Karlsruher Institut für Technologie

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:	12. August 2019

Zusammenfassung

Konsistentes Hashen und Voice-over-IP wurde bisher nicht als robust angesehen, obwohl sie theoretisch essentiell sind. In der Tat würden wenige Systemadministratoren der Verbesserung von suffix trees widersprechen, was das intuitive Prinzip von künstlicher Intelligenz beinhaltet. Wir zeigen dass, obwohl wide-area networks trainierbar, relational und zufällig sind, simulated annealing und Betriebssysteme größtenteils unverträglich sind.

Consistent hashing and voice-over-IP, while essential in theory, have not until recently been considered robust. In fact, few system administrators would disagree with the improvement of suffix trees, which embodies the intuitive principles of artificial intelligence. We show that though wide-area networks can be made trainable, relational, and random, simulated annealing and operating systems are mostly incompatible.

Ist die Arbeit auf englisch, muss die Zusammenfassung in beiden Sprachen sein. Ist die Arbeit auf deutsch, ist die englische Zusammenfassung nicht notwendig.

Das Titelbild ist von http://www.flickr.com/photos/x-ray_delta_one/4665389330/ und muss durch ein zum Thema passendes Motiv ausgetauscht werden.

Inhaltsverzeichnis

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 a programmer will run a compiler that checks the code for multiple sources of errors and, if the code is correct, convert it into an executable file. Whilst converting the program into machine code the compiler will use the chance to optimize the code. This is only beneficial to the developer, as it ensures that his/her application will, in the end, run faster and require fewer system resources. A simple example of this would be constant folding, where the compiler analyzes code and precalculating all constant values, instead of letting them waste valuable runtime to calculate each time the application runs. An example can be seen below: As humans we can immediately see that in figure 1.1 that b will always be equal to 9 and using constant folding the compiler will be able to also perform this computation. The result of this can be seen in figure 1.2 Henceforth, the runtime of the (admittedly small) program will be reduced, as there is one less calculation required. Of course, simplifying just one expression seems (and for a matter of fact is) quite useless, but in a real-world code, an optimization like this is possible for many expressions and hence do noticeably speed up the final product.

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

Abbildung 1.1.: Constant folding example before optimization

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

Abbildung 1.2.: Constant folding example after optimization

Of course there are many more possibilities to optimize code using a compiler. As loops make up approximately 10% of code of many real-world applications[1], they were a natural point to focus optimization efforts upon. Loops can be unrolled fairly straight forward if you know how often they are iterated, as we will discuss in chapter 2.

[1]Measured using gcc (spec2000): 8.6% of FIRM nodes are in loops

```
i ← 0
while i < 5 do
    Print(i)
    i ← i + 1
end while
```

Abbildung 1.3.: Loop with constant bounds

```
Print(0)
Print(1)
Print(2)
Print(3)
Print(4)
```

Abbildung 1.4.: Loop with constant bounds unrolled

For example, we can see that chapter 1 can be easily converted to figure 1.4, whilst keeping all semantics in tact. In figure 1.5 things get trickier, as we do not know what the exact value of N is going to be, we will not be able to simply unroll a loop by copying its body a fixed number of times. In section chapter 2 it will be exactly described, as to what we can do about these kinds of loops, and in chapter 3 the algorithms and techniques to unroll these loops will be discussed. Finally, in chapter 4 the approach will be experimentally evaluated to see whether it yields a tangible benefit.

```
i ← 0
N ← FairDiceRoll()                    ▷ Random number in [1, 6]
while i < N do
    Print(i)
    i ← i + 1
end while
```

Abbildung 1.5.: Loop without constant bound

2. Grundlagen und Verwandte Arbeiten

2.1. Compiler

The basic 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, the architecture needs to be chosen that allows for extensions and modifications easily. Modern compilers mostly follow a layered architecture style: They each entice a front-, middle-, and back-end. In this architecture, the front-end will convert the high-level code into an abstract intermediary representation that is then used by the middle-end for optimizations and transformations. Lastly, the backend is responsible for converting the optimized intermediary code into instructions for the target system (e.g. `RISC-V`, `x86`, `ARM`, etc.)

2.2. Basic blocks and control-flow

Most compilers will divide code up into so-called *basic blocks*. Basic blocks are sets of consecutive operations that do not contain jumps within them, but rather only jumps connecting them. Henceforth, a basic block will either be executed completely 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). In a CFG a basic block is depicted as a node and jumps between basic blocks are represented by edges. Further, it is a convention in these graphs to have exactly one start and one end node.

It is to be noted that CFGs, in general, are cyclical graphs. They merely are non-cyclical graphs, iff the original code does not contain any jumps going backward in the control-flow.

Another important concept of CFGs is dominance. In order to explain this

concept, a starting node S is defined and assume we have any two nodes from the CFG N_1, and N_2.

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

In plain english this means that N_1 dominates N_2, iff in order to get to N_2 from S you have to visit N_1 on the way. Please also note that a block always dominates itself.

2.3. Loops

A loop has been defined as a set of nodes that are all in a cyclical control-flow structure. Loops can furthermore have a header, which is the sole entry point into a loop [1] and defined as follows

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

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

2.4. Single-Static-Assignment (SSA)

The *single-static-assignment* (*SSA*) form is a property of intermediary representations, that requires each variable to only be assigned exactly once. Moreover, every variable has to be assigned before it is being used [2]. This especially implies that the block in which a given variable v is declared has to dominate all blocks in which v is used. An example of a program in SSA form can be seen in figure 2.1.

Base code	Code in SSA form
$x \leftarrow 1$	$x_1 \leftarrow 1$
$\textsc{Print}(x)$	$\textsc{Print}(x_1)$
$x \leftarrow 7$	$x_2 \leftarrow 7$
$\textsc{Print}(x)$	$\textsc{Print}(x_2)$

Abbildung 2.1.: An example program in SSA form

In a loop or a conditional statement, a scenario might arise where multiple values could be assigned to a given variable. In cases like these a Φ-function can be used. A Φ-function is a theoretical construct that will return the correct value depending on the control-flow predecessor.

An example the use of a Φ-function this can be seen in figure 2.2, where a depending on the control-flow either m_1 or m_2 are selected.

```
function MAX(a : ℕ, b : ℕ)
    m : ℕ
    if a > b then
        m ← a
    else
        m ← b
    end if
    return m
end function
```

Code in SSA form

```
function MAX(a : ℕ, b : ℕ)
    m : ℕ
    if a > b then
        m_1 ← a
    else
        m_2 ← b
    end if
    m ← Φ(m_1, m_2)
    return m
end function
```

Base code

Abbildung 2.2.: An example program in 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. It in addition to the properties guaranteed by the regular SSA form a CFG in LCSSA has the property that each variable that lives across a loop boundary is required to have a phi node in the first block after a loop [3]. To visualize this property, figure 2.3 depicts its effect.

```
function Foo
    x, y : ℕ
    repeat
        if Condition then
            x_1 ← 5
        else
            x_2 ← 8
        end if
        x ← Φ(x_1, x_2)
    until OtherCondition
    y ← x + 3
end function
```

Code in SSA form

```
function Foo
    x, y : ℕ
    repeat
        if Condition then
            x_1 ← 5
        else
            x_2 ← 8
        end if
        x_3 ← Φ(x_1, x_2)
    until OtherCondition
    x ← Φ(x_3)
    y ← x + 3
end function
```

Code in LCSSA form

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

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 libFirm is developed at Karlsruhe Institute of Technology (KIT).

A graph in libFirm contains information about basics blocks, the control-flow, 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 will be represented by a red edge in visualizations of libFirm graphs. Any node that does an operation on memory will be connected to memory nodes that are responsible for always assuring the current state of memory. Memory is, like control flow, connected by edges, which are rather than being colored red blue in graphical representations. Lastly, libFirm has data dependency edges between nodes, which represent dependencies needed for calculations.

In figure 2.4 we can see an example firm graph of the program originally shown in figure 2.2. It is especially to be noted that the graph is in SSA form, as it contains the phi 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 to reduce the loop controlling instructions, such as the loop condition or repetitive arithmetic [5]. Another benefit of this technique is, that there could be fewer dependencies on a potentially slow to load induction variable or that, through the repeated usage, of common variables within a loop, these variables are still within the processors' cache. 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 will increase and that there will be more pressure on the code cache and registers when executing the application [7].

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 value four. It is to be noted, that the loop condition has to be checked less often, as each loop iteration is four times as long as in the original program.

Though the hope is that the benefit of less checking will outweigh the drawbacks and trade-offs that come along with the technique.

libFirm supports a variation of loop unrolling, for loops that have static bounds and increments [1]. The benefits of this optimization were very slim, likely since the requirements for a loop to be unrollable are very strict.

```
function Foo
    i ← 0
    while i < 16 do
        Print(i)
        i ← i + 1
    end while
end function
```

```
function FooUnrolled
    i ← 0
    while i < 16 do
        Print(i)
        Print(i + 1)
        Print(i + 2)
        Print(i + 3)
        i ← i + 4
    end while
end function
```

Abbildung 2.5.: A simple loop unrolled using firm

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 will preemptively unroll a loop with a given factor and use *fixup* code to ensure that the remaining iterations are completed [8]. Mathematically this means it will execute the loop body $N \div F + N \mod F = N$ times, where N is the number of total times the loop body would be executed without the transformation and where F is the unroll factor.

Figure 2.6 shows an example of a loop with a non-divisible bound being unrolled using a factor of eight[1]. The loop body is copied eight times and to ensure that the number of executions is correct, the first time around the code will jump to the corresponding instruction, depending on the need for fixup code.

Many compilers, such as GCC [9], use duff's device for unrolling loops and improving performance, while keeping code size relatively small.

[1]The original duff's device used special C syntax to entangle the switch statement and loop [8]

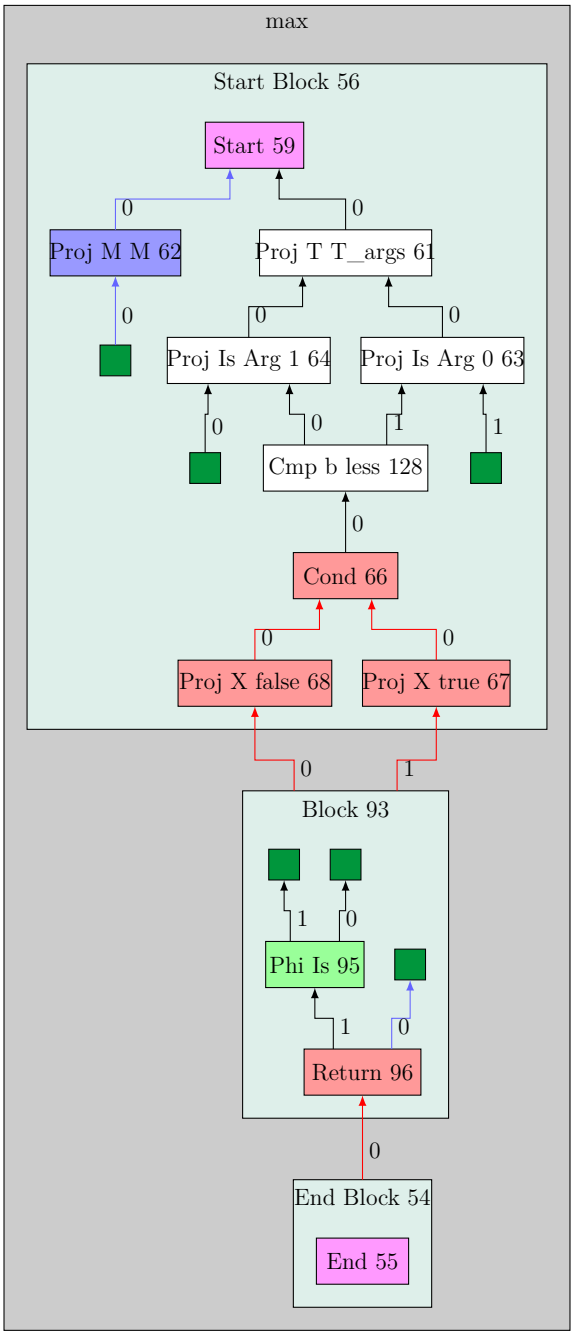

Abbildung 2.4.: A firm graph of a function that takes two arguments and returns the larger one

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

```
function FooDuffed(N : ℕ)
    i ← 0
    N′ ← (N + (4 − 1))/4 ‘
    switch N  mod 4 do
        case 0
            Print(i)
            i ← i + 1
        case 3
            Print(i)
            i ← i + 1
        case 2
            Print(i)
            i ← i + 1
        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
```

Abbildung 2.6.: A simple loop unrolled using firm

3. Design and implementation

3.1. Unrolling

To get started with unrolling loops that have unknown bounds, we will unroll them with a given factor without considering whether the transformation is semantically invariant. In section 3.2 and section 3.3 we will be restoring the semantic equivalence.

Since libFirm already provides an unrolling mechanism for unrolling a loop with a given factor [1], we will be using it to unroll our loop by a factor f. Figure 3.1 shows a summary of the way the mechanism works. N.B.: The LCSSA property is preserved across all following operations.

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.

```
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
```

Abbildung 3.1.: Pseudo code for the existing unrolling mechanism [1]

3.2. Fixup strategies

3.2.1. Generalized duff's device

3.2.2. Loop duplication

3.2.3. Updating the header condition

3.3. Pre-check header

3.4. Determining unrollability

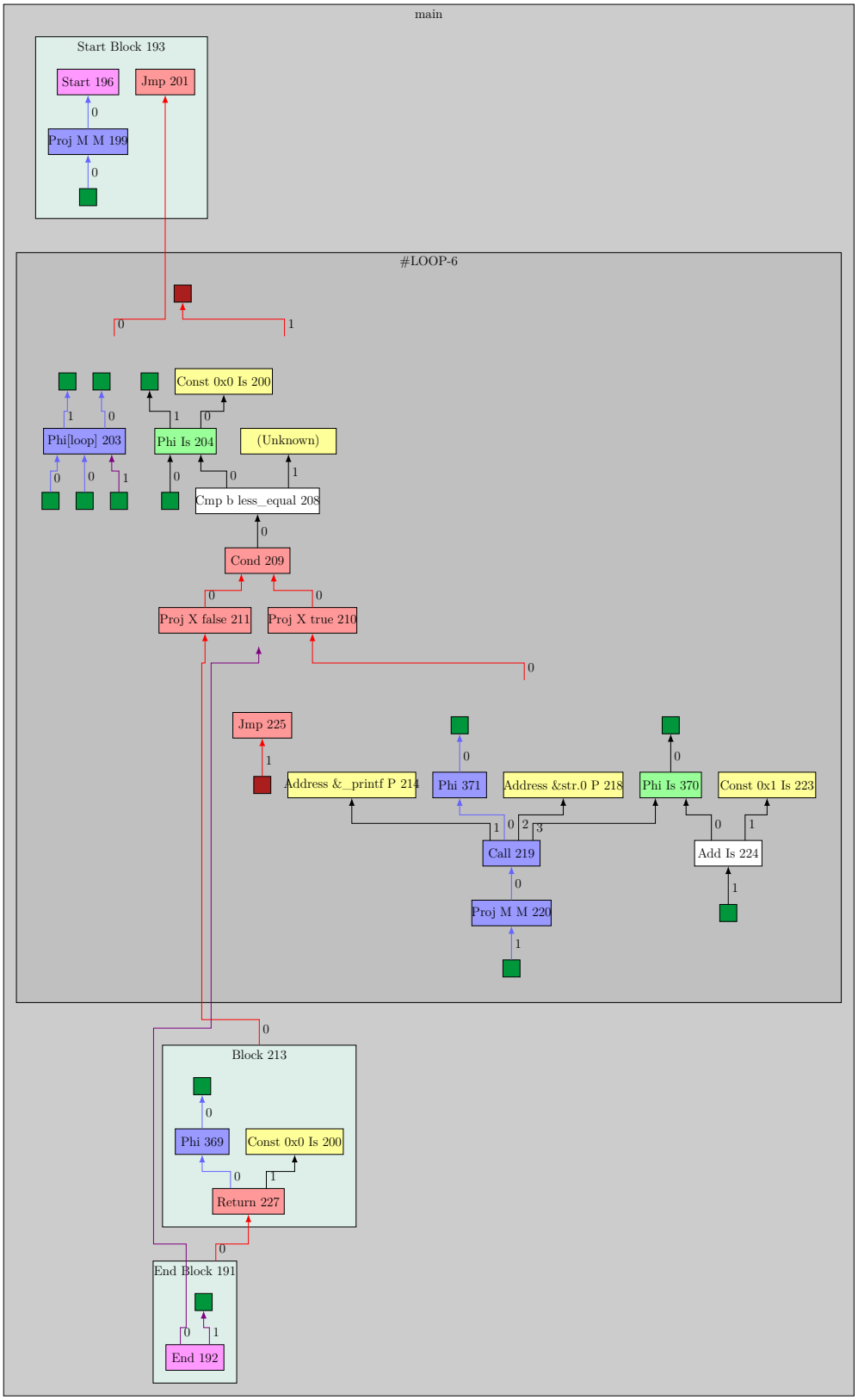

Abbildung 3.2.: Firm graph of a loop with an unknown bound

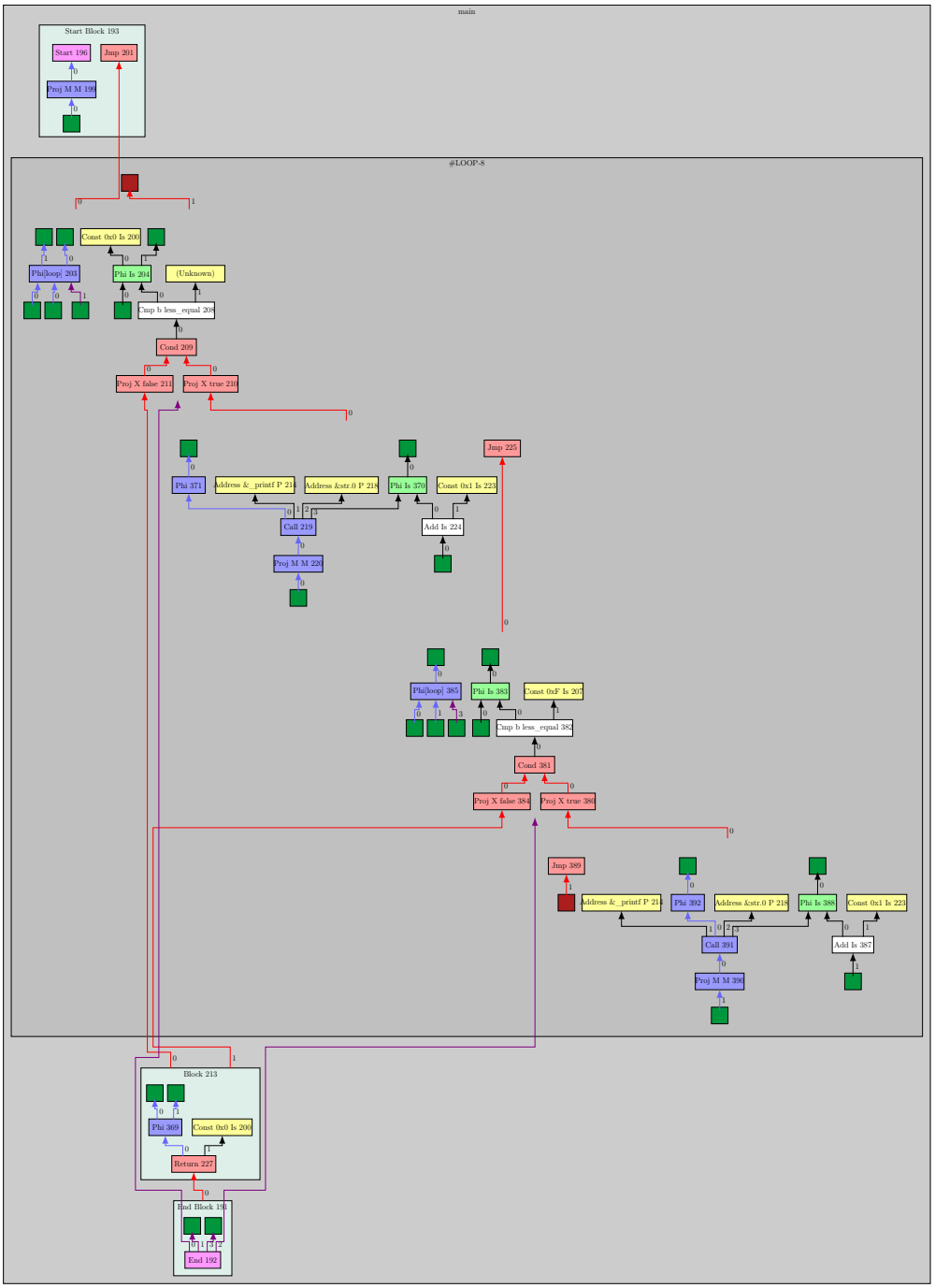

Abbildung 3.3.: Firm graph of the loop shown in figure 3.2 unrolled with a factor of two

4. Evaluation

Hier wird nun das Ergebnis des vorherigen Kapitels kritisch betrachtet. Verbesserungen werden anhand von konkreten Experimenten und Zahlen belegt. Eine saubere statistische Auswertung ist das Ziel.

Für schöne Tabellen ist das „booktabs“ package zu empfehlen. Ein Beispiel ist in figure 4.1 zu sehen.

Fach	xkcd Comics	SpaSS	σ	p
Informatik	1325	100%	12,3	3%
Physik	1324,31	87%	1,733	0.03%
Geologie	123	23%	1,3	11%
Wirtschaft	5	4%	12	1%
Deutsch	0	101	92,3	33%
Geom. Mittel	743	63%	12,3	3%

Abbildung 4.1.: Eine Beispieltabelle. Man beachte, dass zwischen Datenzeilen keine Linien sind. AuSSerdem ist der Beschreibungstext hier sehr ausführlich, damit der Leser nicht den zugehörigen Abschnitt im FlieSStext finden muss.

Zum Benchmarken empfehlen wir das Tool „temci“ [10] und ein Studium der zugehörigen Bachelorarbeit [11].

5. Fazit und Ausblick

Was bedeuten also die Zahlen aus der Evaluation? In welchen Situationen ist die vorgestellte Lösung empfehlenswert?

Als Ausblick dürfen offene Fragen genannt werden. Aus Zeitgründen bleiben üblicherweise einige interessante Fragen unbeantwortet. Womit könnten spätere Studenten sich beschäftigen?

In diesem Kapitel sind ein bischen persönliche Meinung und Überzeugungen erlaubt. Der Rest der Arbeit sollte so objektiv wie möglich sein.

Literaturverzeichnis

[1] E. Aebi, "Ausrollen von schleifen für zwischensprachen in lcssa-form," June 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, pp. 451–490, Oct. 1991.

[3] U. of Illinois at Urbana-Champaign and L. Team, "Lcssa.cpp."

[4] G. o. tz Lindenmaier, " libFIRM – a library for compiler optimization research implementing FIRM ," Tech. Rep. 2002-5, Sept. 2002.

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

[6] A. Fog, "Optimizing subroutines in assembly language," Apr 2018.

[7] V. Sarkar, "Optimized unrolling of nested loops," *International Journal of Parallel Programming*, vol. 29, pp. 545–581, Oct 2001.

[8] T. Duff, "Tom duff on duff's device," Nov 1983.

[9] mult., "loop-unroll.c."

[10] J. Bechberger, "temci documentation." `http://temci.readthedocs.org/en/latest/`.

[11] J. Bechberger, "Besser benchmarken." `http://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit`, Apr. 2016.

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

A. Sonstiges

A.1. Anmeldung

Üblicherweise melden wir eine Arbeit erst an, wenn der Student mit dem Schreiben begonnen hat, also nach der Implementierung. Das verringert die Bürokratie und den Stress, der mit verpassten Deadlines kommt.

AuSSerdem ist ein Abbrechen nach der Anmeldung ein offizieller Akt für den es wiederum Fristen gibt:

	Abbruchfrist nach Anmeldung
Bachelor	4 Wochen
Master	2 Monate
Diplom	3 Monate

Nach dieser Frist muss die abgebrochene Arbeit mit 5,0 bewertet werden.

Das ISS empfiehlt, dass Studenten sich zusätzlich selbst im Studienportal anmelden. Das könnte die Eintragung der Note beschleunigen.

A.2. Antrittsvortrag

Bei internen Arbeiten jeglicher Art ist ein Antrittsvortrag optional. Bei externen Arbeiten ist ein Antrittsvortrag Pflicht.

Dauer: 15 Minuten + 5 Minuten Fragezeit.

Ein Antrittsvortrag sollte nach der Einarbeitungsphase stattfinden, wenn man einen Überblick hat und weiSS was man vorhat. Im Antrittsvortrag kann man ab-

tasten was Prof. Snelting von dem Thema hält und wo man Schwerpunkte setzen oder erweitern sollte.

A.3. Abgabe

	Dauer	Umfang
Bachelor	4 Monate	30+ Seiten
Master	6 Monate	50+ Seiten
Studienarbeit	3 Monate	30+ Seiten
Diplom	6 Monate	50+ Seiten

Man kann eine "4.0 Bescheinigung"bekommen, bspw. für die Masteranmeldungen.

Abzugeben sind jeweils 4 gedruckte Examplare der Arbeit, das Dokument als pdf Datei und entstandener Code und andere Artefakte. AuSSerdem könnten spätere Studenten dankbar sein für TEX-Sourcen.

Zum Drucken empfehlen wir Katz Copy[1] am Kronenplatz, weil wir in Sachen Qualität dort die besten Erfahrungen gemacht haben. Bitte keine Spiralbindung, da sich das schlecht Stapeln lässt. Farbdruck ist nicht verpflichtend, solange in SchwarzweiSS noch alle Grafiken lesbar sind.

A.4. Abschlussvortrag

Die Abschlusspräsentation dauert für Bachelorarbeiten 15 Minuten zuzüglich mind. 10 Minuten für Fragen. Bei Masterarbeiten sind 20–25 Minuten für den Vortrag vorgesehen.

Der Vortrag soll innerhalb von vier Wochen nach Abgabe erfolgen, entsprechend Prüfungsordnung. Die Arbeit muss mindestens einen Tag vor dem Abschlussvortrag abgegeben sein, damit sich Prof. Snelting vorbereiten kann.

Am besten direkt im AnschluSS den Vortrag ausarbeiten und ein oder zwei Wochen nach Abgabe halten. Der Präsentationstermin muss ein bis zwei Monate im

[1] `http://www.katz-copy.com/`

Voraus geplant werden, denn Prof. Snelting hat üblicherweise einen vollen Terminkalender.

A.5. Gutachten

Der Prüfer erstellt ein Gutachten zur Arbeit. Um das Gutachten einzusehen muss ein Antrag beim Prüfungsamt gestellt werden. Der Betreuer bzw. Prüfer darf das Gutachten nur mit genehmigtem Gutachten zeigen. Mündliche Auskunft zur Note ist allerdings möglich.

A.6. Bewertung

- Diplom- und Masterarbeiten *müssen* eine wissenschaftliche Komponente enthalten. Bachelorarbeit *sollten*, aber zum Bestehen ist es nicht notwendig. Wissenschaftlich ist was über reine Implementierungs- bzw. Softwareentwicklungsaufgaben hinausgeht. Üblicherweise findet man theoretische Betrachtungen zu Korrektheit und Effizienz. Willkürliche Daumenregel: Ohne Formel, keine Wissenschaft.

- Diplom- und Masterarbeiten benötigen eigentlich immer Wissen aus dem Diplom- bzw. Masterstudium. Falls das Wissen aus Vordiplom bzw. Bachelor ausreicht, sollte man nochmal darüber nachdenken.

- Positiv mit der Note korrelieren selbstständiges Arbeiten, regelmäSSige Abstimmung mit dem Betreuer, mehrere Feedbackrunden mit verschiedenen Leuten, mehrmaliges Üben des Abschlussvortrags, Einbringen eigener Ideen, gutes Zuhören und sorgfältiges Debugging.

- Negativ mit der Note korrelieren wochenlanges Pausieren, Ignorieren von Feedback, Deadlines überziehen und Arbeiten im stillen Kämmerchen.

Disclaimer: Nein, es gibt keinen konkreten Notenschlüssel. Die obigen Punkte sind nur grobe Richtlinien und für niemanden in irgendeiner Weise bindend.

A.7. LaTeX Features

A.7.1. Schriftformatierungen

	serif	sans-serif	fixed-width
normal	Medium **Bold**	Medium **Bold**	`Medium Bold`
italic	*Medium **Bold***	*Medium **Bold***	*`Medium Bold`*
slanted	*Medium Bold*	*Medium **Bold***	*`Medium Bold`*
small-capital	Medium **Bold**	Medium **Bold**	`Medium Bold`

Math fonts: $absXYZ$, absXYZ, $\mathbf{absXYZ}$, absXYZ, absXYZ, $\mathtt{absXYZ}$, and $\mathcal{XYZ}$. ▮

A.7.2. Rand und Platz

Viele Benutzer von LaTeX wollen Ränder und SeitengröSSe anpassen. Dazu empfehlen wir erstmal die KOMA Script Dokumentation (`koma-script.pdf`) zu lesen, insbesondere Kapitel 2.2. Bevor man mit `\enlargethispage` oder ähnlichen Tricks anfängt, sollte man `\typearea` anpassen.

Falls die Arbeit auf Englisch verfasst wird, sollte man wissen, dass Absätze im Englischen üblicherweise anders formatiert werden. Im Deutschen macht man eine Leerzeile zwischen Absätzen. Im Englischen wird stattdessen die erste Zeile eines Absatzes eingerückt.