Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

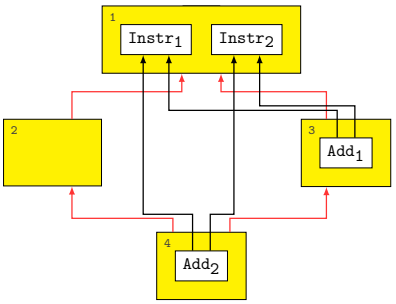
Lehrstuhl Prof. Dr.-Ing. Snelting

SSA-basierte Eliminierung partieller Redundanzen

Diplomarbeit von

Christian Helmer

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 5. April 2012 – 4. Oktober 2012

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

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Für die Erzeugung von effizientem Maschinencode wird der Eliminierung von Redundanzen große Aufmerksamkeit gewidmet. Redundanzen enstehen oft durch Adressberechnungen für Array-Zugriffe, sowie durch automatisch generierten Code in Hochsprachen.

Neben den einfach zu behebenden vollständigen Redundanzen existieren partielle Redundanzen, deren Behebung für gewöhnlich das Einfügen neuer Ausdrücke impliziert. Daraus ergibt sich die Frage der Platzierung dieser neuen Ausdrücke, mit Rücksicht auf den Registerdruck und die Codegröße.

Die Erkennung redundanter Berechnungen erfordert eine Analyse, die sich von Variablen löst, da ein Ausdruck bei Redefinition der verwendeten Operanden andere Werte berechnet. Dies leistet die SSA-Form, die eine geeignete Basis für viele Optimierungen bildet. Allerdings sorgt die durch ϕ-Instruktionen verursachte Verdeckung der Herkunft von Ausdrücken für zusätzlichen Aufwand beim Erkennen von Redundanzen.

Erste Algorithmen waren komplex, oder erkannten einen bedeutenden Teil der Redundanzen nicht, was daran lag, dass Ausdrücke nach wie vor lexikalisch verglichen wurden. Ein anderer Ansatz verfolgt GVN, das es ermöglicht, die Werte von Ausdrücken zu bestimmen. Dadurch kann ein Vergleich auf Wertbasis stattfinden, der die Erkennung von Redundanzen ermöglicht, die auf syntaktisch unterschiedlichen Ausdrücken basieren.

GVN-PRE kombiniert die Eliminierung partieller Redundanzen mit GVN und bietet auf diese Weise ein übersichtliches Verfahren auf Basis der SSA-Form. Im Rahmen dieser Arbeit wird libFirm um GVN-PRE erweitert. Gleichzeitig sollen die Fähigkeiten von GVN-PRE evaluiert und erweitert werden.

2 Grundlagen

Ein optimierender Übersetzer erzeugt nach der Analyse eine Zwischendarstellung des Programms. Diese dient als Basis für eine Vielzahl von Optimierungen und abstrahiert sowohl von der Zielmaschine, als auch von der Quellsprache. Oftmals werden die Ausdrücke dafür in einer 3-Adress-Form dargestellt, wodurch ein Standard geschaffen wird, auf dem viele Optimierungen arbeiten können.

Häufig benötigen Optimierungen die Stelle des Programms, an der eine Variable zuletzt definiert wurde. Dies ist Voraussetzung für die Ermittlung möglicher Positionen eines verwendenden Ausdrucks, oder den Vergleich zwischen Ausdrücken. Bedingt durch die häufige Verwendung dieser Analyse, entstand mit der SSA-Form eine Weiterentwicklung der 3-Adress-Form.

2.1 SSA

Die SSA[1]-Form [RWZ88a, CFR+91] hat sich als vorteilhafte Zwischendarstellung für zahlreiche Optimierungen erwiesen. Sie erlaubt statisch genau eine Zuweisung an jede Variable. Dies bedeutet, dass bei der Transformation in die SSA-Form bei jeder Variablenzuweisung ein neuer Bezeichner für den zugewiesenen Wert erstellt wird. Die Verwendungen werden dementsprechend angepasst, sodass immer der zuletzt zugewiesene Wert verwendet wird. Dadurch entsteht eine Verknüpfung zwischen Verwender und der Definition, wodurch einfache Wertkopien komplett entfallen. Diese Information ist für Datenflussanalysen von Vorteil, da die Herkunft eines Wertes klar ersichtlich ist.

Das Programm aus Listing 2.1 ist in Listing 2.2 in der SSA-Form dargestellt. $a = \bullet$ stellt im Folgenden die Zuweisung eines unbekannten Wertes dar. Während die beiden Zeilen $x = a + b$ in Listing 2.1 gleich aussehen, sind die Operanden

[1] engl.: static single assignment

in der SSA-Form an die letzte Definition von a angepasst und demnach unterschiedlich. Da c nur ein Alias für b darstellt, entfällt es komplett, wodurch die Gleichheit von x_2 und y_1 sofort festgestellt werden kann. Je nach Methode der SSA-Erstellung, wird $y_1 = a_2 + b_1$, als Alias von x_2, nicht erstellt, wodurch die Verwender von y_1 stattdessen direkt auf x_2 verweisen.

Die SSA-Form lässt sich ebenfalls als Graph darstellen, mit Knoten als Befehle und Kanten als Relation zwischen Definition und Verwendung. Der Graph in Abbildung 2.1 ist in Grundblöcke als Ausführungseinheiten gegliedert. Für einen Grundblock gilt, wird er betreten, so werden grundsätzlich alle enthaltenen Befehle ausgeführt. Zwischen den Grundblöcken geben Steuerflusskanten die Reihenfolge an.

Tatsächlich ist es statisch nicht möglich, an jedem Punkt des Programms den zuletzt zugewiesenen Wert zu verwenden, da durch Verzweigungen mehrere mögliche Werte existieren. In Listing 2.3 wird, abhängig von einer Bedingung, ein Pfad genommen, der a redefiniert. Da der Wert von a, als Operand von $a + b$, von dem verwendeten Pfad abhängt, ist keine eindeutige Definition auswählbar. Aus diesem Grund wird eine imaginäre ϕ-Instruktion eingeführt, die beide möglichen Werte vereint. Die ϕ-Instruktion nimmt, abhängig von dem ausgeführten Pfad, den Wert eines Vorgängerknotens an. Die Anzahl der Eingänge entspricht deshalb genau der Anzahl der Vorgängerblöcke. Unter der Annahme, dass, im Fall der positiven Auswertung der Bedingung, in `Grundblock 2` gesprungen wird, ist der Graph in Abbildung 2.1 die Darstellung des Programms in Listing 2.3.

```
a = •
b = •
c = b

x = a + b
a = •
x = a + b
y = a + c
```

Listing 2.1: Eingabe-Programm in 3-Adress-Form

$$
\begin{aligned}
a_1 &= \bullet \\
b_1 &= \bullet \\
\\
x_1 &= a_1 + b_1 \\
a_2 &= \bullet \\
x_2 &= a_2 + b_1 \\
y_1 &= a_2 + b_1
\end{aligned}
$$

Listing 2.2: Eingabe-Programm in SSA-Form

```
a = 2
b = 3

if (..):
    a = 5
else
    ..

x = a + b
```

Listing 2.3: Definition von a statisch nicht bekannt.

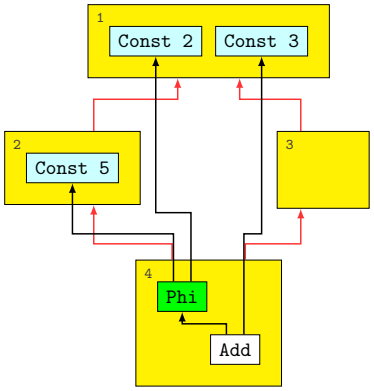


Abbildung 2.1: Notwendigkeit von ϕ-Knoten in der SSA-Form.

2.2 libFirm

Firm [LBBG05] [TLB99] bietet eine Zwischendarstellung in Form von SSA-Graphen. Bei libFirm [Lin02] handelt es sich um eine Implementierung von Firm, die bereits zahlreiche Optimierungen beinhaltet. In Firm-Graphen ist die Reihenfolge bei voneinander unabhängigen Befehlen innerhalb eines Grundblocks nicht explizit vorgegeben. Stattdessen existiert ein Modell des Speichers, das die Reihenfolge von einigen Befehlen – insbesonders Lade- und Speicherbefehle – vorgibt, indem diese durch Speicherkanten verbunden sind. Da eine Speicherkante den Zustand des Speichers repräsentiert, erfordern Verzweigungen möglicherweise ein Speicher-ϕ, um den Zustand anhand des Pfades auszuwählen.

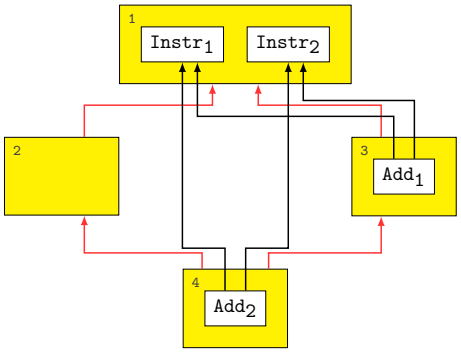


Abbildung 2.2: Beispiel einer partiellen Redundanz.

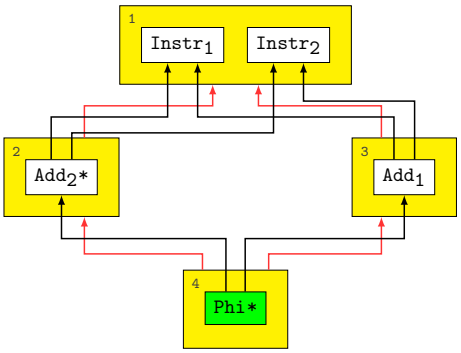


Abbildung 2.3: Auflösung der partiellen Redundanz.

2.3 Partielle Redundanzen

Die allgemeine Definition einer partiellen Redundanz sagt aus, dass ein Ausdruck partiell redundant ist, falls dieser auf einem Teil der zu ihm führenden Pfaden bereits existiert. Diese Definition trifft ebenfalls auf schleifeninvariante Ausdrücke zu. Statisch betrachtet, existiert durch die Rückwärtskante ein Pfad durch die Schleife, der einen in der Schleife befindlichen Ausdruck bereits auswertet. Dadurch kann PRE ebenfalls invariante Berechnungen aus Schleifen entfernen.

In Abbildung 2.2 ist $\texttt{Add}_2$ partiell redundant, da der Pfad über `Grundblock 3` erneut den gleichen Wert berechnet. Dadurch, dass der Pfad über `Grundblock 2` den Ausdruck einmalig auswertet, ist es nicht möglich, den redundanten Ausdruck einfach zu entfernen. Es wird davon ausgegangen, dass $\texttt{Add}_1$ nicht in `Grundblock 4` untergebracht werden kann, da das Resultat in `Grundblock 3`

weiterverwendet wird. Auch die Möglichkeit, beide Ausdrücke in einem gemeinsamen Dominator zu platzieren, kann zu unerwünschten Effekten führen. Die Berechnung wird dadurch möglicherweise auf Pfaden eingefügt, die sie nicht benötigen. Zudem kann dies die Lebenszeit einiger Werte stark verlängern, was sich potentiell negativ auf den Registerdruck auswirkt. Stattdessen wird konservativ vorgegangen, indem ein neues $\texttt{Add}_2\texttt{*}$ in `Grundblock 2` eingefügt wird. Dieser Vorgang wird im Folgenden als *Heben* bezeichnet. $\texttt{Add}_2$ wird schließlich durch einen ϕ-Knoten ersetzt. Das Ergebnis befindet sich in Abbildung 2.3. Sofern die beteiligte Steuerflusskante keine kritische Kante ist, wird dadurch kein unbeteiligter Pfad um die Auswertung ergänzt. Kritische Kanten können durch das Einfügen eines leeren Grundblocks entfernt werden. Für die Anwendung von PRE wird dies oftmals vorausgesetzt, oder bei Bedarf durchgeführt.

2.4 GVN

GVN[2] [Coc69, RL77] ist ein Verfahren, das statische Werte von Ausdrücken bestimmt und somit den Vergleich untereinander, sowie deren Vereinfachung, ermöglicht. Viele Optimierungen und PRE-Verfahren basieren auf dem lexikalischen Vergleich von Termen. Bereits die Kopie eines Wertes in eine andere Variable kann den Vergleich verhindern. In der SSA-Form spielen Wertkopien bereits keine Rolle mehr, da alle Ausdrücke direkt ihre Operanden referenzieren. Jedoch besteht auch hier die Möglichkeit, dass unterschiedliche Rechenwege den gleichen Wert darstellen. Die beteiligten Knoten können dabei auf verschiedene Pfade verteilt sein, sodass dies bei der SSA Erstellung zwar erkannt, allerdings nicht behoben werden kann. Listing 2.4 enthält eine partiell redundante Berechnung, die jedoch aus völlig unterschiedlichen Termen besteht. Zudem ist der Operand a statisch unbekannt. GVN ermöglicht den Vergleich zwischen den Ausdrücken y_1 und y_2. Der Ausdruck x_1 stellt den Wert $a * 2$ dar, y_1 den Wert $(a*2)+2$. Durch algebraisches Normalisieren kann die Äquivalenz von $(a*2)+2$ und $(a + 1) * 2$ erkannt werden.

[2] engl.: global value numbering

```
a = •

if (..):
    x = a * 2
    y = x + 2

x = a + 1
y = x * 2
```

Listing 2.4: Erkennung von partiellen Redundanzen mittels GVN.

Ausgehend von der SSA-Form, erhält jeder gefundene Wert ein Merkmal zur Identifizierung, das als *Wertnummer* bezeichnet wird. Äquivalente Ausdrücke erhalten den gleichen Wert und können somit verglichen werden. Existiert ein Ausdruck an einem Punkt des Programms, an dem ein anderer Ausdruck mit dem gleichen Wert verfügbar ist, so ist er vollständig redundant und kann entfernt werden. Allerdings führt das Einfügen von ϕ-Knoten in Listing 2.5 dazu, dass über den Wert von a_3 in Listing 2.6 keine Aussage getroffen werden kann. Dies ist jedoch essentiell, um PRE durchzuführen. Wird der Pfad genommen, in dem die Bedingung positiv auswertet, so ist $a_2 + b_1$ in der Form von $a_3 + b_1$ weiterhin redundant. Die durch SSA eingebrachten verschiedenen Versionen von a verdecken die Äquivalenz der Ausdrücke.

```
a = •
b = •

if (..):
    a = •
    x = a + b


y = a + b
```

Listing 2.5: PRE bei Quelle in der 3-Adress-Form

```
a1 = •
b1 = •

if (..):
    a2 = •
    x1 = a2 + b1

a3 = φ(a1, a2)
y1 = a3 + b1
```

Listing 2.6: PRE bei Quelle in der SSA-Form

3 Verwandte Arbeiten

PRE[1] [MR79] wurde von Morel und Renvoise vorgestellt. Das Verfahren ermöglicht die Eliminierung vollständiger, sowie partieller Redundanzen. Die Bedingung für partielle Redundanzen greift auch bei schleifeninvarianten Berechnungen, sodass diese ebenfalls optimiert und außerhalb der Schleifen platziert werden.

3.1 PRE

Die Entstehung der Redundanzen wird zu einem wesentlichen Teil den Hochsprachen zugeschrieben. Zugriffe auf Objekte und Arrays generieren sich wiederholende Adressberechnungen, die sich durch den Übersetzer eliminieren lassen. Zwei aufeinanderfolgende Zugriffe auf ein 2-dimensionales Array mit $a[4][0]$ und $a[4][1]$ erzeugt zur Dereferenzierung von $a[4]$ erneut $a + 4 * sizeof(*ptr)$.

Als Grundlage für PRE wird die 3-Adress-Form angenommen sowie die Unterteilung in Grundblöcke. Schleifen benötigen keine zusätzliche Aufmerksamkeit. Die Datenflussanalyse erstellt Listen für in den Grundblöcken *verfügbare* und *erwartete* Ausdrücke. Ein Ausdruck ist, ausgehend vom Programmbeginn, ab der Definition bis zu einem bestimmten Punkt verfügbar, solange er das gleiche Ergebnis berechnet. Das ist der Fall, solange nicht an die Operanden zugewiesen wird. Umgekehrt gilt ein Ausdruck vom Programmende aus in einem Grundblock als erwartet, solange ein Platzieren in diesem Grundblock ebenfalls das gleiche Ergebnis liefert. Eine vollständige Redundanz liegt vor, falls ein Ausdruck sowohl erwartet, als auch zu Beginn des Grundblocks verfügbar ist.

Für die Erkennung partieller Redundanzen wird die Definition der partiellen Verfügbarkeit verwendet, die besagt, dass ein Ausdruck auf einem Teil der eingehenden Pfade verfügbar ist. Wenn möglich, wird eine partielle Redundanz

[1] engl.: partial redundancy elimination

zunächst in eine vollständige überführt und anschließend entfernt. Dies erfordert das Einfügen des partiell redundanten Ausdrucks auf allen Pfaden, auf denen er nicht verfügbar ist. Das stellt eines der zu lösenden Probleme von PRE dar, da auf diese Weise unbeteiligte Pfade verlängert werden können, oder neue Redundanzen entstehen. Abbildung 3.1 zeigt einen Graph mit einer partiellen Redundanz in `Grundblock 5`. Wird diese durch das Platzieren des Ausdrucks in `Grundblock 2` aufgelöst, so entsteht eine weitere in `Grundblock 4`.

Der in [MR79] verfolgte Ansatz erfordert deshalb eine weitere Bedingung, welche dies überprüft. Allgemein stellt sich die Frage einer vorteilhaften Platzierung neuer Knoten.

Bei Lazy Code Motion [KRS92] handelt es sich um ein Verfahren, das, unter Beachtung des Registerdrucks, nachweislich die optimale Platzierung der Ausdrücke bezüglich der benötigten Instruktionen liefert. Die optimale Platzierung bedeutet die Minimierung der Instruktionen auf jedem Pfad, unter Beachtung der erwarteten Ausdrücke. Dies kann stets erreicht werden, indem ein Ausdruck so früh wie möglich ausgewertet wird, solange er weiterhin erwartet ist. Dadurch entfallen alle weiteren Neuauswertungen ab diesem Punkt. Diese `SAFE-EARLIEST`-Platzierung entspricht dem Verfahren von Morel und Renvoise, verwendet jedoch nicht die Verfügbarkeit des Ausdrucks als Kriterium.

Diese wirkt sich allerdings nachteilig auf den Registerdruck aus, falls eine spätere optimale Auswertung ebenfalls möglich ist. Durch weitere Prädikate, `ISOLATED` und `DELAYED`, wird die isolierte Platzierung von Ausdrücken identifiziert, sowie eine möglichst späte Platzierung erreicht. Die Neuplatzierung eines Ausdrucks ist isoliert, falls auf allen davon ausgehenden Pfaden in jedem Fall ein weiterer neuer Ausdruck eingefügt wird. Dadurch ist eine isoliert Platzierung unnötig. `DELAYED` sind Positionen, die durch eine `SAFE-EARLIEST` Position dominiert wer-

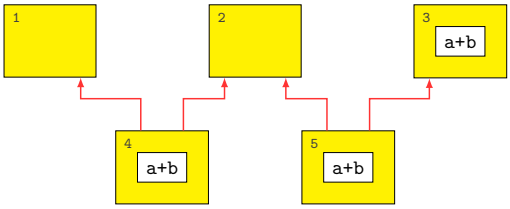


Abbildung 3.1: Kritische Kanten erschweren die Eliminierung partieller Redundanzen.

den und alle darauffolgenden, von `SAFE-EARLIEST` erfassten Ausdrücke dominieren. Dieses Prädikat erlaubt es, die `SAFE-EARLIEST` Lösung auf spätere Positionen zu verschieben und auf mehrere Pfade aufzuteilen. Dadurch verkürzt es die Lebenszeiten, der neu eingefügten Ausdrücke.

Durch die eingebrachten ϕ-Instruktionen sind die Ansätze, die auf dem lexikalischen Vergleich basieren, prinzipbedingt nicht durchführbar. ϕ-Instruktionen behindern den Ansatz, Ausdrücke anhand von Variablenbezeichnern vergleichen zu können.

Mit SSAPRE [CCK+97, KCL+99] wird dieses Problem angegangen, indem bei einem Quellprogramm in SSA-Form auf Variablen zurückgegriffen wird. Ein Ausdruck $a+b$ erhält durch eine zugewiesene Variable einen Namen h. Es werden neue Φ-Instruktionen $h = \Phi(h, .., h)$ an den Dominanzgrenzen des Ausdrucks und dort, wo ein ϕ einen der Operanden des Ausdrucks enthält, eingefügt. Mittels der SSA-Versionierung der neuen Variablen $h_4 = \Phi(h_1, h_2)$, werden gleiche Ausdrücke anhand von der gleichen Variablenversion erkannt. Zudem deuten mehrere Versionen innerhalb eines Pfades auf eine Zuweisung an die Operanden hin. Daraus lässt sich bestimmen, dass ein solcher Wert erwartet wird, wenn sich auf allen Abwärtspfaden die aktuelle Variablenversion befindet. In Kombination mit der Verfügbarkeit der Variablen, kann entschieden werden, welche Φ-Instruktionen in ein ϕ umgewandelt werden und auf welche Pfade ein Ausdruck dazu gehoben werden muss.

3.1.1 Spekulative PRE

Spekulative PRE erlaubt das Einfügen neuer Knoten auf selten genutzten Pfaden. Die Information über die Häufigkeit der Pfadnutzung ist aufgrund von Profiling bekannt. Dadurch erhält jede Steuerflusskante ein positives Gewicht, das bei stärkerer Frequentierung ebenfalls höher ist. Dies stellt einen Unterschied zu konservativer PRE dar, da Ausdrücke für gewöhnlich nur an Positionen eingefügt werden, an denen sie vollständig, und damit auf allen Pfaden bis zum Ende, erwartet werden. Das Problem besteht in der Platzierung der neu eingefügten Knoten, sodass ausschließlich Pfade mit dem geringsten Gewicht um zusätzliche Instruktionen verlängert werden, jedoch alle partiellen Redundanzen entfernt werden können.

[XC06] zeigt mit MC-PRE[2] eine Lösung auf, die dies leistet und gleichzeitig optimal bezüglich der Lebenszeit der eingefügten temporären Variablen ist. Damit erweitern sie das Verfahren aus [CX03] um die Optimalität bezüglich der Lebenszeiten.

MC-PRE führt zunächst die Datenflussanalysen für erwartete und verfügbare Werte aus. Dadurch können für einen Ausdruck *unwesentliche* Kanten entfernt werden, die sich nicht für das Einfügen des Ausdrucks eignen. Eine unwesentliche Kante liegt vor, falls der Ausdruck entlang dieser Kante verfügbar ist, oder falls er nicht erwartet wird. In beiden Fällen muss ebenfalls die Redefinition der Operanden beachtet werden. Daraus ergibt sich ein reduzierter Graph für diesen Ausdruck.

Aus diesem reduzierten Graph wird ein Flussgraph erstellt. Dies geschieht durch das Einfügen einer neuen Quelle und Senke, die jeweils eine Verbindung mit unendlichem Kantengewicht zu Grundblöcken ohne Vorgänger bzw. ohne Nachfolger eingehen.

Mittels dieses Graphen wird ein minimaler Schnitt über die Kantengewichte berechnet. Somit sind dies Kanten, die möglichst selten verwendet werden. Der Ausdruck wird auf allen geschnittenen Kanten eingefügt. An diesen Stellen wird der Ausdruck zwar erwartet, ist jedoch nicht verfügbar. Da allein aus der Nichtverfügbarkeit allerdings noch keine Redundanz folgt, ist diese vorläufige Lösung noch nicht optimal bezüglich der Lebenszeiten. Isolierte Ausdrücke werden bei dieser vorläufigen Lösung unnötigerweise auf Kanten eingefügt und eine neue temporäre Variable erzeugt. Ein Ausdruck ist isoliert, falls er am Ende des Grundblocks seines einzigen Verwenders bereits stirbt. Somit ist der gehobene Ausdruck unnötig und verlängert die Lebenszeit seines Wertes.

Scholz, Horspool und Knoop behandeln in [Sch04] ebenfalls spekulative PRE, allerdings unter Beachtung der Codegröße. Dies wird erreicht, indem möglicherweise Redundanzen belassen werden. Die Ausführungshäufigkeit jedes Grundblocks ist bekannt.

Der Graph wird für das Modell in Grundblöcke mit jeweils einem Ausdruck unterteilt. Somit hat jeder Grundblock bezüglich eines Ausdrucks genau eine der drei folgenden Eigenschaften: Er verhält sich neutral, invalidiert den Ausdruck oder berechnet ihn. Zudem existieren die *Szenarios*, die sich aus den Kombi-

[2] engl.: Min-Cut PRE

nationen ergeben, ob der jeweils Wert zu Beginn und Ende eines Grundblocks verfügbar sein soll. Anhand der Eigenschaft eines Knotens und dem gewünschten Szenario ergibt sich die dazu benötigte Veränderung des Grundblocks. Verhält sich ein Grundblock beispielsweise neutral und der Ausdruck ist nicht zu Beginn verfügbar, soll jedoch am Ende verfügbar sein, so muss er eingefügt werden. Jedes Einfügen wird mit gewissen Kosten verbunden. Die Minimierung dieser Kosten, in Kombination mit den Ausführungshäufigkeiten, erfolgt durch Reduzierung auf Stone's Problem, welches dies in polynomieller Zeit lösen kann.

Die genannten Verfahren verwenden als Basis die 3-Adress-Form. Da die SSA-Form jedoch ein besserer Ausgangspunkt für viele Optimierungen bietet, ist es das Ziel, PRE ebenfalls auf dieser Basis zu betreiben.

Diese Verfahren haben gemeinsam, dass sie auf Ausdrücken operieren und diese syntaktisch vergleichen. Jedoch können Ausdrücke, die nicht äquivalent erscheinen, das gleiche Ergebnis liefern. Während die SSA-Form bereits simple Wert-Kopien aufdeckt, ermöglicht GVN die Betrachtung der Werte von Ausdrücken.

3.2 GVN

Alpern, Wegman und Zadeck [AWZ88] betrachten die Gleichwertigkeit von Ausdrücken im Steuerfluss-Kontext und untersuchen Bedingungen und Schleifen auf Äquivalenz. Dazu wird, auf der SSA-Form aufbauend, ein gerichteter Wertflussgraph[3] erstellt, der die Abhängigkeit zwischen Werten widerspiegelt. Dieser Graph ermöglicht es, gleiche Ausdrücke anhand der gleichen Vorgänger und des gleichen Rechenoperators zu erkennen. Zudem sind ϕ-Knoten enthalten, da diese den Wert nach Bedingungen oder innerhalb von Schleifen repräsentieren. Falls ein ϕ-Knoten die Ergebnisse einer Bedingung kombiniert, wird er zu einem ϕ_{if}, der als Eingänge die beiden möglichen Werte, sowie die Bedingung besitzt. Auf diese Weise ist es möglich, Bedingungen zu vergleichen. Ähnlich wird im Fall von Schleifen vorgegangen, deren Iterationsvariablen gleich sind, falls sie dies vor Schleifeneintritt sind und innerhalb der Schleife die gleichen Berechnungen durchgeführt werden.

Die Eliminierung partieller Redundanzen wurde bereits in [RWZ88b] auf die SSA-Form angewandt. Da ϕ-Instruktionen die Abstammung von Ausdrücken

[3] engl.: value flow graph

verschleiern, wird ein Ausdruck mit von ϕ-Instruktionen abstammenden Operanden in Richtung Programmanfang verschoben. Beim Passieren einer als Operand verwendeten ϕ-Instruktion, wird der Ausdruck auf alle eingehenden Pfade vervielfacht. Dabei wird der Operand durch den Operand der ϕ-Instruktion des jeweiligen Pfades ersetzt. Dies ermöglicht den Vergleich des Ausdrucks mit den vorhandenen und kann auf diese Weise Redundanzen aufdecken.

3.3 GVN-PRE

GVN-PRE [VH04] ist ein Verfahren, das die Optimierung vollständiger und partieller Redundanzen auf SSA-Graphen beherrscht. Es erkennt partielle Redundanzen und überführt sie in vollständige Redundanzen, die schließlich entfernt werden. Da es sich bei schleifeninvarianten Berechnungen ebenfalls um partielle Redundanzen handelt, werden diese ebenso erfasst und aus der Schleife entfernt.

Mit diesem Verfahren werden Knoten ausschließlich innerhalb ihrer Pfade gehoben. Dadurch befinden sie sich stets auf Pfaden, auf denen der Wert des Knotens in jedem Fall berechnet wird. Zudem produziert das Verfahren keine neuen partiellen Redundanzen, indem mehrfach über den Graph iteriert wird. Die benötigte Anzahl dieser Iterationen ist durch die geeignete Bearbeitungsreihenfolge für gewöhnlich sehr gering. Da das Entfernen kritischer Kanten eine Voraussetzung ist, kann jede gefundene partielle Redundanz entfernt werden.

GVN-PRE ist in drei Phasen eingeteilt: Analyse, Einfüge-Phase und Eliminierungsphase. Während der Analyse werden Informationen über den SSA-Graph gesammelt und für jeden Grundblock gespeichert. Die Erkennung partieller Redundanzen findet in der Einfüge-Phase statt. Diese Phase erhält ihre Bezeichnung durch das Heben und damit Einfügen neuer Knoten. Die Eliminierungsphase entfernt vollständige Redundanzen, und erfasst dadurch ebenfalls die durch die Einfüge-Phase umgewandelten partiellen Redundanzen.

3.3.1 Verfahren

Bei GVN-PRE werden Grundblöcke betrachtet, an denen mehrere Pfade zusammentreffen. Die verfügbaren und erwarteten Werte werden mittels Datenflussanalysen für jeden Grundblock bestimmt. Erwartete Ausdrücke befinden sich in

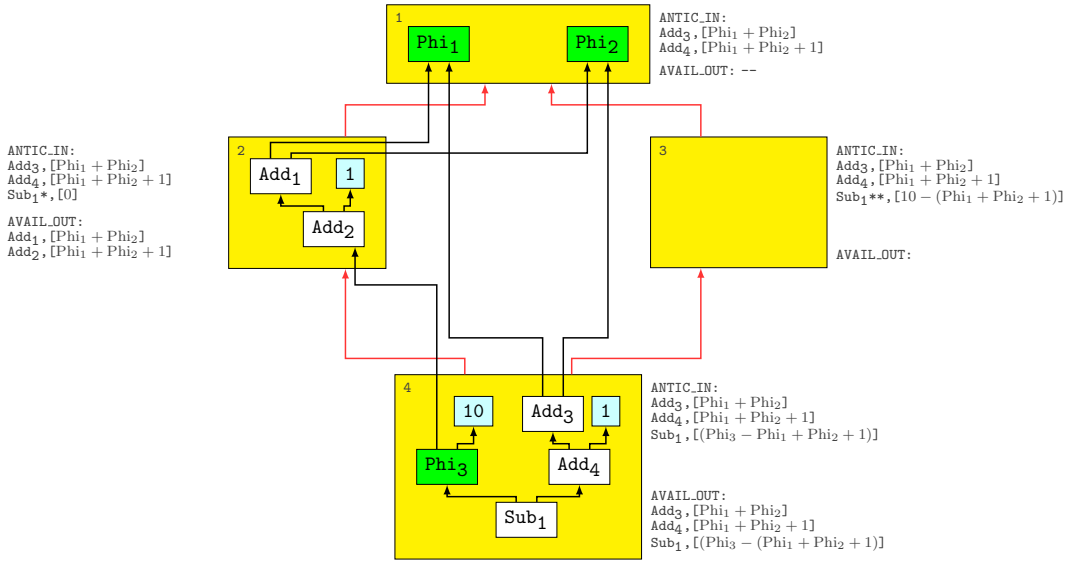


Abbildung 3.2: GVN-PRE Eingabeprogramm

ANTIC Wertemengen, die vollständig verfügbaren Ausdrücke in AVAIL Wertemengen. Die Datenflussanalyse erstellt die Mengen ANTIC_IN und AVAIL_IN für den Blockanfang, sowie ANTIC_OUT und AVAIL_OUT für das Ende eines Grundblocks.

Ausgehend von diesen Mengen, besteht eine partielle Redundanz, wenn sich ein Ausdruck aus ANTIC_IN, bei einem Teil der direkten Vorgängerblöcke in AVAIL_OUT befindet. Der redundante Ausdruck wird anschließend in die Vorgängerblöcke, in denen er bislang nicht verfügbar war, gehoben und somit eine vollständige Verfügbarkeit hergestellt. Der nun vollständig redundante Ausdruck kann durch ein ϕ-Knoten ersetzt werden, dessen Eingänge die verfügbaren und gehobenen Knoten in den Vorgängerblöcken sind. Die Wertemengen werden um die Information über die neu verfügbaren Knoten erweitert.

Abbildung 3.2 zeigt ein Graph, wie ihn GVN-PRE als Eingabe erhält. Das Ergebnis ist Abbildung 3.3. $\texttt{Add}_3$ und $\texttt{Add}_4$ sind partiell redundant und werden deshalb in `Grundblock 3` gehoben. Der Wert von $\texttt{Sub}_1$ wird auf dem Pfad über `Grundblock 2` mit 0 bestimmt, weshalb $\texttt{Sub}_1$ ebenfalls in `Grundblock 3` gehoben wird.

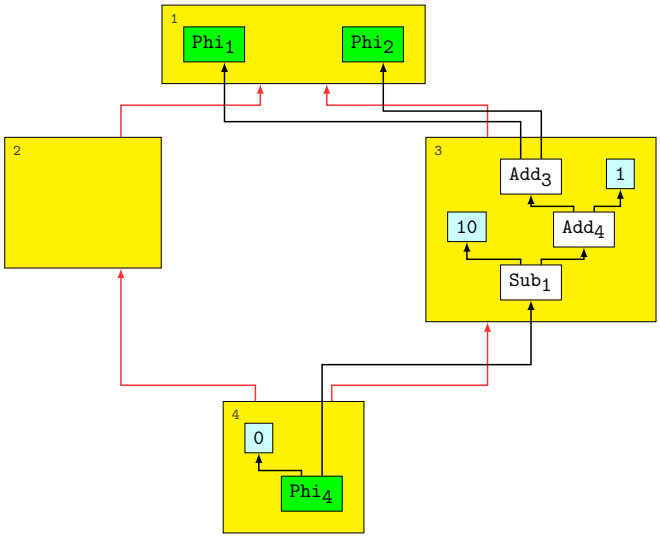


Abbildung 3.3: GVN-PRE Ausgabe

3.3.2 Algorithmus

Die Voraussetzung für den GVN-PRE Algorithmus ist ein Steuerflussgraph über Grundblöcke. Dieser Graph besitzt einen Startblock sowie einen Endblock. Für PRE werden kritische Pfade durch das Einfügen eines leeren Grundblocks entfernt. Globale Wertnummern werden mittels einer Funktion bestimmt, die als Eingabe einen Ausdruck erhält und die zugehörige Wertnummer liefert. GVN-PRE verwaltet sortierte Wertemengen, wobei jedes Element ein Paar aus Wert und Repräsentant darstellt. Ist ein Wert gegeben, so kann mittels einer Hash-Tabelle der Repräsentant nachgeschlagen werden.

Das Datenflussframework erstellt zwei Arten von Wertemengen für den Anfang und das Ende jedes Grundblocks: AVAIL_IN und AVAIL_OUT, sowie ANTIC_IN und ANTIC_OUT. AVAIL_IN enthält diejenigen Werte, die an diesem Punkt des Programms in jedem Fall pfadunabhängig, vorhanden sind. Diese Information ermöglicht klassische GVN, indem festgestellt wird, ob sich der Wert eines Ausdrucks innerhalb eines Grundblocks bereits in dessen AVAIL_IN Wertemenge befindet. Der Ausdruck wäre somit redundant.

Die ANTIC-Wertemenge gibt an, ob ein enthaltener Wert auf allen Pfaden von dem betrachteten Punkt aus bis zum Ende, ausgewertet wird. Dies kann auch so verstanden werden, dass dieser Wert im Folgenden benötigt oder *erwartet*

wird. Für ANTIC wird eine obere Grenze eingeführt. Ist innerhalb eines Grundblocks nicht feststellbar, welchen Wert ein Knoten darstellt, so wird dieser aus ANTIC_IN entfernt. Dies geschieht bei Funktionsaufrufen und Lade-Befehlen, sowie allen davon abhängigen Operatoren und führt dazu, dass sich ausschließlich diejenigen Knoten in ANTIC_IN befinden, die tatsächlich gehoben werden können.

In der Einfüge-Phase werden partielle Redundanzen durch das Einfügen von Knoten in vollständige umgewandelt. Dies erfolgt ausgehend vom Startblock topologisch und grundblockweise. An jedem Grundblock mit mehreren Eingängen werden die Werte in ANTIC betrachtet und auf Verfügbarkeit auf den eingehenden Pfaden geprüft. Wird eine partielle Redundanz erkannt, so wird der Repräsentant aus ANTIC in die Vorgängerblöcke kopiert, auf denen der Wert noch nicht ausgewertet wurde. Damit wird der neue Knoten in AVAIL_OUT des Grundblocks eingefügt. Da der Wert aus ANTIC nun in allen Vorgängerblöcken verfügbar ist, kann in dem aktuell betrachteten Grundblock ein ϕ-Knoten erstellt werden, der den Wert dort bereitstellt. Dieser ϕ-Knoten repräsentiert nun den Wert und wird in AVAIL_OUT eingefügt und ggf. den vorhandenen Repräsentanten ersetzt. Dadurch wird der ϕ-Knoten einer eventuell vorhandenen Neuauswertung eines Ausdrucks vorgezogen. Die Änderungen an der AVAIL Wertemenge müssen an die darauffolgenden Grundblöcke weiterpropagiert werden. Vollständige Redundanzen werden erkannt, indem der Wert aus ANTIC bereits im direkten Dominator verfügbar ist. In diesem Fall erfolgt zunächst keine Bearbeitung.

Die Eliminierungsphase entfernt vollständige Redundanzen mittels der Information über die verfügbaren Werte aus der AVAIL Wertemenge. Dazu werden alle Knoten ausgehend vom Endblock besucht und der Wert in AVAIL_OUT des Grundblocks nachgeschlagen. Existiert bereits ein anderer verfügbarer Ausdruck, so liegt eine vollständige Redundanz vor. Der Knoten ist somit redundant und wird durch den verfügbaren Knoten ersetzt. Im Fall von partiellen Redundanzen handelt es sich dabei um den in der Einfüge-Phase erstellten ϕ-Knoten, andernfalls um einen dominierenden Knoten.

3.3.3 Aufbau der Wertemengen

Führende Knoten

Die Hilfsmenge EXP_GEN wird für jeden Grundblock erstellt und enthält alle Ausdrücke dieses Grundblocks, die für den Algorithmus von Bedeutung sind, sowie deren Werte. Für den Fall, dass mehrere Knoten mit dem gleichen Wert existieren, wird jeweils ein Repräsentant gewählt. Dieser wird als führender Knoten bezeichnet. Als führender Knoten wird immer derjenige verwendet, der als erstes einen neuen Wert in die jeweilige Wertemenge einfügt. Da die Bearbeitung eines Knotens von seinen Vorgängern abhängt, enthält EXP_GEN die Knoten in topologischer Reihenfolge. Diese beiden Anforderungen begünstigen die Implementierung. Die Topologie-Eigenschaft bleibt bei der Konstruktion der Wertemengen erhalten. Liegen mehrere Repräsentanten vor, so wird durch die Verwendung von führenden Knoten das unnötige Speichern, sowie die wiederholte Bearbeitung dieser vermieden.

Erwartete Werte

Die ANTIC Wertemenge gibt an jedem Grundblock an, welche Werte auf dem Pfad bis zum Endblock auf allen Pfaden ausgewertet werden. Dazu erfolgt die Datenflussanalyse blockweise ausgehend vom Programmende und verwendet EXP_GEN, um ANTIC_OUT zu bestimmen. Die Datenflussgleichung lautet:

$$\text{ANTIC\_IN} = \mathit{clean}(EXP\_GEN \cup \text{ANTIC\_OUT})$$

$\mathit{clean}()$ entfernt alle Werte, die zu Beginn des Grundblocks noch nicht bekannt sind und somit nicht über diesen Grundblock hinaus gehoben werden können. Liegt beispielsweise der Ursprung eines Wertes in einem an den Grundblock gebundenen Lade-Knoten, so können die Verwender nicht über diesen Lade-Knoten hinaus geschoben werden. Obwohl die Verwender in diesen Grundblock gehoben werden könnten, werden solche Werte aus ANTIC_IN entfernt, da sie für die Analyse nicht von Bedeutung sind.

ANTIC_OUT erhält Werte von den nachfolgenden Grundblöcken. Hierbei müssen die Fälle unterschieden werden, in denen der Grundblock genau einen, oder aber

mehrere Nachfolger besitzt. Ist genau einen Nachfolger vorhanden, so gilt nach dem Entfernen der kritischen Kanten, dass dieser potentiell mehrere Eingänge und somit ϕ-Knoten besitzt.

Bei dem Versuch, die in für `Grundblock 2` in Abbildung 3.2 zu berechnen, stößt GVN an seine Grenzen, da im Fall von ϕ-Knoten nicht vorhergesagt werden kann, welcher Wert angenommen wird. Erst wenn der Pfad bekannt ist, kann der Wert des ϕ-Knotens bestimmt werden.

Der Wert, den $\texttt{Sub}_1$ in Folge des Pfades durch `Grundblock 2` annimmt, wird bestimmt, indem der Wert von $\texttt{Phi}_3$ in den Kontext von `Grundblock 2` *übersetzt*[4] wird. Dazu wird $\texttt{Sub}_1$ als direkter Nachfolger von $\texttt{Add}_2$ betrachtet.

Für den Fall, dass der Grundblock mehrere Nachfolger aufweist, so besitzt jeder dieser Nachfolger durch das Entfernen der kritischen Kanten jeweils genau einen Eingang. Dies trifft auf `Grundblock 1` zu. Da ausschließlich vollständig erwartete Werte benötigt werden, wird ANTIC_OUT aus dem wert-weisen Schnitt von ANTIC_IN der nachfolgenden Grundblöcke bestimmt.

Somit ergeben sich die Datenflussgleichung:

$$(\#succs(b) = 1) \quad \text{ANTIC\_OUT}(b) = phi\_translate(\text{ANTIC\_IN}(succ(b)), b)$$
$$(\#succs(b) > 1) \quad \text{ANTIC\_OUT}(b) = \bigcap_{succ=succs(b)} \text{ANTIC\_IN}(succ)$$

Verfügbare Werte

Die Bestimmung der verfügbaren Werte erfolgt vom Startblock aus durch folgende Datenflussgleichungen:

$$\text{AVAIL\_IN}(b) = \text{AVAIL\_OUT}(idom(b))$$
$$\text{AVAIL\_OUT}(b) = \text{AVAIL\_IN}(b) \cup EXP\_GEN(b)$$

AVAIL_IN wird aus AVAIL_OUT des direkten Dominators bestimmt. Es erfolgt kein wertweiser Schnitt bei Grundblöcke mit mehreren Vorgängern. Der führende Knoten aus AVAIL_IN wird bei jeder Vereinigung mit EXP_GEN nicht ersetzt, da in dieser Situation eine vollständige Redundanz vorliegt und der Knoten aus EXP_GEN keine neuen Informationen enthält. Somit kann aus einem

[4] engl.: phi translation

Wert mittels der AVAIL Wertemenge direkt der führende Knoten erhalten werden. Die AVAIL Wertemenge benötigt keine Schnittbildung wie ANTIC, da in der Einfüge-Phase explizit auf die Verfügbarkeit eines Wertes innerhalb der Vorgängerblöcke geprüft wird, und AVAIL mit dieser Information aktualisiert wird.

3.3.4 Einfüge-Phase

In der Einfüge-Phase werden Redundanzen erkannt und Knoten eingefügt, um vollständige Redundanzen herzustellen. Jeder Repräsentant aus ANTIC_IN wird dazu mittels ϕ-Übersetzung in alle Vorgängerblöcke übersetzt und der Wert bestimmt. Diese potentiell unterschiedlichen Werte werden nun in AVAIL_OUT des jeweiligen Grundblocks gesucht. Existiert ein Eintrag, so ist der gesuchte Wert bereits verfügbar und wird auf diesem Pfad mehrfach ausgewertet. Gleichzeitig liefert AVAIL_OUT direkt den verfügbaren Knoten. Enthält ein echter Teil dieser Pfade eine solche Redundanz, so ist der Repräsentant aus ANTIC_IN partiell redundant. Besteht die Redundanz auf allen Pfaden, so liegt eine vollständige Redundanz vor. Dies ist auch der Fall, wenn der Wert aus ANTIC bereits im direkten Dominator verfügbar ist, erfordert jedoch in dieser Phase keine Bearbeitung, da keine Knoten eingefügt werden müssen. Liegt eine partielle oder vollständige Redundanz vor, wird der redundante Knoten auf jeden der Pfade gehoben, auf denen der Wert bisher nicht verfügbar ist. Dies erfolgt durch das erneute Übersetzen des Knotens und das Eintragen in AVAIL_OUT der jeweiligen Vorgängerblöcke. Anschließend ist der erwartete Wert auf allen Pfaden verfügbar, sodass ein ϕ-Knoten diese Repräsentanten zusammenführen kann. Der ϕ-Knoten ersetzt ab diesem Grundblock den führenden Knoten in AVAIL_OUT, sodass die Eliminierung der Redundanz den ϕ-Knoten, anstelle des ursprünglichen Knotens verwendet.

Da sich während dieser Phase AVAIL ändert und diese Information propagiert werden muss, wird eine zusätzliche Wertemenge NEW_SET eingeführt, die ausschließlich alle neuen Einträge beinhaltet. Dies erspart das wiederholte Durchsuchen der gesamten AVAIL_OUT Wertemenge. Vor der Bearbeitung eines Grundblocks wird EXP_GEN geleert und aus dem direkten Dominator übernommen, sowie AVAIL_IN mit EXP_GEN ersetzt und erweitert.

Durch die Erweiterung von AVAIL innerhalb von Schleifen, ergibt sich nach

einmaliger Ausführung dieser Phase eine neue Situation, da diese Information ebenfalls über Rückwärtskanten fließt. Es erfordert mehrere Iterationen, um an einen Fixpunkt zu gelangen. Die Anzahl ist allerdings gering, da von bis zu drei ausgegangen wird. Wenn ein Knoten innerhalb des betrachteten Grundblocks bereits optimiert wurde, kann er als bearbeitet markiert werden, und in den folgenden Iterationen ignoriert werden.

3.3.5 Eliminierungsphase

Die Eliminierungsphase entspricht der Eliminierung bei GVN für vollständig redundante Ausdrücke. Es werden alle Knoten auf die Existenz eines verfügbaren Knotens mit dem gleichen Wert untersucht. Existiert ein Knoten, der selbst nicht dem führenden entspricht, so ist er redundant und wird durch den führenden Knoten ersetzt.

4 GVN-PRE Implementierung

4.1 GVN

GVN wurde aufbauend auf einer bestehenden CSE implementiert, die es bereits ermöglicht, die Wert-Äquivalenz bei Knoten mit den exakt gleichen Vorgängern zu erkennen. Für die initiale Wertbestimmung wird der Graph ausgehend vom Startblock topologisch besucht. Als Wert-Repräsentant dient der erste Knoten, der auf diese Weise gefunden wird. Existieren allerdings zwei äquivalente Knoten, die von unterschiedlichen Knoten, jedoch von den gleichen Werten abhängen, werden diese nicht als wertgleich erkannt, da der Vorgängerknoten und nicht dessen Wert überprüft wird. Aus diesem Grund werden die Vorgänger des betrachteten Knotens darauf untersucht, ob sie der Repräsentant ihres Wertes sind. Ist dies der Fall, kann der vorläufige Wert durch CSE bestimmt werden. Andernfalls muss, dadurch bedingt, dass ein Wert ebenfalls ein Knoten ist, ein neuer Knoten mit den Wertrepräsentanten der Vorgänger erzeugt werden, sodass die vorläufige Wertbestimmung durch CSE erfolgen kann. Da sich durch ϕ-Knoten Abhängigkeitszyklen bilden können und die eigentliche Wertbestimmung bei der Erstellung der ANTIC Wertemenge erfolgt, erhält jeder ϕ-Knoten einen eigenen Wert und repräsentiert somit sich selbst. Erst nach der ϕ-Übersetzung wird durch GVN ein Wertflussgraph aufgebaut, mit dem Wertäquivalenz durch algebraische Umformung festgestellt werden kann.

4.2 Bestimmung von EXP_GEN

EXP_GEN wird durch das topologische Besuchen aller Knoten, ausgehend vom Startblock, bestimmt. Für den Algorithmus uninteressante Knoten, wie volatile Ladeinstruktionen und Funktionsaufrufe werden nicht in EXP_GEN aufgenommen. Da die Nachfolger dieser Knoten innerhalb des gleichen Grundblocks

ebenfalls nicht gehoben werden können, werden diese durch *clean*() herausgefiltert. Eingefügt werden dagegen ϕ-Knoten, sowie Knoten, deren Vorgänger sich bereits in EXP_GEN oder außerhalb des Grundblocks befinden.

4.3 Bestimmung der verfügbaren Werte

Da für den Algorithmus ausschließlich AVAIL_OUT verwendet wird, kann die Berechnung von AVAIL_IN ausgelassen werden. Zunächst wird AVAIL_OUT im gleichen Schritt wie die EXP_GEN Erstellung vorbelegt. Allerdings dürfen keine Knoten ausgelassen werden, da selbst ignorierte Knoten Vorgänger von Knoten aus EXP_GEN sein können und die Verfügbarkeit später erfragt werden kann.

Die Bestimmung von AVAIL_OUT, erfolgt durch das Besuchen des Dominatorbaums. Die verfügbaren Werte werden durch AVAIL_OUT des direkten Dominators bestimmt, erweitert um alle Knoten des Grundblöcke. Die Datenflussgleichung lautet somit:

$$\text{AVAIL\_OUT}(b) = \text{AVAIL\_OUT}(idom(b)) \cup EXP\_GEN(b)$$

4.4 Bestimmung der ANTIC_IN Wertemenge

Der Algorithmus verwendet ausschließlich ANTIC_IN, da sich dies vorteilhaft für das Vorgehen in der Einfüge-Phase auswirkt. ANTIC_IN wird ausgehend vom Endblock berechnet. Die ϕ-Übersetzung dient der Wertfindung, wenn der Wert eines Knotens pfadabhängig ist und folglich von einem ϕ-Knoten abstammt. Dies tritt in Grundblöcken mit mehreren Eingängen auf. Um den Wert für ANTIC_IN in den jeweiligen Vorgängern zu berechnen wird jeder Wert aus ANTIC_IN des Nachfolgers übersetzt.

$$\text{ANTIC\_IN}(b) = clean(EXP\_GEN \cup phi\_translate(\text{ANTIC\_IN}(succ(b)), b)$$

In Abbildung 4.1 ist die Situation während der Berechnung von ANTIC_IN für `Grundblock 2` dargestellt. $\texttt{Add}_3\texttt{*}$ wurde von einem darauffolgenden Block in `Grundblock 4` übersetzt. Werte benötigen einen Repräsentanten, der übersetzt werden kann. Da jeder Wert jeweils nur einen Repräsentanten besitzt,

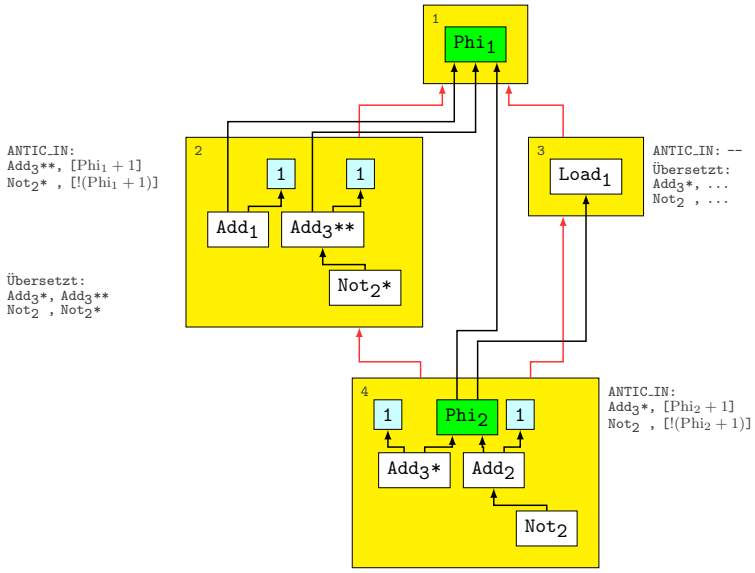


Abbildung 4.1: Bestimmung von ANTIC_IN in `Grundblock 2`: Übersetzen von Knoten.

ist $\texttt{Add}_3\texttt{*}$ in ANTIC_IN der führende Knoten, da er zuerst in ANTIC_IN aufgenommen wurde. Die Wertbestimmung in `Grundblock 2` erfolgt durch Iteration über ANTIC_IN von `Grundblock 4`. Um $\texttt{Add}_3\texttt{*}$ zu übersetzen, wird ein Knoten $\texttt{Add}_3\texttt{**}$ mit den jeweils übersetzten Eingangsknoten erzeugt. Die Konstante befindet sich bei Firm im Startblock und ist somit stets verfügbar, der andere Eingang ist das Ergebnis der ϕ-Übersetzung von $\texttt{Phi}_2$. Der Wert von $\texttt{Add}_3\texttt{**}$ ist somit $\texttt{Phi}_1\texttt{+1}$. Dieses Paar ist sauber und wird in ANTIC_IN von `Grundblock 2` aufgenommen, da sich alle Definitionen der Eingänge außerhalb von `Grundblock 2` befinden.

Die ϕ-Übersetzung, des beispielhaft für diese Situation ausgewählten Knotens $\texttt{Not}_2$, erfordert zur Übersetzung das Auffinden von $\texttt{Add}_3\texttt{**}$ anhand von $\texttt{Add}_2$. Dieser Knoten befindet sich in ANTIC_IN von `Grundblock 2`, allerdings unter einem anderen Wert als in `Grundblock 4`, sodass ein Lookup nicht möglich ist. Es erfordert eine Verknüpfung zwischen $\texttt{Add}_2$ und $\texttt{Add}_3\texttt{**}$, um den übersetzten Knoten aufzufinden. Diese Verknüpfung wurde mit einer zusätzlichen Hashliste implementiert. Sowohl `Grundblock 2`, als auch `Grundblock 3` benötigen diese, um das Ergebnis der Übersetzung vorzuhalten. Übersetzt wird jeweils der führende Knoten, weshalb der Eintrag aus dem Knotenpaar $\texttt{Add}_3\texttt{*}$ und $\texttt{Add}_3\texttt{**}$ eingefügt wird. Dadurch kann, mit Hilfe des Wertes von $\texttt{Add}_2$, der führende Kno-

ten in ANTIC_IN von `Grundblock 4` erfolgen. Dieser wurde in $\texttt{Add}_3\texttt{**}$ übersetzt und kann in der Übersetzungs-Liste gefunden werden. Somit ist der übersetzte Vorgänger von $\texttt{Not}_2$ bekannt, wodurch der Knoten in $\texttt{Not}_2\texttt{*}$ übersetzt werden kann.

$\texttt{Not}_2\texttt{**}$ ist sauber in `Grundblock 2`, da sich sein Vorgängerknoten in ANTIC_IN befindet. Aus dem umgekehrten Grund ist er in `Grundblock 3` nicht sauber. Sowohl in `Grundblock 2`, als auch in `Grundblock 3`, wird der Knoten in die Übersetzungs-Liste aufgenommen. Dies dient in der Einfüge-Phase oder weiteren ANTIC-Iterationen der Wiederherstellung von ANTIC_OUT.

Die Existenz von mehreren Nachfolgeblöcken stellt den anderen Fall dar. Ein Paar aus Wert und Repräsentant wird nicht übernommen, falls sich der Wert nicht in allen ANTIC_IN Wertemengen der Nachfolger befindet.

4.5 Einfüge-Phase

In dieser Phase wird ANTIC_IN zusammen mit AVAIL_OUT verwendet, um partielle Redundanzen festzustellen und in vollständige Redundanzen zu überführen. Die Grundblöcke werden in der Reihenfolge des Dominatorbaumes besucht. Dies ist von Vorteil, da AVAIL_OUT um neue Werte erweitert werden kann, die dadurch problemlos weiterpropagiert werden können. Neu verfügbare Werte befinden sich NEW_SET des direkten Dominators und werden in AVAIL_OUT sowie in NEW_SET eingefügt. Dies stellt den ersten Schritt dieser Phase dar. Grundblöcke mit weniger als zwei Eingängen erfordern keine weiteren Schritte.

Bei Grundblöcken mit mehreren Eingängen werden anschließend die erwarteten Werte betrachtet. Die folgenden Schritte werden für jeden Eintrag vom ANTIC_IN des Grundblocks ausgeführt. Da Werte nach ihrer Bearbeitung in diesem Grundblock verfügbar sind, können sie in darauffolgenden Iterationen ignoriert werden. Dazu werden bearbeitete Werte in die ANTIC_DONE Hilfsmenge des Grundblocks eingefügt. Dadurch kann festgestellt werden, ob der Wert bereits bearbeitet wurde. Dies hat den Vorteil, dass ANTIC_IN dadurch vollständig erhalten bleibt, statt direkt Werte aus der Menge zu entfernen, da beide Wertemengen für die Code Placement Erweiterung in Abschnitt 5.3 benötigt werden.

Falls der erwartete Wert bereits im direkten Dominator verfügbar ist, handelt es sich um eine vollständige Redundanz. Da dies kein Einfügen von Knoten erfordert, kann zum nächsten erwarteten Wert übergegangen werden.

Das Auffinden partieller Redundanzen benötigt ANTIC_OUT der Vorgängerblöcke, da dies die einzige Wertemenge ist, in der die Werte bereits übersetzt sind. Allerdings müsste dazu die ANTIC_OUT Wertemenge jedes Vorgängerblocks gleichzeitig betrachtet werden und die gemeinsame Abstammung der Repräsentanten bekannt sein. ANTIC_OUT ist jedoch aus ANTIC_IN und der Übersetzungs-Liste herstellbar, wobei die Abstammung aus ANTIC_IN hervorgeht. Es wird folglich der Repräsentant aus ANTIC_IN mittels Übersetzungs-Liste in jeden der Vorgängerblöcke übersetzt und der Wert bestimmt. Falls dieser Wert im gleichen Grundblock verfügbar ist, wurde eine partielle Redundanz gefunden. Selbst wenn die Redundanz in allen Vorgängerblöcken besteht, ändert sich das Verfahren nicht.

Für die Überführung in eine vollständige Redundanz ist es erforderlich, dass alle Vorgänger des Knotens, der gehoben wird, im Zielblock verfügbar sind. Dadurch wird verhindert, dass Nachfolger von nicht redundanten Knoten gehoben werden, denn das würde das gierige Heben von Knoten erzwingen.

Ein Knoten wird gehoben, indem er im Zielblock neu erstellt wird. Dies erfordert Kenntnis über seine neuen Vorgänger, die möglicherweise ebenfalls redundant waren und in denselben Grundblock gehoben wurden. Die Vorgänger sind beliebige Knoten, über die zunächst keine Informationen vorliegen. Die benötigte Information ist, welcher Repräsentant für die Verfügbarkeit im Zielblock sorgt.

Der Repräsentant wird mittels AVAIL_OUT ermittelt. Dies erfordert den übersetzten Wert des Vorgängers. Der führende Knoten kann in ANTIC_IN nachgeschlagen werden. Dieser wurde übersetzt und kann somit in der Übersetzungs-Liste gefunden werden. Mit dem Wert des übersetzten Knotens wird der führende Knoten in AVAIL_OUT nachgeschlagen. Dies ist der Grund, weshalb sich die durch ϕ-Übersetzung erstellten Knoten nicht eignen, da diese als Vorgänger führende Knoten aus ANTIC besitzen, und nicht sichergestellt ist, dass diese dominieren bzw. verfügbar sind.

Nicht jeder Vorgänger wurde übersetzt, sodass in jedem dieser Schritte auf den Vorgänger zurückgefallen wird, falls ein Lookup fehlschlägt. Dominiert der Vorgänger beispielsweise den Zielblock, so kann jeder Schritt bis auf das Nach-

schlagen des Wertes in AVAIL_OUT fehlschlagen und ausgelassen werden.

Der neu erstellte Knoten wird in seinem Grundblock in AVAIL_OUT eingefügt. Nachdem die Verfügbarkeit schließlich auf allen Pfaden hergestellt wurde, wird ein ϕ-Knoten erzeugt, dessen Eingänge die Repräsentanten der verfügbaren Werte sind. Der ϕ-Knoten wird neuer Repräsentant des erwarteten Wertes in AVAIL_OUT. Damit ist die Struktur hergestellt, die es später ermöglicht, den redundanten Knoten durch den ϕ-Knoten zu ersetzen. Die Ersetzung findet erst nach dieser Phase statt, da in den Wertemengen Referenzen auf Knoten bestehen, die dadurch ungültig werden würden.

4.6 Eliminierungsphase

Die Eliminierungsphase besucht alle Knoten und bestimmt aus dem Wert eines Knotens die Verfügbarkeit durch AVAIL_OUT des Grundblocks, in dem sich der Knoten befindet. Liefert AVAIL_OUT einen verfügbaren Repräsentanten für den gesuchten Wert, so wird der betrachtete Knoten durch den führenden Knoten ersetzt, falls dieser nicht selbst der führende ist.

5 GVN-PRE Erweiterungsmöglichkeiten

Im Folgenden wird erläutert, wie die im GVN-PRE Paper bereits angeprochene Erweiterbarkeit des Algorithmus auf Lade- und Speicherbefehle für Firm-Graphen realisiert werden kann.

Des Weiteren wird die Behandlung von Endlosschleifen und deren Konsequenzen für die ANTIC Wertemenge untersucht. Die Platzierung der gehobenen Ausdrücke ist ein ebenfalls ein Punkt der aufgegriffen wird, da sich die Vervielfachung von Knoten negativ auf die Codegröße auswirken kann. Zudem bietet dieses Gebiet eine weitere Nutzung der ANTIC Wertemenge, wodurch der Registerdruck verringert werden kann.

Nicht näher behandelt wird eine Klasse von Redundanzen, die GVN-PRE vernachlässigt. Grund dafür, ist der wert-weise Schnitt bei der Erstellung der ANTIC_OUT Wertemenge, im Fall von mehreren Nachfolgeblöcken. Bei dem Graph in Abbildung 5.1 ist $\texttt{Add}_3$ auf beiden möglichen Pfaden durch `Grundblock 4` redundant. $\texttt{Add}_3$ wird in `Grundblock 3` jedoch nicht erwartet, weshalb er nicht als partielle Redundanz – im Sinne der Einfüge-Phase – erkannt wird. Die Kriterien für eine vollständige Redundanz treffen ebenfalls nicht zu, da $\texttt{Add}_1$ und $\texttt{Add}_2$, bedingt durch ihre unterschiedlichen Werte, in `Grundblock 3` nicht verfügbar sind.

5.1 Endlosschleifen

Falls die Anzahl der ANTIC-Iterationen nicht limitiert wird, so terminiert die Fixpunktiteration bei Endlosschleifen nicht, falls ein schleifenvarianter Wert im Schleifenkopf erwartet wird. Da der Schleifenkopf Nachfolger eines Grundblöcke der Schleife ist, erfolgt die Bestimmung von ANTIC_OUT durch Übersetzung der erwarteten Werte. Sei in einer Schleife $while(1) : i = i + 1$ durch Initialisierung mit $i = 0$ zunächst $0 + 1$ erwartet. Die Übersetzung davon liefert bereits

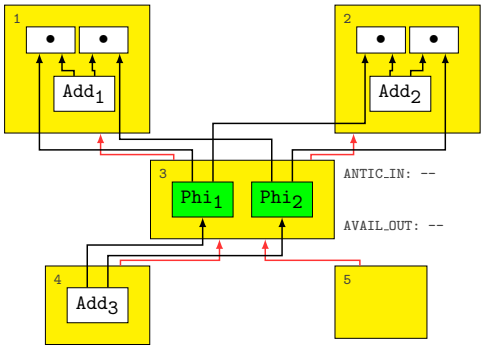


Abbildung 5.1: Durch GVN-PRE nicht erkannte Art von vollständigen Redundanzen

$0 + 1 + 1$, wodurch sich ANTIC_IN wiederum ändert und eine weiter Iteration erfolgt. Dieses Verhalten ist logisch, da bei einer Endlosschleife alle für i möglichen Werte erwartet werden. Ist eine Schleife nicht endlos, tritt dies nicht auf, da ein Pfad zum Endblock existiert, mit dem der Schnitt von ANTIC_IN gebildet wird, was die ANTIC Wertemenge einschränkt. Selbst bei der Limitierung der durchgeführten ANTIC-Iterationen, führt dies dazu, dass ANTIC zusätzlich Werte der ausgerollten Schleife enthält. Um dies zu vermeiden, ist es nötig, Endlosschleifen zu erkennen und ANTIC_IN auf sinnvolle Werte zu begrenzen.

5.1.1 Endlosschleifen im Firm-Graph

Eine Endlosschleife kann nicht durch Steuerflusspfade verlassen werden. Zudem können Endlosschleifen nicht verschachtelt sein, da sie keine starke Zusammenhangskomponente mit der potentiell äußeren Schleife bilden. Allerdings können endliche Schleifen enthalten sein. In libFirm existiert das Konzept der lebenserhaltenden Kanten, die andernfalls nicht mit dem Ende verbundene Knoten und Grundblöcke erhalten. Mindestens ein Grundblock von Endlosschleifen muss durch eine solche Kante erreichbar sein. Allerdings werden diese Kanten großzügig eingefügt, sodass aus der Existenz keine Schlüsse über das Vorhandensein von Endlosschleifen gezogen werden können. Aus diesem Grund wurde der folgende Algorithmus zur Erkennung von Endlosschleifen entwickelt.

Endlosschleifen – bzw. die Grundblöcke, die Teil einer solchen sind – können mit Hilfe einer Steuerflussanalyse erkannt und markiert werden. Voraussetzung ist die Existenz eines Schleifenbaumes, sowie Möglichkeiten zur Markierung von

Schleifen und Grundblöcken. Zunächst werden alle über reguläre Steuerflusskanten vom Endblock aus erreichbaren Grundblöcke als erreichbar markiert. Auf diese Weise bleiben Teilbäume unmarkiert, die eine Endlosschleife enthalten. Wird bei der Markierung topologisch vorgegangen, so wird ein Grundblock des unerreichbaren Teilbaums über eine beliebige lebenserhaltend Kante der Endlosschleife betreten. Da Endlosschleifen nicht verschachtelt sein können, muss es sich bei der äußersten Schleife, in der sich der Grundblock befindet, um die Endlosschleife handeln. Wird die als Endlosschleife erkannte Schleife verlassen, so wird der Grundblock, der nicht mehr Teil der Endlosschleife ist, als erreichbar markiert, da keine weitere Endlosschleife auf diesem Aufwärtspfad existieren kann.

5.1.2 Behandlung von Endlosschleifen

Die Bearbeitung von Endlosschleifen kann komplett ausgelassen werden, falls während der ANTIC-Erstellung ausschließlich echte Steuerflusspfade verfolgt werden. Dadurch wird der gesamte Teilbaum, der die Endlosschleife enthält, nicht besucht. Dies führt allerdings auch dazu, dass auf diesem Teilbaum keine Optimierung erfolgt, und keine ANTIC Wertemengen erstellt wird.

Mit der Information, welche Grundblöcke Teil von Endlosschleifen sind, kann durch das Ignorieren der Rückwärtskanten der Endlosschleife bei der ANTIC-Erstellung verhindern, dass Informationen darüber fließen. Dadurch wird die Endlosschleife als einmalig ausgeführt betrachtet. Dies ermöglicht die Optimierung von verschachtelten Schleifen und schließt den Teilbaum nicht komplett aus.

5.1.3 Implementierung

Die Datenflussanalyse wird durch topologisches Besuchen aller Grundblöcke, vom Endblock ausgehend, durchgeführt. Zuvor sind alle Grundblöcke als *unerreichbar* markiert, bis auf den Endblock. Ist ein Grundblock erreichbar, so werden alle durch echte Steuerflusskanten verbundenen Vorgänger ebenfalls als erreichbar markiert. Liegt ein nicht markierter Grundblock vor, so ist dieser Teil einer Endlosschleife, und markiert alle Vorgänger, die nicht mehr Teil genau dieser Schleife sind, als erreichbar. Grundblöcke, die anschließend nicht markiert

sind, sind Teil von Endlosschleifen. Die ANTIC Datenflussanalyse wird schließlich um eine Bedingung erweitert, die keine Informationen über Rückwärtskanten von Endlosschleifen fließen lässt.

5.2 Lade- und Speicherinstruktionen

Neben den von GVN-PRE bereits abgedeckten Operatoren existieren weitere, die ebenfalls Kandidaten für partielle Redundanzen sind, darunter Lade-, Speicher- und Divisionsoperatoren. Allerdings benötigen diese zusätzliche Aufmerksamkeit, da sie eine festgelegte Ausführungsreihenfolge erfordern. Das Ziel ist, Ladeinstruktionen so in GVN-PRE einzubinden, dass sowohl die Ladeadresse auf Wertebene verglichen werden kann, als auch die partiell redundante Ladeinstruktion. Dies führt dazu, dass auch darauffolgende Operatoren von GVN-PRE erfasst und optimiert werden können. Die Ausführungsreihenfolge wird in libFirm durch das Konzept der Speicherkanten realisiert. Jeder dieser Operatoren besitzt einen Speicherkanten-Eingang, sowie die Möglichkeit eines Speicherkanten-Ausgangs. Da diese Speicherinstruktionen mehrere Werte liefern, werden Projektionsknoten verwendet, um das Ergebnistupel auf den geladenen Wert und die ausgehende Speicherkante zu projizieren. In Abbildung 5.2 sind partiell redundante Lade-Knoten dargestellt, wobei die zu ladenden Adressen den gleichen Wert besitzen und die Projektionen jeweils ein ganzzahliges Ergebnis liefern. Bei Grundblöcken mit mehreren Eingängen wird für Speicherkanten zudem ein Speicher-ϕ benötigt.

Da der Schwerpunkt auf partielle Redundanzen gelegt wird, werden nur Ladeinstruktionen mit direktem Speicher-ϕ als Vorgänger in EXP_GEN eingefügt. Volatile Ladeinstruktionen werden ignoriert, da das Verschieben auf andere Pfade unzulässig ist. Die Bestimmung eines Wertes erfolgt in der ANTIC-Phase nach der ϕ-Übersetzung, weshalb die übersetzte Ladeinstruktion bereits mit einer bestehenden vergleichbar sein muss. Falls eine Situation wie in Abbildung 5.2 besteht, wird zunächst sichergestellt, dass es sich bei dem vorhergehenden Befehl ebenfalls um eine Ladeinstruktion handelt. Da dies in `Grundblock 2` der Fall ist, findet die ϕ-Übersetzung statt, allerdings wird `Load*` direkt an den Speicher-Vorgänger von $\texttt{Load}_2$ gehängt. Dadurch erhält `Load*` den gleichen Wert wie $\texttt{Load}_2$ und die Verfügbarkeit kann ohne umfangreiche Spezialfälle festgestellt werden.

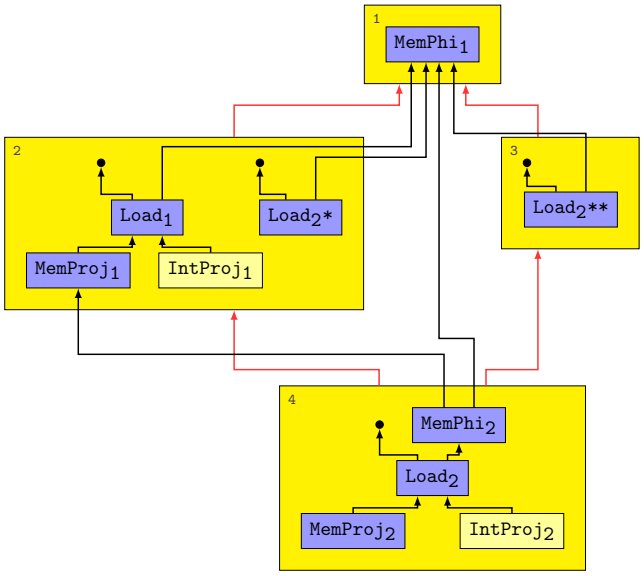


Abbildung 5.2: ϕ-Übersetzung bei partiell redundanter Ladeinstruktion

5.2.1 Implementierung

Für die Änderungen zur Behandlung von Speicherinstruktionen ist es zunächst erforderlich diese und ihre Projektionen ebenfalls in die Wertemengen einzufügen. Der wesentliche Teil wird während der ANTIC-Phase in der ϕ-Übersetzung durchgeführt. Dort wird bei der Übersetzung von $\texttt{Load}_2$ in `Grundblock 2` festgestellt, dass diesem eine Speicherprojektion und eine weitere Ladeinstruktion vorausgehen. Wie bei der bisherigen ϕ-Übersetzung werden zunächst die übersetzten Vorgänger von $\texttt{Load}_2$ bestimmt. Da $\texttt{MemPhi}_2$ in $\texttt{MemProj}_1$ übersetzt wird, kann das adjazente $\texttt{Load}_1$ gefunden werden. $\texttt{Load}_2\texttt{*}$ wird direkt mit dem Speichereingang von $\texttt{Load}_1$ verbunden, wodurch beide Knoten den gleichen Wert erhalten. Da, bei der Übersetzung in `Grundblock 3`, keine adjazente Speicherinstruktion vorhanden ist, erfolgt diese wie bisher.

Projektionen stellen keine Ausnahme dar, da ihr Wert, durch die Abhängigkeit von Speicherinstruktionen, stets korrekt bestimmt wird. In der Einfügephase wird für $\texttt{Load}_2\texttt{*}$ und $\texttt{Load}_2\texttt{**}$ kein ϕ-Knoten erstellt, da sie Tupel als Ergebnis liefern. Stattdessen erzeugen die Projektionen jeweils ein Speicher-ϕ und einen ϕ-Knoten für das Ergebnis der Speicherinstruktion.

5.3 Code Placement

GVN-PRE erzeugt durch das Vervielfachen von Knoten potentiell zusätzlichen Code. Wenn man von einem Beispiel wie in Abbildung 2.2 ausgeht, so wäre die Platzierung von `Add2*` in Grundblock 1 vorteilhaft. Dadurch wird die Redundanz behoben, jedoch die Codegröße nicht erhöht. Zudem ergibt sich der Vorteil, dass die beiden Operanden bereits in `Grundblock 1` keine Verwendung mehr besitzen und durch den `Add`-Knoten in nur einen Ausdruck überführt werden. Dies reduziert den Registerdruck nach `Grundblock 1`.

Dabei muss darauf geachtet werden, dass dadurch keine Pfade verlängert werden. Dies kann durch die vorhandene ANTIC Wertemenge sichergestellt werden. Solange die Werte der Operanden verfügbar sind, und das Ergebnis des Operators erwartet wird, kann der Knoten in dominierenden Grundblöcke platziert werden. In Betracht kommen Knoten, deren Operanden auf dem Pfad von dem neuen Zielblock zum Endblock ausschließlich zur Berechnung des Wertes des Knotens verwendet werden. Dieser Wert wird durch die Platzierung im neuen Zielblock verfügbar und dadurch führender Knoten, wodurch mittels GVN alle redundanten Knoten entfernt werden können.

5.3.1 Implementierung

Code Placement wurde als möglichst unabhängige Phase konzipiert und wird zwischen Einfüge-Phase und Eliminierungsphase ausgeführt. Benötigt werden die Wertemengen ANTIC_IN und AVAIL_OUT, sowie die Übersetzungs-Liste. Da ausschließlich partiell redundante Knoten betrachtet werden, sind die in ANTIC_DONE befindlichen Einträge ausreichend.

Grundsätzlich ist der Vorgang zunächst vergleichbar mit der Einfüge-Phase. An Grundblöcken mit mehreren Eingängen wird jeder Eintrag in ANTIC_DONE betrachtet. Alle in ANTIC_DONE befindlichen Werte sind bereits vollständig redundant und somit in den Vorgängerblöcken verfügbar. Diese Knoten sind von Interesse, da durch ihre Platzierung im direkten Dominator mehrere Ausdrücke eingespart werden können. Dies reduziert nicht die Anzahl der Instruktionen, ist allerdings für die Codegröße vorteilhaft. Zudem soll dadurch der Registerdruck gesenkt, jedoch keinesfalls erhöht werden. Dies wird erreicht, indem sichergestellt wird, dass die Operanden auf allen Pfaden vom Dominator zum Endblock nicht

mehr verwendet werden. Dies ist gleichbedeutend damit, dass die Operanden auf allen Pfaden, die von dem neuen Zielblock dominiert werden, ausschließlich für die Berechnung des Werts verwendet werden. Außerdem muss der Wert im Zielblock erwartet sein, sodass keine Pfade verlängert werden. Bedingung für die Platzierung im Zielblock ist zudem die Verfügbarkeit der Werte aller Vorgängerknoten. Werden alle Anforderungen erfüllt, so wird der Knoten in den Zielblock kopiert und dort in AVAIL_OUT eingetragen. Dies ermöglicht das Platzieren von Knoten, die von diesem Wert abhängen.

Da sich durch das Einfügen neuer führender Knoten AVAIL_OUT ändert, werden alle Grundblöcke des Dominatorbaums besucht, um Änderungen zu propagieren. Dadurch werden die in der Einfüge-Phase gehobenen Knoten vollständig redundant und können in der Eliminierungsphase entfernt werden.

6 Auswertung

Die Messung der Ausführungszeiten erfolgte durch das Ausführen von SPEC CINT2000 Benchmarks auf einem Rechner mit einem 3,20GHz Intel i3 550 Prozessor und 1877 MiB Speicher. Die Anzahl der ANTIC-Iterationen wurde – wie in [VH04] erwähnt – auf 10, die Iterationen der Einfüge-Phase auf 3 begrenzt, da dies keine Auswirkungen auf die Ergebnisse ergab. Alle Messungen erfolgten mit der höchsten Optimierungsstufe 4. Die Tabellen geben die Laufzeiten in Sekunden an, sowie deren Verhältnis.

Benchmark	NO-GCSE	NO-GCSE GVN-PRE	Verhältnis
164.gzip	76.93	76.66	99.65 %
175.vpr	71.56	72.24	100.95 %
176.gcc	33.59	33.56	99.91 %
181.mcf	39.60	38.32	96.76 %
186.crafty	37.91	37.39	98.64 %
197.parser	86.05	86.12	100.08 %
253.perlbmk	66.92	68.07	101.72 %
254.gap	43.48	40.80	93.85 %
255.vortex	68.25	68.31	100.08 %
256.bzip2	70.94	70.91	99.97 %
300.twolf	101.61	99.06	97.50 %
			ø 99.01 %

Tabelle 6.1: O4 ohne GCSE im Vergleich zu GVN-PRE ohne GCSE

Für die Messung in Tabelle 6.1 wurde die GCSE abgeschaltet, um die Auswirkungen von GVN-PRE ohne Erweiterungen, hervorzuheben. Das Ergebnis ist im Durchschnitt eine Beschleunigung um 0,99% durch GVN-PRE. Wesentlich langsamer wurden dagegen 175.vpr und 253.perlbmk.

Dies kann durch die Nebeneffekte von GVN-PRE erklärt werden. Am offen-

sichtlichsten ist, dass GVN-PRE durch das Einfügen neuer Knoten letztlich der Code vergrößert werden kann, was sich durch Cache-Effekte negativ auswirkt. Zudem wird die Erstellung neuer ϕ-Knoten, die durch das Heben entstehen, nicht limitiert. Jeder neue ϕ-Knoten bedeutet ein zusätzlich benötigtes Register innerhalb des Grundblocks und erhöht dadurch den Registerdruck, sodass potentiell mehr Werte in den Speicher ausgelagert werden müssen. Die GCSE behandelt sich nicht dominierende Werte als unterschiedlich, und fügt, im Gegensatz zu GVN-PRE, keine ϕ-Knoten ein, um die Verfügbarkeit dieser Werte herzustellen.

Ein weiterer Punkt ist, dass Neuberechnungen durch die Eliminierung aller gefundenen Redundanzen entfernt werden. Die Berechnung eines Wertes erfolgt möglichst einmalig, wodurch das Ergebnis für alle Verwender vorgehalten werden muss. Kann dieser Informationsgewinn nicht für weitere Optimierungen verwendet werden, ist möglicherweise die Neuberechnung in Situationen vorteilhaft, in denen der Zugriff auf Speicher teurer ist.

Benchmark	O4	GVN-PRE	Verhältnis
164.gzip	76.39	78.16	102.32 %
175.vpr	69.50	71.15	102.37 %
176.gcc	33.46	34.13	102.00 %
181.mcf	38.21	38.53	100.85 %
186.crafty	36.21	36.94	102.01 %
197.parser	85.97	86.64	100.78 %
253.perlbmk	67.84	68.61	101.14 %
254.gap	39.31	39.51	100.51 %
255.vortex	66.41	66.60	100.28 %
256.bzip2	70.65	71.32	100.94 %
300.twolf	98.79	99.44	100.66 %
			ø 101.26 %

Tabelle 6.2: O4 (GCSE inbegriffen) im Vergleich zu O4 mit GVN-PRE.

Tabelle 6.2 zeigt, dass bei eingeschalteter GCSE, die zusätzliche Ausführung von GVN-PRE eine durchschnittliche Verlangsamung ergibt. Um die Ursachen festzustellen, wurde die Einfüge-Phase um eine Bedingung erweitert, welche die Erstellung von ϕ-Knoten einschränkt und gleichzeitig die Anzahl der gehobenen Knoten limitiert. Diese Bedingung erlaubt das Heben partiell redundanter Knoten, falls diese einziger Nachfolger eines ϕ-Knotens innerhalb des gleichen

Grundblocks sind. Dadurch wird der vorhandene ϕ-Knoten durch einen neuen ersetzt und nur ein Knoten gehoben. Die Messergebnisse befinden sich in Tabelle 6.3. Zudem wurde in Tabelle 6.4 die Anzahl der ausgeführten Instruktionen verglichen.

Die geringen Auswirkungen sind wegen der Einschränkung zu erwarten. Allerdings ist 186.crafty weiterhin langsamer und besitzt ebenfalls die größte Zahl an zusätzlichen Instruktionen. Da sowohl 181.mcf, als auch 256.bzip2 ebenfalls eine gesteigerte Anzahl Instruktionen benötigt, wobei 181.mcf eine Geschwindigkeitssteigerung aufweist, liegt die Erklärung in Speichereffekten. Die Eliminierung vollständiger Redundanzen kann hierbei weiterhin neue ϕ-Knoten erstellen und die Lebenszeit von Werten verlängern.

Benchmark	O4	GVN-PRE mit Erweiterungen	Verhältnis
164.gzip	76.39	76.94	100.72 %
175.vpr	69.50	70.28	101.13 %
176.gcc	33.46	33.60	100.42 %
181.mcf	38.21	37.70	98.66 %
186.crafty	36.21	36.58	101.02 %
197.parser	85.97	86.26	100.33 %
253.perlbmk	67.84	68.43	100.87 %
254.gap	39.31	40.01	101.78 %
255.vortex	66.41	66.93	100.78 %
256.bzip2	70.65	70.81	100.23 %
300.twolf	98.79	98.96	100.17 %
			ø 100.56 %

Tabelle 6.5: O4 im Vergleich zu GVN-PRE mit allen implementierten Erweiterungen

Mit aktivierten Erweiterungen ist in Tabelle 6.5, eine Verbesserung gegenüber dem reinen GVN-PRE Algorithmus zu erkennen. Durch die verbesserte Platzierung von Knoten können einige ϕ-Knoten eingespart werden. Zudem kann die Zunahme der Codegröße durch das Zusammenfassen gehobener Knoten im Dominator reduziert werden. Durch die Ergebnisse des Experiments zur Schnittminimierung wird für 186.crafty auch hier keine wesentliche Verbesserung erwartet.

Benchmark	O4	GVN-PRE Register-druckreduzierung	Verhältnis
164.gzip	77.11	76.76	99.55 %
175.vpr	69.50	69.67	100.24 %
176.gcc	33.55	33.34	99.39 %
181.mcf	38.30	37.49	97.87 %
186.crafty	36.46	36.87	101.14 %
197.parser	86.86	86.12	99.15 %
253.perlbmk	73.08	72.36	99.01 %
254.gap	38.92	39.00	100.22 %
255.vortex	66.58	66.67	100.12 %
256.bzip2	69.44	69.63	100.28 %
300.twolf	98.76	99.12	100.37 %
			ø 99.76 %

Tabelle 6.3: Experiment: GVN-PRE mit Registerdruckreduzierung

Benchmark	O4	GVN-PRE Register-druckreduzierung	Verhältnis
164.gzip	316 128 041 442	316 176 126 292	100.02
175.vpr	198 912 787 145	198 913 337 610	100.00
176.gcc	147 868 820 631	147 798 626 050	99.95
181.mcf	47 750 125 496	47 883 978 990	100.28
186.crafty	184 966 774 856	186 342 975 294	100.74
197.parser	306 867 492 740	306 853 183 226	100.00
253.perlbmk	377 756 243 694	377 544 698 315	99.94
254.gap	218 097 439 752	217 837 024 335	99.88
255.vortex	323 758 708 139	323 734 251 330	99.99
256.bzip2	277 173 787 425	278 010 071 781	100.30
300.twolf	294 353 036 854	294 259 217 266	99.97

Tabelle 6.4: Vergleich der ausgeführten Instruktionen bei GVN-PRE mit Registerdruckreduzierung.

7 Zusammenfassung und Ausblick

Zusammenfassend ist GVN-PRE ein Verfahren, das eine gute Basis für die Eliminierung partieller und vollständiger Redundanzen bietet und viele Möglichkeiten zur Erweiterung erlaubt. Einige Erweiterungen, wie Code Placement, sind notwendig, da GVN-PRE gehobene Ausdrücke möglichst spät und ohne Beachtung des Steuerflusses einfügt. Es wurde die Leistung von GVN-PRE evaluiert und einige Ansätze zu Erweiterungen aufgezeigt, die sich überwiegend positiv auf die Laufzeiten auswirken. Die Laufzeiten mit GVN-PRE sind bei dem verwendeten Übersetzer letztendlich schlechter als diejenigen mit GCSE. Ein Grund ist, dass sich GVN-PRE weniger konservativ als die vorhandene GCSE verhält.

Im Folgenden wird eine Reihe weiterer Möglichkeiten erläutert, die der Verbesserung des Algorithmus dienen können.

7.1 Registerdruck

Durch die bei der Behebung partieller Redundanzen neu eingebrachten ϕ-Knoten, erhöht sich der Registerdruck innerhalb des Grundblocks. In Abbildung 7.1 sind die Knoten $\texttt{A}_2$, $\texttt{B}_2$ und $\texttt{C}_2$ partiell redundant. Das Ergebnis nach GVN-PRE in Abbildung 7.2 enthält zusätzliche ϕ-Knoten für in `Grundblock 4`, sodass eine größere Zahl von Registern benötigt wird. Ursache ist der Schnitt zwischen den partiell redundanten Knoten und den nicht redundanten Knoten. Jede Kante, die über die Schnittgrenze hinweg führt, verursacht einen ϕ-Knoten. Um die Anzahl der neu erzeugten ϕ-Knoten zu minimieren, muss der Schnitt minimiert werden.

Da die Einfüge-Phase allerdings topologisch jeden Knoten für sich betrachtet, kann kein Gesamtbild erstellt werden. Bei der Analyse eines Knotens ist nur bekannt, dass das Heben möglich ist und ob er redundant ist. Falls beide Bedingungen erfüllt sind, wird die Redundanz entfernt. Andernfalls ist der Wert des

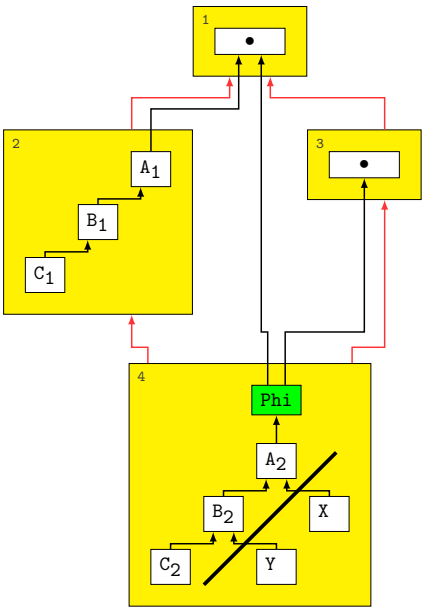


Abbildung 7.1: Schnitt zwischen redundanten und richtredundanten Knoten.

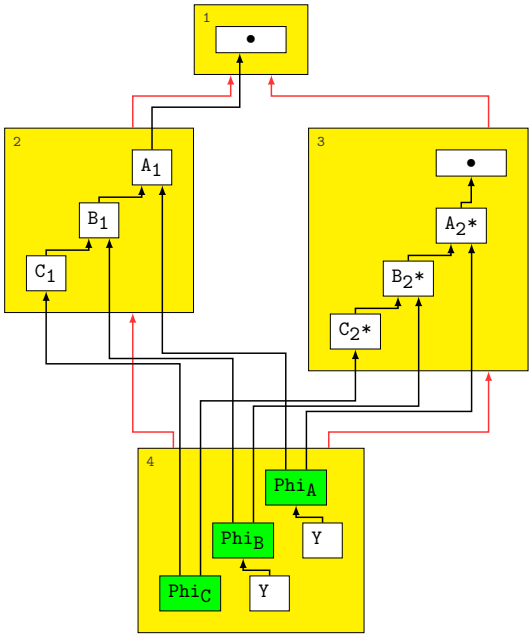


Abbildung 7.2: Erhöhter Registerdruck durch Heben der redundanten Knoten.

Knotens in den Vorgängerblöcken nicht verfügbar, wodurch eine Bearbeitung aller Knoten, die von seinem Wert abhängen, verhindert wird. Bisher werden partielle Redundanzen nicht behoben, falls sie von Operanden abhängen, die selbst nicht redundant sind. Dies verhindert das gierige Heben ihrer Vorgängerknoten.

Es kann von Vorteil sein, zunächst alle partiellen Redundanzen in einem Grundblock zu markieren um anschließend mit einer Heuristik die Entscheidung treffen zu können, welche Knoten gehoben werden. Diese Entscheidung, ob das gierige Heben eines Knotens sowohl den Schnitt minimiert, als auch eine größere Zahl von Redundanzen entfernt, kann verzögert werden.

Die Heuristik kann den Schnitt minimieren, indem nur Knoten mit einem oder wenigen Verwendern gehoben wird. Falls allerdings alle Verwender selbst in dem betrachteten Grundblock liegen und redundant sind, werden diese ebenfalls zu Kandidaten der Optimierung. Die Heuristik bestimmt, ob konservativ vorgegangen wird oder Knoten gierig gehoben werden dürfen. Dies kann durch ein Aufteilen der Einfüge-Phase erreicht werden.

Der erste Schritt, ist das Markieren aller redundanten Knoten. Die Heuristik passt die Markierungen an, sodass bestimmte Knoten möglicherweise gierig gehoben werden, oder partielle Redundanzen belassen werden. Daraufhin kann der zweite Teil der Einfüge-Phase die gewählten Knoten heben.

7.2 Code Placement

Die Code Placement Erweiterung konzentriert sich darauf, gehobene Knoten in dem gemeinsamen Dominator zu platzieren. Dazu wird ANTIC_IN herangezogen, um festzustellen, dass der Wert in dem Dominator erwartet wird, um keine Pfade zu verlängern. Die Definition von ANTIC_IN schließt jedoch Knoten aus, die nicht über den betrachteten Grundblock gehoben werden können. Dadurch sind beispielsweise Nachfolger von Funktionsaufrufen, nicht enthalten, obwohl sie jedoch in diesem Grundblock platziert werden können. Folglich sollte ANTIC_IN mit einer angepassten *clean*() Funktion neu bestimmt werden, um auch diese Werte zu enthalten. Ist es möglich, solche Einträge von ANTIC_IN durch Markieren kenntlich zu machen, können diese Einträge bereits bei der Erstellung der Wertemenge eingefügt werden. Diese Einträge werden bis zur Code

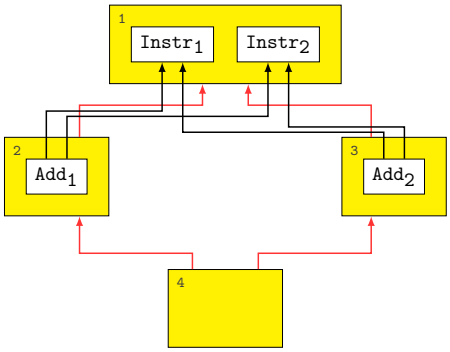


Abbildung 7.3: Erweiterung des Code Placements: Platzierung im Dominator ohne PRE

Placement Phase ignoriert.

Weitere Möglichkeiten ergeben sich, wenn sich die in dieser Phase bearbeiteten Ausdrücke nicht auf auf die durch PRE gehobenen beschränken. Durch die Verwendung von ANTIC_DONE als Quelle der betrachteten Werte, werden die Möglichkeiten limitiert. Ein Beispiel dafür beinhaltet der Graph in Abbildung 7.3. Da die Ausdrücke `Add`$_1$ und `Add`$_2$ in `Grundblock 1` erwartet werden, ist es möglich, sie dort zu platzieren. Da die Ausdrücke jedoch nicht in `Grundblock 4` erwartet werden, sind sie nicht in ANTIC_DONE enthalten. Wird ein Wert nicht in `Grundblock 4` erwartet, allerdings in dem direkten Dominator `Grundblock 1`, so wäre dies ebenfalls ein Kandidat für die Code Placement Erweiterung. Dabei müsste allerdings beachtet werden, dass dieser Wert auf mehreren Pfaden durch unterschiedliche Berechnungen entstehen kann, und ein Repräsentant ausgewählt werden muss, dessen Operanden im Dominator verfügbar sind.

Literaturverzeichnis

[AWZ88] Alpern, B. ; Wegman, M. N. ; Zadeck, F. K.: Detecting equality of variables in programs. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1988 (POPL '88). – ISBN 0–89791–252–7, 1–11

[CCK+97] Chow, Fred ; Chan, Sun ; Kennedy, Robert ; Liu, Shin-Ming ; Lo, Raymond ; Tu, Peng: A new algorithm for partial redundancy elimination based on SSA form. In: *SIGPLAN Not.* 32 (1997), Mai, Nr. 5, 273–286. http://dx.doi.org/10.1145/258916.258940. – DOI 10.1145/258916.258940. – ISSN 0362–1340

[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), Oct, Nr. 4, 451–490. http://doi.acm.org/10.1145/115372.115320

[Coc69] Cocke, John: *Programming languages and their compilers: Preliminary notes*. Courant Institute of Mathematical Sciences, New York University, 1969. – ISBN B0007F4UOA

[CX03] Cai, Qiong ; Xue, Jingling: Optimal and efficient speculation-based partial redundancy elimination. In: *in '1st IEEE/ACM International Symposium on Code Generation and Optimization*, 2003, S. 91–102

[KCL+99] Kennedy, Robert ; Chan, Sun ; Liu, Shin-Ming ; Lo, Raymond ; Tu, Peng ; Chow, Fred: Partial redundancy elimination in SSA form. In: *ACM Trans. Program. Lang. Syst.* 21 (1999), Mai, Nr. 3, 627–676. http://dx.doi.org/10.1145/319301.319348. – DOI 10.1145/319301.319348. – ISSN 0164–0925

[KRS92] Knoop, Jens ; Rüthing, Oliver ; Steffen, Bernhard: Lazy code motion. In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. New York, NY, USA : ACM, 1992 (PLDI '92). – ISBN 0–89791–475–9, 224–234

[LBBG05] Lindenmaier, Götz ; Beck, Michael ; Boesler, Boris ; Geiss, Rubino: Firm, an Intermediate Language for Compiler Research. Version: 3 2005. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003172`. University of Karlsruhe, 3 2005 (2005-8). – Forschungsbericht. – 19 S.

[Lin02] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Version: Sep 2002. `http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps`. Universität Karlsruhe, Fakultät für Informatik, Sep 2002 (2002-5). – Forschungsbericht. – 75 S.

[MR79] Morel, E. ; Renvoise, C.: Global optimization by suppression of partial redundancies. In: *Commun. ACM* 22 (1979), Februar, Nr. 2, 96–103. `http://dx.doi.org/10.1145/359060.359069`. – DOI 10.1145/359060.359069. – ISSN 0001–0782

[RL77] Reif, J. H. ; Lewis, H. R.: Symbolic Evaluation and the Global Value Graph. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, 104–118

[RWZ88a] Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988, 12–27

[RWZ88b] Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1988 (POPL '88). – ISBN 0–89791–252–7, 12–27

[Sch04] Scholz, Bernhard: Optimizing for space and time usage with speculative partial redundancy elimination. In: *In Proceedings of the*

2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2004, S. 221–230

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karlsruhe, Fakultät für Informatik. Version: Dec 1999. `http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz`. Universität Karlsruhe, Fakultät für Informatik, Dec 1999 (1999-14). – Forschungsbericht. – 0–40 S.

[VH04] Vandrunen, Thomas ; Hosking, Antony L.: Value-based partial redundancy elimination. In: *In CC*, 2004, S. 167–184

[XC06] Xue, Jingling ; Cai, Qiong: A lifetime optimal algorithm for speculative PRE. In: *ACM Trans. Archit. Code Optim.* 3 (2006), Juni, Nr. 2, 115–155. `http://dx.doi.org/10.1145/1138035.1138036`. – DOI 10.1145/1138035.1138036. – ISSN 1544–3566