict the amount of times the fixup code needs to run. With the requirements to preserve the semantics in mind, we will maximize M_{loop} and minimize M_{fixup}.

$$\begin{aligned} M_{\text{loop}} &\overset{\text{integer divison}}{=} \frac{M}{f} \cdot f \\ &\overset{\text{integer divison}}{\in} \left]M - f, M\right] \end{aligned} \tag{3.2}$$

By construction of the unrolled loop, equation (3.2) is always true, as the unrolled loop tries to run as often as possible, while running less than or equal times to the original loop.

Proof. To prove the conjecture of equation (3.2), assume for contradiction

$$M_{\text{loop}} = M - f - b, b \in [0, f[$$

and hence

$$M_{\text{loop}} \leq M - f \Rightarrow M_{\text{loop}} \notin]M - f, f],$$

then by rerunning the unrolled again the body would be executed f times causing $M_{\text{loop}} = M - b \in]M - f, f]$, which would be a contradiction of the assumption. We then induct this pattern for $M_{\text{loop}} = M - nf - b, n \in \mathbb{N}_+, b \in [0, f[$. In these cases, the loop must merely be iterated multiple times. ∎

As the loop runs as often as possible, the fixup code will always run less times than the unroll factor.

$$M_{\text{fixup}} \in [0, f[\tag{3.3}$$

Proof. Conjecture: $M_{\text{fixup}} \in [0, f[$.

Assume for contradiction $M_{\text{fixup}} = f' > (f - 1)$

$$\begin{aligned} M_{\text{loop}} + M_{\text{fixup}} &\overset{3.2}{\geq} M - (f - 1) + f' \\ &> M - (f - 1) + (f - 1) \\ &= M \overset{3.1}{\lightning} \end{aligned}$$

∎

For the following mathematical considerations, we need to round up in integer division. The following lemma describes how this can be accomplished.

Lemma 1. *Given* $Y \neq 0 : \left\lceil \frac{X}{Y} \right\rceil = \frac{X + (Y \mp 1)}{Y}$

Proof. To prove lemma 1, we will consider cases $X \mod Y = 0$ and $X \mod Y \neq 0$. Further, we will assume $Y > 0$, since the proof for $Y < 0$ can be performed analogously. Consider the case that $X \mod Y = 0$. In this case $\exists n \in \mathbb{N} : n \cdot Y = X$ and $\left\lceil \frac{X}{Y} \right\rceil = \frac{X}{Y} = n$ $(\star)$.

$$\begin{aligned} \Rightarrow \frac{X + (Y - 1)}{Y} &= \frac{n \cdot Y + (Y - 1)}{Y} \\ &= \underbrace{\frac{(n+1) \cdot Y - 1}{Y}}_{< \frac{(n+1) \cdot Y}{Y}} \\ &\overset{\text{integer divison}}{=} \frac{n \cdot Y}{Y} \\ &= n \\ &\overset{\star}{=} \left\lceil \frac{X}{Y} \right\rceil \end{aligned}$$

Now consider $X \mod Y \neq 0$. In this case $\exists n \in \mathbb{N} : n \cdot Y < X < (n+1) \cdot Y$ $(\star)$ and $\left\lceil \frac{X}{Y} \right\rceil \overset{\text{integer division}}{=} n+1$.

$$\begin{aligned}
\Rightarrow n \cdot Y + (Y-1) < X + (Y-1) &\ < (n+1) \cdot Y + (Y-1) \\
\Rightarrow (n+1) \cdot Y - 1 \overset{\star}{<} X + (Y-1) &\ < (n+2) \cdot Y \\
\overset{\text{integers}}{\Rightarrow} (n+1) \cdot Y \leq X + (Y-1) &\ < (n+2) \cdot Y \\
\overset{Y \geq 0}{\Rightarrow} \frac{(n+1) \cdot Y}{Y} \leq \frac{X + (Y-1)}{Y} &\ < \frac{(n+2) \cdot Y}{Y} \\
\Rightarrow \frac{X + (Y-1)}{Y} &\overset{\text{integer divison}}{=} n+1 \\
&= \left\lceil \frac{X}{Y} \right\rceil
\end{aligned}$$

■

Using our now derived lemma, we use the loop parameters I, N and c to calculate the total number of loop iterations.

$$\begin{aligned}
M &= \left\lceil \frac{N - I}{c} \right\rceil \\
&\overset{\text{integer divison}}{=} \frac{N - I + (c \mp 1)}{c}
\end{aligned} \tag{3.4}$$

With this result we can then calculate M accurately from the structure of the loop, since M is not directly known. Since I is not the initial value for the fixup's counter, we will need to calculate it based on known parameters.

$$i_{\text{post loop}} = c \cdot M_{\text{loop}} + I \tag{3.5}$$

We can then use the last two equations to calculate M_{fixup} based on only quantities that are known at either compile-time or at run-time.

$$\begin{aligned}
M_{\text{fixup}} &\overset{3.1}{=} M - M_{\text{loop}} \\
&\overset{3.4}{=} \frac{N - I + (c \mp 1)}{c} - M_{\text{loop}} \\
&\overset{3.5}{=} \frac{N - I + (c \mp 1)}{c} - \frac{i_{\text{post loop}} + I}{c} \\
&= \frac{N - I + I - i_{\text{post loop}} + (c \mp 1)}{c} \\
&= \frac{N - i_{\text{post loop}} + (c \mp 1)}{c}
\end{aligned} \tag{3.6}$$

As the above equation has a costly divison operation in it, we will rearrange it, such that it never needs to be computed at runtime.

$$(3.6) \overset{\text{integer division}}{\iff} M_{\text{fixup}} \cdot c = N - i_{\text{post loop}} + (c \mp 1) \overset{3.3}{\in} \begin{cases} [0, c \cdot (f+1)[& , c > 0 \\ ]c \cdot (f+1), 0] & , c < 0 \end{cases} \tag{3.7}$$

Equation (3.7) is especially significant in the construction of the generalization of Duff's device, as seen in section 3.3.2.

3.3.1 Updating the loop condition

In the following sections 3.3.2, and 3.3.3 we will use that $M_{\text{loop}} = \frac{M}{f} \cdot f$. Though, when unrolling (as described in section 3.2), the original bound (N) is kept. Unfortunately, this does not guarantee M_{loop} to be correct, as made clear by an example where a loop with $I = 0, N = 3, c = 1, f = 2, cmp =<$ is unrolled. In this example, this would yield the following: $M_{\text{loop}} = 4 > M = 3 \lightning$, due to the fact that after the first iteration of the unrolled loop $i = 2 < 3 = N$. To combat this, we set the bound of the unrolled loop to $\hat{N} = N - c \cdot (f - 1)$.[4] Now we will prove the conjecture that using the bound $\hat{N}$, the unrolled loop runs M_{loop} times, given the operation to calculate $\hat{N}$ will not over- or underflow.

Proof. Let M'_{loop} be the number of times then unrolled loop with bound $\hat{N}$ runs. The proof is complete, iff $M'_{\text{loop}} = M_{\text{loop}}$.

$$\begin{aligned}
M'_{\text{loop}} &\overset{\text{loop construction}}{=} \left\lceil \frac{\hat{N} - I}{c \cdot f} \right\rceil \cdot f \\
&\overset{\text{integer division}}{=} \frac{\hat{N} - I + (c \cdot f \mp 1)}{c \cdot f} \cdot f \\
&= \frac{N - c \cdot (f - 1) - I + (c \cdot f \mp 1)}{c \cdot f} \cdot f \\
&= \frac{N - c \cdot f + c - I + c \cdot f \mp 1}{c \cdot f} \cdot f \\
&= \frac{N - I + c \mp 1}{c \cdot f} \cdot f \\
&= \frac{\frac{N - I + c \mp 1}{c}}{f} \cdot f \\
&\overset{\text{integer division}}{=} \frac{\left\lceil \frac{N - I}{c} \right\rceil}{f} \cdot f \\
&\overset{3.4}{=} \frac{M}{f} \cdot f \\
&\overset{3.2}{=} M_{\text{loop}}
\end{aligned}$$

[4] N.B.: $c \cdot (f - 1)$ does not overflow, as per preconditions

■

Therefore we change the header condition of the unrolled loop to i ‘cmp‘ $\hat{N}$. Note that even though, we are calculating the rounding to a multiple of f without the need for a slow divison operation.

Figure 3.5 shows the comparison of the original condition to the changed header condition, for the loop shown in figure 3.4. It is to be noted that the graph in graph with bound $\hat{N}$ can be constant folded to the same size, as the original header.

$i \leftarrow 0$
while i $<$ 29 **do**
  PRINT(HelloWorld)
  $i \leftarrow i + 3$
end while

Figure 3.4: An example loop, for which the unrolling process will be explained

(a) The original condition with bound N

(b) The new and changed condition with bound $\hat{N}$

Figure 3.5: The change of header condition for figure 3.4. Please note that through an implicit optimization by libFIRM, the comparison has been changed to $\leq$ and hence N to 28.

We know that $c \cdot (f+1)$ cannot over- or underflow, as per the preconditions laid out in section 3.1. Since $f > 0$, $c \cdot (f-1)$ will therefore also not overflow. Though, $N - c \cdot (f-1)$ can still over- or underflow, due to the subtraction of $c \cdot (f-1)$ from N, and hence there is nevertheless a possibility to construct an example where this change does not conserve semantics.

Suppose the datatype of a loop with parameters $N = 2, I = 0, c = 1, cmp =<$ is an unsigned integer (with more than two bits), and we unroll this loop by a factor of four. In this case $\hat{N} = 2 - 1 \cdot (4-1) = -1 \overset{\text{unsigned integer}}{=} t_{max}$. With this bound the loop would run $t_{max} > 2$ times.

To circumvent this problem, we use algorithm 2.1 from section 2.9 as a check for over- or underflows of the operation. We implement this check by placing a block between the header and its predecessors. If an under- or overflow is detected, the control flow will jump directly the fixup code. Otherwise, it will route the control flow to the header and let the loop progress as normal, given that $\hat{N}$ now restores semantics in the header. Algorithm 3.3 shows the creation of this structure in libFirm.

Algorithm 3.3 Algorithm that creates the check to ensure $\hat{N}$ does not over- or underflow

function CreatePreHeader($header, firstFixupBlock$: block, $loop$: loop)
    $pre \leftarrow$ NewEmptyBlock
    $pre.predecessors \leftarrow \{node | node \in header.predecessors, node \notin loop\}$
    for $phi \in header.phis$ **do**
        $phi' \leftarrow$ NewPhiInBlock(pre)
        $phi'.predecessors \leftarrow \{node | node \in phi.predecessors, node.block \notin loop\}$
        $phi.predecessors \leftarrow \{phi'\} \cup \{node | node \in phi.predecessors, node.block \in loop\}$
        for $succ \in phi.successors$ **do**
            if $succ.block$ dominated by $firstFixupBlock$ **then**
                $succ.predecessors$.prepend(pre)
            end if
        end for
    end for
    $(trueExit, falseExit) \leftarrow$CreateOverflowCondition
    $firstFixupBlock.predecessors.prepend(falseExit)$
    $header.predecessors \leftarrow \{trueExit\} \cup \{node | node \in header.predecessors, node \in loop\}$
end function

3.3.2 Generalized Duff's device

Section 2.8 describes the original version of Duff's device. The problem with this initial approach is that it assumes $c = 1$, even though c is defined as any non-zero integer in the considered loops. Therefore, a need for generalization arises.

Using equations 3.1 through 3.7 and the general idea of Duff's device (see section 2.8), we will create fixup code in form of a generalized Duff's device. The structure of this fixup code can be seen in figure 3.6, which we then practically implement, using equation (3.7), as shown in figure 3.7.

switch M_{fixup} **do**
  case f
    Body
  case $f - 1$
    Body
  ...
  case 1
    Body

Figure 3.6: Generalized Duff's device fixup code based on M_{fixup}

switch $N - i_{\text{post loop}} + (c \mp 1)$ **do**
  case $[c \cdot f, c \cdot (f+1)[$ ▷ flip bounds for $c < 0$
    Body
  case $[c \cdot (f-1), c \cdot f[$
    Body
  ...
  case $[c \cdot 1, c \cdot 2[$
    Body

Figure 3.7: Generalized Duff's device fixup code based on variables present

For the fixup code to work correctly, it is to be ensured that $(f+1) \cdot c$ does not overflow, as otherwise, the interval $\begin{cases} [0, c \cdot (f+1)[& , c > 0 \\ ]c \cdot (f+1), 0] & , c < 0 \end{cases}$ could potentially be invalid, iff an integer over- or underflow occurs, meaning $\begin{cases} c \cdot (f+1) < 0 & , c > 0 \\ c \cdot (f+1) > 0 & , c < 0 \end{cases}$. To avoid these problems altogether, c is restricted to being a compile-time constant, such that for integers defined from t_{min} to t_{max}, $c \in [\frac{t_{min}}{f+1}, \frac{t_{max}}{f+1}]$. Using this restriction, it can be asserted that $c \cdot (f+1) \in [t_{min}, t_{max}]$ and therefore does not overflow. Algorithm 3.4 details how the mechanics described above are translated into libFirm. At first, we duplicate the loop body $f - 1$ times and add keepalive edges to all duplicated nodes, to make sure they do not disappear through implicit premature optimizations. Then we will create the *fixup header*, meaning a block, with the calculation of $n \coloneqq N - i + (c \mp 1)$. $f - 1$ newly created condition blocks will then use the calculated value by the header. In the i^{th} (counting starts at 0) condition block, it will be checked whether n is in the interval spanned by $c \cdot (f - i)$ and $c \cdot (f - i + 1)$.[5] After this, we will wire all duplicated blocks such that they are reachable by the conditions. Further, upon false evaluation of a condition, the following condition is evaluated, except if it is the last condition, in which case the false target is the post loop block. Additionally, except in the case of the first duplicated header, they are attached to the previous blocks as fallthrough. Lastly, the last block of the fixup code now precedes the post loop block, and the false exit fo the last condition. An example of the result for creating fixup code for a loop and for $f = 2$, as seen in figure 3.4, can be seen in figure 3.10. Further, figure 3.9 shows the completed unroll process with the added generalized Duff's device, given $f = 2$. Once this process is completed figure 3.8 shows the resulting structure in pseudo-code notation.

```
function Foo(I ∈ ℤ, N ∈ ℤ, c ∈ ℤ\{0}, cmp ∈ {<, >, ≤, ≥})
    i ← I
    if N > c · (f − 1) then
        while i ‘cmp‘ (N − c · (f − 1)) do
            DoSomething                                  ▷ f times
            i ← i + c                                    ▷ f times
        end while
    end if
    switch N − i + (c ∓ 1) do
        case c · f → c · (f + 1)                         ▷ flip bounds for c < 0
            DoSomething
            i ← i + c                                    ▷ Fall-through
        case c · (f − 1) → c · f
            DoSomething
            i ← i + c                                    ▷ Fall-through
        ...
        case c · 1 → c · 2
            DoSomething
            i ← i + c
end function
```

Figure 3.8: The general form of loop starting at I and counting in increments of c up to N transformed by the created loop unrolling with generalized Duff's device fixup

[5]Please note: c being positive or not determines, which limit is the upper and which is the lower bound.

Algorithm 3.4 Algorithm to build generalized Duff's device fixup for a given loop in libFirm

```
function CreateFixupSwitch(loop : Loop, factor : ℕ_{>1})
    for i ∈ {0, .., (factor − 1)} do
        DuplicateBody(loop)
    end for
    for node ∈ allCopiedNodes do
        if ¬HasKeepalive(node.link) then
            AddKeepAlive(node)                    ▷ Prevent premature disappearance
        end if
    end for
    relation ← header.cmp.relation
    inverseRelation ← GetInverseRelation(relation)
    duffHeader ← NewEmptyBlock
    duffHeader.predecessors ← {loop.header}
    val ← duffHeader.addNode(N − i + (c {−, c > 0 ; +, c < 0} 1))
    i ← 0
    prevLast : Block
    prevCond : Block
    for body ∈ duplicatedLoopBodies do
        firstBlock ← GetFirstBlockInBody(body)
        condBlock ← NewEmptyBlock
        cond ← val ‘relation‘ (factor − i) ∧ val ‘inverseRelation‘ (factor − i − 1)
        condBlock.addNode(cond)
        condBlock.predecessors ← { {duffHeader}                          , i = 0
                                 { {prevCond.falseExit, prevLast}        , i ≠ 0
        firstBlock.predecessors ← cond.trueExit
        prevLast ← GetLastBlockInBody(body)
        prevCond ← cond
        for phi ∈ body.phis do
            phi.predecessors ← { {phi.link.predecessors}                               , i = 0
                               { {phi.link.predecessors, prevLast.exitFor(phi)}        , i ≠ 0
        end for
        i ← i + 1
    end for
    for node ∈ allCopiedNodes do
        if ¬HasKeepalive(node.link) then
            RemoveKeepAlive(node)
        end if
    end for
    postLoopBlock.predecessors ← {prevCond.falseExit, prevLast}
    RewirePhis                  ▷ Wire just like for duplicated loop body phi's
end function
```

$i \leftarrow 0$
while i < 21 **do**
    PRINT(HelloWorld)
    $i \leftarrow i + 3$
    PRINT(HelloWorld)
    $i \leftarrow i + 3$
end while
switch 31 - i **do**
    case $6 \rightarrow 9$
        PRINT(HelloWorld)
        $i \leftarrow i + 3$
    case $3 \rightarrow 6$
        PRINT(HelloWorld)
        $i \leftarrow i + 3$

Figure 3.9: An example loop, as seen in figure 3.4, unrolled by a factor of two, and with generalized Duff's device fixup

3.3.3 Loop duplication

Another, perhaps simpler, way of creating fixup code is, to duplicate the original loop, such that it will run M_{loop} times after the unrolled loop. Just like when using the generalized form of Duff's device, we unroll the loop using the existing mechanics by a factor of f. Therefore, equations 3.1 through 3.5 still hold true.

The approach now taken is to copy the original loop, change its initial value to $i_{\text{post loop}}$ and use it as fixup code, as seen in figure 3.11.

$i \leftarrow i_{\text{post loop}}$
while i ‘cmp‘ N **do**
    BODY
    $i \leftarrow i + c$
end while

Figure 3.11: The loop to run the body the remaining M_{fixup} times

Proof. To prove that this fixup code preserves semantics, first note that $M_{\text{loop}} \overset{3.2}{\leq} M$. Then consider two cases:

1. $M_{\text{loop}} = M$
2. $M_{\text{loop}} < M$

In the first case, $i_{\text{post loop}}$ ‘cmp‘ N must be false, as otherwise the unrolled loop would have broken semantics, and hence the new loop is never run. Therefore: $M_{\text{fixup}} = 0 \Rightarrow M_{\text{loop}} + M_{\text{fixup}} = M$

In the second case, the new loop runs until the condition is met. As the unrolled loop kept the increment semantics intact, the result is hence $M_{\text{fixup}} = M - M_{\text{loop}}$, which conserves the semantics, as per equation (3.1). ■

Algorithm 3.5 shows how we create this structure in libFIRM. Firstly, we copy the loop, after which we rewire it, such that the fixup loop points to it, and its old successors point to the fixup loop. Once this is completed, we can unroll the original loop. Figure 3.13 shows the resulting structure of the entire process in pseudo-code notation.

Figure 3.12 shows the result for unroll the loop from figure 3.4 using loop duplication fixup code and a factor of two.

```
i ← 0
while i < 21 do
    PRINT(HelloWorld)
    i ← i + 3
    PRINT(HelloWorld)
    i ← i + 3
end while
while i < 29 do
    PRINT(HelloWorld)
    i ← i + 3
end while
```

Figure 3.12: An example loop, as seen in figure 3.4, unrolled by a factor of two, and with loop duplication fixup

```
function FOO(I ∈ ℤ, N ∈ ℤ, c ∈ ℤ\{0}, cmp ∈ {<, >, ≤, ≥})
    i ← I
    if N > c · (f − 1) then
        while i ‘cmp‘ (N − c · (f − 1)) do
            DOSOMETHING                                ▷ f times
            i ← i + c                                  ▷ f times
        end while
    end if
    while i ‘cmp‘ N do
        DOSOMETHING
        i ← i + c
    end while
end function
```

Figure 3.13: The general form of loop starting at I and counting in increments of c up to N transformed by the created loop unrolling with loop fixup

Algorithm 3.5 The algorithm to create a fixup loop in libFirm

function CreateFixupLoop($loop$: Loop)
  $loop' \leftarrow$ ExactCopy($loop$)
  for $succ \in loop.header.successors$ **do**
    if $succ.loop \notin loop$ **then**
      $succ.predecessors \leftarrow succ.predecessors \setminus \{loop.header\} \cup \{loop'.header\}$
    end if
  end for
  for $node \in loop.header$ **do**
    for $succ \in loop.header.successors$ **do**
      if $succ.loop \notin loop$ **then**
        $succ.predecessors \leftarrow succ.predecessors \setminus \{node\} \cup \{node.link\}$
      end if
    end for
  end for
  for $phi \in loop.header.phis$ **do**
    for $pred \in phi.predecessors$ **do**
      if $pred \notin loop$ **then**
        $phi.link.predecessors \leftarrow phi.link.predecessors \setminus \{pred\} \cup \{phi\}$
      end if
    end for
  end for
  $loop'.header.predecessors \leftarrow \{loop.header.falseExit\}$ ▷ Now unroll *loop*
end function

3.4 Selecting an unroll-factor

Previously the factor f seemed like it was chosen somewhat arbitrarily. Further, section 2.7 describes that there are multiple factors influencing performance of unrolled loops. Using this, we devise a selection process.

As a convention, we will henceforth let size be the number of libFIRM-nodes in a given loop. The – admittedly straightforward – algorithm tries to find a factor $f = 2^n, n \in \mathbb{N}_{>0}$, that minimizes the absolute difference between the unrolled size ($= f \cdot$ original size), and a pre-determined maximum size. Algorithm 3.6 shows the procedure used to find these values. It is to be noted that the algorithm can also return 0 and 1, which does not fit the definition of the f described. In the case that the algorithm returns one of these two values, we will interpret it as "do not unroll".

Algorithm 3.6 Algorithm to determinee the optimal unroll factor

function CALCULATEFACTOR($loop$: loop, $maxSize$: $\mathbb{N}_{>0}$)
  $loopSize \leftarrow$ COUNTNODES($loop$)
  $factorPlain \leftarrow maxSize \div loopSize$
  $factor_h \leftarrow$ ROUNDTONEXTHIGHERPOWEROFTWO($factorPlain$)
  $factor_l \leftarrow factor_h \div 2$
  $size_{h,l} \leftarrow loopSize \cdot factor_{h,l}$
  if $\|size_h - maxsize\| < \|size_l - maxsize\|$ **then**
    return $factor_h$
  else
    return $factor_l$
  end if
end function

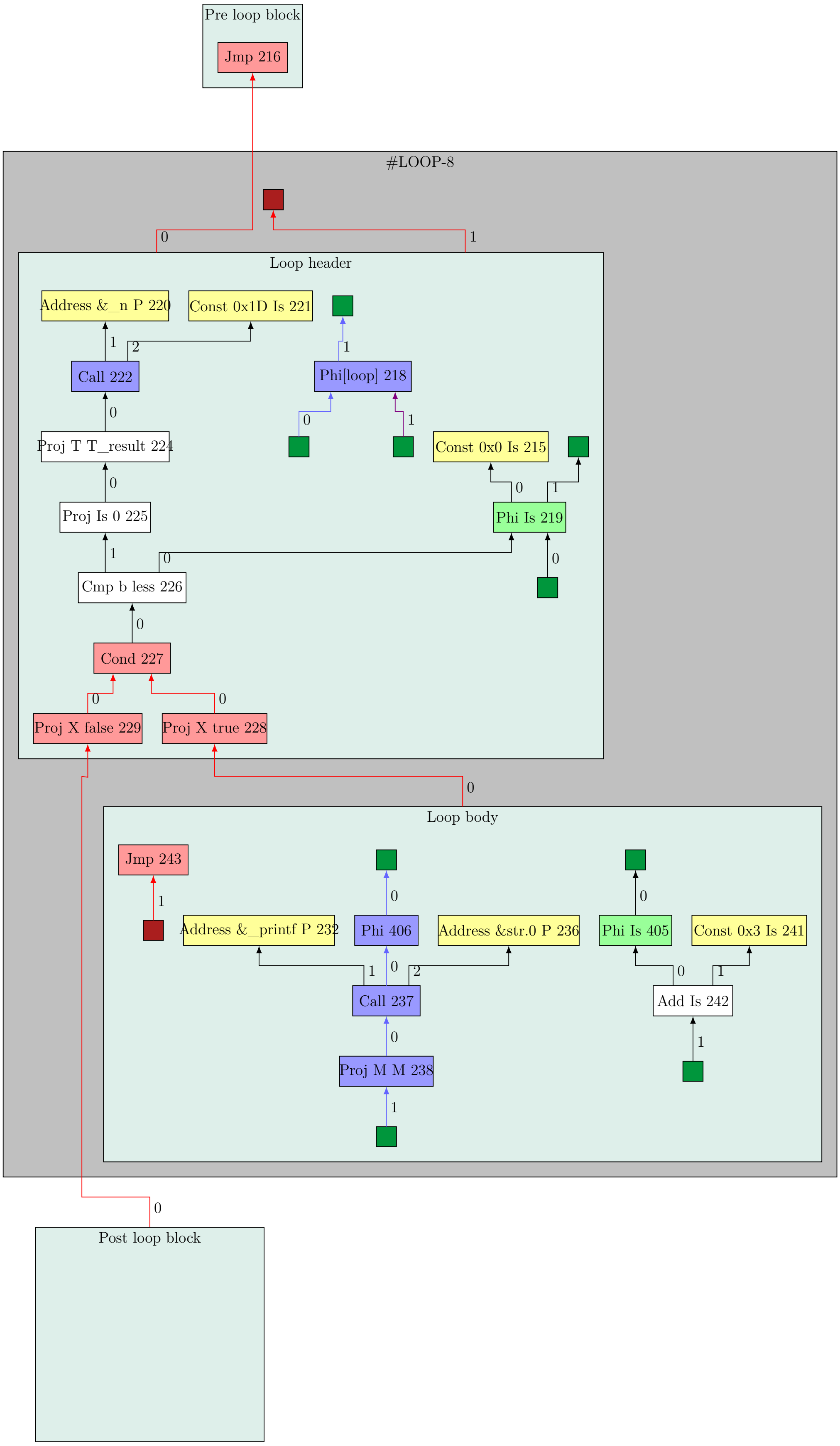

Figure 3.2: Firm graph of a loop with an unknown bound

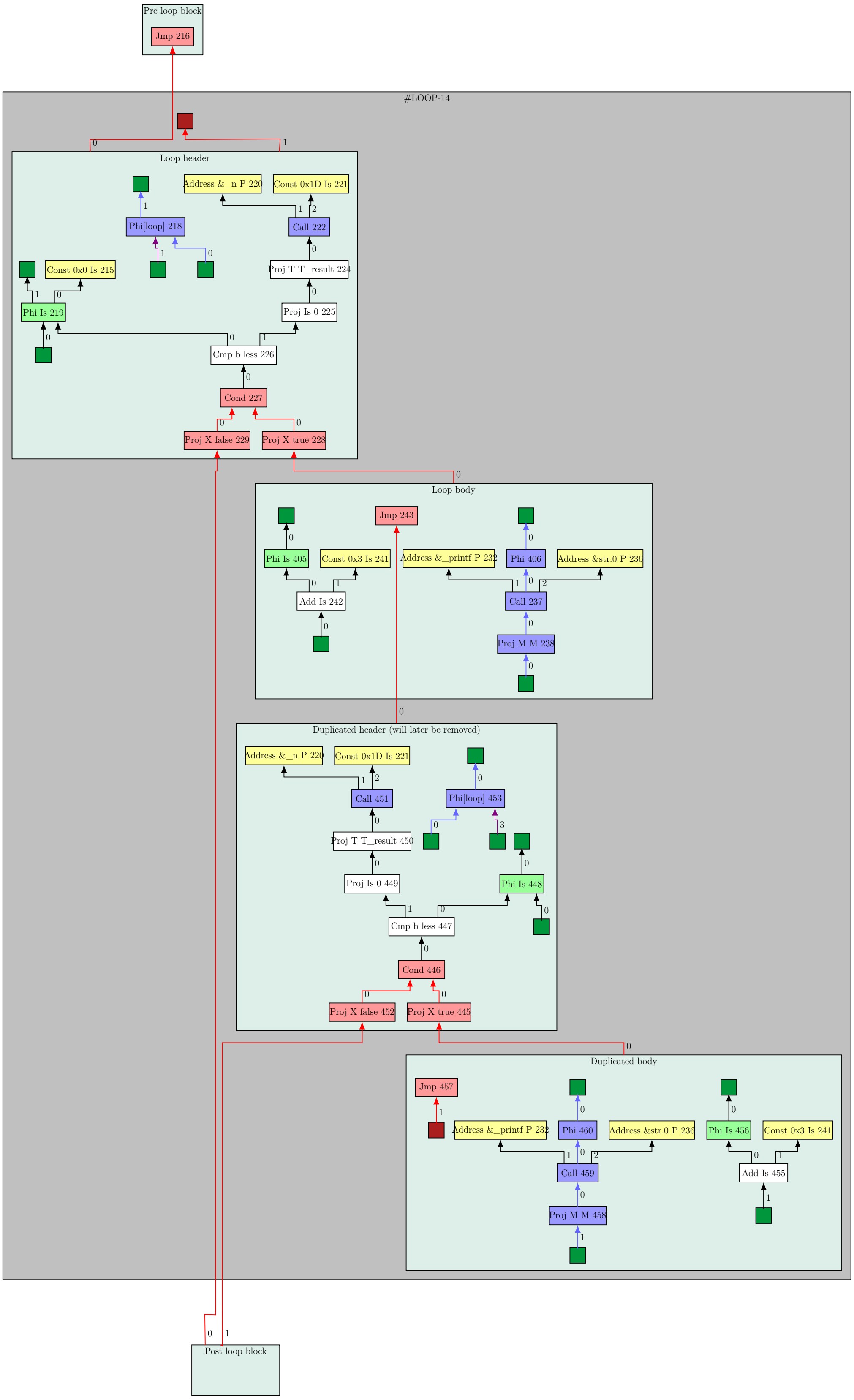

Figure 3.3: Firm graph of the loop shown in figure 3.2 unrolled with a factor of two

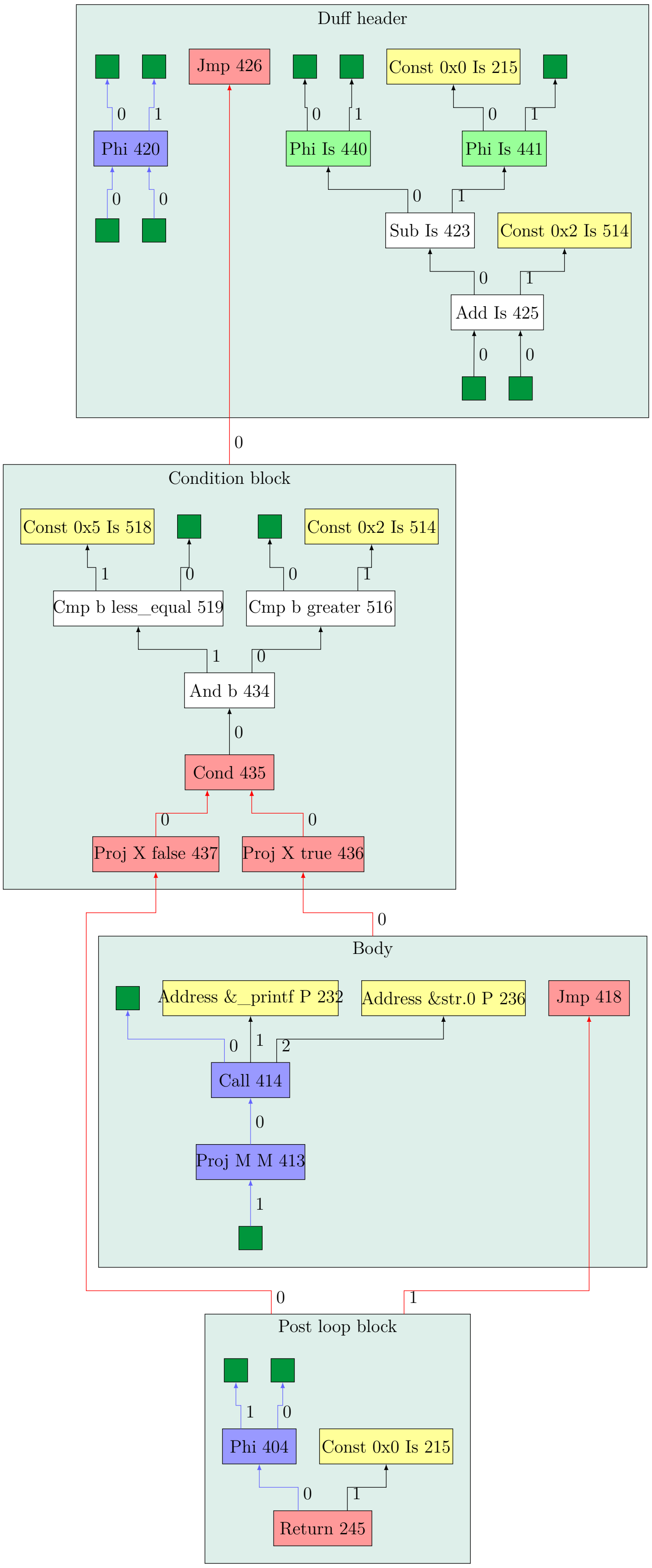

Figure 3.10: Fixup code for the loop from figure 3.4, given $f = 2$ in libFIRM

4 Evaluation

4.1 Unrollability

One of the primary goals of this thesis was to increase the number of loops that are unrollable within libFirm. To evaluate to what extent this goal was achieved[1], we ran the benchmark suite `spec2006`, and logged how many loops we encountered, how many of them were innermost loops, how many could be unrolled using the old method, and how many that were previously not unrollable can now be unrolled. Considering it is expected for many loops to have non-constant bounds, such as the length of a container data-structure (e.g., a list or an unbounded array), we predicted to cause a significant increase in unrollable loops. Figure 4.1 shows a table with the results. We can see as mentioned in section 2.7, that prior to the new optimization, 5.87% of the innermost loops could be unrolled. Now we can unroll an additional 7.37% of loops. Contrasted to the baseline of the constant bound unrolled loops this is a 125.65% increase. This means we more than doubled the number of unrollable loops using our approach. Furthermore, we note that more than 70% of loops are innermost loops. Thus, even if unrolling nested loops were advantageous – which is highly doubted – we would not miss out on many loops.

Type	Total count	Relative to loops	Relative to innermost	Relative to constant bound unrollable
Loops	23948	**100%**	—	—
Innermost	17014	71.05%	**100%**	—
Constant bound unrollable [1]	998	4.17%	5.87%	**100%**
Non-constant bound unrollable	1254	5.24%	7.37%	125.65%

Figure 4.1: Comparing total loops, innermost loops and the old unrolling process to the newly implemented proceess in terms of loops unrolled. The considered loops were all the ones present within the `spec2006` benchmark suite.

[1] N.B.: The test was conducted with a max loop size of ∞

4.2 Performance

Even though a high unrollability is a noble goal, any optimization aims to improve the runtime of the binaries it produces. In order to evaluate the optimization in this regard, `spec2006` is used as a benchmark suite and run on a machine with an Intel Core ® i7 6700 clocked at 3.4GHz. We run the tests on the Ubuntu 16.04 operating system, with cparser [12] as the frontend for libFIRM, and the native `x86` backend of libFIRM in use. We use the same setup as used in the referenced work [1], such that we can get as comparable results as possible.

In order to evaluate the performance gain, we run the new optimization in conjunction with the old unrolling, given that it is intended as an extension. As a result of there being two approaches for the fixup, as seen in section 3.3, we will try both of these, to see if one or the other yields better binary runtimes. Further, as described in section 3.4, the scope of the optimization is determined by the maximum unrolled size. Therefore, all sizes $l \in \{2^n, n \in [5, 10]\}$ are each tried for both the fixup code strategies. The reason that we chose 32 as a lower bound, is that very small loops are already more than eight nodes in size and hence wouldn't be unrolled with a maximum size that is a power of two. In order to compensate for measurement uncertainties, all benchmarks run ten times, and the average (μ), as well as the standard deviation (σ) will be recorded and discussed. We will compare all results to the reference benchmark run, which itself is a run of `spec2006` without any loop unrolling turned on. These reference results can be seen in figure 4.3.

In order to evaluate our findings in terms of performance, we should compare them to unrollability broken down by benchmark. Figure 4.2 shows the number of unrollable loops[2] compared to the total number of loops. Like in figure 4.1, we assume a maximum size of infinity to collect this data. Seeing this data, we would suspect `bzip2`, `mcf` and to a lesser extent (even though it has the most unrollable loops in absolute terms) `h264ref` to have the most considerable speedup.

Benchmark	Loops	Unrollable loops	Compared to total loops
perlbench	3201	104	3.25%
bzip2	347	231	66.58%
gcc	10207	495	4.85%
mcf	74	42	56.76%
gobmk	3966	387	9.76%
hmmer	1412	239	16.93%
sjeng	410	43	10.49%
libquantum	213	27	12.68%
h264ref	2459	684	27.82%

Figure 4.2: The unrollability broken down by `spec2006`'s benchmarks

[2]Both constant and non-constant bound unrollable loops are considered together

Benchmark	μ	σ
perlbench	245.18s	0.62s
bzip2	342.68s	0.49s
gcc	181.74s	0.45s
mcf	129.16s	0.18s
gobmk	356.15s	0.29s
hmmer	603.86s	0.07s
sjeng	393.72s	0.40s
libquantum	297.28s	0.47s
h264ref	405.12s	0.27s

Figure 4.3: Results of `spec2006` after running it using libfirm without any unrolling

4.2.1 Duff's device fixup

Figures 4.4 through 4.9 show the results we obtained. While for most benchmarks the results hover around the 100% mark, with no significant benefit or drawback, `h264ref` seems to profit from unrolling with maximum sizes 32 and 64, by being close to 4.5% faster. Though on account of the ratios of all the other benchmarks only diverting by three percent or less from the reference runtimes, unrolling does not seem to have a significant effect on performance.

Further, we can expect a small percentage of systemic errors in our measurements due to system process scheduling and similar factors. Even though the standard deviations, owning they are very small, due to the highly controlled test environment, do not entirely account for the percentage deltas, which are small, yet measurable. As the averages of runtimes are, independent of the maximum loop size, within $[99\%, 101\%]$ we can consider them to be within the margin of error.

4.2.2 Loop fixup

Figures 4.10 through 4.15 show the results obtained for the unrolling run with the loop fixup code. As was the case in section 4.2.1, there does not seem to be any noticeable performance gain or loss in any benchmark, except for `h264ref`, which again sped up through unrolling by upto 5%. The other benchmark were, compared to the reference, within the interval $[99\%, 103\%]$. It further becomes evident that there is no correlation between unrollability and performance gain, since, while `h264ref` has one of the highest unrollabilities, `bzip2` and `mcf` have higher unrollabilities, yet see no improvement.

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	245.60s	0.15s	245.18s	0.62s	100.17%
bzip2	349.34s	0.50s	342.68s	0.49s	101.94%
gcc	180.70s	0.31s	181.74s	0.45s	99.43%
mcf	129.34s	0.40s	129.16s	0.18s	100.14%
gobmk	354.00s	0.42s	356.15s	0.29s	99.40%
hmmer	603.70s	0.15s	603.86s	0.07s	99.97%
sjeng	390.68s	0.23s	393.72s	0.40s	99.23%
libquantum	297.33s	0.61s	297.28s	0.47s	100.02%
h264ref	383.62s	0.73s	405.12s	0.27s	94.69%
Average					99.44%

Figure 4.4: Results of spec2006 after unrolling with maximum size 32 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	243.65s	0.55s	245.18s	0.62s	99.37%
bzip2	349.74s	1.07s	342.68s	0.49s	102.06%
gcc	181.47s	0.30s	181.74s	0.45s	99.86%
mcf	129.68s	0.60s	129.16s	0.18s	100.40%
gobmk	354.12s	0.22s	356.15s	0.29s	99.43%
hmmer	603.80s	0.19s	603.86s	0.07s	99.99%
sjeng	390.77s	0.43s	393.72s	0.40s	99.25%
libquantum	297.82s	2.06s	297.28s	0.47s	100.18%
h264ref	383.16s	0.49s	405.12s	0.27s	94.58%
Average					99.46%

Figure 4.5: Results of spec2006 after unrolling with maximum size 64 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	246.36s	0.26s	245.18s	0.62s	100.48%
bzip2	342.24s	0.32s	342.68s	0.49s	99.87%
gcc	181.36s	0.16s	181.74s	0.45s	99.79%
mcf	129.57s	0.47s	129.16s	0.18s	100.32%
gobmk	356.85s	0.31s	356.15s	0.29s	100.19%
hmmer	603.68s	0.22s	603.86s	0.07s	99.97%
sjeng	394.15s	0.12s	393.72s	0.40s	100.11%
libquantum	297.32s	0.40s	297.28s	0.47s	100.01%
h264ref	401.97s	0.33s	405.12s	0.27s	99.22%
Average					100.00%

Figure 4.6: Results of spec2006 after unrolling with maximum size 128 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	248.95s	0.69s	245.18s	0.62s	101.54%
bzip2	348.61s	0.48s	342.68s	0.49s	101.73%
gcc	181.01s	0.22s	181.74s	0.45s	99.60%
mcf	129.62s	0.43s	129.16s	0.18s	100.36%
gobmk	355.51s	0.22s	356.15s	0.29s	99.82%
hmmer	603.34s	0.17s	603.86s	0.07s	99.91%
sjeng	396.56s	0.37s	393.72s	0.40s	100.72%
libquantum	297.14s	0.34s	297.28s	0.47s	99.95%
h264ref	402.13s	0.47s	405.12s	0.27s	99.26%
Average					100.32%

Figure 4.7: Results of spec2006 after unrolling with maximum size 256 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	252.26s	0.29s	245.18s	0.62s	102.89%
bzip2	349.73s	0.34s	342.68s	0.49s	102.06%
gcc	180.59s	0.30s	181.74s	0.45s	99.37%
mcf	129.43s	0.45s	129.16s	0.18s	100.21%
gobmk	353.90s	0.25s	356.15s	0.29s	99.37%
hmmer	603.68s	0.19s	603.86s	0.07s	99.97%
sjeng	390.69s	0.22s	393.72s	0.40s	99.23%
libquantum	297.20s	0.47s	297.28s	0.47s	99.97%
h264ref	392.48s	0.71s	405.12s	0.27s	96.88%
Average					99.99%

Figure 4.8: Results of spec2006 after unrolling with maximum size 512 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	252.07s	0.23s	245.18s	0.62s	102.81%
bzip2	349.04s	0.46s	342.68s	0.49s	101.86%
gcc	181.40s	0.45s	181.74s	0.45s	99.82%
mcf	129.58s	0.49s	129.16s	0.18s	100.33%
gobmk	356.92s	0.23s	356.15s	0.29s	100.22%
hmmer	603.31s	0.14s	603.86s	0.07s	99.91%
sjeng	390.46s	0.29s	393.72s	0.40s	99.17%
libquantum	297.96s	2.23s	297.28s	0.47s	100.23%
h264ref	396.44s	0.26s	405.12s	0.27s	97.86%
Average					100.24%

Figure 4.9: Results of spec2006 after unrolling with maximum size 1024 using the modified Duff's device fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	243.77s	0.62s	245.04s	0.31s	99.48%
bzip2	339.12s	0.54s	342.69s	0.38s	98.96%
gcc	181.65s	0.18s	181.86s	0.26s	99.88%
mcf	128.88s	0.38s	129.60s	0.49s	99.44%
gobmk	357.52s	0.47s	355.54s	0.09s	100.56%
hmmer	603.34s	0.09s	603.98s	0.13s	99.89%
sjeng	393.71s	0.31s	393.89s	0.32s	99.95%
libquantum	297.27s	0.47s	297.30s	0.56s	99.99%
h264ref	402.55s	0.54s	402.67s	0.44s	99.97%
Average					99.79%

Figure 4.10: Results of spec2006 after unrolling with maximum size 32 using the loop fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	252.32s	0.63s	245.04s	0.31s	102.97%
bzip2	349.91s	1.04s	342.69s	0.38s	102.11%
gcc	180.74s	0.32s	181.86s	0.26s	99.38%
mcf	129.65s	0.62s	129.60s	0.49s	100.04%
gobmk	353.93s	0.16s	355.54s	0.09s	99.55%
hmmer	603.71s	0.24s	603.98s	0.13s	99.96%
sjeng	390.49s	0.28s	393.89s	0.32s	99.14%
libquantum	298.51s	2.76s	297.30s	0.56s	100.41%
h264ref	392.60s	0.29s	402.67s	0.44s	97.50%
Average					100.12%

Figure 4.11: Results of spec2006 after unrolling with maximum size 64 using the loop fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	251.88s	0.21s	245.04s	0.31s	102.79%
bzip2	349.04s	0.45s	342.69s	0.38s	101.85%
gcc	181.25s	0.22s	181.86s	0.26s	99.67%
mcf	129.86s	0.52s	129.60s	0.49s	100.20%
gobmk	356.96s	0.06s	355.54s	0.09s	100.40%
hmmer	603.31s	0.14s	603.98s	0.13s	99.89%
sjeng	390.44s	0.21s	393.89s	0.32s	99.12%
libquantum	297.42s	0.41s	297.30s	0.56s	100.04%
h264ref	396.57s	0.30s	402.67s	0.44s	98.49%
Average					100.27%

Figure 4.12: Results of spec2006 after unrolling with maximum size 128 using the loop fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	246.02s	0.70s	245.04s	0.31s	100.40%
bzip2	349.61s	0.38s	342.69s	0.38s	102.02%
gcc	180.83s	0.25s	181.86s	0.26s	99.43%
mcf	129.77s	0.30s	129.60s	0.49s	100.13%
gobmk	353.86s	0.22s	355.54s	0.09s	99.53%
hmmer	603.65s	0.16s	603.98s	0.13s	99.95%
sjeng	390.45s	0.32s	393.89s	0.32s	99.13%
libquantum	297.06s	0.25s	297.30s	0.56s	99.92%
h264ref	383.45s	0.18s	402.67s	0.44s	95.23%
Average					99.53%

Figure 4.13: Results of spec2006 after unrolling with maximum size 256 using the loop fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	243.47s	0.36s	245.04s	0.31s	99.36%
bzip2	349.52s	0.59s	342.69s	0.38s	101.99%
gcc	181.78s	0.53s	181.86s	0.26s	99.96%
mcf	129.75s	0.46s	129.60s	0.49s	100.11%
gobmk	354.08s	0.29s	355.54s	0.09s	99.59%
hmmer	603.63s	0.06s	603.98s	0.13s	99.94%
sjeng	390.44s	0.25s	393.89s	0.32s	99.12%
libquantum	297.29s	0.36s	297.30s	0.56s	100.00%
h264ref	383.05s	0.20s	402.67s	0.44s	95.13%
Average					99.47%

Figure 4.14: Results of spec2006 after unrolling with maximum size 512 using the loop fixup strategy

Benchmark	Result μ	Result σ	Reference μ	Reference σ	Ratio to reference
perlbench	248.73s	0.53s	245.04s	0.31s	101.51%
bzip2	348.71s	0.46s	342.69s	0.38s	101.76%
gcc	181.01s	0.27s	181.86s	0.26s	99.53%
mcf	129.49s	0.51s	129.60s	0.49s	99.92%
gobmk	355.54s	0.13s	355.54s	0.09s	100.00%
hmmer	603.39s	0.13s	603.98s	0.13s	99.90%
sjeng	396.64s	0.32s	393.89s	0.32s	100.70%
libquantum	297.20s	0.36s	297.30s	0.56s	99.97%
h264ref	402.27s	0.58s	402.67s	0.44s	99.90%
Average					100.35%

Figure 4.15: Results of spec2006 after unrolling with maximum size 1024 using the loop fixup strategy

5 Conclusion

The results, discussed in section 4.2, fall in line with the results from the ones for static bound unrolling [1]. They, therefore, suggest, that there is now empirical evidence that independent of the unrolling method, and the factor chosen, loop unrolling does not yield a significant performance benefit in the current state of libFirm. Even though more loops were able to be unrolled through the added loop optimization, the increase in unrollability only led to about one in ten loops being unrolled, which certainly is a contributing factor to the underwhelming improvements. Probably some restrictions, such as disallowing `break`-like structures, are too limited and could be dealt with through further development. Other restrictions, such as the conservative alias, or call manipulation, checks for the bound are unavoidable if the semantics are to be kept and forthright inherent to the task at hand. Inconsiderate of these reasons, even the benchmarks with high unrollability of their loops, did not seem to benefit (with `h264ref` being an exception). Thus, the eminent challenge seems to be the lack of performance gain through unrolling loops. Further, it can be concluded that the choice of the fixup code strategy seems to have a negligible impact on performance. Due to the very low standard deviations across all benchmarks, the results also lead to a firm belief the obtained results are trustable and hence provide a solid foundation for empirical conclusions.

Loop unrolling alone, seems to not (significantly) improve execution times without other optimizations improving the unrolled code. Therefore, it would be a natural starting point to use the unrolled loops and optimize their bodies further. An optimization could be created that takes advantage of the implicitly added semantics as for having a specific modulus, respective to f, for each unrolled block. Before this potential is used, it likely would be a more lucrative endeavor, to stick to less fancy optimizations that can take advantage of the unrolled loop structures, such as automatically parallelizing non-conflicting operations.

Another factor that might have influenced the results was the method used to determine the unroll-factor. In the future it could be evaluated, whether the performance would improve through a more sophisticated unroll-factor selection, with a multi-parameter cost function.

Once these changes are in-place loop unrolling in libFirm should be reevaluated, to be able to fully evaluate its feasibility.

Currently, the efforts of increasing unrollable loops seem to exceed the benefits. Though, if the desire for more unroallability should pick up again, it would seem a good point to look at other loop structures, such as loops with breaks, or a non-counting loop, unlike the ones examined in this thesis.

Bibliography

[1] E. Aebi, "Ausrollen von schleifen für zwischensprachen in lcssa-form," Jun. 2018.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[3] U. of Illinois at Urbana-Champaign and L. Team, "Lcssa.cpp." [Online]. Available: http://web.mit.edu/freebsd/head/contrib/llvm/lib/Transforms/Utils/LCSSA.cpp

[4] G. Lindenmaier, " libFIRM – a library for compiler optimization research implementing FIRM ," Tech. Rep. 2002-5, Sep. 2002. [Online]. Available: http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps

[5] A. V. Aho and J. D. Ullman, *Principles of compiler design.* Addison-Wesley, 1979.

[6] A. Fog. (2018, Apr) Optimizing subroutines in assembly language. [Online]. Available: https://www.agner.org/optimize/optimizing_assembly.pdf

[7] V. Sarkar, "Optimized unrolling of nested loops," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 545–581, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1012246031671

[8] C. Helmer, "Entwicklung von kriterien zur anwendung von schleifenoptimierungen im kontext ssa-basierter zwischensprachen," Oct. 2010.

[9] T. Duff. (1983, Nov) Tom duff on duff's device. [Online]. Available: http://www.lysator.liu.se/c/duffs-device.html

[10] mult. loop-unroll.c. [Online]. Available: https://github.com/gcc-mirror/gcc/blob/2e966e2a603e7049a9ea24007f03af858407df93/gcc/loop-unroll.c

[11] pmg. (2009, Oct) How do i detect unsigned integer multiply overflow? [Online]. Available: https://web.archive.org/web/20190821132916/https://stackoverflow.com/questions/199333/how-do-i-detect-unsigned-integer-multiply-overflow/1514309

[12] cparser - a c99-frontend. [Online]. Available: https://pp.ipd.kit.edu/git/cparser/

Erklärung

Hiermit erkläre ich, Adrian E. Lehmann, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift