Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

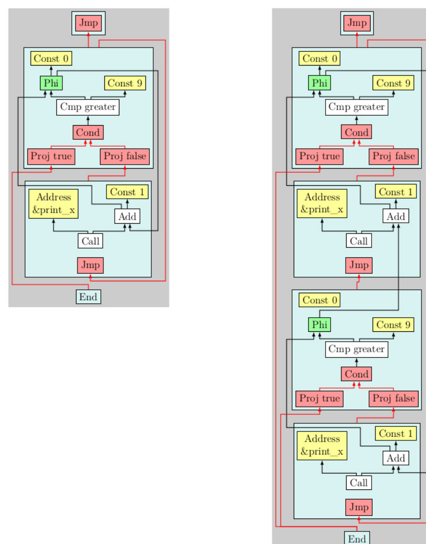
Lehrstuhl Prof. Dr.-Ing. Snelting

Ausrollen von Schleifen für Zwischensprachen in LCSSA-Form

Bachelorarbeit von

Elias Aebi

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 22. Februar 2018 – 21. Juni 2018

Zusammenfassung
Abstract

Das Ausrollen von Schleifen ist eine Optimierungstechnik, die in Compilern eingesetzt wird, um die Performanz von Programmen zu verbessern. Die bisherige Implementierung zum Ausrollen von Schleifen in libFIRM kann Schleifen jedoch nur ausrollen, wenn sie einer bestimmten Form entsprechen. Diese Arbeit stellt eine neue Implementierung vor, die Programme zuerst in LCSSA-Form transformiert, um so Schleifen allgemeiner ausrollen zu können als die bisherige Implementierung. Die Evaluation mittels SPEC CPU2006 zeigt, dass die neue Implementierung die Performanz im Mittel verbessert, jedoch auch in einigen Fällen verschlechtert.

Loop unrolling is a techique that is used in compilers to optimize the performance of generated code. The current implementation of loop unrolling in libFIRM, however, can only unroll certain kinds of loops. This thesis proposes a new implementation that transforms programs into LCSSA form before unrolling in order to be able to unroll loops more generally than the current implementation. Performance evaluation using the SPEC CPU2006 benchmark shows that the new implementation improves the mean performance even though the performance is worse for some cases.

Inhaltsverzeichnis

1 Einführung **7**

2 Grundlagen und Verwandte Arbeiten **9**
2.1 Zwischensprachen . . . 9
2.2 Kontrollflussgraphen . . . 9
2.3 SSA-Form . . . 12
2.4 LCSSA-Form . . . 13
2.5 libFIRM . . . 15
2.6 Ausrollen von Schleifen . . . 17

3 Entwurf und Implementierung **19**
3.1 LCSSA-Form . . . 19
3.2 Ausrollen von Schleifen . . . 20
3.3 Optimierungen . . . 27

4 Evaluation **29**

5 Fazit und Ausblick **41**

1 Einführung

Ein Compiler übersetzt Programmcode in Maschinencode. Der dabei erzeugte Maschinencode soll auf dem Rechner, auf dem er ausgeführt wird, möglichst effizient sein. Um dieses Ziel zu verwirklichen, werden Compileroptimierungen eingesetzt. Das Ausrollen von Schleifen ist eine dieser Compileroptimierungen, die den Inhalt von Schleifen vervielfacht, in der Hoffnung, danach Sprünge eliminieren zu können. Wenn beispielsweise die Anzahl Iterationen einer Schleife zur Compilezeit bekannt ist, kann der Inhalt der Schleife entsprechend dieser Anzahl vervielfacht werden, um so die Rücksprünge komplett zu eliminieren. Um das Ausrollen von Schleifen zu vereinfachen, gibt es die Loop-Closed-SSA-Form (LCSSA-Form), eine spezielle Form der Zwischensprachen, die in Compilern eingesetzt wird.

Diese Arbeit beschreibt die Implementierung und Evaluation des Ausrollens von Schleifen mittels LCSSA-Form in libFIRM, einem am Karlsruher Institut für Technologie (KIT) entwickelten Compiler. Die Arbeit ist folgendermaßen strukturiert: Kapitel 2 geht auf die Grundlagen und verwandten Arbeiten ein, auf denen diese Arbeit aufgebaut ist, und Kapitel 3 beschreibt die Implementierung im Detail. Kapitel 4 vergleicht und evaluiert die Laufzeiten verschiedener Programme vor und nach dem Ausrollen von Schleifen für verschiedene Parameter. Kapitel 5 schließt die Arbeit daraufhin mit einem Fazit und einem Ausblick ab.

2 Grundlagen und Verwandte Arbeiten

Dieses Kapitel erklärt die Begriffe, Konzepte und Technologien, die der Arbeit zugrunde liegen und stellt verwandte Arbeiten vor, die diesbezüglich wichtig sind.

2.1 Zwischensprachen

In Compilern wird Programmcode nach dem Parsen intern auf eine Art und Weise repräsentiert, die die Transformationen, die der Compiler danach ausführt, vereinfacht. Diese Repräsentation wird Zwischensprache genannt. Die folgenden Abschnitte gehen näher auf die Eigenschaften solcher Zwischensprachen ein.

2.2 Kontrollflussgraphen

In Zwischensprachen werden direkt aufeinanderfolgende Instruktionen ohne Sprünge in sogenannte Grundblöcke zusammengefasst. Die Sprünge im Programm werden dabei als Kanten zwischen den Grundblöcken modelliert. Dieser Graph bestehend aus den Grundblöcken als Knoten und den Sprüngen als Kanten wird Kontrollflussgraph genannt. Zusätzlich zu den Grundblöcken enthält jeder Kontrollflussgraph noch einen Start- und einen Endknoten.

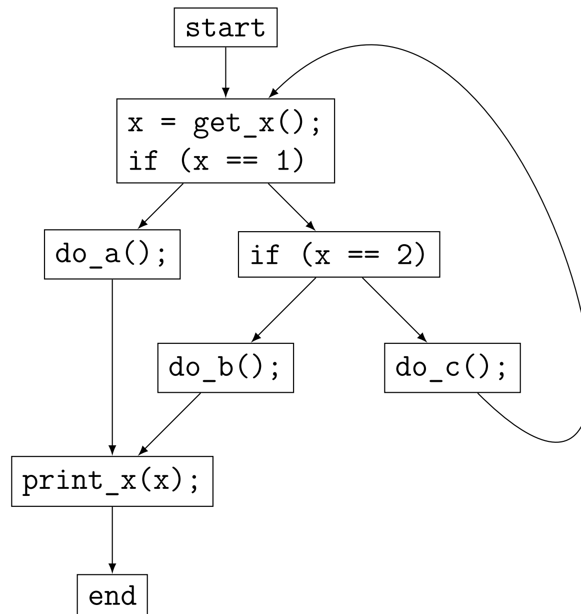
Ein wichtiges Konzept in Kontrollflussgraphen ist die Dominanz. Dabei dominiert ein Knoten a einen Knoten b, wenn jeder Pfad vom Startknoten zu b auch a enthält. Die Dominanzrelation ist reflexiv, da nach Definition jeder Knoten sich selbst dominiert. Die Dominanzrelation ist außerdem transitiv: Wenn jeder Pfad vom Startkonten zu c b enthält und jeder Pfad vom Startknoten zu b a enthält, so muss jeder Pfad vom Startknoten zu c auch a enthalten. Man spricht von strikter Dominanz, wenn ein Knoten a einen Knoten b dominiert und $a \neq b$ gilt. Die direkte Dominanz ist wie folgt definiert: Ein Knoten a dominiert einen Knoten c direkt, wenn a c strikt

dominiert und kein Knoten b existiert, sodass a b strikt dominiert und b c strikt dominiert. Über die direkte Dominanz lässt sich ein Baum definieren, indem man die Knoten des Kontrollflussgraphen nimmt und zwei Knoten mit einer Kante verbindet, genau dann, wenn der eine Knoten den anderen direkt dominiert. Dieser Baum wird Dominatorbaum genannt.

Wenn ein Programm eine Schleife enthält, so ist diese im dazugehörigen Kontrollflussgraphen als Zyklus erkennbar. In dieser Arbeit wird eine Schleife auch genau als ein Zyklus im Kontrollflussgraphen definiert. Falls in der Schleife ein Knoten existiert, der jeden anderen Knoten in der Schleife dominiert, so wird dieser Knoten Schleifenkopf genannt. Der Schleifenkopf ist also der einzige Eintrittspunkt der Schleife. Ein Schleifenkopf muss nicht notwendigerweise existieren, zum Beispiel falls eine Schleife zwei Eintrittspunkte besitzt.

Abbildung 2.1 zeigt ein Beispielprogramm und dessen Kontrollflussgraphen. Das Programm enthält eine Schleife, die aus drei Knoten besteht. Betrachtet man nur den Programmcode, so könnte man denken, dass die beiden Befehle vor den Break-Anweisungen zur Schleife gehören. Dies ist jedoch nicht der Fall, da diese höchstens einmal ausgeführt werden und nicht Teil eines Zyklus sind. Der Knoten, der dem Startknoten folgt, ist der Schleifenkopf. Abbildung 2.2 zeigt den Dominatorbaum für das Programm.

```
int x;
while (true) {
    x = get_x();
    if (x == 1) {
        do_a();
        break;
    } else if (x == 2) {
        do_b();
        break;
    } else {
        do_c();
    }
}
print_x(x);
```



a Programmcode

b Kontrollflussgraph

Abbildung 2.1: Ein Programm und dessen Kontrollflussgraph

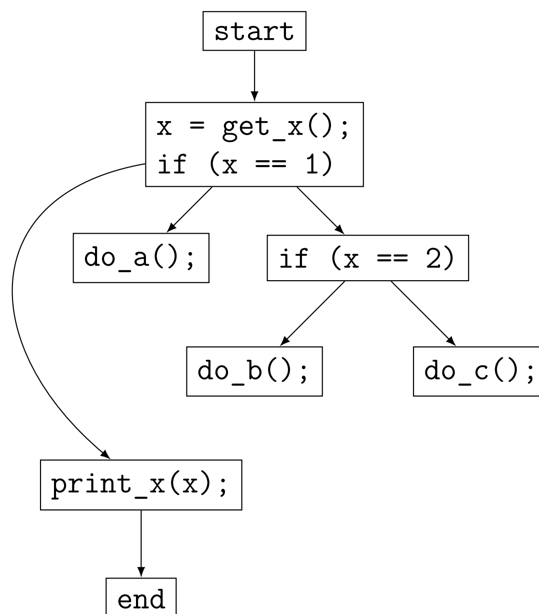


Abbildung 2.2: Der Dominatorbaum zum Kontrollflussgraphen aus Abbildung 2.1

2.3 SSA-Form

Die Static-Single-Assignment-Form (SSA-Form) [1] ist eine spezielle Form einer Zwischensprache. In der SSA-Form darf jeder Variable nur an genau einer Stelle ein Wert zugewiesen werden und diese Zuweisung muss erfolgen, bevor die Variable zum ersten Mal gelesen wird. Dies bedeutet, dass der Block der Definition den Block der Verwendung dominieren muss. Um trotz dieser Einschränkung Verzweigungen im Kontrollfluss und insbesondere Schleifen darstellen zu können, gibt es Phi-Knoten. Ein Phi-Knoten ist ein Knoten mit der gleichen Anzahl an Vorgängern wie der Block, in dem sich der Phi-Knoten befindet. Der Phi-Knoten nimmt dabei seinen Wert abhängig davon an, aus welchem Block in den Block des Phi-Knotens gesprungen wurde. Für Phi-Knoten gilt, dass der Block des Operanden des Phi-Knotens den entsprechenden Vorgänger des Blocks, in dem sich der Phi-Knoten befindet, dominieren muss.

Abbildung 2.3 zeigt ein Beispielprogramm, das eine Schleife enthält, und wie das Programm in SSA-Form aussieht. Um den Wert x in jeder Iteration der Schleife inkrementieren zu können, muss ein Phi-Knoten eingefügt werden, der für die erste Iteration den Wert 0 annimmt und für alle folgenden Iterationen den Wert der Addition aus der vorhergehenden Iteration.

```
x = 0;
while (true) {
    if (x > 9) {
        break;
    }
    x = x + 1;
    print_x(x);
}
```

a Ein Programm

```
x0 = 0;
while (true) {
    x1 = phi(x0, x2);
    if (x1 > 9) {
        break;
    }
    x2 = x1 + 1;
    print_x(x2);
}
```

b Das Programm in SSA-Form

Abbildung 2.3: Ein Programm und dessen Transformation in SSA-Form

2.4 LCSSA-Form

Die LCSSA-Form [2, 3] ist eine weitere Einschränkung der SSA-Form. Sie besagt, dass es für jeden Wert, der innerhalb einer Schleife definiert wird und außerhalb der Schleife verwendet wird, einen Phi-Knoten geben muss, der sich direkt im Ausgangsblock (in dem Block, zu dem aus der Schleife herausgesprungen wird) befindet. Außer diesen Phi-Knoten darf es außerhalb der Schleife keine weiteren Verwender geben.

Diese Einschränkung führt dazu, dass bei Manipulationen der Schleife (wie zum Beispiel dem Ausrollen der Schleife) die Phi-Knoten in Ausgangsblöcken abgesehen von den Ausgangsblöcken selbst die einzigen Knoten außerhalb der Schleife sind, die angepasst werden müssen, was diese Manipulationen unter Umständen stark vereinfacht.

Abbildung 2.4 zeigt das Programm aus Abbildung 2.1 in LCSSA-Form. Das Programm enthält zwei Ausgangsblöcke, in denen jeweils ein Phi-Knoten eingefügt werden muss.

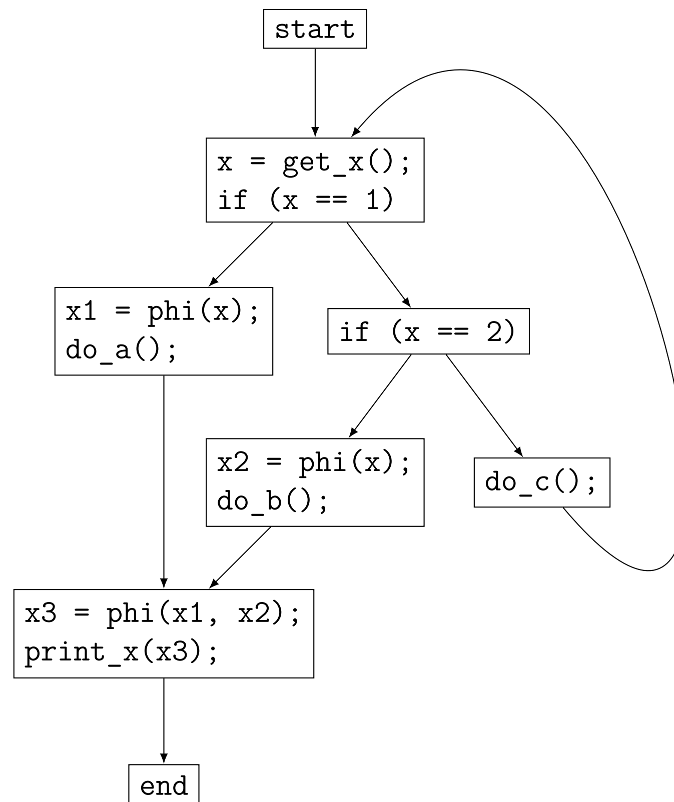


Abbildung 2.4: Das Programm aus Abbildung 2.1 in LCSSA-Form

2.5 libFIRM

libFIRM [4, 5] ist eine C-Programmbibliothek, mit deren Hilfe Programme abgebildet, umgeformt und in ausführbaren Maschinencode umgewandelt werden können. libFIRM wird seit 1996 am KIT entwickelt. Zusammen mit dem cparser [6] ist libFIRM ein vollständiger Compiler, der C-Quellcode in Maschinencode übersetzen kann.

Programme werden in libFIRM in einer Graphen-basierten Zwischensprache in SSA-Form dargestellt. Anders als bei einem normalen Kontrollflussgraphen zeigen in libFIRM die Kontrollflusskanten in die dem Kontrollfluss entgegengesetzte Richtung. Zusätzlich zu den Knoten des Kontrollflussgraphen ist in libFIRM jede Instruktion ein eigener Knoten. Zwischen diesen Knoten gibt es verschiedene Typen von Kanten. Die graphische Repräsentation der Zwischensprache weist den verschiedenen Kantentypen verschiedene Farben zu: Datenabhängigkeiten werden schwarz gezeichnet, Kontrollflusskanten werden rot gezeichnet und Speicherabhängigkeiten werden blau gezeichnet. Knoten, die sonst nicht vom Endknoten erreichbar wären (wie zum Beispiel Endlosschleifen), werden durch eine violette Kante vom Endknoten aus mit diesem verbunden.

In libFIRM besitzt jeder Knoten ein sogenanntes Link-Feld, mit dessen Hilfe ein Knoten mit einem beliebigen anderen Knoten verknüpft werden kann. Dieses Feld ist sehr hilfreich für die Implementierung von Algorithmen und wird auch in der Implementierung dieser Arbeit verwendet.

Abbildung 2.5 zeigt, wie das Programm aus Abbildung 2.3 in libFIRM dargestellt wird. Der Übersichtlichkeit halber wurden Speicherabhängigkeiten und Speicher-spezifische Knoten weggelassen. Die Grundblöcke sind hier hellblau hinterlegt.

```
x0 = 0;
while (true) {
    x1 = phi(x0, x2);
    if (x1 > 9) {
        break;
    }
    x2 = x1 + 1;
    print_x(x2);
}
```

a Das Programm aus Abbildung 2.3 in SSA-Form

Jmp
Const 0
Phi
Const 9
Cmp greater
Cond
Proj true
Proj false
Const 1
Address &print_x
Add
Call
Jmp
End

b Repräsentation des Programms in libFIRM

Abbildung 2.5: Ein Programm und dessen Repräsentation in libFIRM

2.6 Ausrollen von Schleifen

Das Ausrollen von Schleifen ist eine Technik in der Compileroptimierung, bei der der Inhalt einer Schleife mehrfach dupliziert wird. Die Anzahl Instanzen der Schleife nach dem Ausrollen wird Ausroll-Faktor genannt: Ein Ausroll-Faktor n bedeutet, dass die Schleife $n - 1$ mal dupliziert wird. Das Ausrollen von Schleifen ermöglicht in vielen Fällen weitere Optimierungen, wie das Entfernen von bedingten Sprüngen, und führt so zu einer verbesserten Performanz des Programms auf Kosten der Größe des erzeugten Codes.

Abbildung 2.6 zeigt das Programm aus Abbildung 2.4 nachdem es zweifach ausgerollt wurde. Das Beispiel zeigt sehr schön, inwiefern die LCSSA-Form das Ausrollen von Schleifen vereinfacht. Ohne LCSSA-Form müsste man sich hier überlegen, wie man im Block vor dem Endblock den richtigen Wert für x erhält.

In libFIRM existiert bereits eine Implementierung für das Ausrollen von Schleifen [7], welche jedoch nicht die LCSSA-Form verwendet und Schleifen nur ausrollen kann, wenn sie eine Zählvariable besitzen, die in jeder Iteration um einen konstanten Wert erhöht wird und deren Abbruchbedingung nur die Zählvariable mit einer weiteren Variablen oder Konstanten vergleicht.

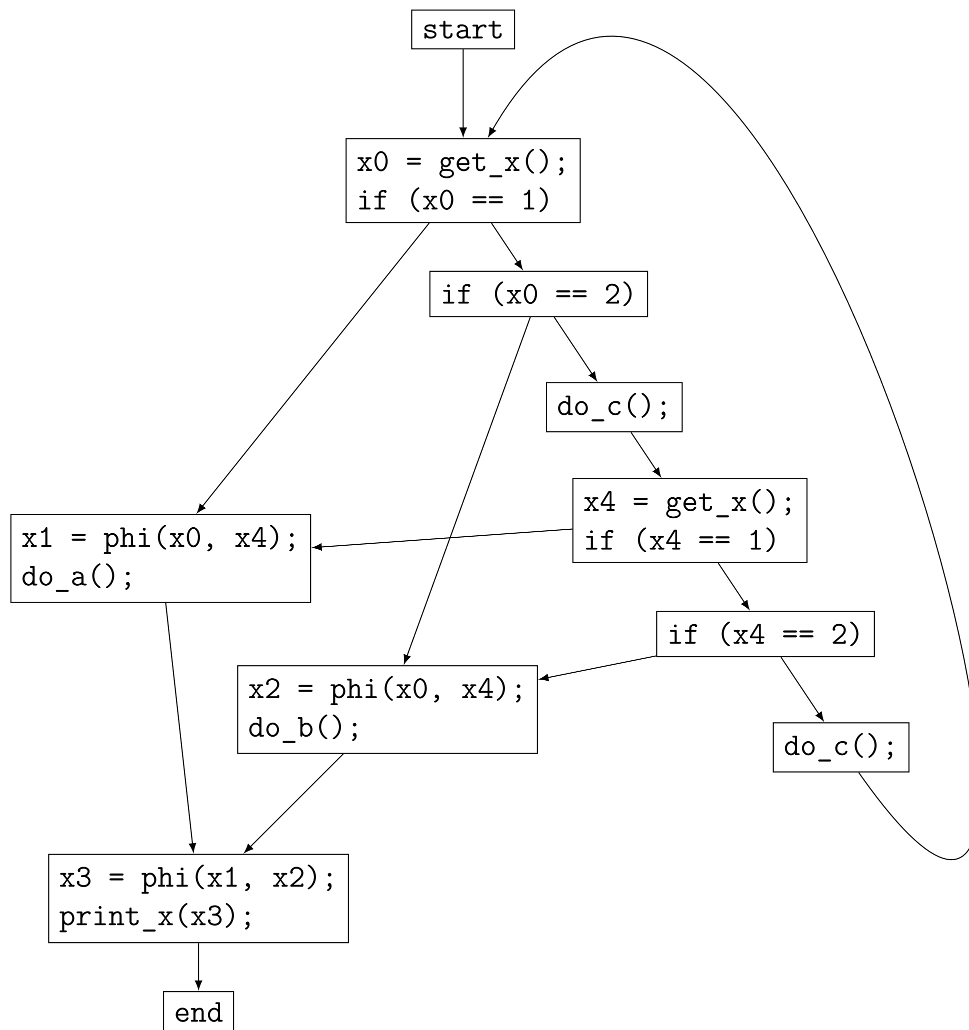


Abbildung 2.6: Das Programm aus Abbildung 2.4 zweifach ausgerollt

3 Entwurf und Implementierung

Dieses Kapitel ist in drei Abschnitte gegliedert. Zuerst wird die Transformation eines Programms in LCSSA-Form erläutert und anschließend darauf aufbauend das Ausrollen von Schleifen. Schließlich wird noch eine zusätzliche Optimierung beschrieben.

3.1 LCSSA-Form

Im Folgenden wird der Algorithmus 1 vorgestellt, der ein Programm, das sich in SSA-Form befindet, so transformiert, dass es sich danach in LCSSA-Form befindet. Der rekursive Algorithmus ist an den in [8] vorgestellten Algorithmus zur Konstruktion der SSA-Form angelehnt. Es wird jede Datenfluss-Kante, die von außerhalb einer Schleife in die Schleife hineinführt, einzeln betrachtet und der Algorithmus wird für jede dieser Kanten ausgeführt.

Der Algorithmus 1 beginnt beim Block, der die Verwendung der Variable enthält (beziehungsweise beim entsprechenden Vorgänger des Blocks falls die Verwendung ein Phi-Knoten ist) und traversiert dann mittels Tiefensuche alle Blöcke bis zum Block in der Schleife, der die Definition der Variable enthält. Da sich das Programm in SSA-Form befindet, muss der Block der Definition den Block der Verwendung dominieren und somit wird der Block der Definition in der Tiefensuche auch immer erreicht, bevor der Startblock erreicht wird. In jedem dieser traversierten Blöcke wird ein Phi-Knoten eingefügt. Um eine Endlosrekursion in der Tiefensuche zu vermeiden, werden die Phi-Knoten eingefügt, bevor die Vorgängerblöcke besucht werden, und die Rekursion wird abgebrochen, wenn in einem Block bereits ein Phi-Knoten für die Kante eingefügt wurde.

Da der Algorithmus in allen traversierten Blöcken Phi-Knoten einfügt und nicht nur an den Schleifengrenzen, sind im Allgemeinen einige der eingefügten Phi-Knoten für die LCSSA-Form unnötig. Das Entfernen dieser Phi-Knoten würde jedoch die Komplexität des Algorithmus erhöhen und wurde aus Zeitgründen nicht umgesetzt.

Die von dem Algorithmus 1 benötigte Funktionalität, Schleifen in einem Programm

zu finden, wird bereits von libFIRM zur Verfügung gestellt.

```
Function assureLCSSA()
  foreach edge in dataflow edges do
    if edge.def is inside loop and edge.use is outside loop then
      block ← getBlock(edge.use)
      if edge.use is phi then
        block ← block.predecessors[index of edge]
      end
      edge.def ← insertPhisRecursive(edge, block)
    end
  end
end

Function insertPhisRecursive(edge, block)
  if getBlock(edge.def) = block then
    return edge.def
  end
  if phis[edge][block] then
    return phis[edge][block]
  end
  phi ← new phi node in block
  phis[edge][block] ← phi
  foreach pblock in block.predecessors do
    phi.predecessors.pushBack(insertPhisRecursive(edge, pblock))
  end
  return phi
end
```

Algorithmus 1: Transformation eines Programms in LCSSA-Form

3.2 Ausrollen von Schleifen

Sobald sich eine Schleife in LCSSA-Form befindet, kann der folgende Algorithmus 2 ausgeführt werden, der die Schleife ausrollt.

Der Algorithmus arbeitet ausschließlich auf innersten Schleifen (Schleifen, die selbst keine Schleifen enthalten). Für Schleifen, die innere Schleifen enthalten, müssten die Rücksprungkanten der inneren und der äußeren Schleife unterschieden werden. Der hier vorgestellte Algorithmus verzichtet auf diese Unterscheidung, was den Algorithmus vereinfacht. Im Normalfall will man auch nur innerste Schleifen ausrollen,

da innere Schleifen meist öfter durchlaufen werden und somit mehr vom Ausrollen profitieren.

Bevor jedoch der eigentliche Algorithmus beginnt, muss der Schleifenkopf bestimmt werden. Dazu läuft man, beginnend bei einem beliebigen Grundblock in der Schleife, den Dominatorbaum hoch, bis der letzte Knoten gefunden wurde, der sich noch in der Schleife befindet. Wenn dieser Knoten jeden anderen Knoten in der Schleife dominiert, so hat man den Schleifenkopf gefunden. Anderenfalls wird der komplette Algorithmus abgebrochen und die Schleife wird nicht ausgerollt.

Im folgenden Algorithmus 2 wird eine Schleife iterativ entsprechend dem Ausroll-Faktor n-fach ausgerollt. Die folgenden Schritte werden also $n-1$ mal ausgeführt.

Zuerst wird jeder Knoten (also auch jeder Grundblock), der sich in der ursprünglichen Schleife befindet, dupliziert und der alte Knoten wird über das Link-Feld mit seinem Duplikat verknüpft. Somit ist sichergestellt, dass das Duplikat erreichbar bleibt. Zusätzlich wird auch das Duplikat mit seinem Original verknüpft. Dies hat zur Folge, dass das Link-Feld für einen Knoten genau dann gesetzt ist, wenn sich ein Knoten in der ursprünglichen Schleife oder in einem ihrer Duplikate befindet. Somit kann über das Link-Feld festgestellt werden, ob sich ein Knoten in der Schleife befindet. Falls aus der vorhergehenden Iteration bereits ein Duplikat besteht, wird dieses ebenfalls mit dem neuen Duplikat verlinkt. Dieser Schritt stellt sicher, dass auch die Vorgänger der ursprünglichen Schleife, die sich in der Schleife befinden, korrekt mit den neuen Duplikaten verknüpft sind, was für den nächsten Schritt essenziell ist.

Als nächster Schritt müssen die Kanten angepasst werden, wie in Algorithmus 3 und Algorithmus 4 beschrieben. Dabei wird über die Knoten der ursprünglichen Schleife iteriert und es werden drei Fälle unterschieden: Entweder handelt es sich bei dem Knoten um den Schleifenkopf, um einen Phi-Knoten im Schleifenkopf oder um einen anderen Knoten.

Für den Schleifenkopf werden die Kanten so abgeändert, dass die neuen Knoten auf die alten zeigen und die alten auf die neuen. Dieser Schritt ist notwendig, damit die ausgerollte Schleife tatsächlich nur eine Schleife ist und nicht zwei einzelne unabhängige Schleifen bildet. Dabei müssen jedoch Vorgänger beachtet werden, die außerhalb der Schleife liegen, also die Sprungkanten von außen in die Schleife hinein. Diese Vorgänger sollen nur bei den alten Knoten erhalten bleiben, denn der alte Schleifenkopf soll auch weiterhin der Schleifenkopf der ausgerollten Schleife sein.

Da Phi-Knoten die gleiche Anzahl an Vorgängern haben müssen wie der Block, in dem sie sich befinden, werden Phi-Knoten in Schleifenköpfen analog zu den Schleifenköpfen behandelt. Es muss jedoch zwischen den Vorgängern der Phi-Knoten und den entsprechenden Vorgängern der Blöcke unterschieden werden, da es vorkommen kann, dass der Vorgänger des Blockes innerhalb der Schleife liegt, der Vorgänger des

Phi-Knotens jedoch nicht.

Alle anderen Knoten werden wie folgt behandelt: Für jede Kante, die von einem alten Knoten ausgeht, erhält der neue Knoten eine Kante. Falls sich der Vorgänger des alten Knotens in der Schleife befindet, so zeigt die Kante des neuen Knotens auf das Duplikat des Vorgängers. Anderenfalls zeigt die Kante auf den gleichen Vorgänger wie die Kante des alten Knotens. Die Kanten der alten Knoten werden in diesem Fall nicht verändert.

Als letzter Schritt müssen zusätzlich zu den Kanten innerhalb der Schleife auch die Kanten, die von außerhalb in die Schleife hineinführen, angepasst werden. Dieser Schritt wird in Algorithmus 5 beschrieben. Da sich das Programm in LCSSA-Form befindet, müssen nur die Ausgangsblöcke, in die aus der Schleife herausgesprungen wird, und die Phi-Knoten in diesen Blöcken betrachtet werden. Für jede dieser Sprungkanten wird im betreffenden Block und jedem seiner Phi-Knoten eine neue Kante eingefügt, die auf den neuen Knoten zeigt, sodass auch aus den neu duplizierten Knoten in diesen Block gesprungen wird und die Phi-Knoten in diesem Fall den richtigen Wert annehmen.

Zusätzlich zum Ausroll-Faktor kann das Ausrollen noch durch einen weiteren Parameter beeinflusst werden. Dieser Parameter gibt eine maximale Schleifengröße an. Eine Schleife wird dann nur ausgerollt, wenn sie kleiner ist als die maximale Größe. Die Größe einer Schleife wird hierbei mit der Anzahl Knoten, die sie enthält, gemessen. Die Überlegung dabei ist, dass es sich bei kleineren Schleifen eher lohnt, diese auszurollen, da der Rücksprung einen größeren Prozentsatz der Gesamtzeit, die in der Schleife verbracht wird, einnimmt. Analog ergibt es im Allgemeinen wenig Sinn, große Schleifen auszurollen, da diese den erzeugten Maschinencode stärker vergrößern und so tendenziell eher für mehr Cache-Misses als für eine Verbesserung der Performanz sorgen. Diese beiden Parameter können dem Compiler als Kommandozeilenargumente übergeben werden.

```
Function unrollLoop(loop, factor)
    clear links
    for i ← 1 to factor − 1 do
        duplicateLoop(loop)
        rewire(loop)
    end
end

Function duplicateLoop(loop)
    foreach node in loop do
        newNode ← duplicate node
        if node.link then
            node.link.link ← newNode
        end
        node.link ← newNode
        newNode.link ← node
    end
end
```

Algorithmus 2: Ausrollen von Schleifen. Die Funktion rewire wird in Algorithmus 3 beschrieben.

```
Function rewire(loop)
    foreach node in loop do
        if node is loop header then
            rewireLoopHeader(node, loop)
        else if node is phi in loop header then
            rewireLoopHeaderPhi(node, loop)
        else
            rewireOtherNode(node, loop)
        end
    end
    rewireSuccessorBlocks(loop)
end
```

Algorithmus 3: Anpassen der Kanten. Die Hilfsfunktionen werden in Algorithmus 4 und Algorithmus 5 beschrieben.

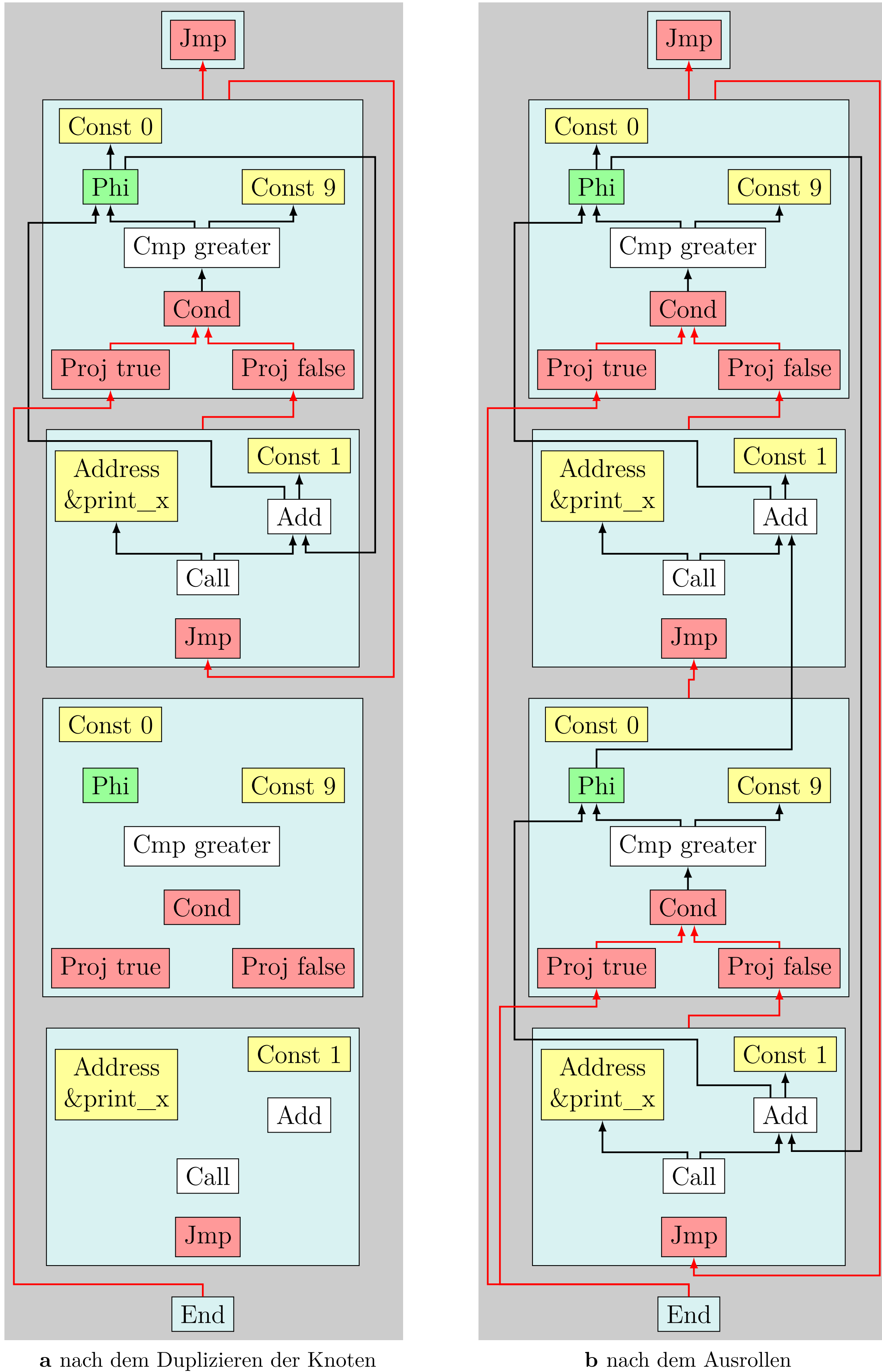
```
Function rewireLoopHeader(block, loop)
    preds ← []
    newPreds ← []
    foreach pred in block.predecessors do
        if pred.link then
            preds.pushBack(pred.link)
            newPreds.pushBack(pred)
        else
            preds.pushBack(pred)
        end
    end
    block.predecessors ← preds
    block.link.predecessors ← newPreds
end

Function rewireLoopHeaderPhi(node, loop)
    preds ← []
    newPreds ← []
    foreach pred in node.predecessors do
        predBlock ← getBlock(node).predecessors[index of pred]
        if predBlock.link then
            if pred.link then
                preds.pushBack(pred.link)
            else
                preds.pushBack(pred)
            end
            newPreds.pushBack(pred)
        else
            preds.pushBack(pred)
        end
    end
    node.predecessors ← preds
    node.link.predecessors ← newPreds
end

Function rewireOtherNode(node, loop)
    newPreds ← []
    foreach pred in node.predecessors do
        if pred.link then
            newPreds.pushBack(pred.link)
        else
            newPreds.pushBack(pred)
        end
    end
    node.link.predecessors ← newPreds
end
```

Algorithmus 4: Funktionen für das Anpassen der Kanten

```
Function rewireSuccessorBlocks(loop)
    foreach edge in controlflow edges do
        if edge.dest is outside loop and edge.source is inside loop then
            edge.dest.predecessors.pushBack(edge.source.link)
            foreach phi in edge.dest do
                phiPred ← phi.predecessors[index of edge]
                if phiPred.link then
                    phi.predecessors.pushBack(phiPred.link)
                else
                    phi.predecessors.pushBack(phiPred)
                end
            end
        end
    end
end
```

Algorithmus 5: Funktion für das Anpassen der Kanten in Nachfolger-Blöcken

Um die Funktionsweise des Algorithmus zu verdeutlichen, wird der Algorithmus im Folgenden beispielhaft für einen Ausroll-Faktor von 2 auf Abbildung 2.5 angewandt. Die Schleife besteht hier aus den beiden Blöcken, die nicht der Start- und nicht der Endblock sind. Der erste Block in der Schleife ist der Schleifenkopf. Das Programm befindet sich bereits in LCSSA-Form, da keine Datenabhängigkeiten von außerhalb der Schleife in die Schleife hineinführen. Somit entfällt der Schritt zur Transformation in LCSSA-Form. Der erste Schritt ist also das Duplizieren aller Knoten in der Schleife. Abbildung 3.1a zeigt das Programm nach dem ersten Schritt. Nun müssen noch die Kanten angepasst werden. Der Schleifenkopf besitzt zwei Vorgänger, von denen einer (der Startblock) außerhalb der Schleife liegt. Somit erhält der alte Schleifenkopf zwei Vorgänger: den Startblock und das Duplikat des zweiten Blocks in der Schleife. Das Duplikat des Schleifenkopfes erhält nur den alten zweiten Block als Vorgänger. Der Phi-Knoten im Schleifenkopf wird analog abgehandelt: Der alte Phi-Knoten erhält die Konstante 0 und die neue Addition als Vorgänger und der neue Phi-Knoten nur die alte Addition. Für alle anderen Knoten in der Schleife werden alle Kanten dupliziert, sodass die neuen Kanten von neuen auf neue Knoten zeigen. Der Endknoten ist der einzige Knoten außerhalb der Schleife mit einer Kante in die Schleife. Dieser erhält eine zusätzliche Kante auf das Duplikat des Proj-True-Knotens. Abbildung 3.1b zeigt das Resultat.



a nach dem Duplizieren der Knoten

b nach dem Ausrollen

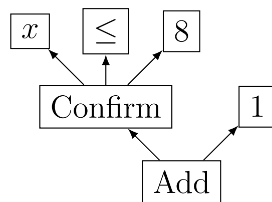
Abbildung 3.1: beispielhafte Anwendung des Algorithmus

3.3 Optimierungen

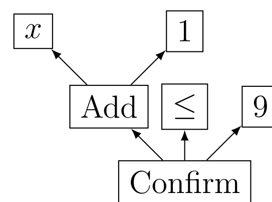
Neben den Algorithmen zur Transformation in LCSSA-Form und dem Ausrollen wurde noch eine zusätzliche Optimierung implementiert, die es ermöglicht, in einfachen Schleifen nach dem Ausrollen die zusätzlichen Sprünge zu eliminieren.

In libFIRM gibt es sogenannte Confirm-Knoten, die bestätigen, dass ein Wert eine bestimmte Bedingung erfüllt. Der Confirm-Knoten selbst hat den gleichen Wert wie der Vorgänger, über den eine Aussage getroffen wird. Die neu hinzugefügte Optimierung betrachtet Additionsknoten, die einen Confirm-Knoten und eine Konstante als Vorgänger haben und stellt diese so um, dass der Confirm-Knoten den Additionsknoten als Vorgänger hat. Anschaulich wird also die Bestätigung, dass ein Wert x vor der Addition einer Konstanten z kleiner oder gleich y ist, zu einer Bestätigung umgeformt, dass der Wert der Addition $x + z$ kleiner oder gleich $y + z$ ist. Die Optimierung wird nur durchgeführt, wenn $y + z$ keinen Überlauf generiert.

Für eine einfache Schleife wie die in Abbildung 2.5, die eine konstante gerade Anzahl mal durchlaufen wird, indem eine Zählvariable in jeder Iteration um 1 erhöht wird, bewirkt die Optimierung folgendes: Wenn diese Schleife wie in Abbildung 3.1b ausgerollt wird, dann erkennt libFIRM, dass der Wert des ersten Phi-Knotens immer gerade ist (beziehungsweise dass das niederwertigste Bit immer 0 ist), da der Wert zu Beginn mit 0 initialisiert wird und dann immer jeweils 2 mal um 1 erhöht wird. Zudem wird erkannt, dass der Wert im zweiten Block der Schleife kleiner oder gleich 9 sein muss, da anderenfalls ja aus der Schleife herausgesprungen würde. Da der Wert aber gerade ist, kann er nicht 9 sein und ist demzufolge kleiner oder gleich 8. Somit fügt libFIRM hier vor der Addition einen Confirm-Knoten ein, der bestätigt, dass der Wert kleiner oder gleich 8 ist. Die hier beschriebene Optimierung zieht den Confirm-Knoten durch die Addition durch, sodass der Confirm-Knoten nun besagt, dass der Wert der Addition kleiner oder gleich 9 ist. Dies ermöglicht nun libFIRM den der Addition folgenden Vergleich zu entfernen und den bedingten Sprung durch einen unbedingten Sprung zu ersetzen, da das Ergebnis des Vergleichs offensichtlich nie wahr ist. Abbildung 3.2 zeigt die Transformation dieses Beispiels graphisch.



a vor der Optimierung



b nach der Optimierung

Abbildung 3.2: Ein Ausschnitt eines Programms vor und nach der Optimierung

4 Evaluation

Zur Evaluation der Arbeit wird der Benchmark SPEC CPU2006 [9] verwendet. Dieser wird auf einem Intel Core i7-6700 Prozessor mit 3,4 GHz Taktfrequenz und 64 GB RAM unter Ubuntu 16.04.4 ausgeführt. Es wird das x86-Backend von libFIRM verwendet.

Es werden die Laufzeiten für verschiedene Werte der Parameter Ausroll-Faktor und maximale Schleifengröße gemessen. Jedes Programm wird 10 mal ausgeführt. Die Laufzeiten sind das arithmetische Mittel der 10 Ausführungszeiten in Sekunden. Wie in [10] empfohlen, wird für jedes Mittel die Standardabweichung σ angegeben, um die Aussagekraft der Ergebnisse einschätzen zu können. Die Referenzwerte sind die Laufzeiten für den Fall, dass keine Schleifen ausgerollt werden. Für jede Konfiguration ist schließlich noch das geometrische Mittel der Verhältnisse von Laufzeiten zu Referenz-Laufzeiten angegeben, um allgemeinere Aussagen über die Konfigurationen treffen zu können.

Es wird erwartet, dass sich die Laufzeiten für verschiedene Konfigurationen verschieden stark verbessern und für manche Konfigurationen verschlechtern, da die Codegröße zu sehr zunimmt. Das Ziel ist es, eine Konfiguration zu finden, in der die Performanzgewinne klar sichtbar sind.

Es folgen die Messergebnisse und deren Evaluation.

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	254.874 s	1.321 s	254.025 s	1.056 s	100.33%
401.bzip2	340.562 s	2.015 s	339.501 s	2.716 s	100.31%
403.gcc	177.818 s	7.362 s	177.011 s	2.045 s	100.46%
429.mcf	136.280 s	1.824 s	136.314 s	1.761 s	99.98%
433.milc	435.410 s	35.596 s	431.110 s	11.625 s	101.00%
445.gobmk	351.418 s	0.648 s	351.002 s	0.615 s	100.12%
456.hmmer	549.442 s	0.442 s	551.022 s	0.317 s	99.71%
458.sjeng	384.749 s	1.192 s	388.709 s	1.598 s	98.98%
462.libquantum	295.388 s	2.754 s	298.711 s	2.353 s	98.89%
464.h264ref	404.244 s	1.105 s	406.405 s	1.413 s	99.47%
470.lbm	230.441 s	2.534 s	230.106 s	0.884 s	100.15%
482.sphinx3	488.370 s	15.037 s	488.693 s	16.519 s	99.93%
Geom. Mittel					99.94%

Abbildung 4.1: Ausroll-Faktor 2, maximale Schleifengröße 32

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	253.912 s	1.282 s	254.025 s	1.056 s	99.96%
401.bzip2	340.594 s	1.764 s	339.501 s	2.716 s	100.32%
403.gcc	172.826 s	1.631 s	177.011 s	2.045 s	97.64%
429.mcf	137.614 s	1.546 s	136.314 s	1.761 s	100.95%
433.milc	432.736 s	14.264 s	431.110 s	11.625 s	100.38%
445.gobmk	349.877 s	1.378 s	351.002 s	0.615 s	99.68%
456.hmmer	549.267 s	0.362 s	551.022 s	0.317 s	99.68%
458.sjeng	383.834 s	1.038 s	388.709 s	1.598 s	98.75%
462.libquantum	300.661 s	1.201 s	298.711 s	2.353 s	100.65%
464.h264ref	403.539 s	2.209 s	406.405 s	1.413 s	99.29%
470.lbm	230.792 s	2.352 s	230.106 s	0.884 s	100.30%
482.sphinx3	479.252 s	3.217 s	488.693 s	16.519 s	98.07%
Geom. Mittel					99.63%

Abbildung 4.2: Ausroll-Faktor 2, maximale Schleifengröße 64

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	254.823 s	0.874 s	254.025 s	1.056 s	100.31%
401.bzip2	346.316 s	3.057 s	339.501 s	2.716 s	102.01%
403.gcc	173.891 s	0.756 s	177.011 s	2.045 s	98.24%
429.mcf	133.131 s	1.450 s	136.314 s	1.761 s	97.66%
433.milc	435.389 s	16.926 s	431.110 s	11.625 s	100.99%
445.gobmk	352.471 s	0.729 s	351.002 s	0.615 s	100.42%
456.hmmer	554.484 s	1.849 s	551.022 s	0.317 s	100.63%
458.sjeng	382.083 s	0.904 s	388.709 s	1.598 s	98.30%
462.libquantum	295.544 s	2.973 s	298.711 s	2.353 s	98.94%
464.h264ref	399.654 s	1.692 s	406.405 s	1.413 s	98.34%
470.lbm	230.978 s	2.878 s	230.106 s	0.884 s	100.38%
482.sphinx3	480.138 s	3.221 s	488.693 s	16.519 s	98.25%
Geom. Mittel					99.53%

Abbildung 4.3: Ausroll-Faktor 2, maximale Schleifengröße 128

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	256.330 s	0.787 s	254.025 s	1.056 s	100.91%
401.bzip2	342.991 s	3.099 s	339.501 s	2.716 s	101.03%
403.gcc	174.202 s	0.820 s	177.011 s	2.045 s	98.41%
429.mcf	131.492 s	0.865 s	136.314 s	1.761 s	96.46%
433.milc	429.181 s	17.962 s	431.110 s	11.625 s	99.55%
445.gobmk	349.336 s	1.423 s	351.002 s	0.615 s	99.53%
456.hmmer	559.510 s	0.524 s	551.022 s	0.317 s	101.54%
458.sjeng	384.473 s	0.828 s	388.709 s	1.598 s	98.91%
462.libquantum	294.900 s	1.319 s	298.711 s	2.353 s	98.72%
464.h264ref	398.416 s	1.840 s	406.405 s	1.413 s	98.03%
470.lbm	230.384 s	1.507 s	230.106 s	0.884 s	100.12%
482.sphinx3	482.827 s	6.512 s	488.693 s	16.519 s	98.80%
Geom. Mittel					99.33%

Abbildung 4.4: Ausroll-Faktor 2, maximale Schleifengröße 256

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	256.238 s	0.874 s	254.025 s	1.056 s	100.87%
401.bzip2	341.754 s	1.276 s	339.501 s	2.716 s	100.66%
403.gcc	177.805 s	1.061 s	177.011 s	2.045 s	100.45%
429.mcf	136.500 s	1.178 s	136.314 s	1.761 s	100.14%
433.milc	433.546 s	8.846 s	431.110 s	11.625 s	100.57%
445.gobmk	350.848 s	1.084 s	351.002 s	0.615 s	99.96%
456.hmmer	554.962 s	0.103 s	551.022 s	0.317 s	100.71%
458.sjeng	386.352 s	0.961 s	388.709 s	1.598 s	99.39%
462.libquantum	298.062 s	1.384 s	298.711 s	2.353 s	99.78%
464.h264ref	401.078 s	1.515 s	406.405 s	1.413 s	98.69%
470.lbm	230.684 s	2.267 s	230.106 s	0.884 s	100.25%
482.sphinx3	486.940 s	14.051 s	488.693 s	16.519 s	99.64%
Geom. Mittel					100.09%

Abbildung 4.5: Ausroll-Faktor 4, maximale Schleifengröße 32

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	257.700 s	0.717 s	254.025 s	1.056 s	101.45%
401.bzip2	340.378 s	1.167 s	339.501 s	2.716 s	100.26%
403.gcc	173.082 s	1.456 s	177.011 s	2.045 s	97.78%
429.mcf	136.446 s	1.110 s	136.314 s	1.761 s	100.10%
433.milc	437.175 s	90.423 s	431.110 s	11.625 s	101.41%
445.gobmk	342.741 s	0.600 s	351.002 s	0.615 s	97.65%
456.hmmer	546.940 s	0.424 s	551.022 s	0.317 s	99.26%
458.sjeng	385.696 s	1.348 s	388.709 s	1.598 s	99.22%
462.libquantum	301.708 s	1.997 s	298.711 s	2.353 s	101.00%
464.h264ref	396.864 s	0.834 s	406.405 s	1.413 s	97.65%
470.lbm	230.747 s	2.320 s	230.106 s	0.884 s	100.28%
482.sphinx3	524.315 s	411.228 s	488.693 s	16.519 s	107.29%
Geom. Mittel					100.25%

Abbildung 4.6: Ausroll-Faktor 4, maximale Schleifengröße 64

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	257.285 s	1.478 s	254.025 s	1.056 s	101.28%
401.bzip2	344.652 s	4.740 s	339.501 s	2.716 s	101.52%
403.gcc	172.927 s	0.726 s	177.011 s	2.045 s	97.69%
429.mcf	134.189 s	0.551 s	136.314 s	1.761 s	98.44%
433.milc	428.900 s	14.825 s	431.110 s	11.625 s	99.49%
445.gobmk	342.173 s	2.851 s	351.002 s	0.615 s	97.48%
456.hmmer	547.934 s	0.170 s	551.022 s	0.317 s	99.44%
458.sjeng	384.521 s	0.944 s	388.709 s	1.598 s	98.92%
462.libquantum	301.560 s	1.360 s	298.711 s	2.353 s	100.95%
464.h264ref	391.880 s	1.616 s	406.405 s	1.413 s	96.43%
470.lbm	230.541 s	2.174 s	230.106 s	0.884 s	100.19%
482.sphinx3	481.278 s	9.200 s	488.693 s	16.519 s	98.48%
Geom. Mittel					99.18%

Abbildung 4.7: Ausroll-Faktor 4, maximale Schleifengröße 128

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	259.402 s	1.256 s	254.025 s	1.056 s	102.12%
401.bzip2	345.802 s	1.953 s	339.501 s	2.716 s	101.86%
403.gcc	172.383 s	0.956 s	177.011 s	2.045 s	97.39%
429.mcf	131.827 s	1.681 s	136.314 s	1.761 s	96.71%
433.milc	412.513 s	9.294 s	431.110 s	11.625 s	95.69%
445.gobmk	341.366 s	0.796 s	351.002 s	0.615 s	97.25%
456.hmmer	554.713 s	0.971 s	551.022 s	0.317 s	100.67%
458.sjeng	385.623 s	0.338 s	388.709 s	1.598 s	99.21%
462.libquantum	301.651 s	3.159 s	298.711 s	2.353 s	100.98%
464.h264ref	389.777 s	1.212 s	406.405 s	1.413 s	95.91%
470.lbm	230.611 s	2.110 s	230.106 s	0.884 s	100.22%
482.sphinx3	483.688 s	8.568 s	488.693 s	16.519 s	98.98%
Geom. Mittel					98.89%

Abbildung 4.8: Ausroll-Faktor 4, maximale Schleifengröße 256

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	263.691 s	0.986 s	254.025 s	1.056 s	103.81%
401.bzip2	339.070 s	1.709 s	339.501 s	2.716 s	99.87%
403.gcc	178.134 s	0.576 s	177.011 s	2.045 s	100.63%
429.mcf	136.807 s	1.514 s	136.314 s	1.761 s	100.36%
433.milc	441.842 s	83.255 s	431.110 s	11.625 s	102.49%
445.gobmk	351.426 s	0.537 s	351.002 s	0.615 s	100.12%
456.hmmer	550.940 s	0.122 s	551.022 s	0.317 s	99.98%
458.sjeng	384.231 s	0.978 s	388.709 s	1.598 s	98.85%
462.libquantum	296.691 s	1.469 s	298.711 s	2.353 s	99.32%
464.h264ref	399.458 s	0.779 s	406.405 s	1.413 s	98.29%
470.lbm	230.415 s	2.068 s	230.106 s	0.884 s	100.13%
482.sphinx3	490.951 s	18.577 s	488.693 s	16.519 s	100.46%
Geom. Mittel					100.35%

Abbildung 4.9: Ausroll-Faktor 8, maximale Schleifengröße 32

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	261.209 s	1.072 s	254.025 s	1.056 s	102.83%
401.bzip2	344.075 s	2.785 s	339.501 s	2.716 s	101.35%
403.gcc	173.697 s	1.143 s	177.011 s	2.045 s	98.13%
429.mcf	136.984 s	1.265 s	136.314 s	1.761 s	100.49%
433.milc	433.031 s	52.494 s	431.110 s	11.625 s	100.45%
445.gobmk	336.590 s	0.886 s	351.002 s	0.615 s	95.89%
456.hmmer	549.509 s	0.232 s	551.022 s	0.317 s	99.73%
458.sjeng	386.522 s	1.475 s	388.709 s	1.598 s	99.44%
462.libquantum	291.270 s	1.346 s	298.711 s	2.353 s	97.51%
464.h264ref	397.474 s	0.722 s	406.405 s	1.413 s	97.80%
470.lbm	230.535 s	2.548 s	230.106 s	0.884 s	100.19%
482.sphinx3	477.730 s	5.258 s	488.693 s	16.519 s	97.76%
Geom. Mittel					99.28%

Abbildung 4.10: Ausroll-Faktor 8, maximale Schleifengröße 64

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	265.561 s	1.040 s	254.025 s	1.056 s	104.54%
401.bzip2	344.751 s	3.143 s	339.501 s	2.716 s	101.55%
403.gcc	173.106 s	0.957 s	177.011 s	2.045 s	97.79%
429.mcf	130.778 s	1.238 s	136.314 s	1.761 s	95.94%
433.milc	425.632 s	10.469 s	431.110 s	11.625 s	98.73%
445.gobmk	342.086 s	0.879 s	351.002 s	0.615 s	97.46%
456.hmmer	539.037 s	0.271 s	551.022 s	0.317 s	97.82%
458.sjeng	386.190 s	17.402 s	388.709 s	1.598 s	99.35%
462.libquantum	288.869 s	0.979 s	298.711 s	2.353 s	96.71%
464.h264ref	390.243 s	2.741 s	406.405 s	1.413 s	96.02%
470.lbm	230.559 s	1.987 s	230.106 s	0.884 s	100.20%
482.sphinx3	480.211 s	12.463 s	488.693 s	16.519 s	98.26%
Geom. Mittel					98.67%

Abbildung 4.11: Ausroll-Faktor 8, maximale Schleifengröße 128

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	264.371 s	1.153 s	254.025 s	1.056 s	104.07%
401.bzip2	347.754 s	3.241 s	339.501 s	2.716 s	102.43%
403.gcc	174.359 s	0.885 s	177.011 s	2.045 s	98.50%
429.mcf	143.208 s	1.524 s	136.314 s	1.761 s	105.06%
433.milc	412.111 s	8.374 s	431.110 s	11.625 s	95.59%
445.gobmk	339.807 s	0.552 s	351.002 s	0.615 s	96.81%
456.hmmer	550.295 s	0.687 s	551.022 s	0.317 s	99.87%
458.sjeng	385.454 s	0.858 s	388.709 s	1.598 s	99.16%
462.libquantum	289.161 s	1.649 s	298.711 s	2.353 s	96.80%
464.h264ref	389.194 s	0.975 s	406.405 s	1.413 s	95.77%
470.lbm	230.290 s	0.363 s	230.106 s	0.884 s	100.08%
482.sphinx3	530.433 s	321.753 s	488.693 s	16.519 s	108.54%
Geom. Mittel					100.15%

Abbildung 4.12: Ausroll-Faktor 8, maximale Schleifengröße 256

Programm	Resultat Laufzeit	Resultat σ	Referenz Laufzeit	Referenz σ	Verhältnis
400.perlbench	269.344 s	1.024 s	254.025 s	1.056 s	106.03%
401.bzip2	338.415 s	1.710 s	339.501 s	2.716 s	99.68%
403.gcc	177.937 s	0.544 s	177.011 s	2.045 s	100.52%
429.mcf	136.583 s	1.680 s	136.314 s	1.761 s	100.20%
433.milc	436.303 s	16.740 s	431.110 s	11.625 s	101.20%
445.gobmk	350.015 s	0.500 s	351.002 s	0.615 s	99.72%
456.hmmer	549.982 s	0.242 s	551.022 s	0.317 s	99.81%
458.sjeng	382.448 s	0.789 s	388.709 s	1.598 s	98.39%
462.libquantum	293.743 s	1.935 s	298.711 s	2.353 s	98.34%
464.h264ref	400.992 s	0.984 s	406.405 s	1.413 s	98.67%
470.lbm	230.410 s	1.093 s	230.106 s	0.884 s	100.13%
482.sphinx3	494.541 s	24.434 s	488.693 s	16.519 s	101.20%
Geom. Mittel					100.31%

Abbildung 4.13: Ausroll-Faktor 16, maximale Schleifengröße 32

Programm	Resultat Laufzeit	Resultat σ	Referenz Laufzeit	Referenz σ	Verhältnis
400.perlbench	266.907 s	1.551 s	254.025 s	1.056 s	105.07%
401.bzip2	344.203 s	4.412 s	339.501 s	2.716 s	101.39%
403.gcc	173.405 s	0.729 s	177.011 s	2.045 s	97.96%
429.mcf	138.327 s	0.534 s	136.314 s	1.761 s	101.48%
433.milc	429.859 s	9.680 s	431.110 s	11.625 s	99.71%
445.gobmk	338.529 s	0.843 s	351.002 s	0.615 s	96.45%
456.hmmer	547.785 s	1.069 s	551.022 s	0.317 s	99.41%
458.sjeng	383.117 s	0.961 s	388.709 s	1.598 s	98.56%
462.libquantum	288.453 s	1.576 s	298.711 s	2.353 s	96.57%
464.h264ref	390.883 s	1.132 s	406.405 s	1.413 s	96.18%
470.lbm	230.449 s	1.996 s	230.106 s	0.884 s	100.15%
482.sphinx3	477.753 s	2.284 s	488.693 s	16.519 s	97.76%
Geom. Mittel					99.19%

Abbildung 4.14: Ausroll-Faktor 16, maximale Schleifengröße 64

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	270.064 s	2.987 s	254.025 s	1.056 s	106.31%
401.bzip2	349.609 s	1.496 s	339.501 s	2.716 s	102.98%
403.gcc	173.725 s	0.710 s	177.011 s	2.045 s	98.14%
429.mcf	143.856 s	0.905 s	136.314 s	1.761 s	105.53%
433.milc	430.020 s	42.107 s	431.110 s	11.625 s	99.75%
445.gobmk	341.850 s	0.726 s	351.002 s	0.615 s	97.39%
456.hmmer	547.992 s	0.333 s	551.022 s	0.317 s	99.45%
458.sjeng	382.448 s	1.448 s	388.709 s	1.598 s	98.39%
462.libquantum	284.095 s	1.344 s	298.711 s	2.353 s	95.11%
464.h264ref	387.168 s	0.971 s	406.405 s	1.413 s	95.27%
470.lbm	230.102 s	0.230 s	230.106 s	0.884 s	100.00%
482.sphinx3	485.914 s	23.144 s	488.693 s	16.519 s	99.43%
Geom. Mittel					99.76%

Abbildung 4.15: Ausroll-Faktor 16, maximale Schleifengröße 128

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	273.671 s	1.762 s	254.025 s	1.056 s	107.73%
401.bzip2	350.806 s	2.596 s	339.501 s	2.716 s	103.33%
403.gcc	175.325 s	2.171 s	177.011 s	2.045 s	99.05%
429.mcf	145.089 s	1.642 s	136.314 s	1.761 s	106.44%
433.milc	412.246 s	7.802 s	431.110 s	11.625 s	95.62%
445.gobmk	342.430 s	1.032 s	351.002 s	0.615 s	97.56%
456.hmmer	548.327 s	0.270 s	551.022 s	0.317 s	99.51%
458.sjeng	382.225 s	0.954 s	388.709 s	1.598 s	98.33%
462.libquantum	284.041 s	1.721 s	298.711 s	2.353 s	95.09%
464.h264ref	388.626 s	1.521 s	406.405 s	1.413 s	95.63%
470.lbm	230.715 s	2.645 s	230.106 s	0.884 s	100.26%
482.sphinx3	488.063 s	9.389 s	488.693 s	16.519 s	99.87%
Geom. Mittel					99.79%

Abbildung 4.16: Ausroll-Faktor 16, maximale Schleifengröße 256

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	274.811 s	1.077 s	254.025 s	1.056 s	108.18%
401.bzip2	339.153 s	1.899 s	339.501 s	2.716 s	99.90%
403.gcc	178.047 s	1.276 s	177.011 s	2.045 s	100.59%
429.mcf	137.808 s	1.084 s	136.314 s	1.761 s	101.10%
433.milc	436.632 s	32.839 s	431.110 s	11.625 s	101.28%
445.gobmk	351.108 s	0.882 s	351.002 s	0.615 s	100.03%
456.hmmer	555.634 s	0.248 s	551.022 s	0.317 s	100.84%
458.sjeng	384.675 s	1.034 s	388.709 s	1.598 s	98.96%
462.libquantum	299.370 s	3.710 s	298.711 s	2.353 s	100.22%
464.h264ref	399.997 s	0.965 s	406.405 s	1.413 s	98.42%
470.lbm	230.761 s	2.546 s	230.106 s	0.884 s	100.28%
482.sphinx3	489.531 s	14.767 s	488.693 s	16.519 s	100.17%
Geom. Mittel					100.80%

Abbildung 4.17: Ausroll-Faktor 32, maximale Schleifengröße 32

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	272.193 s	3.025 s	254.025 s	1.056 s	107.15%
401.bzip2	346.120 s	1.745 s	339.501 s	2.716 s	101.95%
403.gcc	174.205 s	1.236 s	177.011 s	2.045 s	98.41%
429.mcf	137.013 s	1.153 s	136.314 s	1.761 s	100.51%
433.milc	432.662 s	23.126 s	431.110 s	11.625 s	100.36%
445.gobmk	338.525 s	0.635 s	351.002 s	0.615 s	96.45%
456.hmmer	558.035 s	0.238 s	551.022 s	0.317 s	101.27%
458.sjeng	389.548 s	1.195 s	388.709 s	1.598 s	100.22%
462.libquantum	294.101 s	1.640 s	298.711 s	2.353 s	98.46%
464.h264ref	392.400 s	1.448 s	406.405 s	1.413 s	96.55%
470.lbm	230.675 s	2.444 s	230.106 s	0.884 s	100.25%
482.sphinx3	482.695 s	11.569 s	488.693 s	16.519 s	98.77%
Geom. Mittel					99.99%

Abbildung 4.18: Ausroll-Faktor 32, maximale Schleifengröße 64

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	276.119 s	0.937 s	254.025 s	1.056 s	108.70%
401.bzip2	354.526 s	2.824 s	339.501 s	2.716 s	104.43%
403.gcc	173.963 s	0.825 s	177.011 s	2.045 s	98.28%
429.mcf	143.752 s	1.551 s	136.314 s	1.761 s	105.46%
433.milc	427.796 s	19.841 s	431.110 s	11.625 s	99.23%
445.gobmk	344.239 s	1.294 s	351.002 s	0.615 s	98.07%
456.hmmer	547.596 s	0.534 s	551.022 s	0.317 s	99.38%
458.sjeng	383.170 s	0.888 s	388.709 s	1.598 s	98.57%
462.libquantum	301.390 s	4.080 s	298.711 s	2.353 s	100.90%
464.h264ref	390.330 s	1.253 s	406.405 s	1.413 s	96.04%
470.lbm	230.517 s	2.046 s	230.106 s	0.884 s	100.18%
482.sphinx3	483.126 s	5.300 s	488.693 s	16.519 s	98.86%
Geom. Mittel					100.62%

Abbildung 4.19: Ausroll-Faktor 32, maximale Schleifengröße 128

	Resultat		Referenz		
Programm	Laufzeit	σ	Laufzeit	σ	Verhältnis
400.perlbench	279.865 s	1.740 s	254.025 s	1.056 s	110.17%
401.bzip2	357.917 s	2.026 s	339.501 s	2.716 s	105.42%
403.gcc	176.884 s	1.165 s	177.011 s	2.045 s	99.93%
429.mcf	147.686 s	1.315 s	136.314 s	1.761 s	108.34%
433.milc	413.190 s	11.008 s	431.110 s	11.625 s	95.84%
445.gobmk	346.634 s	0.621 s	351.002 s	0.615 s	98.76%
456.hmmer	548.233 s	0.488 s	551.022 s	0.317 s	99.49%
458.sjeng	387.360 s	1.181 s	388.709 s	1.598 s	99.65%
462.libquantum	300.602 s	6.135 s	298.711 s	2.353 s	100.63%
464.h264ref	389.324 s	0.700 s	406.405 s	1.413 s	95.80%
470.lbm	230.627 s	2.328 s	230.106 s	0.884 s	100.23%
482.sphinx3	505.648 s	117.718 s	488.693 s	16.519 s	103.47%
Geom. Mittel					101.39%

Abbildung 4.20: Ausroll-Faktor 32, maximale Schleifengröße 256

Für eine maximale Schleifengröße von 32 verbessert sich die Performanz nicht. Diese Schleifengröße scheint zu klein zu sein. Für eine maximale Schleifengröße von 256 hingegen verbessert sich die Performanz für alle getesteten Ausroll-Faktoren außer für den Ausroll-Faktor 32, wenn man die Laufzeiten des Programms 482.sphinx3 ignoriert, deren Standardabweichung viel zu hoch ist.

Es ist unklar, warum die Standardabweichung für so viele Resultate deutlich zu hoch ist. Eventuell hätten mehr als 10 Durchläufe pro Programm und Konfiguration hier Abhilfe geschaffen. Dies war zeitlich jedoch nicht mehr möglich, da das Ausführen der Benchmarks viel Zeit in Anspruch nimmt. Einige einzelne Ergebnisse sind ohnehin so weit vom Mittelwert entfernt, dass auch 20 Durchläufe die Standardabweichung nicht signifikant verbessert hätten.

Betrachtet man die Ausroll-Faktoren, so erkennt man keine klaren Tendenzen. In vielen Fällen scheint ein Ausroll-Faktor von 4 oder 8 besser zu sein als ein Ausroll-Faktor von 16.

Verschiedene Programme profitieren sehr unterschiedlich von den Optimierungen. Das Programm 464.h264ref zum Beispiel wird für jede getestete Konfiguration schneller, um bis zu 4%. Das Programm 400.perlbench hingegen wird für fast jede Konfiguration langsamer.

5 Fazit und Ausblick

Die Ergebnisse der Evaluation zeigen, dass die hier vorgestellte Implementierung für das Ausrollen von Schleifen die Performanz von gewissen Programmen verbessern kann. Auch wenn sich die Performanz für viele Konfigurationen im Mittel verbessert, gibt es dennoch für jede Konfiguration, die getestet wurde, immer mindestens ein Programm, dessen Performanz sich verschlechtert. Somit wurde keine Konfiguration gefunden, die für jeden Fall empfohlen werden könnte.

Die verwendete Heuristik zur Bestimmung des Ausroll-Faktors ist jedoch sehr einfach. Sie basiert nur auf der Anzahl Knoten einer Schleife und geht nicht darauf ein, wie teuer die Berechnungen der einzelnen Knoten sind. Vermutlich könnte hier mit einer besseren Heuristik, die die tatsächlichen Kosten besser abschätzt, noch mehr an Performanz gewonnen werden. Für eine Schleife mit einer Zählvariablen und einer konstanten Anzahl Iterationen könnte der Ausroll-Faktor beispielsweise durch die Anzahl Iterationen bestimmt werden.

An dieser Stelle ist es auch wichtig, anzumerken, dass die Performanz-Gewinne des Ausrollens von Schleifen (zumindest in dieser Implementierung) alleine von den danach folgenden Optimierungen abhängen. Die Implementierung entfernt keine Rücksprünge und ohne weitere Optimierungen würde die Performanz vermutlich nur verschlechtert, da mehr Code generiert wird, was die Cache-Ausnutzung verschlechtert. Somit ist es gut möglich, dass durch die Implementierung von weiteren spezifischen Optimierungen, die es ermöglichen, mehr Sprünge zu entfernen, oder dem Ändern der Reihenfolge der Optimierungen die Performanz nochmal deutlich verbessert werden könnte.

Literaturverzeichnis

[1] 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.

[2] "GNU Compiler Collection (GCC) Internals: LCSSA." `https://gcc.gnu.org/onlinedocs/gccint/LCSSA.html`. Stand 18. Juni 2018.

[3] C. Lattner, "Loop Optimizer Notes." `http://nondot.org/sabre/LLVMNotes/LoopOptimizerNotes.txt`, Aug. 2004.

[4] "Firm - Optimization and Machine Code Generation." `https://pp.ipd.kit.edu/firm/`. Stand 12. Juni 2018.

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

[6] "libfirm/cparser: C99 parser and frontend for libfirm." `https://github.com/libfirm/cparser/`. Stand 11. Juni 2018.

[7] C. Helmer, "Entwicklung von Kriterien zur Anwendung von Schleifenoptimierungen im Kontext SSA-basierter Zwischensprachen," Oct. 2010.

[8] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, 2013.

[9] "SPEC CPU 2006." `https://www.spec.org/cpu2006/`. Stand 12. Juni 2018.

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

Erklärung

Hiermit erkläre ich, Elias Aebi, 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