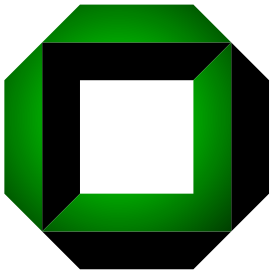


Universität Karlsruhe (TH)

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

If-Konversion auf SSA

Studienarbeit von Christoph H. Mallon

März 2007

Betreuer:
Dipl.-Inform. Sebastian Hack
Dipl.-Inform. Rubino Geiß
Dipl.-Inform. Michael Beck

Verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Zusammenfassung

Diese Studienarbeit stellt einen Algorithmus zur Durchführung der If-Konversion vor. Die If-Konversion ist eine Optimierung mit dem Ziel bedingte Sprünge zu eliminieren. Dies ist für heutige Prozessoren, die üblicherweise nach dem Fließbandprinzip arbeiten, eine interessante Optimierung.

Die Implementierung des Algorithmus erfolgt auf der SSA basierten Zwischensprache Firm, welche am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe im Rahmen des CRS (Compiler Research System) entwickelt wurde. In dieser Arbeit wird daher besonders auf die durch statische Einmalzuweisung gegebenen Eigenschaften eingegangen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Bei heutigen Prozessoren, die üblicherweise Befehle in mehreren Stufen verarbeiten, bedeutet ein falsch vorhergesagter Sprung den Verlust vieler Takte und somit eine Verringerung der Ausführungsgeschwindigkeit. Die in dieser Arbeit vorgestellte If-Konversion versucht einige dieser bedingten Sprünge zu eliminieren.

Ziel dieser Studienarbeit ist es, einen Algorithmus für die If-Konversion für das am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelte Compiler Research System (CRS) zu implementieren und zu bewerten.

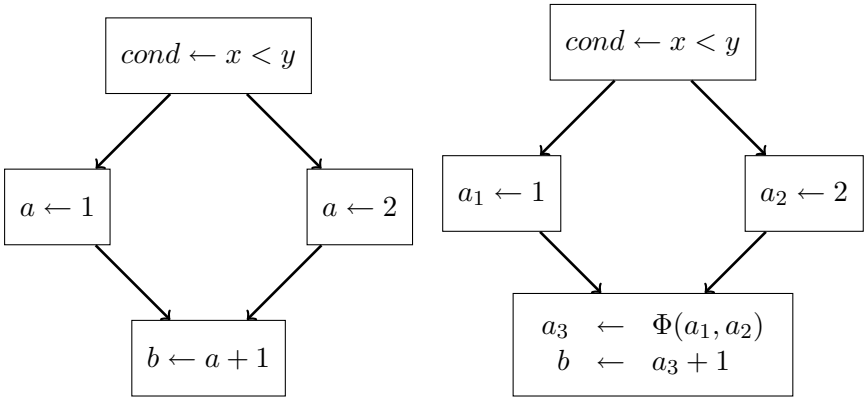
Zu diesem Thema existieren bereits viele Arbeiten, jedoch stellte sich bei deren Durchsicht heraus, dass diese statische Einmalzuweisung (engl. SSA), eine Eigenschaft auf die im CRS verwendeten Zwischensprache Firm besonderes Augenmerk gelegt wird, wenig beachten. So verschob sich der Schwerpunkt auf das Finden eines in Bezug auf SSA geeigneten Algorithmus und Implementierung dessen.

Kapitel 2

Grundlagen

In diesem Abschnitt wird ein kurzer Überblick über die SSA-Eigenschaft von Programmen, die in der Implementierung verwendete Zwischensprache Firm sowie eine Motivation und Beschreibung der Idee hinter der If-Konversion gegeben.

2.1 SSA



(a) Klassisch (b) SSA-Form

Abbildung 2.1: Programmdarstellungen

Statische Einmalzuweisung (Static Single Assignment, SSA) ist eine Darstellung für Programme, in der jede Variable genau eine Definitionsstelle besitzt. Somit ist an jeder Benutzungstelle eines Wertes dessen Definition eindeutig bestimmbar. Diese Darstellung wird durch eine Umformung des Programms erreicht, bei der für jede Zuweisung an eine Variable ein eindeutiger neuer Name vergeben wird. Erreichen mehrere solche Variablen einen Grundblock, so wird mit Hilfe einer speziellen Φ-Auswahlfunktion der gemäß

des Steuerflusses gültige Wert ausgewählt. Diese Φ-Funktionen werden den Ansatzpunkt für die If-Konversion bilden.

Abbildung 2.1 zeigt einen einfachen Programmgraphen in klassischer und in SSA-Form. In der klassischen Variante hat die Variable a zwei Definitionsstellen. In der SSA-Form werden diese beiden Definitionen eindeutig mit a_1 bzw. a_2 benannt. Beim Aufeinandertreffen der beiden Definitionen werden diese durch eine Φ-Funktion zu einer neuen Definition a_3 zusammengeführt.

Ein Algorithmus zum effizienten Aufbau der SSA-Form wird in [CFR+91] vorgestellt.

2.2 Firm und libFirm

Firm ist eine graphbasierte Zwischendarstellung, die ausschließlich SSA-Darstellung unterstützt. Details über diese Zwischendarstellung sind in [TLB99] nachzulesen.

Eine Besonderheit von Firm besteht darin, dass Datenabängigkeiten und nicht Datenfluss verwendet werden. So verweist zum Beispiel ein Additionsknoten auf seine beiden Operanden.

In Anlehnung daran verweisen Grundblöcke auf ihre Vorgänger im Steuerfluss und nicht auf ihre Nachfolger. Allerdings ist hier zu beachten, dass es sich nicht wie bei den Datenabhängigkeiten um eine Abhängigkeit handelt. So kann eine Addition nur durchgeführt werden, wenn alle Operanden verfügbar sind. Im Programmlauf wird jedoch exakt ein Vorgänger ausgeführt. Dies vereinfacht lediglich die Handhabung bei Programmtransformationen, insbesondere in Zusammenhang mit Φ-Funktionen, da diese genau so viele Operanden besitzen wie ihr zugeordneter Grundblock Steuerflussvorgänger besitzt und Transformationen daran – zum Beispiel das Entfernen eines Vorgängers im Steuerfluss – im Einklang behandelt werden müssen.

Die Implementierung des später vorgestellten If-Konversionsalgorithmus erfolgte basierend auf libFirm, einer Implementierung dieser Zwischendarstellung in C.

2.3 If-Konversion

Auf Architekturen mit langen Fließbändern und spekulativer Ausführung bedeutet ein falsch vorhergesagter bedingter Sprung den Verlust vieler Takte, da die spekulativ ausgeführten Befehle verworfen und das Fließband zunächst mit neuen Befehlen bestückt werden muss.

Die Idee der If-Konversion besteht darin, die Werte aller alternativen Programmpfade auszurechnen und danach anhand der Bedingungen die gewünschten Werte auszuwählen. Es werden bei diesem Vorgehen zwar die Werte aller Pfade berechnet, was zusätzlichen Aufwand bedeutet, jedoch

werden dadurch bedingte Sprünge und somit Quellen für falsche Sprungvorhersagen eliminiert. Ist der Mehraufwand geringer als die verlorene Zeit durch Fehlvorhersagen, bedeutet das insgesamt eine schnellere Ausführung. Dies ist besonders für schwer vorhersagbare Sprünge in oft ausgeführten Programmteilen interessant. Zudem besteht bei superskalaren Archtitekturen, die mehrere Befehele parallel ausführen können, die Möglichkeit, dass der Mehraufwand durch bessere Nutzung von zuvor nicht ausgelasteten Auführungseinheiten aufgefangen wird.

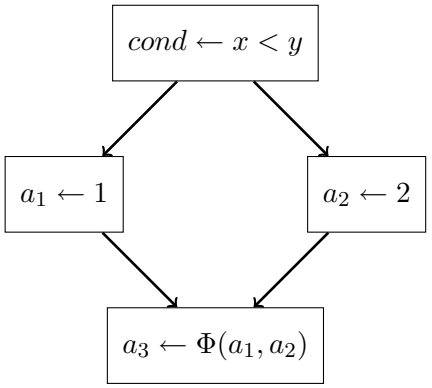
Die maschinenspezifische Umsetzung fällt sehr unterschiedlich aus. Bei Architekturen, die prädizierte Befehlsausführung unterstüzen (z.B. IA64), kann diese direkt für die bedingte Berechnung der Programmpfade genutzt werden. Auf andere Architekturen, die einen Auswahlbefehl (POWER) oder bedingten Zuweisungsbefehl (x86) besitzen, kann dieser benutzt werden, um damit das Ergebnis aus allen berechneten Werten auszuwählen. Manche Prozessoren hingegen bieten spezielle Befehle zur Bildung des Absolutwertes einer Zahl oder dem Bestimmen des Minimums bzw. Maximus zweier Zahlen. Diese Spezialfälle lassen sich bei der If-Konversion durch einfache lokale Betrachtung des Transformationsergebnisses erkennen. Auch bei Architekturen, die keine dieser Mechanismen besitzen, ist es möglich einige Fälle mit Hilfe von geschickt angewandten Bitoperationen umzusetzen.

Kapitel 3

Lösungsansatz

Die Herangehensweise an die If-Konversion wird im Folgenden an einem einfachen Beispiel erläutert. Hierbei wird eine neue Operation Ψ, die den zentralen Aspekt darstellt, eingeführt und deren genaue Semantik im darauf folgenden Abschnitt dargelegt. Das eigentliche Verfahren zur Durchführung der Transformation ist Inhalt des nächsten Kapitels.

3.1 Einführendes Beispiel



(a) Ausgangsprogrammgraph

$$\begin{aligned} a_1 &\leftarrow 1 \\ a_2 &\leftarrow 2 \\ cond &\leftarrow x < y \\ a_3 &\leftarrow \Psi(cond, a_1, a_2) \end{aligned}$$

(b) Transformierter Programmgraph

Abbildung 3.1: Elementares Beispiel für If-Konversion

Abbildung 3.1 zeigt ein einfaches Beispiel, in dem der Steuerfluss mit einem bedingten Sprung verzweigt. Die Bedingung ist hier mit der Wertmarke *cond* versehen. Das nachfolgende Φ liefert abhängig vom gewählten Pfad den Wert 1 oder 2.

Durch die If-Konversion wird dieses Konstrukt aus vier Grundblöcken aufgelöst und stattdessen eine Ψ genannte Funktion eingefügt, die anhand der

Bedingung den gewünschten Wert auswählt. Übrig bleibt nur ein einzelner Grundblock ohne bedingten Steuerfluss.

3.2 Semantik der Ψ-Funktion

Ähnlich der Φ-Funktion wählt die Ψ-Funktion aus mehreren Werten einen aus und liefert diesen als Ergebnis. Anders als die Φ-Funktion, welche den Wert anhand des Steuerfluss bestimmt, wählt die Ψ-Funktion durch Datenfluss in Form einer oder mehrerer Bedingungen ihr Ergebnis aus.

Ein Ψ besitzt eine beliebige Anzahl Dateneingänge, zwischen welchen ausgewählt werden soll. Mit jedem dieser Eingänge – außer dem letzten – ist eine Bedingung assoziiert.

$$\Psi(b_1, w_1, ..., b_n, w_n, w_{\text{default}}) = \begin{cases} w_i & \forall j < i : \neg b_j \wedge b_i \\ w_{\text{default}} & sonst \end{cases}$$

b_i bzw. w_i bezeichnen hier die Bedingungen bzw. die Werte. Bedingungen und Werte mit dem selben Index sind einander zugeordnet. Beim Auswerten der Ψ-Funktion wird die Liste der Bedingungen von links nach rechts ausgewertet. Der zu der ersten wahren Bedingungen gehörende Wert ist das Ergebnis der Ψ-Funktion. Falls keine der Bedingungen wahr ist, so ist das Ergebnis der Wert des bedingungslosen Dateneingangs (hier mit w_{default} bezeichnet).

Die einfachste Form eines Ψs besitzt genau einen Bedingungs- und zwei Dateneingänge und entspricht damit einem einfachen If-Then-Else, das einer Variablen abhängig von der Bedingung einen von zwei Werten zuweist.

Kapitel 4

Algorithmus

Im Folgenden wird der verwendete Algorithmus zur Durchführung der If-Konversion dargelegt. Zunächst wird der Begriff der Steuerabhängigkeit eingeführt, welcher für die If-Konversion wichtige Information liefert. Danach wird die Problematik im Zusammenhang mit kritischen Kanten erläutert. Weiter folgt die Beschreibung des Basisfalls des Algorithmus und darauf die Erweiterung auf die Behandlung mehrerer Steuerabhängigkeiten. Zuletzt wird der Spezialfall von Funktionen mit mehrerern Rücksprüngen gelöst.

4.1 Elementare Eigenschaften

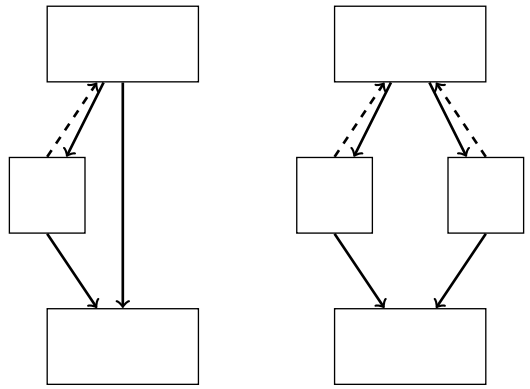
4.1.1 Steuerabhängigkeit

Ein Grundblock B ist steuerabhängig von einem anderen Grundblock A, falls A darüber bestimmt, ob B ausgeführt wird. Genauer ist ein Block B steuerabhängig von einem Block A, genau dann wenn ein Steuerflusspfad von A nach B existiert, jeder Grundblock auf diesem Pfad A nachdominiert und Block B Block A nicht nachdominiert. So sind zum Beispiel die "Then"- und "Else"-Blöcke eines If-Ausdrucks steuerabhängig vom Grundblock, der die Verzweigung in diese enthält. Ein Algorithmus zur Konstruktion des Steuerabhängigkeitsgraphen ist in [Muc97] zu finden.

4.1.2 Kritische Kanten

Eine Kante in einem Graphen ist kritisch, genau dann wenn ihr Zielknoten mehrere eingehende, und ihr Quellknoten mehrere ausgehende Kanten besitzt. Im Kontext von Programmgraphen sind dies Steuerflusskanten, deren inzidente Grundblöcke diese Eigenschaft erfüllen.

Für die If-Konversion führen kritische Steuerflusskanten zu fehlenden Steuerabhängigkeitskanten, wie zum Beispiel in Abbildung 4.1 zu sehen ist. Dort ist der Grundblockgraph, annotiert mit Steuerabhängigkeiten (gestrichelte Kanten), für ein einfaches If-Then-Else-Konstrukt mit leerem Else-



(a) Grundblockgraph mit kritischer Kante

(b) Nach Elimination der kritischen Kante

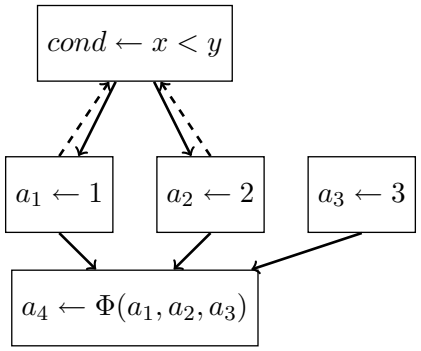
Abbildung 4.1: Kritische Kante und Steuerabhängigkeiten (gestrichelte Kanten)

Teil – und folglich fehlendem Grundblock – zu sehen. Der im nächsten Abschnitt vorgestellte Algorithmus basiert jedoch auf dem Auffinden zweier Grundblöcke, die gemeinsam von einem dritten steuerabhängig sind. Durch die Entfernung aller kritischen Kanten – also durch Einfügen eines leeren Grundblocks auf jeder kritischen Kante – wird die gewünschte Eigenschaft hergestellt.

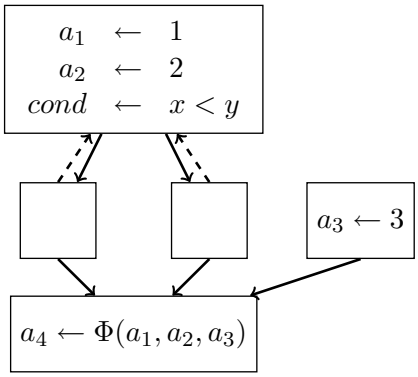
4.2 Kern des Algorithmus

Der Startpunkt des Algorithmus ist ein Grundblock, der mindestens eine Φ-Funktion enthält. Ausgehend von diesem Block werden zwei direkte Vorgängerblöcke gesucht, die eine gemeinsame Steuerabhängigkeit besitzen. Zunächst werden nur Grundblöcke mit genau einer Steuerabhängigkeit betrachtet. Das Vorgehen beim Auftreten mehrerer Steuerabhängigkeiten wird im folgenden Abschnitt dargelegt. Diese beiden Grundblöcke haben somit als gemeinsamen direkten Vorgänger den Block, der den bedingten Sprung in diese enthält.

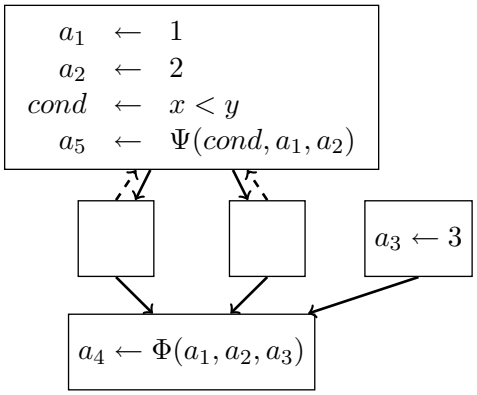
Nun wird überprüft, ob die beiden Blöcke nur seiteneffktfreie Operationen enthalten. Ein Seiteneffekt sei hier eine Änderung des beobachtbaren Zustandes. Darunter fallen schreiben eines Wertes in den Speicher oder auslösen einer Ausnahme bei einem Speicherzugriff. Eine Operation wie Addition auf Registern gilt hier als nicht seiteneffektbehaftet, da die Register nicht zum beobachtbaren Zustand gezählt werden. Falls die verwendete Zielarchitektur Effekte wie das Schreiben auf den Speicher oder das Auslösen einer Ausnahme maskieren kann (z.B. der IA64 unter Verwendung von Prädikatsregistern),



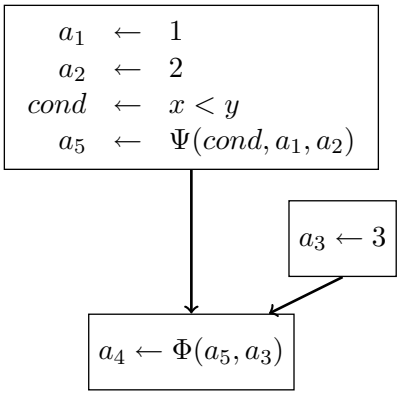
(a) Die Ausgangssituation: Ein Block mit einem Φ. Der Block hat zwei Steuerflussvorgänger, die gemeinsam von einem weiteren Block steuerabhängig (gestrichelte Kanten) sind. Der Block mit dem Wert "3" ist unbeteiligt.



(b) Die Berechnungen in den beiden Blöcken, in diesem Fall nur zwei Konstanten, sind seiteneffektfrei und können somit in ihren gemeinsamen Vorgänger verschoben werden.



(c) Einfügen des Ψ, das anhand der Bedingung einen der beiden Werte auswählt



(d) Verbinden des Φ mit dem Wert des Ψ, Einfügen einer unbedingten Steuerflusskante und Entfernen des bedingten Sprunges sowie seiner enteleerten Nachfolgeblöcke

Abbildung 4.2: Schrittweise Durchführung der If-Konversion

so können weitere Befehlsklassen in die Menge der seiteneffktfreien Befehle aufgenommen werden. Enthalten die beiden Blöcke nur seiteneffektfreie Berechnungen, so kann die If-Konversion durchgeführt werden. Dazu werden zuerst die Inhalte dieser beiden Blöcke in ihren gemeinsamen Vorgänger, der im Speziellen jene auch dominiert, verschoben. Da zuvor überprüft wurde, dass diese Operationen seiteneffektfrei sind, ändert sich das beobachtete Programmverhalten nicht. Dies ist notwendig, da es das Ziel des If-Konversion ist, den bedingten Steuerfluss zu eliminieren, wodurch folglich diese beiden Blöcke entfernt werden.

Danach werden die Φ-Funktionen angepasst. Hierzu werden die Vorgänger jedes Φs, die mit den beiden Grundblöcken assoziiert sind, von den Φs entfernt und zusammen mit der Bedingung des bedingten Sprunges zu einem Ψ mit einer Bedingung und zwei Dateneingängen zusammengefasst. Dieses Ψ wird in den Block, in welchem sich der bedingte Sprung befindet, platziert. Der Ergebniswert des Ψs wird als neuer Eingang mit dem jeweiligen Φ verbunden.

Dieser neue Eingang entspricht dem Grundblock mit der Bedingung, der jetzt neuer Vorgänger des Grundblocks, der die Φs enthält, ist. Die beiden entleerten Blöcke und die bedingten Sprünge in diese werden nun entfernt. Somit ist die If-Konversion an dieser Stelle vollzogen. Es können jedoch mit dem selben Ausgangsgrundblock weitere If-Konversionen mit anderen Vorgängern – einschließlich des neu verbundenen – durchgeführt werden.

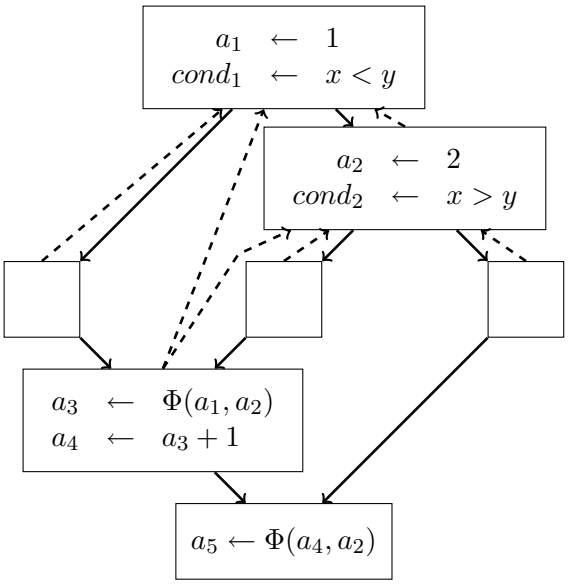
Das Vorgehen im Detail wird graphisch in den Abbildungen 4.2(a) bis 4.2(d) dargelegt.

4.3 Mehrfache Steuerabhängigkeiten

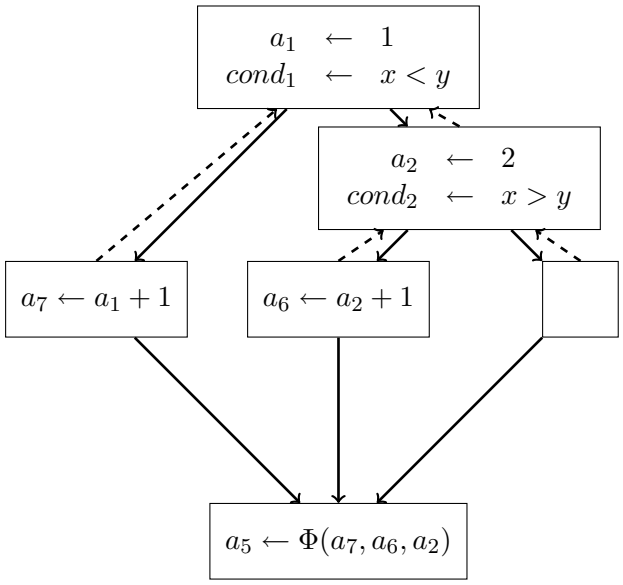
Bei Betrachtung von Abbildung 4.3(a), findet man zwischen der Steuerflusssenke und dem Block mit der Bedingung $cond_2$ beinahe die zuvor beschriebene Ausgangsstellung der If-Konversion: Ein Block mit einem Φ sowie zwei Vorgängerblöcke, die gemeinsam von einem weiteren Grundblock steuerabhängig sind. Jedoch stört der Grundblock mit zwei Steuerabhängigkeiten auf die Blöcke, die $cond_1$ bzw. $cond_2$ enthalten. Dies kann durch Transformation des Programmgraphen behoben werden. Ziel ist es, den Steuerfluss so zu verändern, dass der Basisfall des Algorithmus angewandt werden kann. Hierzu wird der Programmpfad, der über den Block mit mehreren Steuerabhängigkeiten führt, „abgespalten“.

Im Detail bedeutet das, dass alle Operationen in dem Block mit mehreren Steuerabhängigkeiten in den Vorgänger, der an der If-Konversion beteiligt sein soll, kopiert werden. Einzige Ausnahme hierbei bilden die Φs: Diese werden nicht kopiert, sondern der dem abzuspaltenden Pfad entsprechende Datenvorgänger wird verwendet.

Danach wird der Steuerfluss angepasst, d.h. der abgespaltene Pfad wird als Steuerflussvorgänger gestrichen, ebenso alle dazugehörigen Φ-Eingänge.



(a) Ein Programmgraph, der einen Block mit mehreren Steuerabhängigkeiten (gestrichelte Kanten) enthält. Diese Struktur entsteht typischerweise bei logischem Und oder Oder mit Kurzauswertung.



(b) Der Programmgraph nach beseitigen der mehrfachen Steuerabhängigkeiten. Mit dem Block, der $cond_2$ enthält, der Steuerflusssenke und den beiden Blöcken dazwischen kann nun die If-Konversion durchgeführt werden.

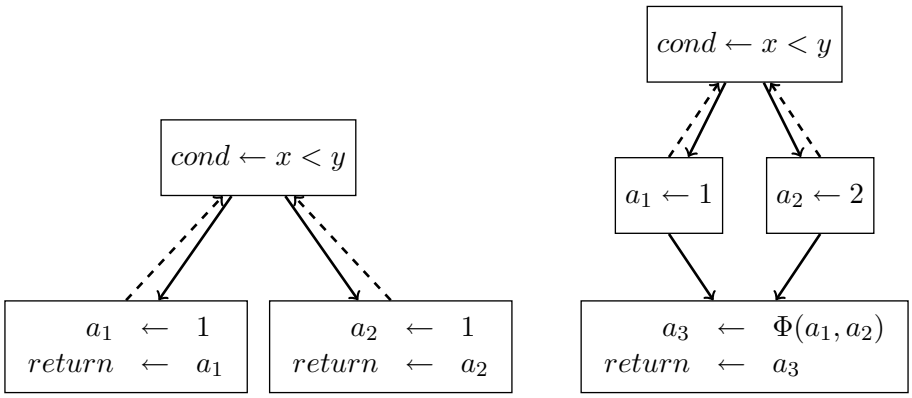
Abbildung 4.3: Auflösen mehrfacher Steuerabhängigkeiten

Sollte jetzt nur noch ein Vorgängerblock verblieben sein, können die beiden verschmolzen werden. Als Steuerflussnachfolger des abgespalteten Blocks fungiert nun der Nachfolger des Blocks mit (eventuell ehemals) mehreren Steuerabhängigkeiten.

Diese Transformation ist möglicherweise mehrfach, falls mehrere Blöcke mit mehrfachen Steuerabhängigkeiten aufeinanderfolgen, oder auf beiden Pfaden der If-Konversion anzuwenden, um den Basisfall herzustellen. Schließlich kann die eigentliche If-Konversion, die zuvor beschrieben wurde, durchgeführt werden.

Das Ergebnis der Transformation am vorgigen Beispielprogrammgraphen ist in 4.3(b) abgebildet.

4.4 Mehrere Rücksprungpunkte



(a) Programmgraph mit mehreren Rückpsprungpunkten

(b) Normalisierte Form mit genau einem Rücksprungpunkt

Abbildung 4.4: Transformation eines Programmgraphen mit mehreren Rücksprungpunkten um ihn mit If-Konversion verarbeiten zu können

Eine Funktion kann mehrere Rücksprünge besitzen. Dies ist eine Auswahl mehrerer möglicher Rückgabewerte, aber ohne Verwendung eines Φs und vor allem ohne eines Grundblockes, der mehrere Steuerflüsse vereinigt, da der Steuerfluss an den jeweiligen Rücksprüngen endet. Dieser Grundblock und das darin enthaltene Φ sind jedoch Vorraussetzung für den Ablauf des vorgestellten Algorithmus.

Die Bearbeitung mehrerer Rücksprungpunkte ist vergleichsweise simpel: Die Rückgabewerte der Rücksprünge werden durch ein Φ in einem neuen Grundblock zusammengefasst. Das Ergebnis des Φs wird nun durch einen einzigen Rücksprung zurückgegeben. Somit wurden die Rückspünge durch ein einziges Φ ersetzt, das nun wiederrum als Ansatzpunkt für die If-Konversion dienen kann. Ein Beispiel hierzu ist in Abbildung 4.4 zu sehen.

4.5 Operationen auf Ψs

Beim Aufbau der Ψs im Verlauf des oben beschriebenen Verfahrens entstehen nur einfache Ψs mit genau einer Bedingung. In einem weiteren Schritt können Kaskaden von diesen zusammengefasst und eventuell vereinfacht werden.

Der erste Fall ist die Verkettung von Ψs über den unbedingten Standardwert:

$$\Psi(\ldots, b_n, w_n, \Psi(b'_1, w'_1, w'_2)) \longrightarrow \Psi(\ldots, b_n, w_n, b'_1, w'_1, w'_2)$$

Alternativ kann ein am letzten bedingten Dateneingang anhängiges Ψ angefügt werden. Hierzu wird die letzte Bedingung negiert und der dazugehöhrige Dateneingang mit dem unbedingten Dateneingang vertauscht.

$$\Psi(\ldots, b_n, \Psi(b'_1, w'_1, w'_2), w_{\text{default}}) \longrightarrow \Psi(\ldots, \neg b_n, w_{\text{default}}, b'_1, w'_1, w'_2)$$

Es ist auch möglich, mehrere Ψs, die an beliebigen bedingten Dateneingängen kaskadiert sind, zu einem Ψ zu verschmelzen. Allerdings enstehen hierbei komplexere logische Bedingungen, die zur Optimierung wenig hilfreich sind und nur mit zusätzlichem Aufwand wieder zu einer einfachen Form abgebaut werden können.

Sind zwei aufeinander folgende Dateneingänge identisch, können einfach die dazugehörigen Bedingungen durch ein logisches Oder verknüpft und so einer der beiden Dateneingänge eliminiert werden:

$$\Psi(\ldots, b_i, w, b_{i+1}, w, \ldots) \longrightarrow \Psi(\ldots, b_i \vee b_{i+1}, w, \ldots)$$

Diese Transformation ist vorteilhaft, falls die Zielarchitektur effizient die Ergebnisse von logischen Operationen verknüpfen kann, z. B. in Form von Prädikatregistern. Ist dies nicht der Fall, kann die Transformation analog rückgängig gemacht werden.

Ein Speziallfall stellt eine Übereinstimmung des letzten bedingten und des unbedingten Dateneingangs dar. Hierdurch entfällt einfach die letzte Bedingung:

$$\Psi(\ldots, b_n, w, w) \longrightarrow \Psi(\ldots, w)$$

Die bisher genannten Transformationen beschreiben elementare Operationen. Durch zusätzliche Informationen sind weitere Vereinfachungen möglich. Ist bekannt, dass $\bigvee_{i \leq k} c_i = \mathit{true}$, so können alle Bedingunen ab k abgeschnitten werden, da sie niemals das Ergebnis des Ψs bestimmen werden:

$$\Psi(b_1, w_1, \ldots, b_k, w_k, \ldots, b_n, w_n, w_{\text{default}}) \longrightarrow \Psi(b_1, w_1, \ldots, b_{k-1}, w_{k-1}, w_k)$$

Auch ist es möglich, dass eine Bedingung durch andere überdeckt wird. Gilt $c_k \Rightarrow \bigvee_{i<k} c_i$, so kann die Bedingung c_k und ihr zugeordneter Wert ausgelassen werden:

$$\Psi(b_1, w_1, \ldots, b_{k-1}, w_{k-1}, b_k, w_k, \ldots) \longrightarrow \Psi(b_1, w_1, \ldots, b_{k-1}, w_{k-1}, \ldots)$$

Dies sind jedoch keine einfachen Transformationen, sondern verlangen weiterführende Analyse der in den Ψs verwendeten Bedingungen.

Kapitel 5

Implementierung

Die Implementierung, bestehend aus vorbereitenden Schritten, der eigentlichen If-Konversion und anschließenden Nacharbeiten, wird in sechs größeren Phasen abgearbeitet.

5.1 Vorbereitende Transformationen

Als erstes werden die normalisierenden Transformationen vorgenommen. Der zu bearbeitende Programmgraph wird umgeformt, sodass er genau einen Rücksprung enthält. Darauf folgend werden die kritischen Steuerflusskanten entfernt. Beide Transformationen werden bereits von libFirm zur Verfügung gestellt.

5.2 Aufbau der Analyseinformation

Im zweiten Schritt werden die Steuerabhängigkeiten der Grundblöcke berechnet. Diese Analyse war noch nicht in libFirm vorhanden, wurde daher im Zuge dieser Arbeit implementiert. Sie wurde generisch gehalten, da auch andere Übersetzungsphasen eventuell von dieser Information profitieren können. Die Implementierung folgt dem in [Muc97] beschriebenen Algorithmus. Zur Durchführung des Algorithmus wird der Nachdominanzbaum über die Grundblöcke benötigt. Diese Information wird ebenfalls bereits von libFirm bereitgestellt.

5.3 Transformationsphase

Die eigentliche If-Konversion erfolgt nun in einem Lauf über den Grundblockgraphen. Dazu wird für jeden Grundblock die Funktion *IfConvert* aufgerufen. In dieser werden der Reihe nach alle Stuerflussvorgänger dieses Blocks mit dessen Steuerabhängikeiten betrachtet. Hierbei ist *#Pred* die Anzahl der

Vorgänger eines Blocks (oder auch die Anzahl der Dateneingänge eines Befehls). *Pred* liefert den durch den Index bestimmten Vorgänger eines Blocks oder Befehls. Desweiteren ist *ControlDeps* die Menge der Steuerabhängigkeiten eines Blocks.

Zu jeder Steuerabhängigkeit des Vorgängerblocks wird die dazugehörige Bedingung mittels *FindCondition* bestimmt. Falls diese Suche nicht fehlschlägt, wird unter den noch nicht betrachteten Steuerflussvorgängern der zweite Vorgänger mit der entsprechenden Steuerabhängigkeit gesucht. Ist dies geglückt, werden durch *PreparePath* diese beiden Wege für die Transformation vorbereitet, indem etwaige mehrfache Steuerabhängigkeiten beseitigt werden.

Danach wird die eigentliche Transformation durchgeführt: Für jedes Φ wird im Block *dep* mit der Bedingung *cond* und den *i*-ten und *j*-ten Eingängen des Φ ein Ψ erzeugt. Jene beiden werden nun als Vorgänger des Φ gestrichen und dafür das Ψ an einem neuen Eingang hinzugefügt. Analog werden der *i*-te und *j*-te Vorgänger des Augangsgrundblocks *block* entfernt und der Block, der die Bedingung und das Ψ enthält, als Vorgänger angefügt.

Hiernach wird die Betrachtung des Grundblocks von Neuem begonnen, da durch den Veränderten Steuerfluss eventuell eine weiterer Angriffspunkt für die If-Konversion gegeben ist.

```
function IfConvert(block)
  restart:
  for (i ← 1 ... #Pred(block))
    pred_i ← Pred(block, i)
    for (dep ← ControlDeps(pred_i))
      cond ← FindCondition(pred_i, dep)
      if (cond = null) continue
      for (j ← i + 1 ... #Pred(block))
        pred_j ← Pred(block, j)
        if (dep ∉ ControlDeps(pred_j)) continue
        if (FindCondition(pred_j, dep) = null) continue
        PreparePath(block, i, dep)
        PreparePath(block, j, dep)
        for (phi ← Instructions(block, Φ))
          psi ← New(Ψ, dep, cond,
            Pred(phi, i), Pred(phi, j))
          RemovePred(phi, i)
          RemovePred(phi, j)
          AddPred(phi, psi)
        RemovePred(block, i)
        RemovePred(block, j)
        AddPred(block, dep)
        goto restart
```

Die Funktion *FindCondition* sucht ausgehend von einem Grundblock *start* die Bedingung des Blocks *dependency*. Dabei verfolgt sie rekursiv die durch *dependency* gegebene Steuerabhängigkeit. Diese Suche kann aus zwei Gründen fehlschlagen: Enthält auf dem Weg ein Block nicht seiteneffekfreie Befehle (*HasSideEffect*), so ist eine weitere Betrachtung unnötig, da hier keine If-Konversion durchgeführt werden kann. Es wird ebenfalls abgebrochen, falls ein Block gefunden wird, der eine Verzweigung enthält. Ansonsten wird die Bedingung des bedingten Sprungs aus dem Block *dependency* zurückgegeben.

```
function FindCondition(start, dependency)
  if (HasSideEffect(start) return null
  for (i = 1 ... #Pred(arity))
    pred ← Pred(start, i)
    if (pred = dependency) return ConditionIn(pred)
    if (ConditionIn(pred) ≠ null) return null
    if (pred ∈ ControlDeps(dependency))
      return FindCondition(pred, dependency)
```

PreparePath bereitet die beiden gefundenen Programmpfade zum Block *dependency* für die eigentliche Transformation vor, indem es mehrfache Steuerabhängigkeiten beseitigt. Hierzu verfolgt sie rekursiv ausgehend vom i-ten Vorgänger des Ausgangsgrundblocks *block* den Weg zum Grundblock *dependency*. Generell besitzen alle Blöcke außer dem direkten Vorgänger des Blocks *dependency* mehrere Steuerabhängigkeiten. Wäre dies nicht der Fall, dann hätten die jeweiligern Blöcke nur einen Vorgänger, was aufgrund der Maximalität von Grundblöcken nicht möglich ist. Also wird auf jeden dieser Blöcke die Funktion *SplitBlock* angewandt.

```
function PreparePath(block, i, dependency)
  pred ← Pred(block, i)
  for (j ← 1 ... #Pred(pred))
    pred_pred = Pred(pred, j)
    if (dependency ∈ ControlDeps(pred_pred))
      PreparePath(pred, j, dependency)
      SplitBlock(block, i, j)
      return
```

In der Funktion *SplitBlock* wird der gewünschte Programmpfad abgespalten, um mehrfache Steuerabhängigkeiten zu vermeiden. Ausgangspunkt ist hierbei der Grundblock *block* und dessen i-ter Vorgänger (dieser sei Block B_i). Dieser ist der Grundblock mit mehrfachen Steuerabhängigkeiten. Alle seine Befehle werden mittels *CopyTo* in seinen j-ten Vorgänger (Block B_j) kopiert. Im konkreten Fall ist dies immer der Block, dessen direkter Steuerflussvorgänger der Block mit der an der If-Konversion beteiligten Bedingung ist. Der Grundblock *block* wird in diesem Schritt nicht verändert, jedoch werden dessen Φs als Ausgangspunkt für das Kopieren der Befehle benötigt. Anschlie-

ßend wird *block* um B_j als Steuerflussvorgänger erweitert. Entsprechend wird B_j als Vorgänger von B_i entfernt. Somit ist die Abspaltung des gewünschten Programmpfades in Bezug auf diese Programmstelle abgeschlossen.

Das Vorgehen kann am besten an den Abbildungen 4.3(a) und 4.3(b) nachvollzogen werden: Die Steuerflusssenke entspricht hierbei *block*. Dessen *i*-ter Vorgänger ist der Grundblock mit den zwei Steuerabhängigkeiten. Wiederrum dessen rechter Vorgänger ist *j*.

```
function SplitBlock(block, i, j)
  pred_block ← Pred(block, i)
  for (phi ← Instructions(block, Φ))
    copy = CopyTo(Pred(phi, i), pred_block, j)
    AddPred(phi, Pred(phi, i))
    SetPred(copy)
  AddPred(block, Pred(block, i))
  SetPred(block, i, Pred(pred_block, j))

  for (phi ← Instructions(pred_block, Φ))
    RemovePred(phi, j)
  RemovePred(pred_block, j)
```

Zur Durchführung des Abspaltens müssen Berechnungen dupliziert werden. Dies geschieht mit Hilfe von *CopyTo*. Diese Funktion kopiert einen Berechnungsbaum ausgehend vom Befehl *node* rekursiv in den *i*-ten Vorgänger des Grundblocks *block*. Hierbei wird mit *CopyNodeTo* eine Kopie eines Befehls in dem gewünschten Block erzeugt und danach durch Rekursion die Abhängikeiten dieses Befehls ebenfalls kopiert. Die Rekursion bricht in zwei Fällen ab: Entweder befindet sich der zu kopierende Befehl nicht in *block*, dann ist keine Kopie notwendig. Ist der Befehl ein Φ in *block* so wird der *i*-te Vorgänger dieses Φs als Resultat zurückgeliefert. Dies ist der Wert des Φs in dem Fall, dass der Programmfluss von eben dem *i*-ten Vorgänger des Grundblocks *block* kommt.

```
function CopyTo(node, block, i)
  if (Block(node) ≠ block) return node
  if (Type(node) = Φ) return Pred(node, i)
  copy = CopyNodeTo(node, Pred(block, i))
  for (j ← 1 ... #Pred(copy))
    SetPred(copy, j, CopyTo(Pred(copy, j), block, i)
  return copy
```

5.4 Lokale Optimierungen

Danach werden übliche lokale Optimierungen, die von libFirm zur Verfügung gestellt werden, angestoßen. Dies ist nicht zwingend notwendig, jedoch er-

leichtert es die folgende Vereinfachung von Ψs. Durch die If-Konversion bieten sich besonders für Elimination gemeinsamer Teilausdrücke (Common Subexpression Elimination, CSE) und Konstantenfaltung Angriffspunkte aufgrund der durch die If-Konversion verschmolzenen Grundblöcke.

5.5 Vereinfachung von Ψs

Nun werden diverse Ψ-Ausdrücke vereinfacht. Hierzu werden alle Ψs betrachtet und gegenenfalls entsprechend der im vorigen Kapitel beschriebenen Umwandlungen vereinfacht. Jedoch wurden nur die elementaren Umformungen implementiert, da eine aufwändige Analyse von logischen Bedingungen und daraus resultierende Vereinfachungen nicht das Ziel dieser Arbeit sind. Insbesondere ist für eine effektive Umsetzung Information über statisch bestimmbare Wertebereiche notwendig.

5.6 Abbau komplexer Ψs

Zuletzt werden Ψs mit mehr als einer Bedingung in mehrere Ψs mit je einer Bedingung zerlegt. Dies stellt keine prinzipielle Notwendigkeit dar. Es wird jedoch durch die von Prozessorarchitekturen üblicherweise zur Verfügung gestellten Befehle, die nur eine Bedingung auf einmal betrachten, bedingt:

$$\Psi(\ldots, b_n, w_n, w_{\text{default}}) \rightarrow \Psi(\ldots, \Psi(b_n, w_n, w_{\text{default}}))$$

Hierbei wird die letzte Bedingung mit ihrem zugeordneten Wert und der Standardwert zu einem Ψ mit einer Bedingung zusammengefasst und somit das ursprüngliche Ψ um eine Bedingung verkürzt. Dieser Vorgang wird solange wiederholt, bis alle Ψs mit mehreren Bedingungen zu Ψs mit je einer Bedingung abgebaut sind. Dies ist die entsprechende Rücktransformation zu der im vorigen Kapitel aufgeführten Verschmelzung mehrerer Ψs.

Kapitel 6

Verwandte Arbeiten

Fließbandarchitekturen sind schon seit vielen Jahren in Gebrauch. Entsprechend wurde die Frage nach der Vermeidung bedingter Sprünge schon oft gestellt und auf verschiedene Weise beantwortet.

In [CSC+99] ist die Herangehensweise prädizierte Hyperblöcke. Das heißt eine durch Steuerfluss verbundene Abfolge mit einem Einsprungpunkt und möglicherweise mehreren Ausgängen wird zu einem Hyperblock zusammengefasst. Die Grundblöcke werden jeweils mit einem Prädikat – der Bedinungung unter der sie ausgeführt werden – versehen. Das Programm befindet sich in SSA-Form. Durch den Aufbau der sogenannten Predicated SSA-Form können mehrere Definitionen für einen Wert existieren, allerdings sind diese mit unterschiedlichen Prädikaten versehen. Dies können sogar dieselben Berechnungen sein, die jedoch über jeweils andere Ausführungspfade erreicht werden. Ziel ist es, auf den jeweiligen Pfaden eine möglichst geringe Befehlslatenz zu erreichen. Dieser Ansatz unterscheidet sich gänzlich vom hier verwendeten und setzt zur Umsetzung Architekturen vorraus, die Befehle bedingt ausführen können. Insbesondere ist es auf die Verwendung von EPIC-Archtitekturen gedacht, die nicht selbst Befehle umordnen, um Ausführungseinheiten besser auszulasten. Auch ist eine Zwischensprache notwendig, die SSA auch unter Einbezug von Prädikaten versteht.

Die Arbeit mit der größten Ähnlichkeit zu dieser ist [Sd01]. Der dortige Ausgangspunkt ist ein Programm, dessen Befehle, insbesondere Zuweisungen, mit Prädikaten versehen sind. Das Programm befindet sich an diesem Punkt nicht in SSA-Form. Nach jeder Zuweisung wird ein Ψ-Operation eingefügt, die alle bisherigen prädizierten Zuweisungen an diese Variable zusammenfasst. An diese Schreibweise ist das in dieser Arbeit vorgestellte Ψ angelehnt. Ein Unterschied besteht darin, dass die hier verwendeten Ψs explizit die Bedingungen der Dateneingänge mitführen und keine prädizierten Befehle verwendet werden. Zudem befindet sich das Programm bereits in SSA-Form, wodurch nicht erreichbare Definitionen direkt nicht als Eingang für Ψs auftreten, und nicht in einer weiteren Phase eliminiert werden müssen.

In [HC00] hingegen wird zur Laufzeit die Leistung gemessen und an kritischen, d.h. häufig ausgeführten, Stellen nach Bedarf eine If-Konversion durchgeführt oder auch rückgängig gemacht, falls sie sich als für die Laufzeit nachteilig herausgestellt hat. Der Schwerpunkt dieser Arbeit ist die Erstellung einer Metrik, wann, abhängig von Trefferrate der Sprungvorhersage und Latenz der beteiligten Befehle, eine Programmstelle transformiert wird. Auf den genauen Ablauf der Transformation wird nicht eingegangen, was auch nicht Ziel der Arbeit ist.

Die Arbeit [CCF03] unterscheidet vier Klassen von Befehlen, die im Zuge der If-Konversion auftreten: Φs, die abhängig von einem Prädikat ihr Ergebnis auswaählen; dies ähnelt den in dieser Arbeit verwendeteten Ψs. Speicher- und Ladebefehle, die abhängig von einem Prädikat einen Wert in den Speicher schreiben bzw. von diesem lesen. Das Verknüpfen von Prädikaten mit logischem Oder und zuletzt Vergleichsbefehle, die Prädikate erzeugen. Ein mächtigeres Konzept stellen die prädizierten Speicherzugriffe dar. Firm stellt Speicher als einen Wert dar, der wie gewöhnliche Werte bei Steuerflusszusammenflüssen durch Φs zusammengeführt wird. Entsprechend könnte das hier vorgestellte Konzept um Speicheroperationen erweitert werden, jedoch war das im Hinblick auf die primäre Zielarchitektur (x86), die keine prädizierten Speicheroperationen unterstützt, nicht sinnvoll.

Kapitel 7

Bewertung und Ausblick

In dieser Arbeit wurde ein Algorithmus vorgestellt und implementiert, der eine If-Konversion auf der SSA basierten Zwischensprache Firm durchführt.

7.1 If-Konversion und SSA

Zur Durchführung der If-Konversion erwiesen sich die durch die SSA-Form gegebenen Eigenschaften als nützlich. Im Verlauf der Transformation werden Werte verschoben. Ohne SSA können sich dadurch Zuweisungen zu gleichnamigen Variablen überlagern, jedoch werden alle diese berechneten Werte benötigt. Deswegen wäre eine explizite Umbenennung der Variablen notwendig. Dies entfällt, da dir SSA Eigenschaft garantiert, dass es für jeden Wert im Programm nur genau eine Zuweisung existiert. Dieses Problem kann auch in klassischer Darstellung umgangen werden, indem Zuweisungen mit Prädikaten, die aus den Bedingungen der bedingten Sprünge erzeugt werden, versehen werden. Allerdings zielt diese Vorgehensweise darauf ab, dass in der Kodererzeugungsphase eine Architektur, die Prädikate besitzt, als Ziel dient. In der vorgestellten Vorgehensweise ist dies nicht notwendig, jedoch kann der Ansatz auf prädizierte Architekturen angewendet werden und auch insofern erweitert werden, dass Befehle, die auf den Speicher zugreifen, dabei in Betracht gezogen werden, falls eine Prädizierung dieser möglich ist. Eine weitere Problemstellung ist, für welche Werte Ψs eingefügt werden müssen. Durch die SSA Eigenschaft ist diese Information direkt durch die vorhandenen Φs gegeben. Ohne sie müssen explizit die erreichenden Definitionen berechnet werden.

7.2 If-Konversion als Optimierung

Die If-Konversion verwandelt Steuerfluss in Datenfluss und beseitigt somit bedingte Sprünge. Dies allein kann, falls die Bedingung des Sprunges – aus

Sicht des Prozessors – von Zufallsdaten abhängt, die Ausführungsgeschwindigkeit des Programms verbessern. Es ist jedoch besonders interessant, diverse typische Spezialfälle zu betrachten. Im folgenden werden einige Beispiele genannt und kurz erläutert. Hiervon wurden in der Implementierung die Bestimmung des Absolutwertes und das Testen von Flags umgesetzt.

7.2.1 Absolutwert

Die Bestimmung des Absolutwertes (manchmal auch das Negative des selben) einer Zahl ist eine Operation, die in vielen Algorithmen Anwendung findet. Nach der If-Konversion findet sich diese Struktur als $\Psi(x < 0, -x, x)$ (oder einfache Varianten wie $x \leq 0$) im Programmgraph. Durch die SSA-Eigenschaft ist die Feststellung, dass es sich bei den x um den selben Wert handelt, direkt gegeben.

Einige Prozessoren bieten hierfür einen speziellen Befehl an. Auf den meisten anderen lässt sich dies mit wenigen Befehlen sprungfrei realisieren. Diese spezielle Optimierung existierte bereits als Implementierung in libFirm und konnte mit minimaler Modifikation für diese Arbeit übernommen werden.

7.2.2 Minimum und Maximum

Diese beiden Funktionen sind ebenfalls sehr häufig anzutreffen. Für das Minimum ensteht durch die If-Konversion die Struktur $\Psi(x < y, x, y)$, das Maximum ist analog dazu. Wiederum bieten einige Prozessoren hierfür spezielle Befehle, auf anderen kann dies auch sprungfrei umgesetzt werden.

7.2.3 Test eines Bitflags

Das Testen eines einzelnen beliebigen Bits in einem Wort und Verarbeitung dessen zu einem boolschen Wert in Form einer Null oder Eins bzw. eines anderen Bitflags ist eine häufige Operation, die nach Erkennen dieser Struktur – $\Psi(x\&2^a, 2^b, 0)$ oder Abwandlungen wie negierte Bedingung oder vertauschte Dateneingänge – leicht durch Bitschieben und Maskieren ersetzt werden kann, was zumeist nicht mehr Befehle in Anspruch nimmt als eine Variante mit bedingtem Sprung und im Speziellen eben diesen meidet.

7.3 If-Konversion als Normalisierung

If-Konversion kann nicht nur als Optimierung per se betrachtet werden, sondern auch als normalisierende Transformation, die nachgeschalteten Optimierungen weitere Möglichkeiten bietet.

Im Zuge der If-Konversion werden bedingte Sprünge entfernt und somit auch Grundblöcke verschmolzen. Effektiv bleiben alle Berechnungen in den an einem Transformationosschritt beteiligten Grundblöcken erhalten,

befinden sich nun aber nicht mehr in verschiedenen Blöcken. Dies kann der Eliminierung gemeinsamer Teilausdrücke weitere Angriffspunkte bieten. Auch im weiteren Verlauf der Kodeerzeugungsphase kann dies von Vorteil für die Durchführung der Befehlsanordnung sein, da nun mehr Befehle innerhalb eines Blocks zur Auswahl stehen und desweiteren weniger Blöcke angeordnet werden müssen.

7.4 Ausblick

Bei einfachen Tests zeigte sich, dass durch If-Konversion signifikante Laufzeitverbesserungen möglich sind. So enthält eine typische Implementierung zur Lösung des N Damenproblems (eine Verallgemeinerung des 8 Damenproblems, das vermutlich erstmals in [Bez48] formuliert wurde) an einer laufzeitkritischen Stelle einen bedingten Sprung, der durch hardwareseitige Sprungvorhersage nicht zuverlässig vorhergesagt wird. Hier führte die Beseitigung des bedingten Sprunges zu einer deutlichen Beschleunigung. Allerdings zeigte sich bei anderen Programmen kein bis negativer Einfluss auf die Laufzeit. Dies ist vermutlich darauf zurückzuführen, dass die If-Konversion dazu neigt, den Registerdruck zu erhöhen. Besonders bei der x86-Architektur führt dies aufgrund der wenigen zur Verfügung stehenden Registern dazu, dass Werte in den Speicher ausgelagert werden müssen. Somit ist es notwendig, aufwendigere Tests durchzuführen, um, abhängig von Gegebenheiten der Hardware, wie Anzahl der Register und Befehlslatenz, geeignete Heuristiken zu erstellen, um zu entscheiden, an welchen Stellen es von Nutzen ist, If-Konversion durchzuführen.

Desweiteren können weitere Transformationen von Ψs implementiert werden, um mehr des gebotenen Optimierungspotentials auszuschöpfen. Insbesondere werden in der aktuellen Implementierung nicht die in den Ψs verwendeten Bedingungen in Betracht gezogen. Dies kann zur Elimination von Berechnungen führen, was sich durch weniger Befehle und niedrigeren Registerdruck positiv auf die Laufzeit und Programmgröße auswirken kann.

Literaturverzeichnis

[Bez48] Bezzel, Max: In: *Berliner Schachzeitung* (1848)

[CCF03] Chuang, Weihaw ; Calder, Brad ; Ferrante, Jeanne: Phi-Predication for light-weight if-conversion. In: *CGO '03: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1913–X, S. 179–190

[CFR+91] Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Oktober, Nr. 4, 451–490. citeseer.ist.psu.edu/cytron91efficiently.html

[CSC+99] Carter, L. ; .Simon, E ; Calder, B. ; Carter, L. ; Ferrante, J.: Predicated Static Single Assignment, 1999

[HC00] Hazelwood, Kim M. ; Conte, Thomas M.: A Lightweight Algorithm for Dynamic If-Conversion during Dynamic Optimization. In: *2000 International Conference on Parallel Architectures and Compilation Techniques*, 2000, 71-80

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. – ISBN 1–55860–320–4

[Sd01] Stoutchinin, Arthur ; de Ferriere, Francois: Efficient static single assignment form for predication. In: *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, 2001. – ISBN 0–7695–1369–7, S. 172–181

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karlsruhe, Fakultät für Informatik. Universität Karlsruhe, Fakultät für Informatik, Dezember 1999 (1999-14). – For-

schungsbericht. – 0–40 S. – http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz