TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Programmtransformation zur Vereinfachung des Kontrollflusses

Bachelor-Thesis

von

Marvin Drewniok

2. Januar 2020

Technische Universität Kaiserslautern,
Department of Computer Science,
67653 Kaiserslautern,
Germany

Examiner: Prof.Dr. Klaus Schneider
M.Sc. Anoop Bhagyanath

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Programmtransformation zur Vereinfachung des Kontrollflusses" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 2.1.2020

____
Marvin Drewniok

Abstract

To further increase the performance of processors, more and more instructions must be parallelized. One form of parallelization is instruction-level paralism (ILP). ILP specifies how many of the instructions of a sequential program can be executed in parallel on a processor. Compilers try to increase ILP as much as possible without the programmer's intervention. One way for compilers to increase ILP is to reduce control dependencies so that instructions can only be executed in parallel depending on their data dependencies. Previous methods include the If conversion and its extension the hyperblock. However, these methods have one problem in common. They are not generally applicable to arbitrarily cyclic program parts. Therefore, a new transformation, loop conversion, is presented here. This transformation reduces any programs, with the exception of recursion, to one loop. This makes it possible to obtain an ILP increase by a factor of 2.8 on average without further optimization.

Zusammenfassung

Zur weiteren Steigerung der Performance von Prozessoren müssen immer mehr Instruktionen parallelisiert werden. Eine Form der Parallelisierung ist Instruction-Level-Paralism (ILP). ILP gibt an wie viele der Instruktionen eines sequenziellen Programmes auf einem Prozessor parallel ausgeführt werden können. Compiler versuchen, ohne Zutun des Programmierers, ILP so weit wie möglich zu erhöhen. Eine Möglichkeit für Compiler ILP zu erhöhen besteht darin Kontrollabhängigkeiten zu reduzieren, so dass Instruktionen nur abhängig von ihren Datenabhängigkeiten parallel ausgeführt werden können. Zu bisherigen Methoden zählen beispielsweise die If-Conversion und dessen Erweiterung, der Hyperblock. Diese haben jedoch ein Problem gemeinsam. Sie sind nicht allgemein auf beliebig zyklische Programmteile anwendbar. Deshalb wird hier eine neue Transformation, die Loop-Conversion, vorgestellt. Diese Transformation reduziert beliebige Programme, mit Ausnahme von Rekursion, auf eine Schleife. So ist es möglich, ohne weitere Optimierungen, im Schnitt einen ILP Zuwachs um den Faktor 2,8 zu erhalten.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

1. Einführung 1

2. Stand der Technik 3
- 2.1. Predicated Execution 3
- 2.2. If-Conversion 4
- 2.3. Hyperblock 4
- 2.4. Reverse If-Conversion 5

3. Algorithmus: Loop-Conversion 7
- 3.1. MiniC 7
- 3.2. Grundlagen 9
- 3.3. Transformation 9
  - 3.3.1. Function Inlining 10
  - 3.3.2. For-Schleifen Ersetzung 10
  - 3.3.3. LFB Einteilung 11
  - 3.3.4. Label Zuweisung 11
  - 3.3.5. Strukturumwandlung 12
  - 3.3.6. If-Conversion 17
- 3.4. Darstellungen 17
- 3.5. Beweis 19
- 3.6. Auswertung 19

4. Fazit und Ausblick 23

Literatur 25

A. Tests: Loop-Conversion 27
- A.1. Basisfälle 27
- A.2. Sequenzielle Schleifen 29
- A.3. Verschachtelte Schleifen 32

Abbildungsverzeichnis

1.1. Beispiel: Instruction-level parallelism. Quelle: [Wal91] 1
1.2. Beispielprogramm mit Quellcode, Assemblercode und Kontrollflussgraph. Quelle: [Hwu+95] 2

2.1. Zusammenhang: If-Conversion, Hyperblock, Reverse-If-Conversion. Quelle [War+93] 5

3.1. Syntaxbeispiel: If-Anweisung und Schleifen 8
3.2. Syntaxbeispiel: Bubblesort in MiniC 8
3.3. Codebeispiel Loop-Free-Blocks 9
3.4. Beispiel Function Inlining vorher 10
3.5. Beispiel Function Inlining nachher 10
3.6. Codebeispiel einer allgemeinen For-Schleife 11
3.7. Codebeispiel nach For-Schleifen Ersetzung 11
3.8. Beispiel Loop-Free-Blocks mit entsprechendem Label 12
3.9. Vergleich der allgemeinen Programmstruktur anhand eines Beispielprogramms, vor und nach Strukturkonversion 13
3.10. Vergleich Beispielprogramm vor und nach Strukturumwandlung 14
3.11. Beispielprogramm vollständig transformiert 17
3.12. Vergleich der möglichen Darstellungen von *currentLabel* und deren Auswirkung auf die Programmstruktur 18
3.13. Vergleich Kontrollfluss vor und nach Transformation 20

Tabellenverzeichnis

3.1. Umstrukturierung While-Schleifen 14
3.2. Umstrukturierung Do-While-Schleifen 15
3.3. Umstrukturierung If-Anweisungen 16
3.4. Umstrukturierung Sequenzen 16
3.5. Zuwachs von ILP und ausgeführte Instruktionen nach Loop-Conversion 21

1. Einführung

Für Prozessoren wird in der heutigen Zeit die parallele Ausführung eines Programms immer bedeutender. Mit einem höheren Grad an Parallelität geht ein direkter Leisungszuwachs einher. Eine Form von Parallelität ist Instruction-Level-Paralism (ILP). ILP gibt an, wie viele Instruktionen in einem sequenziell definierten Programm, von einem Prozessor parallel ausgeführt werden können. Der Hauptvorteil von ILP besteht darin, dass es die Parallelität ausnutzt, ohne dass der Programmierer bestehende Anwendungen neu schreiben muss[Sch+97]. Diese Art von Parallelität wird von Daten- und Kontrollabhängigkeiten eingeschränkt[Wal91]. In Abbildung 1.1 sind zwei Programme mit jeweils drei Instruktionen dargestellt. Davon können in Fall a) alle Instruktionen parallel ausgeführt werden. Während in Fall b) die jeweils nachfolgende Instruktion von dem Ergebnis der Vorherigen abhängt. Dies sind Datenabhängigkeiten.

r1 := 0[r9]	r1 := 0[r9]
r2 := 17	r2 := r1 + 17
4[r3] := r6	4[r2] := r6
(a) parallelism=3	*(b) parallelism=1*

Abbildung 1.1.: *Beispiel: Instruction-level parallelism. Quelle: [Wal91]*

Dagegen geben Kontrollstrukturen an, welche Instruktionen als nächstes ausgeführt werden sollen. So können bedingte Ausführungen und Schleifen realisiert werden. Zur Ausführung eines Programms zerteilen Compiler dieses in Basic Blocks. Die Unterteilung erfolgt an Kontrollstrukturen. Dies ist in Abbildung 3.13 zu sehen. In a) wird ein kleines Beispielprogramm vorgestellt, dessen compilierte Form in Instruktionen in b) dargestellt wird. Die Querstriche in b) geben die Unterteilung in Basic Blocks an. In c) ist der entsprechende Kontrollflussgraph visualisiert.

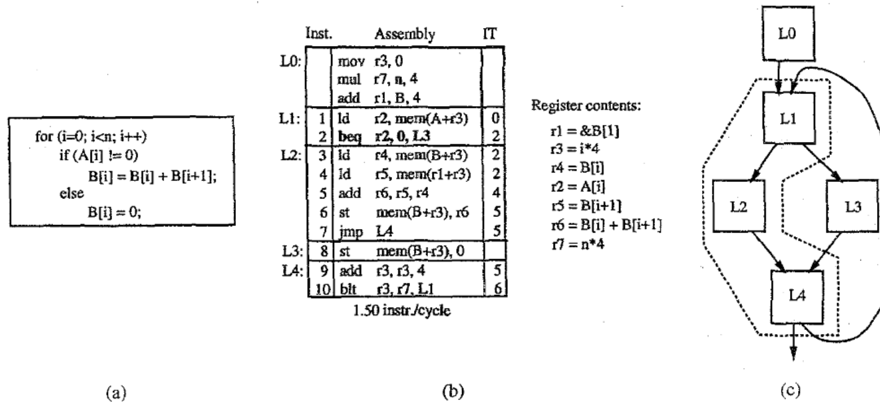


Abbildung 1.2.: *Beispielprogramm mit Quellcode, Assemblercode und Kontrollflussgraph. Quelle: [Hwu+95]*

Ohne weitere Techniken können nur alle Instruktionen eines Basic Blocks, unter Beachtung der Datenabhängigkeiten, parallel ausgeführt werden. Daher ist es das Ziel dieser Arbeit eine neue Möglichkeit aufzuzeigen, welche den Kontrollfluss zusätzlich verringert. So können größere Basic Blocks gefunden werden und Compilern werden mehr Optionen gegeben ILP auszunutzen.

Im Folgenden wird eine Grundlage geschaffen, indem Predicated Execution erklärt wird. Außerdem wird aufgezeigt wie ein bestehender Befehlssatz um teilweisen Predicated Support erweitert werden kann. Anschließend folgen die darauf aufbauenden Kapitel: If-Conversion, Hyperblock, Reverse-If-Conversion und Loop-Conversion, in denen diese Instruktionen genutzt werden, um Kontrollfluss effektiv zu vereinfachen.

In dem letzten dieser Kapitel wird ein neuer Algorithmus vorgestellt. Dieser ermöglicht es, im Gegensatz zu bisher vorhandenen Verfahren, den Kontrollfluss beliebiger Programme auf eine Schleife zu reduzieren.

2. Stand der Technik

2.1. Predicated Execution

Bei Predicated Execution, oft auch Guarded Execution oder Conditional Execution genannt, handelt es sich um spezielle Instruktionen eines Befehlssatzes. Diese Befehle werden abhängig von dem Wert eines Predicate-Registers unterschiedlich ausgeführt. Dieses Registers wird auch als Predicate oder Guard bezeichnet. Besitzt das Predicate den Wert *wahr* beziehungsweise 1, wird die Instruktion regulär ausgeführt. Wenn das Predicate den Wert $falsch$ beziehungsweise 0 aufweist, wird die Instruktion als No-Operation(Nop) ausgeführt[All+83][PS91]. Der Gesamtzustand eines Systems wird durch die ausgeführte Nop nicht verändert. So ist es möglich beide Nachfolger eines Branches auszuführen und nur das Ergebnis des eigentlich auszuführenden Block zu erhalten. Dadurch wurde der Kontrollfluss dieses Branches in Datenfluss überführt.

Man unterscheidet zwischen Befehlssätzen mit teilweiser und mit voller Unterstützung für Predicated Execution[Mah+95]. Bei teilweiser Unterstützung können nur manche Instruktionen bedingt ausgeführt werden, während bei voller Unterstützung alle Instruktionen abhängig von einem Predicate ausgeführt werden können.

Für eine teilweise Unterstützung sind nur wenige Änderungen an einer bestehenden Befehlssatzarchitektur notwendig [PS94][Mah+95]. Eine Möglichkeit, dies zu erreichen, besteht darin den Befehlssatz um einen conditional move Befehl zu erweitern.

cmove dest,src,cond
if (cond) dest = source

Ist die Bedingung $condwahr$, wird der Inhalt aus Quell Register src in das Ziel Register $dest$ kopiert. Das $cond$ Register kann ein bereits vorhandenes Integer oder Floating Point Register sein [Mah+95].
Eine weitere Möglichkeit teilweise Predicated execution zu unterstützen ist der select Befehl.

select dest,srcl,src2,cond
dest = ((cond) ? src1 : src2)

Ist die Bedingung $condwahr$ wird dem Ziel Register $dest$ der Wert aus Register $src1$ zugewiesen, anderfalls der Wert aus Register $src2$. So kann ein Compiler effektiv „then “ und „else “ Pfade unterstützen [Mah+95]. Zusätzlich zu diesen

Befehlen sind weitere notwendig, welche das Predicate Register modifizieren können.

2.2. If-Conversion

If-Conversion bezeichnet die Umwandlung von Kontrollfluss zu Datenfluss, welche ursprünglich zur Vektorisierung von Schleifen genutzt wurde [All+83]. In dem von Allen, Kennedy, Porterfield und Warren vorgestelltem Algorithmus werden Branches einer innersten Schleife entfernt, sodass eine Vektorisierung jener innerster Schleife möglich ist. Hierzu werden die Branches in drei Typen unterteilt, welche in [All+83] wie folgt beschrieben sind:

1. Exit Branch: Das Ziel eines solchen Branches liegt außerhalb des Schleifenlevels, in welchem der Branch auftritt. Folglich terminiert ein Exit Branch mindestens eine Schleife.
2. Forward Branch: Ein Branch dessen Ziel zum Einen innerhalb des Schleifenlevels, in welchem der Branch auftritt, und zum Anderen nach dem Auftreten des Branches liegt.
3. Backward Branch: Dieser Branch ist analog zum Forward Branch mit dem Unterschied, dass das Ziel vor dem Auftreten des Branches liegt.

Der eigentliche Algorithmus ist unterteilt in zwei Teile [All+83]:

1. Branch Relocation: Hier werden Exit Branches in einen der anderen Typen von Branches umgewandelt. Dazu werden Branches aus Schleifen herausgezogen, sodass der Branch und das Ziel des Branches auf einem Schleifenlevel liegen [All+83].
2. Branch Removal: Forward Branches werden durch Ausdrücke der Form $IF(guard)statement$ ersetzt. Wobei der *guard* ein Predicate ist, welches die ursprüngliche Reihenfolge der Ausführung des Programms erhalten soll. [All+83]

Eine weitere Form der If-Conversion findet ihre Anwendung unter anderem in [PS91]. Es stellt einen allgemeineren Ansatz dar, der versucht möglichst viele Branches einer innersten Schleife zu entfernen. Als Ziel sollen alle Anweisungen einer innersten Schleife unter Beachtung der Datenabhängigkeiten parallel ausgeführt werden können. Zusätzlich wird dadurch effektives Softwarepipelining ermöglicht. [PS91]. If-Conversion kann bereits mit einer teilweisen Unterstützung für Predicated Execution ausgeführt werden.

2.3. Hyperblock

Der Hyperblock stellt eine Erweiterung der If-Conversion da. Das Ziel der Hyperblockbildung ist eine Gruppierung von predicated Basisblöcken aus vielen

verschiedenen Kontrollflusspfaden zu einem einzigen Block, dessen Ausführung so besser geplant werden kann[Mah+94]. Ein Hyperblock hat genau eine von oben eingehende Kante und beliebig viele ausgehende Kanten im Kontrollflussgraph[Mah+92]. Ein Hyperblock besteht typischerweise aus dem Rumpf einer innersten Schleife, dennoch sind weitere Basic Blocks möglich.

Zu Beginn besteht ein Hyperblock aus nur einem Basic Block. Es werden sukzessiv weitere Blöcke hinzugenommen. Jeder Basic Block bekommt eine Priorität zugewiesen, anhand der entschieden wird, ob dieser Basic Block in einen Hyperblock aufgenommen wird. Selten ausgeführte, sowie große Blöcke erhalten eine geringere Priorität. Je größer die Priorität, desto eher wird ein Basic Block zu einen Hyperblock hinzugefügt.

Der systematische Ausschluss von Basic Blocks aus Hyperblöcken bietet zusätzliche Optimierungssmöglichkeiten für Strukturen innerhalb eines Hyperblocks [Mah+92].

2.4. Reverse If-Conversion

Die Reverse-If-Conversion ist wiederum eine Erweiterung des Hyperblocks (siehe Abbildung 2.1) [War+93]. Diese findet nach der Hyperblockformung statt. Manche predicated Ausdrücke innerhalb eines Hyperblocks werden wieder in Branches transformiert. Dies wird gemacht um den Schedule weiter zu optimieren. So werden einzelne Pfade herausgelöst, bis ein Optimum gefunden ist. Werden alle predicated Ausdrücke zurückgeformt, können auch Prozessoren ohne Predicated Execution Support, vom Hyperblock Scheduling profitieren [AHM97].

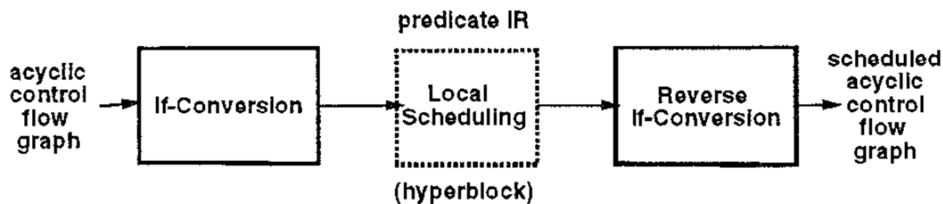


Abbildung 2.1.: *Zusammenhang: If-Conversion, Hyperblock, Reverse-If-Conversion. Quelle [War+93]*

3. Algorithmus: Loop-Conversion

Die Loop-Conversion ist ein neuartiger Algorithmus zur Vereinfachung des Kontrollflusses von Programmen. Dieser Algorithmus stellt eine Erweiterung zur bereits existierenden If-Conversion dar. Es handelt sich hierbei um eine Programmtransformation, welche mittels Predcicated Execution sämtliche Schleifen eines Programms zu einer einzigen Schleife zusammenfasst. Ziel der Transformation ist es Basic Blocks, welche durch Kontrollstrukturen getrennt sind, zusammenzufügen und so einen höheren Grad an ILP zu erzielen. Rekursive Programme werden nicht unterstützt.

3.1. MiniC

Der Algorithmus wurde für Programme der Sprache MiniC implementiert. MiniC ist eine vereinfachte Form der Programmiersprache C und eine, vom Lehrstuhl für eingebettete Systeme der TU Kaiserslautern zu Lehr- und Forschungszwecken entwickelte, Programmiersprache [BSS15]. Diese Programmiersprache soll den Studierenden grundlegende Arbeitsweisen von Prozessoren und Compilern näher bringen.

Der Compiler übersetzt MiniC Programme in den Abacus Befehlssatz. Der Abacus Befehlssatz und gleichnamiger Prozessor wurde zusammen mit der MiniC Sprache entwickelt. Der MiniC Compiler wurde für diese Arbeit modifiziert und um den Loop-Conversion Algorithmus erweitert.

Es werden folgende Datentypen unterstützt: Bool, natürliche und ganze Zahlen, sowie den daraus bestehenden Arrays und Tupel. Für diese Datentypen werden gängige Arithmetische Operationen unterstützt. Deklarationen müssen zu Beginn eines Programms oder Funktion stehen. Des weiteren ermöglicht MiniC Vektoroperationen und Multithreading mit entsprechendem Support für Synchronisation. Außerdem existiert bereits in dieser Programmiersprache eine Anweisung für Predicated Execution. Diese Anweisung hat die Syntax:

$$\mathbf{x = (\sigma)\ ?\ y : z}$$

Diese Anweisung verhält sich analog zu dem *select* Befehl. Wird σ zu 1 ausgewertet erhält x den Wert y, ansonsten den Wert z.

Zu den Kontrollflussstrukturen zählen die bedingte Ausführung mit und ohne sonst Fall, sowie For-, while- und do-while-Schleifen. Die Strukturen haben folgende Syntax:

```
1  if(σ1){
2       s1;
3  }
4  if(σ2){
5       s2;
6  } else {
7       s3;
8  }
9  for(start..end){
10      s4;
11 }
12 while(σ3){
13      s5;
14 }
15 do
16      s6;
17 while(σ4)
```

Abbildung 3.1.: *Syntaxbeispiel: If-Anweisung und Schleifen*

Der nachfolgenden Code zeigt den Bubblesort Algorithmus als vollständiges MiniC Programm.

```
1  thread t {
2       [10]nat x;
3       nat i,y,swapped,xlen;
4
5       for(i=0..xlen-1)
6            x[i] = xlen-i;
7
8       do {
9            swapped = 0;
10           for(i=1..xlen-1) {
11                if(x[i-1] > x[i]) {
12                     y = x[i-1];
13                     x[i-1] = x[i];
14                     x[i] = y;
15                     swapped = 1;
16                }
17           }
18      } while(swapped==1)
19 }
```

Abbildung 3.2.: *Syntaxbeispiel: Bubblesort in MiniC*

MiniC unterstützt keine rekursiven Funktionen.

3.2. Grundlagen

Zum besseren Verständnis der Loop-Conversion werden die folgenden zwei Definition eingeführt.

Definition 3.1.1 Loop-Free-Block(LFB): Ein Loop-Free-Block ist ein maximal großer, zusammenhängender Codeausschnitt, der nicht durch den Start oder das Ende einer Schleife unterbrochen wird. Loop-Free Blocks werden aus dem Ausgangscode festgelegt.

Beispiel 3.1.2 Im nachfolgenden Programmcode stehen $s1$, $s2$, $s3$, $s4$ und $s5$ für beliebige Programmsequenzen ohne Kontrollfluss. Wobei $s1$, $s2$ und $s3$ jeweils einen eigenen Loop-Free-Block bilden. Die gesamte If-Anweisung, mit Ausdruck $\sigma 4$ und den Sequenzen $s4$, $s5$, hingegen bildet einen einzigen LFB.

```
1  while(σ1){
2      s1;         // LFB
3      do{
4          s2;  // LFB
5      }while(σ2)
6  }
7  s3;  // LFB
8  while(σ3){
9      if(σ4){   //
10             s4;  //
11       else      // LFB
12             s5;  //
13       }          //
14 }
```

Abbildung 3.3.: *Codebeispiel Loop-Free-Blocks*

Definition 3.1.4 Label: Ein Label ist eine eindeutige Kennung eines LFBs. Label sind von der Form x_0-x_1-x_2-... . Bei den einzelnen Stellen x_0, x_1, x_2,... handelt es sich um natürliche Zahlen. Das erste Label ist 1 und alle weiteren Label werden fortlaufend zugewiesen.

3.3. Transformation

Die Transformation besteht aus folgenden Schritten:

1. Function Inlining
2. Ersetzung der For-Schleifen
3. Einteilung des Codes in LFB
4. Zuweisung der Label

5. Strukturumwandlung
6. If-Conversion

3.3.1. Function Inlining

Zuerst wird im gesamten Programm jeder Funktionsaufruf durch den Code der aufgerufenen Funktion ersetzt. Dies nennt man Function Inlining [Das03] [Arn+00]. Die Ersetzung geschieht, da eine Funktion Kontrollflussstrukturen enthalten kann und auch diese reduziert werden sollen.

```
1  procedure Initialize([]nat x,nat xlen) {
2      nat i;
3      for(i=0..xlen-1)
4          x[i] = xlen-i;
5      return;
6  }
7
8  thread t {
9      [10]nat x;
10     Initialize(x,10);
11 }
```

Abbildung 3.4.: *Beispiel Function Inlining vorher*

Abbildung 3.4 zeigt ein simples Programm, in welchem die vorher spezifizierte Funktion „Initialize“ aufgerufen wird.

```
1 thread t {
2     [10]nat x;
3     nat i;
4     for(i=0..(10 - 1)) {
5         x[i] = (10 - i);
6     }
7 }
```

Abbildung 3.5.: *Beispiel Function Inlining nachher*

Nach dem Function Inlining wurde die Funktion „Initialize“ durch ihren Funktionscode ersetzt (Abbildung 3.5).

3.3.2. For-Schleifen Ersetzung

Anschließend werden sämtliche For-Schleifen innerhalb des Programms durch äquivalente While-Schleifen ersetzt, da in der späteren Strukturumwandlung boolsche-Ausdrücke als Schleifenbedingung benötigt werden.

```
1 for(i=m..n) {
2     s1;
3 }
```

Abbildung 3.6.: *Codebeispiel einer allgemeinen For-Schleife*

Abbildung 3.6 zeigt eine allgemeine For-Schleife vor der Ersetzung, welche bei $i = m$ startet und nach $i = n$ terminiert. Pro Iteration wird die beliebige Programmsequenz $s1$ ausgeführt.

```
1 nat i;
2 i = m;
3 while(i<=n){
4     s1;
5     i = i + 1;
6 }
```

Abbildung 3.7.: *Codebeispiel nach For-Schleifen Ersetzung*

Abbildung 3.7 zeigt die äquivalente While-Schleife nach der Ersetzung, welche gleiches Verhalten aufweist. Es wird die Variable i deklariert und mit Wert m initialisiert. In jeder Iteration wird die Programmsequenz $s1$ ausgeführt und anschließend i inkrementiert, bis die Schleife nach $i = n$ terminiert.

3.3.3. LFB Einteilung

In diesem Schritt wird der Code gemäß Definition 3.1.1 in Loop-Free-Blocks eingeteilt. Ein entsprechendes Beispiel findet sich in 3.1.2.

3.3.4. Label Zuweisung

Nun wird jedem LFB ein Label nach 3.1.4 zugewiesen. Bei der Zuweisung wird zwischen drei Fällen unterschieden:

- Fall 1: Der nächste LFB befindet sich auf der gleichen Codeebene. Dann wird die letzte Stelle im Label um 1 erhöht.
- Fall 2: Der nächste LFB befindet sich eine Codeebene tiefer. Dann wird das vorhergehende Label um -1 ergänzt.
- Fall 3: Der nächste LFB befindet sich auf einer höheren Codeebene. Dann werden entsprechend viele Stellen von rechts gestrichen und die nun letzte Stelle um 1 erhöht.

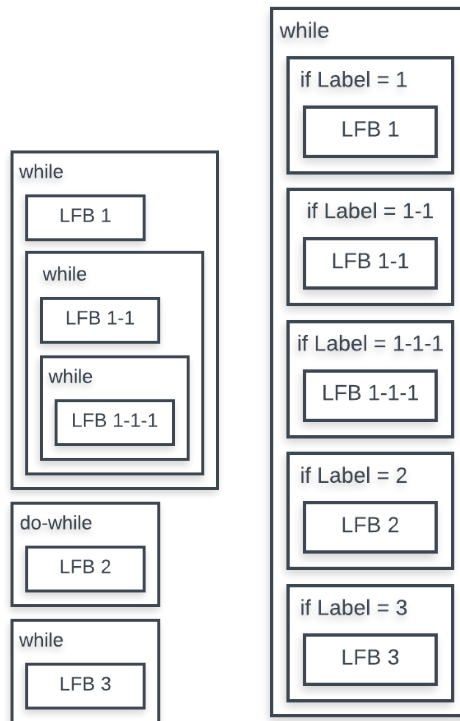
```
1  while(σ1){
2      s1; // LFB mit Label 1
3      do{
4          s2; // Label 1-1
5      }while(σ2)
6  }
7  s3; // Label 2
8  while(σ3){
9       if(σ4){ //
10          s4; //
11      else    // Label 3
12          s5; //
13      }       //
14 }
```

Abbildung 3.8.: *Beispiel Loop-Free-Blocks mit entsprechendem Label*

3.3.5. Strukturumwandlung

Nun werden alle verbleibenden Schleifen in eine einzige While-Schleife transformiert. Es wird eine neue Variable *currentLabel* eingeführt, welche den auszuführenden LFB angibt. Jeder LFB bekommt die Überprüfung, ob sein Label dem des *currentLabel* entspricht, als Predicate bzw. Guard zugewiesen. Durch dieses Predicate werden pro Iteration, der neuen While-Schleife, entsprechende LFBs ausgeführt. Dies ist in Abbildung 3.9 anhand eines Beispiels visualisiert.

Zu Beginn hat *currentLabel* den Wert des Labels des ersten LFB. Um den vorherigen Programmfluss beizubehalten, bekommt *currentLabel* nach dem Ausführen des aktuellen LFB, das Label von dem als nächsten auszuführenden LFBs zugewiesen. Dabei wird unterschieden, ob ein LFB von einer Schleife abstammt und um welche Schleifenart es sich handelt. Ebenso wird beachtet auf welcher Codeebene sich der nachfolgende LFB befindet.



(a) Beispiel Struktur vorher **(b)** Beispiel Struktur nachher

Abbildung 3.9.: *Vergleich der allgemeinen Programmstruktur anhand eines Beispielprogramms, vor und nach Strukturkonversion*

In Abbildung 3.10 ist ein Beispielprogramm vollständig umstrukturiert. Alle Schleifen wurden auf eine Schleife reduziert. Es ist zu sehen, dass jeder LFB sein Label als Guard besitzt und abhängig der boolschen Ausdrücke $\sigma 1, \sigma 2, \sigma 3$ wird das *currentLabel* auf den, als nächstes auszuführenden, LFB gesetzt. So ist es möglich die einzelnen Codesequenzen $s1, s2, ..., s5$ über Kontrollflussschranken ihrer ursprünglichen Schleife hinweg zu verschieben. In den nachfolgenden Tabellen wird beschrieben, wie einzelne LFBs entsprechend ihrem Codeursprung umstrukturiert werden.

```
1 while(σ1){
2     s1; // LFB mit Label 1
3     do{
4         s2; // Label 1−1
5     }while(σ2)
6 }
7 s3; // Label 2
8 while(σ3){
9     if(σ4){ //
10        s4; //
11    else      // Label 3
12        s5; //
13    }         //
14 }
```

(a) Vor Strukturumwandlung

```
1 currentLabel = 1;
2 while(currentLabel != 4){
3     if (currentLabel = 1){
4         if(σ1){
5             s1;
6             currentLabel = 1−1;
7         } else {
8             currentLabel = 2;
9         }}
10    if (currentLabel = 1−1){
11        s1;
12        if(!σ2){
13            currentLabel = 1;
14        }}
15    if (currentLabel = 2){
16        s3;
17        currentLabel = 3;
18        }
19    if (currentLabel = 3){
20        if(σ3){
21            if(σ4){
22                s4;
23            } else {
24                s5;
25            }
26        } else {
27            currentLabel = 4;
28        }}
29 }
```

(b) Nach Strukturumwandlung

Abbildung 3.10.: *Vergleich Beispielprogramm vor und nach Strukturumwandlung*

<table>
<tr><th>Ausgangscode</th><th>Nach Umstrukturierung</th></tr>
<tr><td><pre>while(σ){
    s1; //Label = x
}
LFB //Label = y</pre></td><td><pre>if(currentLabel = x){
    if(σ){
        s1;
    } else {
        currentLabel = y
    }
}</pre></td></tr>
<tr><td><pre>while(σ){
    s1; //Label = x
    LFB //Label = x−1
}
LFB //Label = y</pre></td><td><pre>if(currentLabel = x){
    if(σ){
        s1;
        currentLabel = x−1
    } else {
        currentLabel = y
    }
}</pre></td></tr>
</table>

Tabelle 3.1.: *Umstrukturierung While-Schleifen*

In der ersten Zeile der Tabelle 3.1 ist eine While-Schleife ohne inneren LFB zu sehen. Die Sequenz $s1$ wird so lange ausgeführt bis der boolsche Ausdruck σ nicht mehr zu wahr bzw. zu 1 ausgewertet wird. Anschließend wird der LFB mit Label y ausgeführt. Das selbe Verhalten wird nach der Umstrukturierung erreicht, indem das *currentLabel* erst auf den nachfolgen LFB gesetzt wird, sobald σ zu falsch bzw. 0 ausgewertet wird. In der zweiten Zeile dieser Tabelle besitzt die While-Schleife zusätzlich einen inneren LFB mit dem Label $x - 1$. Also muss in jeder Iteration nach dem Ausführen der Sequenz $s1$ das *current-Label* auf $x - 1$ gesetzt werden.

<table>
<tr><th>Ausgangscode</th><th>Nach Umstrukturierung</th></tr>
<tr><td><pre>do
    s1; //Label = x
while(σ)
LFB //Label = y</pre></td><td><pre>if(currentLabel = x){
    s1;
    if(!σ){
        currentLabel = y
    }
}</pre></td></tr>
<tr><td><pre>do
    s1; //Label = x
    LFB //Label = x−1
while(σ) //Label = z
LFB //Label = y</pre></td><td><pre>if(currentLabel = x){
        s1;
        currentLabel = x−1
}
if(currentLabel = z){
    if(σ){
        currentLabel = x
    } else {
        currentLabel = y
    }
}</pre></td></tr>
</table>

Tabelle 3.2.: *Umstrukturierung Do-While-Schleifen*

In Tabelle 3.2 wird die Umstrukturierung einzelner Do-While-Schleifen dargestellt. Bei der oberen Spalte handelt es sich um eine einfache Do-While-Schleife ohne innere LFBs. Erreicht das Programm diese Schleife wird die Sequenz $s1$ mindestens einmal ausgeführt. Erst wenn σ zu falsch ausgewertet wird, wird der Code im nachfolgenden LFB ausgeführt. Dieses Verhalten erreicht man nach der Umstrukturierung indem die Auswertung von σ und das Umsetzen der Variable *currentLabel* erst nach $s1$ geschieht. In der unteren Spalte enthält die Do-While-Schleife einen inneren LFB, was zu einem Sonderfall führt. Sowohl die Sequenz $s1$, als auch die Schleifenbedingung bekommen ein Label zugewiesen. So ergeben sich zwei getrennte Blöcke nach der Umstrukturierung. Der erste Block führt $s1$ aus und setzt danach das *currentLabel* auf $x - 1$. Der zweite Block, mit dem Label z, wird vom letzten LFB innerhalb der Do-While-Schleife aufgerufen. Dieser Block führt keinen Code aus dem originalen Programm aus, sondern steuert nur welcher LFB als nächstes ausgeführt werden soll. Wird der Ausdruck σ zu wahr ausgewertet, muss $s1$ wieder

ausgeführt werden und *currentLabel* wird auf x gesetzt. Andernfalls folgt der nachfolgende LFB mit dem Label y.

<table>
<tr><th>Ausgangscode</th><th>Nach Umstrukturierung</th></tr>
<tr><td><pre>
if(σ){
    s1;   //Label = x
    LFB //Label = x−1
}
LFB //Label = y
</pre></td><td><pre>
if(currentLabel = x){
    if(σ){
        s1;
        currentLabel = x−1
    } else {
        currentLabel = y
    }
}
</pre></td></tr>
<tr><td><pre>
if(σ){
    s1;
} else {
    s2;
    LFB //Label = x−1
}
LFB //Label = y
</pre></td><td><pre>
if(currentLabel = x){
    if(σ){
        s1;
        currentLabel = y
    } else {
        s2;
        currentLabel = x−1
    }
}
</pre></td></tr>
</table>

Tabelle 3.3.: *Umstrukturierung If-Anweisungen*

In der ersten Zeile der Tabelle 3.3 wird die Umstrukturierung einer If-Anweisung mit einem innerem und einem nachfolgenden LFB dargestellt. Hält der Ausdruck σ wird die Sequenz $s1$ und dann der LFB mit Label $x-1$ ausgeführt. Nach der Umstrukturierung wird *currentLabel* abhängig von σ gesetzt. In der letzten Spalten besitzt die If-Then-Else-Anweisung einen inneren LFB im Else-Fall und einen nachfolgenden LFB. Nach der Umwandlung ergibt sich ein äquivalentes Verhalten, durch entsprechendes setzen des *currentLabels*.

<table>
<tr><th>Ausgangscode</th><th>Nach Umstrukturierung</th></tr>
<tr><td><pre>
s1; //Label = x
LFB //Label = y
</pre></td><td><pre>
if(currentLabel = x){
    s1;
    currentLabel = y
}
</pre></td></tr>
</table>

Tabelle 3.4.: *Umstrukturierung Sequenzen*

Tabelle 3.4 zeigt die vorgegebene Umstrukturierung bei einer Sequenz von LFBs auf gleicher Codeebene. Nach Ausführen der Codesequenz $s1$ bekommt *currentLabel* das Label des nachfolgenden Blocks zugewiesen.

Handelt es sich um den letzten LFB einer Codeebene, hängt die Zuweisung des nachfolgenden Labels von der Art des ersten LFBs dieser Ebene ab. Stammt der erste LFB von einer:

- While-Schleife ab, verweist der letzte LFB auf das Label des ersten LFB.

- Do-While-Schleife ab, wird dem *currentLabel* der Wert des zusätzlichen Block mit der Do-While Bedingung zugewiesen.
- If-Anweisung ab, wird dem *currentLabel* der Wert des Labels des nachfolgenden LFB der If-Anweisung zugeordnet.

3.3.6. If-Conversion

Abschließend wird das Programm mit einer modifizierten If-Conversion transformiert. So werden allen Branches innerhalb der nun einzig vorhanden Schleife eliminiert und der Kontrollfluss maximal reduziert. Abbildung 3.11 zeigt

```
1  _label = 1;
2  while((_label != 4)) {
3      _b2 = ((_label == 1) ? σ1 : false);
4      _b3 = ((_label == 1) ? !(_b2) : false);
5      _label = ((_label == 1) ? (_b2 ? 1 : 2) : _label);
6      s1 = ((_label == 1) ? s1_new : s1);
7      _label = ((_label == 1) ? 1-1 : _label);
8      s2 = ((_label == 1-1) ? s2_new : s2);
9      _b0 = ((_label == 1-1) ? σ2: false);
10     _b1 = ((_label == 1-1) ? !(_b0) : false);
11     _label = ((_label == 1-1) ? (_b0 ? 1-1 : 1) : _label);
12     s3 = ((_label == 2) ? s3_new : s3);
13     _label = ((_label == 2) ? 3 : _label);
14     _b6 = ((_label == 3) ? σ3 : false);
15     _b7 = ((_label == 3) ? !(_b6) : false);
16     _label = ((_label == 3) ? (_b6 ? 3 : 4) : _label);
17     _b4 = ((_label == 3) ? σ4 : false);
18     _b5 = ((_label == 3) ? !(_b4) : false);
19     s4 = (((_label == 3) & _b4) ? s4_new : s4);
20     s5 = (((_label == 3) & _b5) ? s5_new : s5);
21 }
```

Abbildung 3.11.: *Beispielprogramm vollständig transformiert*

das vollständig transformierte Beispielprogramm aus Abbildung 3.9. Sämtliche If-Anweisungen werden durch Predicate Anweisungen ersetzt. Dafür werden zusätzlich zu dem *currentLabel*, hier als *_label* bezeichnet, weitere Guards $_b0, ..., _b7$ eingefügt. Da es sich bei $s1, ..., s5$ um beliebige Codesequenzen ohne Schleifen handelt, können diese auch durch die If-Conversion transformiert werden. Dies wird durch $s = guard?s_new : s$ repräsentiert. Bei s_new handelt es sich um den Programmcode aus der Sequenz s. Dadurch soll verdeutlicht werden, dass s sich nur ändert wenn der Guard *true* ist.

3.4. Darstellungen

Es gibt zwei Alternativen das endgültig transformierte Programm darzustellen. Diese Darstellungen beziehen sich vor allem auf das *currentLabel*. Entweder

wird es als ein Zahlendatentyp repräsentiert, der den aktuell auszuführenden LFB angibt. Dann werden die einzelnen LFBs durchnummeriert und erhalten als Guard ihre entsprechende Zahl. Oder aber für jedes vorhandene Label wird eine bool Variable angelegt und jeder LFB bekommt seine entsprechende Variable als Guard. Dies ist in Abbildung 3.12 visualisiert.

```
1  nat currentLabel = 1;
2  while(currentLabel != 5){
3      if (currentLabel = 1){
4          if(σ1){
5              s1;
6              currentLabel = 2;
7          } else {
8              currentLabel = 3;
9          }}
10     if (currentLabel = 2){
11         s1;
12         if(!σ2){
13             currentLabel = 1;
14         }}
15     if (currentLabel = 3){
16         s3;
17         currentLabel = 4;
18         }
19     if (currentLabel = 4){
20         if(σ3){
21             if(σ4){
22                 s4;
23             } else {
24                 s5;
25             }
26         } else {
27             currentLabel = 5;
28         }}
29 }
```

(a) *Möglichkeit 1: Label als Zahl*

```
1  bool label1 = true;
2  bool label11 = false;
3  bool label2 = false;
4  bool label3 = false;
5  bool label4 = false;
6  while(!label4){
7      if (label1){
8          if(σ1){
9              s1;
10             label1 = false;
11             label11 = true;
12         } else {
13             label1 = false;
14             label2 = true;
15         }}
16     if (label11){
17         s1;
18         if(!σ2){
19             label11 = false;
20             label1 = true;
21         }}
22     if (label2){
23         s3;
24         label2 = false;
25         label3 = true;
26         }
27     if (label3){
28         if(σ3){
29             if(σ4){
30                 s4;
31             } else {
32                 s5;
33             }
34         } else {
35             label3 = false;
36             label4 = true;
37         }}
38 }
```

(b) *Möglichkeit 2: Label als bool*

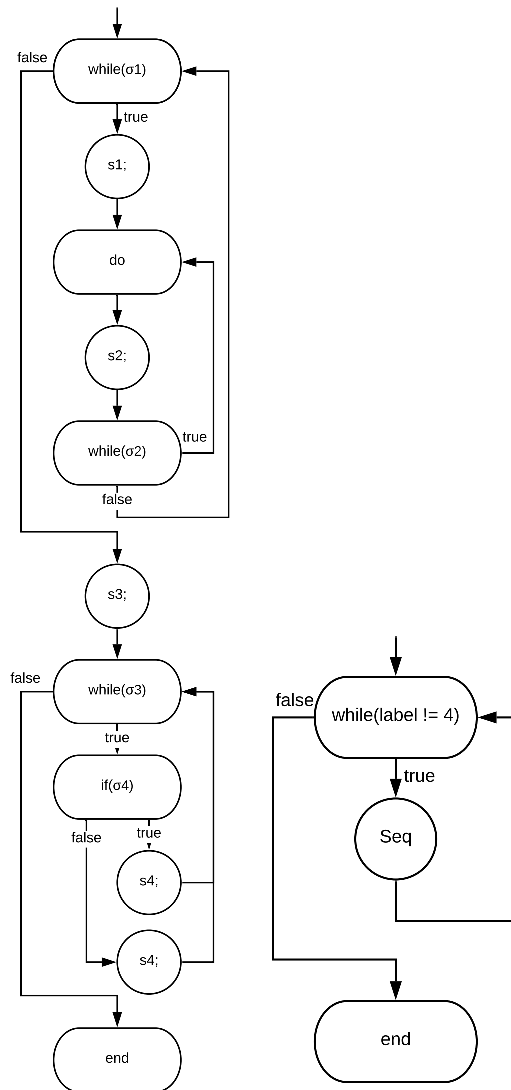
Abbildung 3.12.: *Vergleich der möglichen Darstellungen von currentLabel und deren Auswirkung auf die Programmstruktur*

3.5. Beweis

Im Folgenden wird gezeigt, dass ein beliebiges Programm vor und nach der Loop-Conversion das gleiche Verhalten aufweist. Es wird ein beliebiges Programm nach Inline und For-Schleifen Ersetzung betrachtet.
O.B.d.A hat das Programm $n \in \mathbb{N}$ LFBs. Vor jeden LFB werden Zustandsmarkierungen z_0 bis z_{n-1} eingefügt. Zu einen LFB x wird die Zustandsteiländerungsmarkierung a_x zum Schluss hinzugefügt. Also hat das Programm vor Ausführung des LFB x den Zustand z_{x-1} und danach den Zustand z_{x-1} plus so viele Teiländerungen a_x wie oft der LFB ausgeführt wurde. Dieser Zustand plus jene Teiländerungen, entsprechen dann dem nachfolgenden Zustand z_x.
Eine Schleife erreicht den nächsten Zustand nur, wenn die Schleifenbedingung nicht hält. Und eine Sequenz erreicht den nachfolgenden Zustand, wenn der Programmcode ausgeführt ist. Nun bekommt jeder LFB seine vorhergehende Zustandsmarkierung m_v als gedachten Guard zugewiesen. Also kann ein LFB erst ausgeführt werden wenn der Zustand m_v herrscht. Dies entspricht dem Label nach der Umstrukturierung.
Angenommen die Zustandsteiländerungsmarkierungen werden, da sie hier Teil eines LFBs sind, mit der Umstrukturierung übernommen. Da der Programmcode eines LFBs bei der Umstrukurierung erhalten bleibt, ändert sich die Schleifenbedingung nach der selben Anzahl an Iterationen. Also werden vor und nach Umstrukturierung die selbe Anzahl an Zustandsteiländerungsmarkierungen ausgeführt. Nun herrscht sowohl vor, als auch nach der Umstrukturierung der gleiche, neue Zustand. Diese Zustandsänderung entspricht dem umsetzen des Labels.
Also werden vor und nach der Umstrukturierung die selben LFBs in gleicher Anzahl und Reihenfolge ausgeführt.

3.6. Auswertung

Der vorgestellte Algorithmus erreicht das Ziel den Kontrollfluss beliebiger Programme auf eine einzige Schleife zu reduzieren. Zum Vergleich ist in Abbildung 3.13 der Kontrollflussgraph des Beispielprogramms aus Abb. 3.10 (vor Transformation) und Abb. 3.11 (nach Transformation) dargestellt. Der Kontrollflussgraph beliebiger Programme nach Transformation unterscheidet sich von 3.13 b) nur in der Bedingung der While-Schleife.



(a) Beispiel Kontrollfluss vorher **(b)** Beispiel Kontrollfluss nachher

Abbildung 3.13.: *Vergleich Kontrollfluss vor und nach Transformation*

Um eine quantitative Auswertung des Algorithmus zu ermöglichen wurde dieser für Programme der Sprache MiniC implementiert. Anschließenend mithilfe, der in Anhang A aufgeführten, Tests überprüft.
Die Auswertung hat das Ziel die Transformation möglichst Plattform unabhängig zu bewerten, um Hardwareeffekte auszuschließen. Sowohl auf die Ausgangsprogramme, als auch auf die transformierten Programme, wurden keine weiteren Optimierungen angewandt, außer der Transformation selbst. Für beide Programme, vor und nach der Transformation, wurde ein Durchschnittswert für ILP gebilden. Dafür wurde die Häufigkeit jedes Basic Blocks bei konkreter Ausführung der Programme gemessen und mithilfe von Force Directed Scheduling[PK87] , unter der Annahme von unbegrenzten Ressourcen,

der geringste Schedule pro Basic Block gefunden. Die Anzahl der Instruktionen pro Basic Block wurde durch die ermittelte Länge des Zeitplans geteilt, um einen Durchschnittlichswert für ILP für jeden Basic Block zu bestimmen. Anschließend wurde das gewichtete Arithmetische Mittel aus Häufigkeit und ILP pro Block bestimmt.

Algorithmus	Faktor ILP	Faktor Instruktionen
ShellSort	3.87	21.08
TransHull_OlogN	3.65	23.885
InsertionMaxSort	3.35	18.63
BubbleSort	3.29	9.2
Eratosthenes	3.15	19.16
Daxpy	2.83	4.01
InnerProduct	2.52	7.89
SumUp	2.5	4.83
Heron	1.6	3.59
Fibonacci	1.59	4.95

Tabelle 3.5.: *Zuwachs von ILP und ausgeführte Instruktionen nach Loop-Conversion*

In Tabelle 3.5 ist der Zuwachs sowohl von ILP, als auch von ausgeführten Instuktionen für getestete Algorithmen dargestellt. Es zeigt sich, dass ILP mit dieser Transformation im Schnitt um den Faktor 2,8 erhöht wird. Gleichzeitig steigt aber auch die Anzahl an ausgeführter Instruktionen um den Faktor 11,7.

4. Fazit und Ausblick

In dieser Arbeit wurden mehrere Techniken vorgestellt, welche den Kontrollfluss reduzieren. Zu solchen gehören die If-Conversion, Hyperblocks oder auch die Reverse-If-Conversion. Jedoch sind diese Verfahren nicht für allgemeine Programme geeignet oder minimieren den Kontrollfluss nicht vollständig. Daher wurde in dieser Arbeit Loop-Conversion Algorithmus vorgestellt. Der Fokus bei diesem Algorithmus liegt auf der Strukturumwandlung, welche sich im wesentlichen auf die Reduzierung von beliebig vielen Kontrollstrukturen auf eine einzige Kontrollstruktur konzentriert.

In dieser Phase findet die eigentliche Schleifenreduktion statt, indem die einzelnen LFBs ihr Label als Guard zugeteilt bekommen. Abhängig vom Ursprünglichen Kontrollfluss wird das Label nach dem Ausführen eines LFB umgesetzt. So wurde das Ziel dieser Arbeit, den Kontrollfluss zu reduzieren und ILP ohne weitere Optimierungen zu erhöhen, erreicht. Im Schnitt wird ein ILP Zuwachs um den Faktor 2,8 erzielt. Allerdings erhöht sich im Gegenzug die Anzahl an auszuführenden Instruktionen stark.

Für zukünftige Arbeiten ist noch zu untersuchen, wie der Loop Conversion Algorithmus an sich verbessert werden könnte. Insbesondere könnte der Zuwachs an auszuführenden Instruktionen verkleinert werden. Denkbar wäre hier, analog zu Hyperblöcken, das auschließen selten ausgeführter Basic Blocks. Zusätzlich könnte noch das Zusammenspiel des Loop Conversion Algorithmus mit verschiedenen, bereits bestehenden, Optimierungen untersucht werden. Zu nennen wäre hier vor allem Software-Pipelining.

Literatur

[AHM97] D.I. August, W.W. Hwu und S.A. Mahlke. „A Framework for Balancing Control Flow and Predication“. In: *Microarchitecture (MICRO)*. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, S. 92–103.

[All+83] J.R. Allen, K. Kennedy, C. Porterfield und J. Warren. „Conversion of control dependence to data dependence“. In: *Principles of Programming Languages (POPL)*. Austin, Texas, USA: ACM, 1983, S. 177–189.

[Arn+00] Matthew Arnold, Stephen Fink, Vivek Sarkar und Peter F. Sweeney. „A Comparative Study of Static and Profile-Based Heuristics for Inlining“. In: *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*. DYNAMO ’00. New York, NY, USA: Association for Computing Machinery, 2000, S. 52–64. ISBN: 1581132417. DOI: 10.1145/351397.351416.

[BSS15] N. Bhardwaj, M. Senftleben und K. Schneider. „Abacus - A Processor Family for Education“. In: *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*. Hrsg. von M. Törngren und M.E. Grimheden. New Delhi, India: ACM, 2015, 2:1–2:8.

[Das03] Dibyendu Das. „Function Inlining versus Function Cloning“. In: *SIGPLAN Not.* 38.4 (Apr. 2003), S. 18–24. ISSN: 0362-1340. DOI: 10.1145/844091.844097.

[Hwu+95] W.W. Hwu, R.E. Hank, D.M. Gallagher, S.A. Mahlke, D.M. Lavery, G.E. Haab, J.C. Gyllenhaal und D.I. August. „Compiler technology for future microprocessors“. In: *Proceedings of the IEEE* 83.12 (1995), S. 1625–1640.

[Mah+92] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank und R.A. Bringmann. „Effective compiler support for predicated execution using the hyperblock“. In: *Microarchitecture (MICRO)*. Portland, Oregon, USA: IEEE Computer Society, 1992, S. 45–54.

[Mah+94] S.A. Mahlke, R.E. Hank, R.A. Bringmann, J.C. Gyllenhaal, D.M. Gallagher und W.W. Hwu. „Characterizing the impact of predicated execution on branch prediction“. In: *Microarchitecture (MICRO)*. San Jose, California, USA: IEEE Computer Society, 1994, S. 217–227.

[Mah+95] S.A. Mahlke, R.E. Hank, J.E. McCormick, D.I. August und W.W. Hwu. „A comparison of full and partial predicated execution support for ILP processors“. In: *ACM SIGARCH Computer Architecture News* 23.2 (1995), S. 138–150.

[PK87] P.G. Paulin und J.P. Knight. „Force-Directed Scheduling in Automatic Data Path Synthesis“. In: *Design Automation Conference (DAC)*. Hrsg. von A. O'Neill und D. Thomas. Miami Beach, Florida, USA: ACM, 1987, S. 195–202.

[PS91] J.C.H. Park und M. Schlansker. *On Predicated Execution*. Technical Report HPL-91-58. Hewlett-Packard Company, 1991.

[PS94] D.N. Pnevmatikatos und G.S. Sohi. „Guarded Execution and Branch Prediction in Dynamic ILP Processors“. In: *International Symposium on Computer Architecture (ISCA)*. Chicago, Illinois, USA, 1994, S. 120–129.

[Sch+97] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang und C.L. Thompson. „Compilers for Instruction-level Parallelism“. In: *IEEE Computer* 30.12 (1997), S. 63–69.

[Wal91] D.W. Wall. „Limits of instruction-level parallelism“. In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Santa Clara, California, USA: ACM, 1991, S. 176–188.

[War+93] N.J. Warter, S.A. Mahlke, W.W. Hwu und B. Ramakrishna Rau. „Reverse If Conversion“. In: *Programming Language Design and Implementation (PLDI)*. Hrsg. von R. Cartwright. Albuquerque, New Mexico, USA: ACM, 1993, S. 290–299.

A. Tests: Loop-Conversion

A.1. Basisfälle

<table>
<tr><th>Qualitativ</th><th>Eing. MiniC</th><th>Ausgabe MiniC</th></tr>
<tr><td>s1;</td><td>thread t {
nat x1, x2, x3;
x1 = 1;
x2 = 2;
x3 = 3;
}</td><td>thread t{
{
nat _label;
{
nat x1,nat x2,nat x3;
_label = 0;
x1 = 1;
x2 = 2;
x3 = 3;
}}}</td></tr>
<tr><td>while(b1){
s1;
}</td><td>thread t {
bool b1;
nat x1;
while (b1){
x1 = 1;
}
}</td><td>thread t{
{
nat _label,bool _c0,bool _c1;
{
bool b1,nat x1;
_label = 0;
_label = ((_label == 0) ? 2 : _label);
while((_label != 1)) {
_c0 = ((_label == 2) ? b1 : false);
_c1 = ((_label == 2) ? !(_c0) : false);
_label=((_label == 2)?(_c0 ? 2 : 1):_label);
x1 = ((_label == 2) ? 1 : x1);
}}}}</td></tr>
<tr><td>s1;
while(b1){
s2;
}
s3;</td><td>thread t {
bool b1;
nat x1,x2,x3;
x1 = 1;
while (b1) {
x2 = 2;
}
x3 = 3;
}</td><td>thread t{
{
nat _label,bool _c0,bool _c1;
{
bool b1,nat x1,nat x2,nat x3;
_label = 0;
x1 = 1;
_label = ((_label == 0) ? 2 : _label);
while((_label != 1)) {
_c0 = ((_label == 2) ? b1 : false);
_c1 = ((_label == 2) ? !(_c0) : false);
_label=((_label == 2)?(_c0 ? 2 : 1):_label);
x2 = ((_label == 2) ? 2 : x2);
}
x3 = ((_label == 1) ? 3 : x3);
}}}</td></tr>
</table>

Qualitativ	Eing. MiniC	Ausgabe MiniC

do s1;
while(b1)
	thread t {
	    bool b1;
	    nat x1,x2,x3;
	    do
	      x1 =1;
	    while(b1)
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1;
		{
		bool b1,nat x1,nat x2,nat x3;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		    x1 = ((_label == 2) ? 1 : x1);
		    _c0 = ((_label == 2) ? b1 : false);
		    _c1 = ((_label == 2) ? !(_c0) : false);
		    _label=((_label == 2)?(_c0 ? 2 : 1):_label);
		}}}}

s1;
do s2;
while(b1)
s3;
	thread t {
	    bool b1;
	    nat x1,x2,x3;
	    x1 = 1;
	    do
	      x2 = 2;
	    while(b1)
	    x3 = 3;
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1;
		{
		bool b1,nat x1,nat x2,nat x3;
		_label = 0;
		x1 = 1;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		    x2 = ((_label == 2) ? 2 : x2);
		    _c0 = ((_label == 2) ? b1 : false);
		    _c1 = ((_label == 2) ? !(_c0) : false);
		    _label=((_label == 2)?(_c0 ? 2 : 1):_label);
		}
		x3 = ((_label == 1) ? 3 : x3);
		}}}

for(i=x1..x2){
    s1
};
	thread t {
	    nat x1,x2,x3,i;
	    x1 = 1;
	    x2 = 2;
	    for (i = x1..x2){
	        x3 = 3;
	    }
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1;
		{
		nat x1,nat x2,nat x3,nat i,nat i;
		_label = 0;
		x1 = 1;
		x2 = 2;
		i = x1;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		    _c0 = ((_label == 2) ? (i <= x2) : false);
		    _c1 = ((_label == 2) ? !(_c0) : false);
		    _label=((_label == 2)?(_c0 ? 2 : 1):_label);
		    x3 = ((_label == 2) ? 3 : x3);
		    i = ((_label == 2) ? (i + 1) : i);
		}}}}

Qualitativ	Eing. MiniC	Ausgabe MiniC

if(b1) s1;
else s2;
	thread t {
	nat x1,x2,x3;
	bool b1;
	if (b1)
	x1 = 1;
	else
	x2 = 1;
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1;
		{
		nat x1,nat x2,nat x3,bool b1;
		_label = 0;
		_c0 = ((_label == 0) ? b1 : false);
		_c1 = ((_label == 0) ? !(_c0) : false);
		x1 = (_c0 ? 1 : x1);
		x2 = (_c1 ? 1 : x2);
		}}}

A.2. Sequenzielle Schleifen

Qualitativ	Eing. MiniC	Ausgabe MiniC

while(b1){s1;}
while(b2){s2;}
	thread t {
	nat x1,x2;
	bool b1,b2;
	while (b1) {
	x1 =1;
	}
	while (b2) {
	x2 = 2;
	}
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{
		nat x1,nat x2,bool b1,bool b2;
		_label = 0;
		_label = ((_label == 0) ? 3 : _label);
		while((_label != 1)) {
		_c0 = ((_label == 3) ? b1 : false);
		_c1 = ((_label == 3) ? !(_c0) : false);
		_label=((_label == 3)?(_c0 ? 3 : 2):_label);
		x1 = ((_label == 3) ? 1 : x1);
		_label = ((_label == 2) ? 5 : _label);
		_c2 = ((_label == 5) ? b2 : false);
		_c3 = ((_label == 5) ? !(_c2) : false);
		_label=((_label == 5)?(_c2 ? 5 : 1):_label);
		x2 = ((_label == 5) ? 2 : x2);
		}}}}

while(b1){s1;}
s2;
while(b2){s3;}
	thread t {
	nat x1,x2,x3;
	bool b1,b2;
	while (b1) {
	x1 =1;
	}
	x2 = 2;
	while (b2) {
	x3 = 3;
	}
	}
		thread t{
		{ nat _label,bool _c0,bool _c1,bool _c2,bool _c3; {
		nat x1,nat x2,nat x3,bool b1,bool b2;
		_label = 0;
		_label = ((_label == 0) ? 3 : _label);
		while((_label != 1)) {
		_c0 = ((_label == 3) ? b1 : false);
		_c1 = ((_label == 3) ? !(_c0) : false);
		_label=((_label == 3)?(_c0 ? 3 : 2):_label);
		x1 = ((_label == 3) ? 1 : x1);
		x2 = ((_label == 2) ? 2 : x2);
		_label = ((_label == 2) ? 5 : _label);
		_c2 = ((_label == 5) ? b2 : false);
		_c3 = ((_label == 5) ? !(_c2) : false);
		_label=((_label == 5)?(_c2 ? 5 : 1):_label);
		x3 = ((_label == 5) ? 3 : x3);
		}}}}

<table>
<tr><th>Qualitativ</th><th>Eing. MiniC</th><th>Ausgabe MiniC</th></tr>
<tr><td>do s1;
while(b1)
do s2;
while(b2)</td><td>thread t {
nat x1,x2;
bool b1,b2;
do {
x1 =1;
} while (b1)
do {
x2 = 2;
} while (b2)
}</td><td>thread t{
{
nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
{
nat x1,nat x2,bool b1,bool b2;
_label = 0;
_label = ((_label == 0) ? 3 : _label);
while((_label != 1)) {
x1 = ((_label == 3) ? 1 : x1);
_c0 = ((_label == 3) ? b1 : false);
_c1 = ((_label == 3) ? !(_c0) : false);
_label=((_label == 3)?(_c0 ? 3 : 2):_label);
_label = ((_label == 2) ? 5 : _label);
x2 = ((_label == 5) ? 2 : x2);
_c2 = ((_label == 5) ? b2 : false);
_c3 = ((_label == 5) ? !(_c2) : false);
_label=((_label == 5)?(_c2 ? 5 : 1):_label);
}}}}</td></tr>
<tr><td>if(b1) s1;
else s2;
if(b2) s3;
else s4;</td><td>thread t {
nat x1,x2;
bool b1, b2;
if (b1) {
x1 = 1;
} else {
x1 = 11;
}
if (b2) {
x2 = 2;
} else {
x2 = 22;
}
}</td><td>thread t{
{
nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
{
nat x1,nat x2,bool b1,bool b2;
_label = 0;
_c0 = ((_label == 0) ? b1 : false);
_c1 = ((_label == 0) ? !(_c0) : false);
x1 = (_c0 ? 1 : x1);
x1 = (_c1 ? 11 : x1);
_c2 = ((_label == 0) ? b2 : false);
_c3 = ((_label == 0) ? !(_c2) : false);
x2 = (_c2 ? 2 : x2);
x2 = (_c3 ? 22 : x2);
}}}</td></tr>
<tr><td>while(b1){s1;}
if(b2) s2;
else s3;</td><td>thread t {
nat x1,x2;
bool b1, b2;
while (b1) {
x1 = 1;
}
if (b2) {
x2 = 2;
} else {
x2 = 22;
}
}</td><td>thread t{
{
nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
{
nat x1,nat x2,bool b1,bool b2;
_label = 0;
_label = ((_label == 0) ? 2 : _label);
while((_label != 1)) {
_c0 = ((_label == 2) ? b1 : false);
_c1 = ((_label == 2) ? !(_c0) : false);
_label=((_label == 2)?(_c0 ? 2 : 1):_label);
x1 = ((_label == 2) ? 1 : x1);
}
_c2 = ((_label == 1) ? b2 : false);
_c3 = ((_label == 1) ? !(_c2) : false);
x2 = (((_label == 1) & _c2) ? 2 : x2);
x2 = (((_label == 1) & _c3) ? 22 : x2);
}}}</td></tr>
</table>

Qualitativ	Eing. MiniC	Ausgabe MiniC

if(b1) s1;
else s2;
while(b2){s3;}
	thread t {
	nat x1,x2;
	bool b1, b2;
	if (b1) {
	x1 = 2;
	} else {
	x1 = 22;
	}
	while (b2) {
	x2 = 1;
	}
	}
		thread t{
		{
		nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{
		nat x1,nat x2,bool b1,bool b2;
		_label = 0;
		_c0 = ((_label == 0) ? b1 : false);
		_c1 = ((_label == 0) ? !(_c0) : false);
		x1 = (_c0 ? 2 : x1);
		x1 = (_c1 ? 22 : x1);
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		_c2 = ((_label == 2) ? b2 : false);
		_c3 = ((_label == 2) ? !(_c2) : false);
		_label=((_label == 2)?(_c2 ? 2 : 1):_label);
		x2 = ((_label == 2) ? 1 : x2);
		}}}}

A.3. Verschachtelte Schleifen

Qualitativ	Eing. MiniC	Ausgabe MiniC

while(b1){
while(b2){
s1;
}}
	thread t {
	nat x1;
	bool b1, b2;
	while (b1) {
	while (b2) {
	x1 = 1;
	}
	}
	}
		thread t{
		{nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{nat x1,bool b1,bool b2;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		_c2 = ((_label == 2) ? b1 : false);
		_c3 = ((_label == 2) ? !(_c2) : false);
		_label = ((_label == 2)?(_c2 ? 2:1):_label);
		_label = ((_label == 2) ? 4 : _label);
		_c0 = ((_label == 4) ? b2 : false);
		_c1 = ((_label == 4) ? !(_c0) : false);
		_label = ((_label == 4)?(_c0 ? 4:3):_label);
		x1 = ((_label == 4) ? 1 : x1);
		_label = ((_label == 3) ? 2 : _label);
		}}}}

while(b1){
s1;
while(b2){
s2;
}}
	thread t {
	nat x1,x2;
	bool b1, b2;
	while (b1) {
	x1 = 1;
	while (b2) {
	x2 = 2;
	}
	}
	}
		thread t{
		{nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{nat x1,nat x2,bool b1,bool b2;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		_c2 = ((_label == 2) ? b1 : false);
		_c3 = ((_label == 2) ? !(_c2) : false);
		_label = ((_label == 2)?(_c2 ? 2 : 1):_label);
		x1 = ((_label == 2) ? 1 : x1);
		_label = ((_label == 2) ? 4 : _label);
		_c0 = ((_label == 4) ? b2 : false);
		_c1 = ((_label == 4) ? !(_c0) : false);
		_label = ((_label == 4)?(_c0 ? 4 : 3):_label);
		x2 = ((_label == 4) ? 2 : x2);
		_label = ((_label == 3) ? 2 : _label);
		}}}}

if(b1){
while(b2){
s1;
}}
else {s2;}
	thread t {
	nat x1,x2;
	bool b1, b2;
	if (b1){
	while(b2) {
	x1 = 1;
	}
	} else {
	x2 = 2;
	}
	}
		thread t{
		{nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{nat x1,nat x2,bool b1,bool b2;
		_label = 0;
		_c0 = ((_label == 0) ? b1 : false);
		_c1 = ((_label == 0) ? !(_c0) : false);
		_label = ((_c0 & (_label == 0)) ? 2 : _label);
		x2 = (_c1 ? 2 : x2);
		while((_label != 1)) {
		_c2 = ((_c0 & (_label == 2)) ? b2 : false);
		_c3 = ((_c0 & (_label == 2)) ? !(_c2) : false);
		_label = ((_c0&(_label == 2))?(_c2?2:1):_label);
		x1 = (((_label == 2) & _c0) ? 1 : x1);
		}}}}

Qualitativ	Eing. MiniC	Ausgabe MiniC

if(b1){
while(b2){
s1;
}}
else{
while(b3){
s2;
}}
	thread t {
	nat x1, x2;
	bool b1,b2,b3;
	if (b1){
	while(b2) {
	x1=1;
	}
	} else {
	while(b3) {
	x2=2;
	}
	}
	}
		thread t{
		{nat _label,bool _c0,bool _c1,bool _c2;
		bool _c3,bool _c4,bool _c5;
		{
		nat x1,nat x2,bool b1,bool b2,bool b3;
		_label = 0;
		_c0 = ((_label == 0) ? b1 : false);
		_c1 = ((_label == 0) ? !(_c0) : false);
		_label = ((_c0 & (_label == 0)) ? 2 : _label);
		_label = ((_c1 & (_label == 0)) ? 4 : _label);
		while((_label != 1)) {
		_c2 = ((_c0 & (_label == 2)) ? b2 : false);
		_c3 = ((_c0 & (_label == 2)) ? !(_c2) : false);
		_label=((_c0&(_label == 2))?(_c2?2:1):_label);
		x1 = (((_label == 2) & _c0) ? 1 : x1);
		_c4 = ((_c1 & (_label == 4)) ? b3 : false);
		_c5 = ((_c1 & (_label == 4)) ? !(_c4) : false);
		_label=((_c1&(_label == 4))?(_c4?4:1):_label);
		x2 = (((_label == 4) & _c1) ? 2 : x2);
		}}}}

while(b1){
if(b2) s1;
else s2;
}
	thread t {
	nat x1, x2;
	bool b1,b2,b3;
	while(b1){
	if (b2)
	x1=1;
	else
	x2=2;
	}
	}
		thread t{
		{nat _label,bool _c0,bool _c1,bool _c2,bool _c3;
		{
		nat x1,nat x2,bool b1,bool b2,bool b3;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		_c2 = ((_label == 2) ? b1 : false);
		_c3 = ((_label == 2) ? !(_c2) : false);
		_label=((_label == 2)?(_c2?2:1):_label);
		_c0 = ((_label == 2) ? b2 : false);
		_c1 = ((_label == 2) ? !(_c0) : false);
		x1 = (((_label == 2) & _c0) ? 1 : x1);
		x2 = (((_label == 2) & _c1) ? 2 : x2);
		}}}}

Qualitativ	Eing. MiniC	Ausgabe MiniC

do {
    s1;
    while(b2){
        s2;
    }
    s3;
} while (b1)
	thread t {
	    nat x1, x2, x3;
	    bool b1, b2;
	    do {
	        x1 = 1;
	        while (b2) {
	            x2 = 2;
	        }
	        x3 = 3;
	    } while (b1)
	}
		thread t{
		{
		nat _label, bool _c0, bool _c1, bool _c2, bool _c3;
		{
		nat x1, nat x2, nat x3, bool b1, bool b2;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		    x1 = ((_label == 2) ? 1 : x1);
		    _label = ((_label == 2) ? 4 : _label);
		    _c0 = ((_label == 4) ? b2 : false);
		    _c1 = ((_label == 4) ? !(_c0) : false);
		    _label=((_label == 4)?(_c0 ? 4 : 3):_label);
		    x2 = ((_label == 4) ? 2 : x2);
		    x3 = ((_label == 3) ? 3 : x3);
		    _c2 = ((_label == 3) ? b1 : false);
		    _c3 = ((_label == 3) ? !(_c2) : false);
		    _label=((_label == 3)?(_c2 ? 2 : 1):_label);
		}}}}

while(b1){
    s1;
    do
        s2;
    while(b2)
    s3;
}
	thread t {
	    nat x1, x2, x3;
	    bool b1, b2;
	    while(b1) {
	        x1=1;
	        do{
	            x2=2;
	        }while(b2)
	        x3 =3;
	    }
	}
		thread t{
		{
		nat _label, bool _c0, bool _c1, bool _c2, bool _c3;
		{
		nat x1, nat x2, nat x3, bool b1, bool b2;
		_label = 0;
		_label = ((_label == 0) ? 2 : _label);
		while((_label != 1)) {
		    _c2 = ((_label == 2) ? b1 : false);
		    _c3 = ((_label == 2) ? !(_c2) : false);
		    _label = ((_label == 2)?(_c2 ? 2 : 1):_label);
		    x1 = ((_label == 2) ? 1 : x1);
		    _label = ((_label == 2) ? 4 : _label);
		    x2 = ((_label == 4) ? 2 : x2);
		    _c0 = ((_label == 4) ? b2 : false);
		    _c1 = ((_label == 4) ? !(_c0) : false);
		    _label = ((_label == 4)?(_c0 ? 4 : 3):_label);
		    x3 = ((_label == 3) ? 3 : x3);
		    _label = ((_label == 3) ? 2 : _label);
		}}}}