Registerzuteilung für Prozessor−Cluster mit dynamisch rekonfigurierbaren Registerbänken

Diplomarbeit

Ralf Dreesen

Universität Paderborn
FG Kastens

Diplomarbeit vorgelegt bei

Prof. Dr. Uwe Kastens

und

Prof. Dr. Odej Kao

Danksagung

Mein Dank gilt Herrn Prof. Dr. Uwe Kastens für die konstruktiven Ratschläge und die Erstkorrektur der Diplomarbeit, sowie Herrn Prof. Dr. Odej Kao für die Zweitkorrektur.

Für die hervorragende Betreuung bin ich Michael Thies und Michael Hussmann dankbar, die mir in zahlreichen Diskussionen wertvolle Anregungen für die Diplomarbeit gaben.

Gewidmet ist die Arbeit meinen Eltern, die mich stets selbstlos unterstützt haben und mir das Studium so ermöglichten.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe sowie ohne Benutzung anderer als den angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Diplomarbeit wird ein Registerzuteilungsverfahren für einen Übersetzer konzipiert. Die Zielarchitektur des Übersetzers ist ein Prozessor-Cluster, bei dem mehrere Prozessoren ein Programm auf Instruktionsebene parallel ausführen. Die Prozessoren verfügen dabei über gemeinsame Registerbänke, die dynamisch, also zur Laufzeit, rekonfiguriert werden können. Um diese Registerbänke zu nutzen, muss im Übersetzer ein spezielles Registerzuteilungsverfahren verwendet werden. Dieses Verfahren wird in der Diplomarbeit beschrieben.

1.1 Motivation

Die Leistungsfähigkeit eines Prozessors kann gesteigert werden, indem die Anzahl der Prozessorregister erhöht wird. Dadurch können während der Programmausführung mehr Werte in Registern gespeichert werden, wodurch sich die Anzahl der Zugriffe auf den Hauptspeicher verringert.

Um eine größere Menge an Registern ansprechen zu können, muss allerdings auch das Instruktionsformat geändert werden. Die Änderung ist notwendig, weil nun mehr Platz benötigt wird, um die größeren Registernummern in einer Maschineninstruktion zu codieren. Solche Erweiterungen sind in der Regel aber nicht möglich, weil die Größe der Maschineninstruktionen begrenzt ist.

Durch *dynamisch rekonfigurierbare Registerbänke* wird dieses Problem gelöst. Es kann sowohl eine größere Menge an Registern genutzt, als auch das Instruktionsformat beibehalten werden. Um dies zu ermöglichen, wird ein Einblendungsmechanismus verwendet. Für diesen Mechanismus werden Registerrahmen eingeführt, in welche die Register eingeblendet werden können. In einer Instruktion wird nun nicht mehr ein Register codiert, sondern ein Registerrahmen. Damit das richtige Register angesprochen wird, muss es zuvor durch eine Rekonfigurationsanweisung in den Registerrahmen eingeblendet werden. Diese Rekonfigurationen der Registerbank werden dynamisch, also zur Laufzeit, durchgeführt. Wird durch eine Rekonfiguration eine Einblendung hergestellt, bleibt diese bestehen, bis ein anderes Register in den Rahmen eingeblendet wird.

Damit nicht jedes Register einzeln eingeblendet werden muss und um die Hardwarekomplexität zu verringern, werden die Register zudem in Blöcke einer festen Größe unterteilt. Statt nun ein einzelnes Register in einen Rahmen einzublenden, wird ein ganzer Block von Registern in einen Block von Registerrahmen eingeblendet. Wie sich später in der Evaluierung zeigen wird, verringert sich dadurch der Rekonfigurationsaufwand zur Laufzeit.

Eine dynamisch rekonfigurierbare Registerbank eignet sich auch hervorragend für einen synchronen Prozessor-Cluster mit gemeinsamen Registern. Weil die Menge der gemeinsamen Register für mehrere Prozessoren relativ groß sein muss, würden ansonsten wieder Probleme bei dem Instruktionsformat auftreten. Auch wenn die Register gemeinsam genutzt werden, verfügt jeder Prozessor des Clusters über eine eigene Menge an Registerrahmen. Dadurch können die Prozessoren die Einblendungen unabhängig voneinander herstellen.

Anstelle der Begriffe Register und Registerrahmen werden in dieser Arbeit die Begriffe *architektonisches Register* und *physikalisches Register* verwendet. Dabei entspricht ein architektonisches Register einem Registerrahmen und ein physikalisches Register einem Register im vorherigen Sinne. Dementsprechend werden auch die Begriffe architektonischer Block und physikalischer Block verwendet, um einen Block von Rahmen bzw. einen Block von Registern zu bezeichnen.

1.2 Aufgabenstellung

In der Diplomarbeit soll ein Registerzuteilungsverfahren für einen Prozessor-Cluster mit rekonfigurierbaren Registerbänken konzipiert werden.

Die Register sollen bei dem Verfahren so zugeteilt werden, dass möglichst viele Werte in Registern gehalten werden. Dadurch wird die Anzahl der benötigten Hauptspeicherzugriffe gering gehalten, was sich positiv auf die Ausführungsgeschwindigkeit des Programms auswirkt.

Zudem sollen zur Laufzeit möglichst wenige Rekonfigurationsanweisungen ausgeführt werden. Die Register sollen deshalb so zugeteilt werden, dass durch eine einzige Rekonfiguration viele zukünftige Zugriffe ermöglicht werden. Dazu kann im Übersetzer ausgenutzt werden, dass für jede Stelle des Programms Informationen über einen Teil der zukünftigen Programmausführung zur Verfügung stehen.

Neben der Konzipierung soll das Registerzuteilungsverfahren auch in den Übersetzer des Score Prozessors integriert werden.

1.3 Vorgehensweise

Im Folgenden wird der Aufbau der Diplomarbeit skizziert. Dabei wird vom Kern der Arbeit, der Registerzuteilung, ausgegangen. Danach werden die begleitenden Kapitel aufgeführt und in Bezug zur Registerzuteilung gesetzt.

In der Arbeit wird in Kapitel 5 zunächst ein Registerzuteilungsverfahren für einen einzelnen Prozessor mit rekonfigurierbaren Registerbänken konzipiert. Am Ende des Kapitels wird dieses Verfahren in Abschnitt 5.4 auf einen Prozessor-Cluster übertragen.

Zur Durchführung der Registerzuteilung wird das Verfahren von Chaitin, wie in Abschnitt 5.1 beschrieben, erweitert. Dieses Verfahren hat sich für die globale Registerzuteilung bei normalen Registerbänken bewährt und wird heute in kommerziellen Übersetzern eingesetzt. Durch die Erweiterung wird versucht, die Register so zuzuteilen, dass aufeinander folgende Befehle möglichst auf Register des gleichen physikalischen Blocks zugreifen. Auf diese Weise sollen Rekonfigurationen möglichst vermieden werden.

Ist eine Rekonfiguration notwendig, wird, wie in Abschnitt 5.2 beschrieben, nach der Strategie "Latest Used Again" (LUA) entschieden, welcher physikalische Block dadurch verdrängt wird. Es wird dabei der physikalische Block verdrängt, der am spätesten wiederverwendet wird.

Um die dafür notwendigen Informationen zu erlangen, wird eine Analyse der Registerzugriffe durchgeführt, die vorweg in Kapitel 4 beschrieben wird.

Nach der Konzeption des Registerzuteilungsverfahrens wird in Kapitel 6 die Implementierung beschrieben. Abschließend werden in Kapitel 7 einige Ergebnisse des Verfahrens präsentiert und erläutert.

Bevor das Registerzuteilungsverfahren behandelt wird, werden im folgenden Kapitel 2 zunächst die Grundlagen dafür vorgestellt. In dem darauf folgenden Kapitel 3 werden mehrere Eigenschaften beschrieben, die bei rekonfigurierbaren Registerbänken variiert werden können. Einige dieser Eigenschaften haben große Auswirkungen auf das Registerzuteilungsverfahren, sodass in 3.4 eine Registerarchitektur festgelegt wird, für die das Zuteilungsverfahren im folgenden Kapitel konzipiert wird.

Kapitel 2

Grundlagen der Registerzuteilung

In diesem Kapitel werden einige grundlegende Verfahren aus dem Übersetzerbau vorgestellt und verwandte Arbeiten behandelt.

Zunächst wird in Abschnitt 2.1 der Aufbau eines Übersetzers skizziert, um die Registerzuteilung in den Übersetzungsprozess einordnen zu können.

Die Methoden, die für das Registerzuteilungsverfahren benötigt werden, bzw. auf die das Verfahren aufbaut, werden in den beiden darauf folgenden Abschnitten 2.2 und 2.3 beschrieben.

In dem letzten Abschnitt 2.4 werden schließlich zwei verwandte Arbeiten vorgestellt, die Registerarchitekturen mit rekonfigurierbaren Registerbänken behandeln.

2.1 Übersetzerphasen

Das Übersetzen eines Quellprogramms erfolgt in mehreren Phasen, deren Abfolge in Abbildung 2.1 dargestellt ist.

Der Anwender nimmt von einem Übersetzer in der Regel nur den *Driver* wahr, den er zum Übersetzen eines Programms aufruft. Der Driver wiederum ruft für die vier Übersetzungsphasen in der Mitte der Abbildung weitere

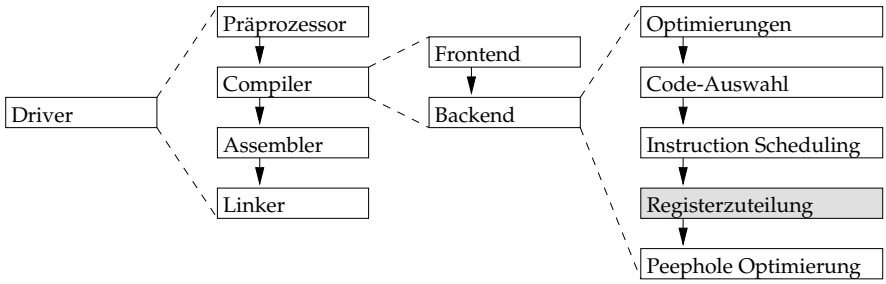

Abbildung 2.1: Übersetzerphasen

eigenständige Anwendungen[1] auf.

Die erste dieser Anwendungen ist der *Präprozessor*, der von der Programmiersprache unabhängig ist und in der Quelldatei Textersetzungen durchführt. Dazu zählt zum Beispiel das Einfügen von anderen Dateien wie auch das Expandieren von Makros oder das Entfernen von Kommentaren. Das Resultat ist ein Text in der Programmiersprache, der die Eingabe des Compilers ist.

Der *Compiler* übersetzt die Programmiersprache in die Maschinensprache des Zielprozessors. Die Ausgabe ist eine Folge von Assemblerinstruktionen. Auf die Phasen des Übersetzers wird gleich näher eingegangen.

Die Assemblerinstruktionen, die vom Compiler erzeugt wurde, werden in der nächsten Phase vom *Assembler* in binäre Maschineninstruktionen übersetzt. Die Ausgabe wird auch als Maschinencode bezeichnet und in einer Objektdatei gespeichert.

Mehrere dieser Objektdateien werden in der letzten Phase vom *Linker* zu einer ausführbaren Datei verknüpft.

Im Folgenden soll der Compiler aus der eben genannten Kette von Anwendungen näher betrachtet werden. Der Übersetzungsvorgang des Compilers kann in mehrere Phasen unterteilt werden, die auf der rechten Seite der Abbildung 2.1 aufgelistet sind.

Zunächst wird der Quelltext durch das *Frontend* eingelesen, analysiert und dem *Backend* in einer Zwischensprache zur Verfügung gestellt. Im Backend werden zu Beginn diverse *Optimierungen* an dem Programmbaum vorgenommen. In der darauf folgenden *Code-Auswahl* werden schließlich Sequenzen von Assemblerbefehlen für die noch abstrakten Befehle im Programmbaum gewählt. Die Assemblerbefehle werden optional im *Instruction Scheduling* umgeordnet, um die Ausführungsgeschwindigkeit zu erhöhen.

Bis zu diesem Zeitpunkt operieren die Assemblerbefehle auf sog. *virtuellen Registern*[2] statt auf *physikalischen Registern*, wie sie durch den Prozessor zur Verfügung gestellt werden. Virtuelle Register stehen in beliebiger Anzahl zur Verfügung, wodurch bei der Code-Auswahl zunächst für jede Variable ein eigenes Register gewählt werden kann. Im nächsten Schritt, der *Registerzuteilung*, wird jedem virtuellen Register ein physikalisches Register zugeteilt. Mit dieser Phase befasst sich diese Diplomarbeit.

Nach der Registerzuteilung werden in der letzten Phase, der *Peephole Optimierung*, noch einige lokale Optimierungen durchgeführt, bevor der Assembler-Code ausgegeben wird.

2.2 Datenflussanalyse

Die Datenflussanalyse [Kas90] ist eine *generische Methode*, um Programmeigenschaften zu bestimmen. Sie wird in Kapitel 4 verwendet, um zukünftige Zugriffe auf Register zu analysieren.

Bei einer Datenflussanalyse wird der *gesamte Kontrollflussgraf* einer Funktion betrachtet, um eine Eigenschaft zu bestimmen. Ein exemplarischer Kontrollflussgraf ist in Abbildung 2.2 dargestellt. Er besteht aus den Grundblöcken B1,B2,B3,B4 und den dazwischen eingezeichneten Kontrollflusskanten. Anders

[1] Assembler und Compiler sind häufig auch in einer einzigen Anwendung integriert

[2] Die virtuellen Register werden auch als *symbolische Register* bezeichnet. In dieser Arbeit wird jedoch ausschließlich der Begriff "virtuelle Register" verwendet.

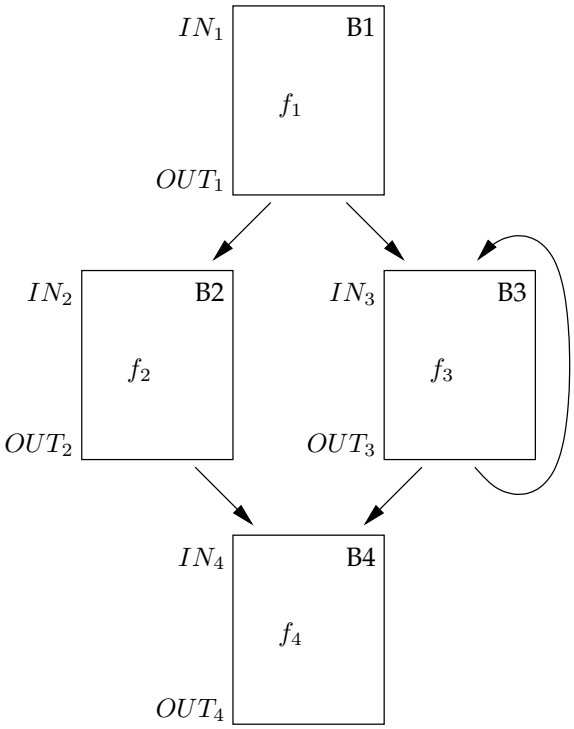

Abbildung 2.2: Beispiel eines Kontrollflussgrafen.

als bei einer lokalen Analyse, die nur einen Grundblock betrachtet, können Eigenschaften bei einer Datenflussanalyse auch über Grundblockgrenzen hinweg propagiert werden. Eine Datenflussanalyse ist also leistungsfähiger, als eine lokale Analyse. Zum Beispiel wird bei dem Datenflussproblem "Konstantenweitergabe" propagiert, ob eine Variable einen konstanten Wert hat und wie dieser lautet. Wird in dem Kontrollflussgrafen aus Abbildung 2.2 einer Variablen v *einmalig* die Konstante 3 im Block B1 zugewiesen, wird diese Eigenschaft der Variablen in die Blöcke B2,B3 und B4 propagiert.

Weil zur Übersetzungszeit nicht bekannt ist, wie die Programmausführung bei einer Verzweigung des Kontrollflusses fortfährt, müssen bei der Datenflussanalyse pessimistische Abschätzungen gemacht werden. Das heißt, dass die ermittelten Eigenschaften für *jede* Ausführung des Programms gelten müssen. Zum Beispiel kann $v = 3$ am Anfang des Grundblocks B4 nur angenommen werden, wenn $v = 3$ am Ende der Grundblöcke B2 *und* B3 gilt.

Ein Datenflussproblem, wie das der Konstantenweitergabe, kann durch einen *Halbverband* und eine *Transferfunktion pro Grundblock* angegeben werden. Diese Spezifikation wird im Algorithmus der generischen Datenflussanalyse verwendet, um das Datenflussproblem zu lösen.

Der Halbverband des Datenflussproblems definiert eine *Grundmenge* (M) und eine *Meet*-Operation ($\cap$) darauf. Aus der Meet-Operation leitet sich wiederum durch $a \subseteq b \Leftrightarrow a \cap b = a$ eine *Halbordnung* $\subseteq$ auf der Grundmenge ab.

Bei der Konstantenweitergabe besteht die Grundmenge des Halbverban-

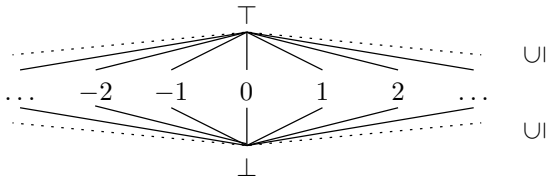

Abbildung 2.3: Halbverband für das Datenflussproblem "Konstantenweitergabe".

des, wie in Abbildung 2.3 dargestellt, aus allen Konstanten, die der Variablen zugewiesen werden können, sowie einem Top- ($\top$) und einem Bottom-Element ($\bot$).

Die Halbordnung $\subseteq$ ist für die Konstantenweitergabe so definiert, dass $x \subseteq \top$ und $\bot \subseteq x$ für alle $x \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ gilt. In der Abbildung 2.3 sind die Elemente demnach von oben nach unten angeordnet. Die Meet-Operation, liefert bezüglich dieser Halbordnung das größte Element der Menge, das nicht größer ist, als die beiden Operanden. Es gilt also zum Beispiel $2 \cap 1 = \bot$, $2 \cap \top = 2$, $2 \cap 2 = 2$ und $\top \cap \bot = \bot$.

Die Transferfunktion, die durch eine lokale Analyse aus einem Grundblock gewonnen werden kann, beschreibt die Auswirkungen des Grundblocks auf die propagierte Eigenschaft. Die Transferfunktion besitzt die Signatur $M \to M$ und muss zudem monoton steigen.

Bei der Konstantenweitergabe ist diese Funktion für eine Variable v und einen Grundblock B durch

$$f(e) = \begin{cases} e & \text{wenn in B nicht an } v \text{ zugewiesen wird} \\ x & \text{wenn in B als letztes die Konstante } x \text{ an } v \text{ zugewiesen wird} \\ \bot & \text{wenn in B als letztes ein variabler Wert an } v \text{ zugewiesen wird} \end{cases}$$

definiert.

Wurde ein Datenflussproblem wie beschrieben durch einen Halbverband und eine Transferfunktion definiert, berechnet der Algorithmus der generischen Datenflussanalyse für den Anfang und das Ende aller Grundblöcke die Eigenschaft. Dabei wird die Eigenschaft am Anfang eines Grundblocks i mit IN_i und am Ende mit OUT_i bezeichnet.

In einem Grundblock i hängt die OUT Eigenschaft OUT_i durch die Gleichung

$$OUT_i = f_i(IN_i)$$

von der IN Eigenschaft des Grundblocks ab. Dabei ist f_i die Transferfunktion des Grundblocks i. Weitere Gleichungen ergeben sich dadurch, dass IN_i durch

$$IN_i = \bigcap_{j \in \text{vorgänger}(i)} OUT_j$$

von den OUT Eigenschaften der Vorgängergrundblöcke des Grundblocks i abhängt. Dabei ist $\cap$ die Meet-Operation, die im Halbverband definiert wurde.

Wenn der Kontrollflussgraf, eine Schleife enthält, wie es bei Block B3 in Abbildung 2.2 der Fall ist, hängen die IN und OUT Eigenschaften wechselseitig

voneinander ab. Sie können also nicht durch einmalige Auswertung der Gleichungen in einer bestimmten Reihenfolge berechnet werden.

Das Gleichungssystem kann aber durch eine Fixpunktberechnung gelöst werden. Dabei werden die IN_i und OUT_i Eigenschaften mit dem $\top$-Element initialisiert. Nun werden wiederholt die IN_i und OUT_i durch eine Auswertung der Gleichungen neu berechnet. Ändern sich die IN und OUT Eigenschaften nicht mehr, terminiert das Verfahren. Die IN_i und OUT_i enthalten nun die Eigenschaften, die zu Beginn und am Ende des Grundblocks i gelten.

Wird der Variablen v im Block B1 die Konstante 3 und im Block B3 die Konstante 5 zugewiesen, leiten sich daraus die Transferfunktionen

$$f_1(e) = 3, f_2(e) = e, f_3(e) = 5, f_4(e) = e$$

ab. Durch die Fixpunktberechnung ergeben sich für IN_i und OUT_i somit die Eigenschaften:

$$IN_1 = \top, OUT_1 = IN_2 = OUT_2 = 3, OUT_3 = 5, IN_3 = IN_4 = OUT_4 = \bot$$

Bemerkenswert ist, dass v am Anfang des Grundblocks B3 keinen konstanten Wert hat, weil v am Ende der Vorgänger B1 und B3 zwei unterschiedliche Werte hat. Dennoch hat v am Ende des Grundblocks B3 durch die Zuweisung im Grundblock den konstanten Wert 5.

Bei der bisher beschriebenen Form der Datenflussanalyse wurde ein Meet-Halbverband verwendet. Es gibt aber auch Datenflussprobleme, die einen Join-Halbverband ($\cup$) verwenden. Bei der Datenflussanalyse werden dann die IN und OUT Eigenschaften mit $\bot$ initialisiert und ein *minimaler Fixpunkt* berechnet. Ein Datenflussproblem das einen Join-Halbverband verwendet wird auch als *Vereinigungsproblem* und ein Datenflussproblem das einen Meet-Halbverband verwendet wird auch als *Schnittproblem* bezeichnet.

Bei der bisherigen Datenflussanalyse wurden die Eigenschaften in Richtung des Kontrollflusses propagiert. Ein Datenflussproblem, das sich mit einer solchen Datenflussanalyse lösen lässt, wird als *Vorwärtsproblem* bezeichnet. Bei einem *Rückwärtsproblem* breitet sich eine Eigenschaft folglich gegen den Kontrollfluss aus.

Bei dem Datenflussproblem "lebendige Variablen" werden die Variablen bestimmt, deren Wert noch evtl. im weiteren Verlauf des Programms in einem Ausdruck verwendet wird. Es wird zum Beispiel bei der Berechnung der Lebensspannen von Variablen verwendet. Dieses Datenflussproblem ist ein Rückwärts-Vereinigungs-Problem. Es handelt sich um ein Rückwärtsproblem, weil eine Variable lebendig ist, wenn sie in den darauf folgenden Instruktionen lebendig ist. Das Datenflussproblem ist ein Vereinigungsproblem, weil eine Variable am Ende eines Grundblocks lebendig ist, wenn sie in *einem* der Nachfolgergrundblöcke lebendig ist.

Weiterhin wird die Datenflussanalyse zum Beispiel eingesetzt, um nicht initialisierte Variablen aufzufinden, gemeinsame Teilausdrücke zu ermitteln, nicht verwendete Zuweisungen zu finden oder Informationen zum Eliminieren unnötiger Kopieroperationen zu gewinnen. Die Datenflussanalyse ist also eine vielseitig einsetzbare Methode, die nicht nur für Analyseaufgaben im Übersetzer, sondern auch in Entwicklerwerkzeugen verwendet werden kann.

2.3 Registerzuteilung auf Funktionsebene (Chaitin)

Die Registerzuteilung ist, wie bereits in Abschnitt 2.1 beschrieben eine Phase des Übersetzungsvorgangs. In dieser Phase werden die virtuellen Register, die sich in den Operanden der Instruktionen befinden, durch physikalische ersetzt. Dazu wird jedem virtuellen Register ein physikalisches *zugeteilt*.

Für diese Aufgabe existieren verschiedene Verfahren, die sich in der Größe des betrachteten Programmausschnitts unterscheiden.

Bei dem Registerzuteilungsverfahren nach Sethi-Ullman findet die Registerzuteilung nur auf Ausdrucksebene statt. In den Registern werden also nur Zwischenergebnisse der Ausdrucksauswertung gespeichert. Damit die benötigte Anzahl von Registern minimal ist, werden Ergebnisse aus gemeinsamen Teilausdrücken nicht wiederverwendet, sondern mehrfach berechnet. Ein Wert wird also nur einmal in ein Register geschrieben und wieder ausgelesen. Die Registerzuteilung wird dabei nicht in einer eigenen Phase durchgeführt, sondern ist in die Code-Auswahl Phase integriert.

Bei dem Registerzuteilungsverfahren nach Belady wird bereits ein ganzer Grundblock für die Zuteilung betrachtet. Ein Wert kann daher im Grundblock in ein Register geschrieben und mehrfach wieder ausgelesen werden. Allerdings können Werte nicht über Grundblockgrenzen in Registern gehalten werden. Es sind also am Anfang und Ende eines Grundblocks Lade- und Schreiboperationen notwendig, um die Werte in den Hauptspeicher zu schreiben.

Im Gegensatz zu diesen *lokalen* Verfahren wird bei der Registerzuteilung nach Chaitin [Cha82] die *gesamte Funktion* bei der Zuteilung betrachtet. Durch diese *globale* Zuteilung ist es nun möglich, dass Werte über Grundblockgrenzen hinaus in Registern gehalten werden. Wenn eine Funktion nur wenige lokale Variablen und Zwischenergebnisse verwendet, können diese sogar permanent in Registern gehalten werden. Die Werte werden also niemals in den Hauptspeicher geschrieben. In den heutigen kommerziellen Übersetzern wird deshalb meist dieses Registerzuteilungsverfahren eingesetzt, um die Anzahl der Speicherzugriffe gering zu halten.

Die Aufgabe der Registerzuteilung wird bei Chaitins Verfahren auf eine Graffärbung reduziert. In dem Grafen sind die virtuellen Register die Knoten, zwischen denen eine Kante existiert, wenn sie an *einer Stelle* des Programms gleichzeitig lebendig sind. Die beiden virtuellen Register können in diesem Fall nicht demselben physikalischen Register zugeteilt werden. Die Kanten werden deshalb auch als *Konfliktkanten* und der Graf als *Konfliktgraf* bezeichnet.

Die beiden virtuellen Register v und w aus Abbildung 2.4 werden im Konfliktgrafen zum Beispiel durch eine solche Konfliktkante verbunden, weil sie unter anderem in Block B1 gleichzeitig lebendig sind.

Um festzustellen, in welchen Teilen einer Funktion ein virtuelles Register lebendig ist, wird eine Datenflussanalyse, wie am Ende von Abschnitt 2.2 beschrieben, durchgeführt.

In der Abbildung 2.4 kann man gut erkennen, wie sich die Lebensspannen von den Verwendungen der virtuellen Register gegen den Kontrollfluss ausbreiten, bis eine Definition des virtuellen Registers auftritt. Die Lebensspanne des virtuellen Registers w geht von der letzten Verwendung in Block B4 aus und endet in den Blöcken B1 und B3. Die Lebensspanne des Registers v geht sowohl von B2, als auch von einer Verwendung im Block B3 aus und endet in B1, wie auch B3. Eine Lebensspanne kann dabei auch, wie im Block B3,

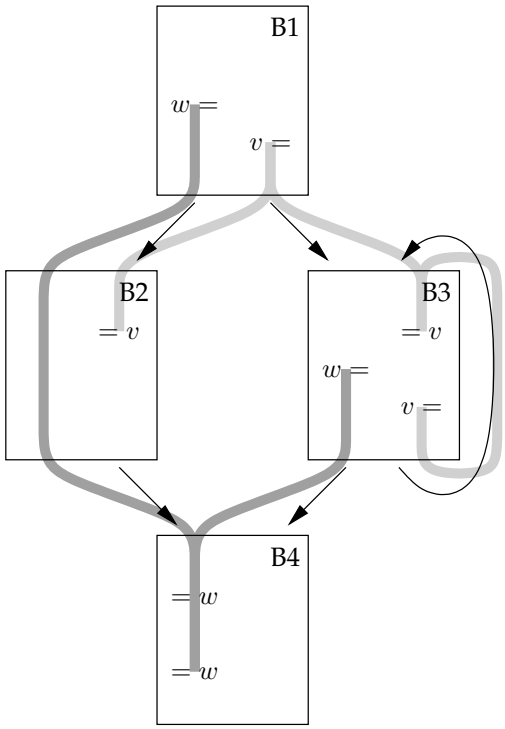

Abbildung 2.4: Kontrollflussgraf mit Lebensspannen (grau).

vom Anfang des Grundblocks über eine Kontrollflusskante bis zum Ende des Grundblocks verlaufen. Eine Lebensspanne ist also nicht zwingend ein Intervall, wenn nur ein Grundblock als Ausschnitt betrachtet wird.

Wurde der Konfliktgraf erzeugt, werden die Knoten mit den physikalischen Registern gefärbt. Es wird also jedem virtuellen Register ein physikalisches zugeordnet. Durch die Konfliktkanten wird dabei sichergestellt, dass zwei virtuelle Register, die an einer Stelle des Programms gleichzeitig jeweils einen Wert speichern müssen, nicht dem gleichen physikalischen Register zugeteilt werden.

Die optimale Färbung des Grafen, also die Färbung mit der minimalen Anzahl von physikalischen Registern, ist ein NP-Vollständiges Problem. In einem Übersetzer wird deshalb üblicherweise eine Heuristik verwendet, um den Grafen mit n Farben zu färben, wobei n die Anzahl der physikalischen Register ist.

Dabei werden die Knoten mit einem Grad kleiner n sukzessive aus dem Grafen entfernt, bis der Graf leer ist. Anschließend werden die Knoten wieder in umgekehrter Reihenfolge in den Grafen eingefügt und dabei gefärbt. Weil ein eingefügter Knoten maximal $n - 1$ gefärbte Nachbarn hat, kann er problemlos gefärbt werden. Tritt beim Entfernen der Knoten die Situation ein, dass alle Knoten einen Grad größer oder gleich n haben, wird der Knoten mit dem größten Grad entfernt und für das *Spilling* vorgemerkt. Dem Knoten wird in diesem Fall kein physikalisches Register zugeordnet. Stattdessen wird der Wert vor einem Lesezugriff aus dem Speicher in ein temporäres Register geladen und nach einem Schreibzugriff direkt wieder in den Speicher geschrieben.

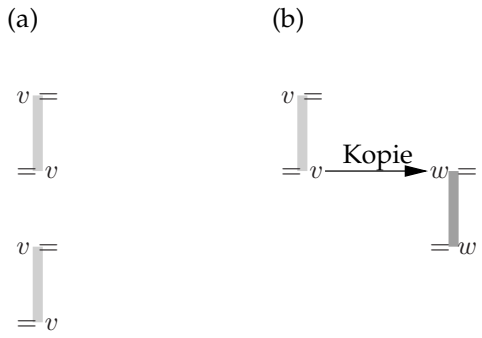

Abbildung 2.5: (a) Unterteilung einer Lebensspanne. (b) Beispiel für Coalescing.

Diese Technik wird als Spilling bezeichnet.

Eine Verbesserung des Verfahrens von Chaitin wurde von Briggs [BCT94] vorgestellt. Dabei wird nicht schon beim Entfernen der Knoten entschieden, ob ein virtuelles Register gespillt wird, sondern erst beim Einfügen. Nur wenn sich das einzufügende Register tatsächlich nicht färben lässt, weil alle Nachbarn mit einer der n Farben gefärbt sind, wird das Register gespillt.

Bei einer anderen Verbesserung des Verfahrens können einem virtuellen Register auch mehrere disjunkte Lebensspannen zugeordnet werden. Wird zum Beispiel ein virtuelles Register v wie in Abbildung 2.5a wiederverwendet, zerfällt die Lebensspanne in zwei Teile. Anstatt nun beide Teile als eine Lebensspanne anzusehen, kann jeder Teil als separate Lebensspanne betrachtet und gefärbt werden. Der Vorteil dabei ist, dass sich dadurch die Anzahl der Konfliktkanten im Konfliktgrafen verringern kann.

Durch ein Verfahren namens *coalescing* können nach der Bestimmung der Lebensspannen unnötige Kopieranweisungen eliminiert werden. Existieren, wie in Abbildung 2.5b, zwei disjunkte Lebensspannen der virtuellen Register v und w, die durch eine Kopieranweisung verbunden sind, können beide Lebensspannen vereinigt werden. Die beiden virtuellen Register werden somit *demselben* physikalischen Register zugeteilt. Die Kopieranweisung wird dadurch überflüssig und kann gelöscht werden.

2.4 Grundlagen der Registerarchitektur

In den vorherigen Abschnitten wurden allgemeine *Methoden* vorgestellt, die für die Konzeption eines Registerzuteilungsverfahrens benötigt werden. Daneben ist es für das Verfahren wichtig, welche Eigenschaften die rekonfigurierbaren Registerbänke haben. Deshalb werden nun zwei Papiere von Ravindran und Kiyohara vorgestellt, die Prozessorarchitekturen mit rekonfigurierbaren Registerbänken behandeln.

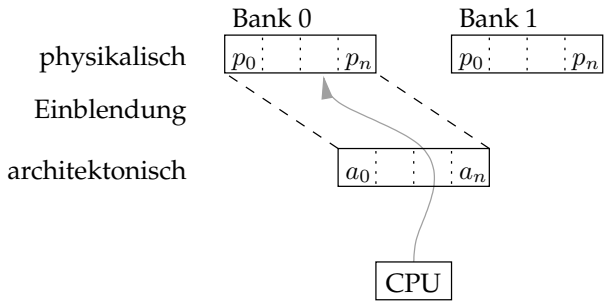

Abbildung 2.6: Registerarchitektur nach Ravindran.

2.4.1 Verwendung von mehreren Registerbänken (Ravindran)

In dem Papier von Ravindran [RSM+03] wird eine Einprozessor-Architektur vorgestellt, die wie in Abbildung 2.6 dargestellt, über mehrere Registerbänke verfügt.

Die Registerbänke bestehen dabei jeweils aus n gleichartigen Registern $p_0, \ldots, p_n$. Daneben verfügt der Prozessor über n adressierbare Register $a_0, \ldots, a_n$. Bevor der Prozessor auf ein Register einer Bank zugreifen kann, muss er die Bank mittels einer *Einblendungs-Instruktion*, wie in Abbildung 2.6 angedeutet, einblenden. Zu jedem Zeitpunkt kann dabei jedoch nur genau eine Bank eingeblendet werden. Als Konsequenz daraus kann eine Instruktion nur auf Register *derselben* Bank zugreifen. Weil dies die Registerzuteilung stark einschränken würde, wird in dem Papier noch eine *Bank-Kopier-Instruktion* eingeführt, die Registerinhalte zwischen den Bänken kopiert. Mit dieser Instruktion kann also zum Beispiel der Inhalt des Registers p_3 der Bank 0 direkt in das Register p_5 der Bank 1 kopiert werden.

Das Ziel des im Papier vorgestellten Registerzuteilungsverfahrens ist es nun, die Anzahl der Einblendungen und Bank-Kopien wie auch das Auftreten von Spillcode möglichst gering zu halten.

In einem ersten Schritt wird dafür zunächst ein Affinitätsgraf aufgebaut. Die Affinität zwischen zwei virtuellen Registern gibt dabei die Kosten[3] an, die entstehen, wenn die beiden Register unterschiedlichen Registerbänken zugeordnet werden. Mit Hilfe dieses Grafen werden nun die virtuellen Register auf die Registerbänke partitioniert. Die eigentliche Registerzuteilung wird für jede Registerbank und die dazugehörigen virtuellen Register separat durchgeführt. Zum Einsatz kommt dabei der globale Registerzuteilungsalgorithmus von Chaitin aus Abschnitt 2.3.

In der zweiten Phase werden noch Einblendungs- und Bank-Kopier-Instruktionen in den Code eingefügt, um die jeweils als nächstes benötigte Registerbank einzublenden und wenn nötig Operanden zwischen Registerbänken zu kopieren.

[3]Summe von potentiellen Bank-Kopien und Einblendungen

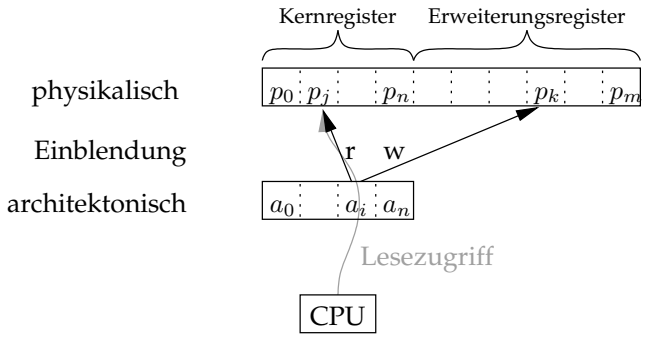

Abbildung 2.7: Registerarchitektur nach Kiyohara.

2.4.2 Verwendung von Registerverbindungen (Kiyohara)

Bei dem vom Kiyohara vorgestellten Verfahren [KMC+93] "Register Connection" (RC) greift der Prozessor nicht *direkt* auf die Register zu. Stattdessen adressiert der Prozessor wie in Abbildung 2.7 Registerrahmen $a_0, \ldots, a_n$, in die eine beliebige Auswahl der Register $p_0, \ldots, p_m$ *eingeblendet* werden kann. Die Registerrahmen werden in dem Papier als *architektonische Register* und die einblendbaren Register als *physikalische Register* bezeichnet. Diese Namensgebung wird auch durchgängig in dieser Arbeit verwendet.

Anstatt von "Einblendungen" ist in dem Papier jedoch von dem "Herstellen einer Verbindung zwischen einem architektonischen und einem physikalischen Register" die Rede. Daraus leitet sich auch die Bezeichnung "Register Connection" ab.

Die Einblendungen können für jedes architektonische Register getrennt für Lese- und Schreibzugriffe angegeben werden. In ein architektonisches Register a_i kann also, wie in Abbildung 2.7 dargestellt, zugleich für einen Lesezugriff das Register p_j und für einen Schreibzugriff das Register p_k eingeblendet werden.

Der Instruktionssatz der Architektur verfügt dazu über einen eigenen Maschinenbefehl zum Einblenden eines physikalischen Registers in ein gewünschtes architektonisches Register. Dieser Befehl wird im Folgenden als *Rekonfigurationsanweisung* bezeichnet.

Einblendungen können jedoch nicht nur auf diese Weise explizit vorgenommen werden, sondern auch implizit. Dabei wird nach einem Lese- bzw. Schreibzugriff auf das architektonische Register a_i das physikalische Register p_i in a_i eingeblendet. Weil durch implizite Einblendungen nur die physikalischen Register $p_0, \ldots, p_n$ in die architektonischen Register eingeblendet werden können, werden diese auch als *Kernregister* bezeichnet. Die übrigen physikalischen Register $p_{n+1}, \ldots, p_m$ werden als *Erweiterungsregister* bezeichnet.

Weil die Kernregister durch die impliziten Einblendungen in der Regel eingeblendet sind, muss der Übersetzer nur selten Rekonfigurationsanweisungen einfügen, um auf ein Kernregister zuzugreifen. Den Kernregistern werden deshalb virtuelle Register zugeteilt, auf die häufig zugegriffen wird. Ansonsten basiert das Registerzuteilungsverfahren auf dem Verfahren von Chaitin, das in Abschnitt 2.3 vorgestellt wurde.

Kiyohara stellt in seinem Papier mehrere Varianten des impliziten Einblen-

dens vor, die sich darin unterscheiden, *wie* sich die Lese- und Schreibeinblendungen durch einen Registerzugriff ändern. Darunter ist auch eine Variante, bei der *keine* impliziten Einblendungen vorgenommen werden. In diesem Fall braucht nicht zwischen Kernregistern und Erweiterungsregistern unterschieden werden, weil sie gleich behandelt werden.

In dem Papier von Kiyohara wird abschließend beschrieben, wie sich eine bestehende Prozessorarchitektur um das "Register Connection" Konzept erweitern lässt. Dabei war es wichtig, dass der erweiterte Prozessor rückwärtskompatibel ist. Werden einer bestehenden Architektur Erweiterungsregister und der Einblendungsmechanismus hinzugefügt, lassen sich noch weiterhin Programme der ursprünglichen Architektur ausführen, wenn zu Beginn der Programmausführung die Kernregister $p_0, \ldots, p_n$ eingeblendet sind. Das Programm nutzt dabei jedoch nicht die Erweiterungsregister und kann deshalb auch nicht effizienter auf der erweiterten Architektur ausgeführt werden.

Bei der Evaluierung des Verfahrens ist die Registerarchitektur laut dem Papier so schnell wie ein Prozessor, der *direkt* m Register adressieren kann. Allerdings wird bei der Simulation des Prozessors davon ausgegangen, dass eine Rekonfigurationsanweisung in 0 Taktzyklen ausgeführt werden kann. Aus dem Papier geht jedoch nicht klar hervor, wie diese Eigenschaft realisiert werden kann. Dabei müsste nämlich auch das Laden und Decodieren des Instruktionswortes in Nullzeit ausgeführt werden.

Kapitel 3

Registerarchitektur

Bevor die Algorithmen zur Registerzuteilung behandelt werden können, muss zunächst geklärt werden, wie die Zielarchitektur aufgebaut ist. Einige unscheinbare Eigenschaften der Registerarchitektur wirken sich nämlich gravierend auf das Registerzuteilungsverfahren aus. Es gibt bei der Registerarchitektur mehrere Eigenschaften, die variiert werden können und in der Regel unabhängig voneinander sind. Weil nicht jede Kombination der Eigenschaften behandelt werden kann, werden zunächst in Abschnitt 3.2 die Auswirkungen der Eigenschaften beschrieben und am Ende in Abschnitt 3.3 einige Beispiele erläutert. Zuvor werden noch die Begriffe definiert, die zur Beschreibung der Registerarchitektur verwendet werden.

3.1 Begriffsbildung

Um die Architektur eindeutig beschreiben zu können, werden im Folgenden einige Begriffe und Symbole eingeführt, die in der Arbeit durchgängig verwendet werden.

In Abbildung 3.1 werden die Begriffe einer Registerarchitektur zugeordnet. Die abgebildete Architektur weist viele der Eigenschaften auf, die in diesem Kapitel beschrieben werden. Sie verfügt über die Prozessoren CPU_0 und CPU_1, die jeweils über eine architektonische Registerbank für Lese- und Schreibzugriffe verfügen. Zudem verfügt jeder Prozessor über zwei physikalische Registerbänke k_0 und k_1, auf die auch die anderen Prozessoren zugreifen können.

Die Register der architektonischen und physikalischen Registerbänke sind in zwei gleich große Blöcke b_0 und b_1 unterteilt. Die Pfeile in der Abbildung stellen die blockweisen Einblendungen von physikalischen Registern in architektonische Register dar. Diese Einblendungen sind *eine* exemplarische Konfiguration der Registerbänke, wie sie zur Laufzeit auftreten kann.

Architektonische Register $\mathcal{R}_A = \{a_1, \ldots, a_n\}$ Architektonische Register werden vom Prozessor *direkt* adressiert, also in den Instruktionen als Operanden codiert. Sie stellen selber keinen Speicherplatz zur Verfügung, sondern sind Rahmen, in die physikalische Register eingeblendet werden. In den physikalischen Registern werden die Werte letztendlich gespeichert.

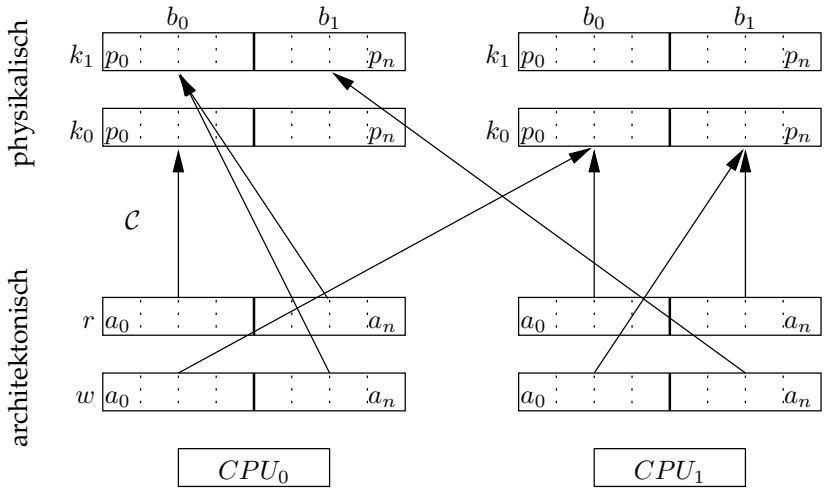

Abbildung 3.1: Mächtige Registerarchitektur mit *einer* zulässigen Einblendung.

Physikalische Register $\mathcal{R}_P = \{p_1, \ldots, p_n\}$ In physikalischen Registern werden Werte gespeichert. Sie werden jedoch nicht in den Operanden einer Instruktion codiert. Stattdessen wird ein physikalisches Register in ein architektonisches eingeblendet, das dann als Operand verwendet werden kann. Auf ein physikalisches Register wird also *indirekt* über ein architektonisches zugegriffen.

Physikalische Registerbänke $\mathcal{K} = \{k_0, k_1\}$ Jede CPU kann eine oder mehrere Bänke von physikalischen Registern haben. Die Anzahl der Register einer solchen Bank ist gleich der Anzahl der architektonischen Register. Auch wenn eine physikalische Registerbank einem Prozessor zugeordnet wird, können alle Prozessoren in gleicher Weise auf die Register der Bank zugreifen.

Registerblöcke $\mathcal{B}_A = \{b_0, b_1\}$ Die architektonischen, bzw. physikalischen Register einer Bank werden in disjunkte Registerblöcke einer festen Größe unterteilt. Der Grund dafür ist, dass Register nicht einzeln, sondern in ganzen Blöcken eingeblendet werden sollen. In der architektonischen Registerbank wird ein solcher Block als *architektonischer Block* und in der physikalischen Registerbank als *physikalischer Block* bezeichnet.

Physikalische Blöcke $\mathcal{B}_P = \{P_0, \ldots, P_7\}$ Die Menge der physikalischen Blöcke $\mathcal{B}_P$ besteht aus den Registerblöcken aller physikalischen Registerbänke. In der Abbildung 3.1 sind das also die Registerblöcke b_0, b_1 aus den physikalischen Registerbänken k_0, k_1 der beiden Prozessoren CPU_0, CPU_1. Diese 8 physikalischen Blöcke werden mit $P_0, \ldots, P_7$ bezeichnet.

Architektonische Blöcke $\mathcal{B}_A = \{A_0, A_1\}$ Analog zu den physikalischen Blöcken besteht die Menge der architektonischen Blöcke $\mathcal{B}_A$ aus den Registerblöcken aller architektonischen Bänke. Wenn zwischen Lese- und Schreibzugriffen nicht unterschieden wird, existiert nur eine architektonische Bank und die Menge $\mathcal{B}_A$ entspricht somit der Menge B.

Prozessoren $CPU = \{CPU_0, CPU_1\}$ Die Prozessoren sind gleichartig, besitzen also die gleichen Fähigkeiten. Zudem kann jeder Prozessor *jeden* physikalischen Block einblenden. Zum Beispiel kann der Prozessor CPU_0 den Block b_1 aus der Registerbank k_0 des Prozessors CPU_1 einblenden.

Zugriffsart $\mathcal{M} = \{r, w\}$ Die Zugriffsart gibt an, ob der Inhalt eines Registers durch einen Befehl gelesen oder geschrieben wird. Wird bei der Registerarchitektur in Abbildung 3.1 ein Wert von Prozessor CPU_0 in das architektonische Register a_0 geschrieben, wird dieser in dem physikalischen Register p_0 der Bank k_0 des Prozessors CPU_1 gespeichert. Wird hingegen lesend auf das architektonische Register a_0 zugegriffen, wird der Wert aus p_0 der Bank k_0 des Prozessors CPU_0 gelesen.

Zulässige Einblendungen $\mathcal{C} : \mathcal{B} \times \mathcal{M} \times CPU \rightarrow \mathcal{B} \times \mathcal{K} \times CPU$ Die Menge der zulässigen Einblendungen in der Registerarchitektur wird durch eine Signatur C beschrieben. Eine Funktion zu dieser Signatur ist dabei *eine* Konfiguration der rekonfigurierbaren Registerbänke. Die Funktion bildet einen architektonischen Block auf einen physikalischen Block ab. In der Registerarchitektur aus Abbildung 3.1 lautet die Signatur $C : \mathcal{B} \times \mathcal{M} \times CPU \rightarrow \mathcal{B} \times \mathcal{K} \times CPU$. Dabei wird ein Registerblock $\mathcal{B}$ aus einer der beiden architektonischen Bänke $\mathcal{M}$ eines Prozessors CPU abgebildet auf einen Registerblock $\mathcal{B}$ der beiden physikalischen Bänke $\mathcal{K}$ eines Prozessor CPU.

Soll hingegen ausgedrückt werden, dass der Registerblock aus der architektonischen Bank mit dem der physikalischen Bank übereinstimmen muss, wird im rechten Teil der Signatur die Menge $\mathcal{B}$ weggelassen. Es wird dann ein architektonischer Block nur auf eine physikalische Registerbank abgebildet und implizit angenommen, dass der Registerblock mit dem gleichen Index eingeblendet wird.

Ports einer Registerbank Die Anzahl der Ports einer Registerbank gibt an, wie viele Register der Bank gleichzeitig gelesen, bzw. geschrieben werden können. Wenn nur ein Prozessor auf eine Registerbank zugreift und in einem Befehl maximal ein Register geschrieben und zwei gelesen werden, wird eine Registerbank mit zwei Lese- und einem Schreibport benötigt.

3.2 Eigenschaften

In diesem Abschnitt wird erläutert, welche Eigenschaften bei einer Registerarchitektur mit rekonfigurierbaren Registerbänken variiert werden können, und welchen Einfluss die Eigenschaften auf das Registerzuteilungsverfahren haben.

Die ersten drei Eigenschaften, die in der folgenden Tabelle aufgelistet werden, legen die Kapazitäten der Architektur fest. Die beiden darauf folgenden Eigenschaften gehören dem Einblendungsmechanismus an.

Abschnitt	Eigenschaft
3.2.1	Anzahl der Prozessoren
3.2.2	Anzahl der Registerbänke
3.2.3	Anzahl der Registerblöcke
3.2.4	Zulässige Einblendungen
3.2.5	Unterscheidung zwischen Lese/Schreibzugriffen
3.2.6	Anzahl der Ports einer Registerbank
3.2.7	Operanden in mehreren Registern

3.2.1 Anzahl der Prozessoren

Das Registerzuteilungsverfahren soll nicht nur auf einen Prozessor ausgelegt sein, sondern sich auch auf eine *VLIW-Architektur* mit *gemeinsamen physikalischen Registern* übertragen lassen.

Jeder Prozessor verfügt dabei über eine *eigene* Bank von architektonischen Registern, wodurch die Einblendungen eines Prozessors P_i unabhängig von den Einblendungen eines Prozessors P_j hergestellt werden können. Der Zugriff aller Prozessoren über eine *einzige* architektonische Bank auf die physikalischen Register führt zu vielen Problemen und wird deshalb nicht weiter betrachtet.

Im Gegensatz zu den architektonischen Registern werden die physikalischen Register von allen Prozessoren gemeinsam genutzt. Das Kopieren von Registerinhalten zwischen zwei Prozessoren über den Hauptspeicher oder durch spezielle Kommunikationsbefehle ist deshalb nicht notwendig. Ein weiterer Vorteil der gemeinsamen Register ist, dass sie besser ausgelastet werden, weil ein Prozessor Register nutzen kann, die ein anderer Prozessor nicht benötigt.

Weil die Prozessoren eine Menge von physikalischen Registern gemeinsam verwenden, muss auch ein gemeinsamer Konfliktgraf erstellt werden. Dafür ist es notwendig zu wissen, welche Instruktionen parallel ausgeführt werden. Diese Information ist bei einer VLIW-Maschine direkt gegeben, weil festgelegt ist, welche Instruktionen parallel ausgeführt werden.

3.2.2 Anzahl der Registerbänke

Je größer die Anzahl der Registerbänke ist, desto mehr Register können durch den Prozessor verwendet werden und umso seltener muss Spillcode eingefügt werden. Ansonsten hat die Anzahl der Registerbänke keinen weiteren Einfluss auf die Registerzuteilung.

3.2.3 Anzahl der Registerblöcke

Wie sich die Anzahl der Registerblöcke auf die Leistungsfähigkeit der Registerarchitektur auswirkt, ist nicht unmittelbar ersichtlich. Wählt man die maximale Anzahl von Registerblöcken, sodass sich in einem Block nur ein einziges Register befindet, erhält man eine Registerarchitektur, die der von Kiyohara aus 2.4.2 ähnlich ist. Die physikalischen Register können dann zwar flexibel eingeblendet und verdrängt werden, allerdings sind auch mehrere Rekonfigurationsanweisungen notwendig, um mehrere physikalische Register einzublenden.

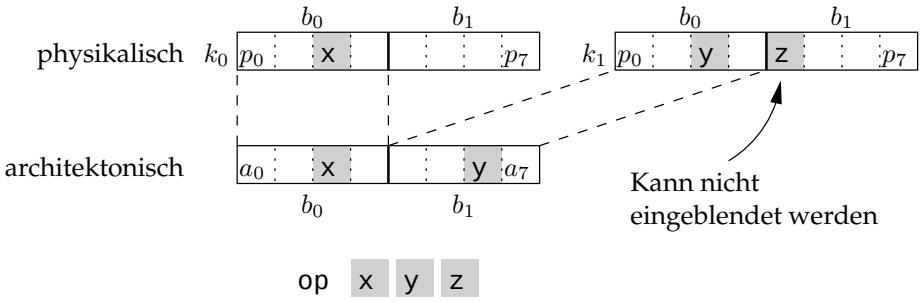

Abbildung 3.2: Einblendungsproblem bei zu wenigen architektonischen Blöcken.

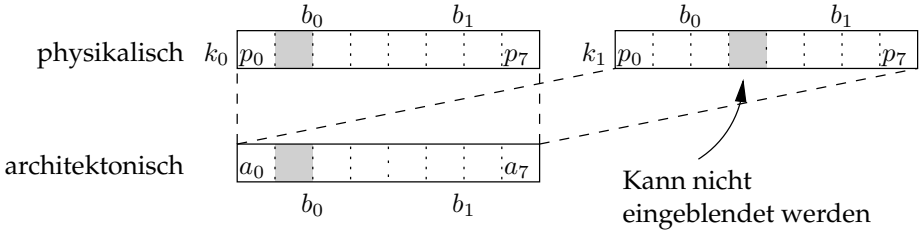

Abbildung 3.3: Einblendungsproblem bei nur einem architektonischen Block.

Wählt man hingegen eine geringe Anzahl von Registerblöcken, können zwar mit einer Rekonfigurationsanweisung gleich mehrere physikalische Register eingeblendet werden, allerdings büßt man dafür die Flexibilität bei der Wahl der einzublendenden physikalischen Register ein.

Wenn die Anzahl der Registerblöcke sogar geringer ist, als die maximale Anzahl der Operanden einer Instruktion, kann es vorkommen, dass nicht alle Operanden gleichzeitig eingeblendet werden können. In diesem Fall wäre das Kopieren von Registerinhalten zwischen physikalischen Blöcken erforderlich. In Abbildung 3.2 ist ein Beispiel zu sehen, bei dem die drei Operanden `x`, `y`, `z` nicht gleichzeitig eingeblendet werden können, weil die architektonische Bank nur aus zwei Blöcken besteht.

Besteht die architektonische Registerbank, wie in Abbildung 3.3 dargestellt, sogar nur aus einem einzigen Block, kann wie bei der Registerarchitektur von Ravindran (2.4.1) nur zwischen den physikalischen Bänken umgeschaltet werden. Es können daher keine Registerinhalte zwischen den Bänken mit einer `mov` Instruktion kopiert werden, weil dazu die beiden Bänke gleichzeitig eingeblendet sein müssten.

3.2.4 Zulässige Einblendungen

Welche physikalischen Blöcke in einen architektonischen Block eingeblendet werden können, wird durch eine Signatur C festgelegt. In diesem Abschnitt werden zwei solche Signaturen vorgestellt.

Bei der ersten Signatur

$$\mathcal{C} : \mathcal{B} \to \mathcal{K}$$

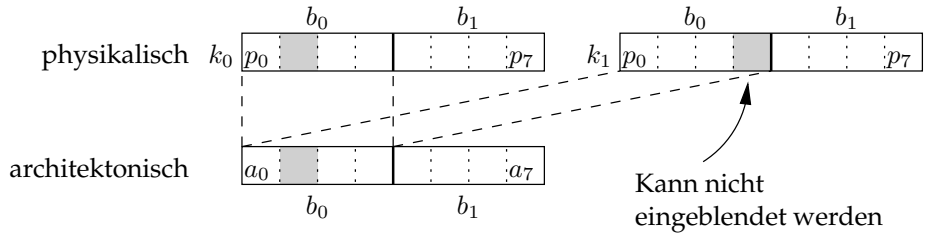

Abbildung 3.4: Einblendungsproblem bei physikalischen Blöcken mit gleichem Index.

kann in einen architektonischen Block $\mathcal{B}$ ein physikalischer Block mit dem gleichen Index aus einer Bank $\mathcal{K}$ eingeblendet werden. Unterschiedliche Registerblöcke mit dem gleichen Index stehen dabei im *Konflikt* miteinander, weil sie nicht gleichzeitig eingeblendet werden können.

In dem Beispiel in Abbildung 3.4 können die beiden b_0 Blöcke aus den Registerbänken k_0 und k_1 also nur in den Block b_0 der architektonischen Registerbank eingeblendet werden. Die beiden grau hinterlegten physikalischen Register können somit nicht gleichzeitig eingeblendet werden. Daraus folgt auch, dass ein physikalisches Register stets in das architektonische Register mit dem *gleichen Index* eingeblendet wird.

Bei dem Registerzuteilungsverfahren müssen nun die Folgen bedacht werden, wenn zwei virtuelle Register ...

1. ... demselben physikalischen Block zugeteilt werden. Die beiden Register können dann als Operanden einer gemeinsamen Instruktion verwendet werden. Dafür muss maximal eine Einblendung vorgenommen werden.

2. ... Blöcken mit unterschiedlichen Indizes zugeteilt werden. Die beiden Register können dann in einer Instruktion verwendet werden und es müssen zwei physikalische Blöcke eingeblendet werden.

3. ... dem gleichen Block von zwei unterschiedlichen Registerbänken oder Prozessoren zugeteilt werden. Die beiden Register können dann *nicht* in einer gemeinsamen Instruktion verwendet werden, weil nicht beide Register gleichzeitig eingeblendet werden können.

Im Folgenden wird diese Registerarchitektur als eine mit *eingeschränkten Einblendungen* bezeichnet.

Bei der zweiten Signatur

$$\mathcal{C} : \mathcal{B} \rightarrow \mathcal{B} \times \mathcal{K}$$

kann in einen physikalischen Block $\mathcal{B}$ ein beliebiger physikalischer Block $\mathcal{B} \times \mathcal{K}$ eingeblendet werden. Als Vorteil ergibt sich daraus, dass physikalische Blöcke nicht mehr wie zuvor im Konflikt miteinander stehen. Ein weiterer Vorteil ist, dass beim Einblenden eines physikalischen Blocks nicht festgelegt ist, welcher architektonische Block verdrängt wird. Es kann also zum Beispiel der physikalische Block verdrängt werden, der am spätesten wieder verwendet wird. Dafür wird ein physikalisches Register aber nicht mehr zwingend in ein architektonisches Register mit dem gleichen Index eingeblendet.

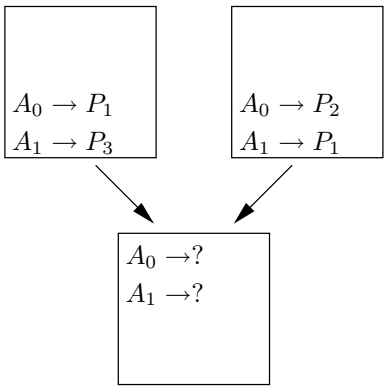

Abbildung 3.5: Problem bei der Weiterverwendung von Einblendungen an Grundblockgrenzen.

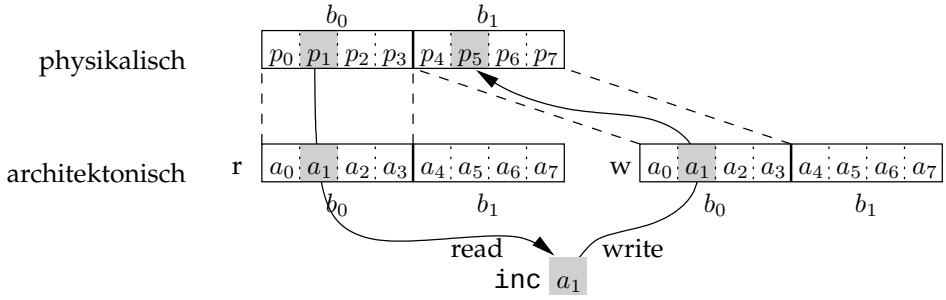

Abbildung 3.6: Veränderte Semantik von Lese-/Schreiboperanden.

Hat ein Grundblock wie in Abbildung 3.5 zwei Vorgänger, kann es zudem vorkommen, dass in beiden ein physikalischen Block P_1 am Ende eingeblendet ist, allerdings in unterschiedlichen architektonischen Blöcken. Am Anfang des Grundblocks muss daher durch eine Rekonfigurationsanweisung P_1 erneut eingeblendet werden, weil nicht bekannt ist ob P_1 in A_1 oder in A_2 eingeblendet ist. Diese Registerarchitektur wird im Folgenden als eine mit *beliebigen Einblendungen* bezeichnet.

3.2.5 Unterscheidung zwischen Lese-/Schreibzugriffen

Eine Registerarchitektur kann bei den Einblendungen auch zwischen lesenden und schreibenden Zugriffen unterscheiden. Dabei existiert eine architektonische Bank für lesende und eine für schreibende Zugriffe.

In Abbildung 3.6 sind die Zugriffe einer `inc` Instruktion zu sehen, die den Inhalt eines Registers um 1 erhöht. Der Operand a_1 ist in diesem Fall also ein Lese-/Schreiboperand. Für das Lesen des Registerinhaltes wird nun die Lese-Einblendung der linken architektonischen Bank verwendet und für das Schreiben die Schreib-Einblendung der rechten architektonischen Bank. Weil in die beiden architektonischen Blöcke jedoch unterschiedliche physikalische Blöcke eingeblendet sind, wird dabei das Register p_1 gelesen und der erhöhte Wert in

das Register p_5 geschrieben. Durch diesen Effekt kann somit eine, wenn auch leicht eingeschränkte, 3-Adress-Addition auf einer 2-Adress-Maschine durchgeführt werden. Wenn der Effekt allerdings nicht gewünscht ist, müssen die Lese- und Schreibeinblendungen identisch sein. Es stellt sich dann die Frage, wie häufig solche Operationen auftreten und ob durch die Unterscheidung die Ausführungsgeschwindigkeit noch erhöht wird.

Prinzipiell kann sich die Unterscheidung wie eine Verdoppelung der architektonischen Register auswirken, weil jeweils $|\mathcal{B}|$ physikalische Blöcke für Lese- und Schreibzugriffe eingeblendet werden können.

Auf der anderen Seite müssen jedoch auch zwei Rekonfigurationsanweisungen durchgeführt werden, um ein nicht eingeblendetes physikalisches Register zu erhöhen.

Bei Registerarchitekturen mit nur einem Block (Abbildung 3.3) oder eingeschränkten Einblendungen und einem Konflikt kann die Unterscheidung genutzt werden, um einen Wert zwischen zwei Blöcken durch eine `mov` Instruktion zu kopieren.

In die Signatur C, die die Menge zulässiger Konfigurationen beschreibt, wird für die Lese-/Schreibunterscheidung zusätzlich die Menge $\mathcal{M}$ aufgenommen.

$$\mathcal{C} : \mathcal{B} \times \mathcal{M} \rightarrow \mathcal{B} \times \mathcal{K} \times CPU$$

Der eingeblendete physikalische Block $\mathcal{B} \times \mathcal{K}$ hängt nun also von dem architektonischen Block $\mathcal{B}$ und der Art des Zugriffs $\mathcal{M}$ ab.

3.2.6 Anzahl der Ports einer Registerbank

Die Anzahl der Ports einer Registerbank gibt an, auf wie viele Register in der Bank gleichzeitig zugegriffen werden kann. Wenn die Anzahl der Ports geringer ist als die Anzahl gleichzeitiger Zugriffe aller Prozessoren, können Konflikte auftreten.

Wenn diese Konflikte nicht durch die Hardware behandelt werden, muss der Übersetzer sicherstellen, dass in einem Takt nicht zu viele Zugriffe auf die gleiche Registerbank erfolgen. Das kann zum Beispiel durch ein spezielles Instruction-Scheduling erfolgen oder durch das Einfügen von Warteinstruktionen. Damit solche Konflikte nicht entstehen können, müsste die Registerbank bei einer n-Adress-Maschine $|CPU| * n$ viele Ports besitzten.

3.2.7 Operanden in mehreren Registern

Bei einigen Prozessoren befindet sich ein Operand nicht nur in einem einzigen, sondern in mehrere physikalischen Registern. Zum Beispiel werden zwei aufeinander folgende 16-Bit-Register für einen 32-Bit-Operanden verwendet, indem nur das erste physikalische Register in der Instruktion codiert wird. Geschrieben, bzw. gelesen werden jedoch beide Register, wobei das eine das höherwertige 16-Bit-Wort und das andere das niederwertige 16-Bit-Wort enthält.

Um dieser Architektureigenschaft Rechnung zu tragen, werden so genannte *logische Register* eingeführt, die aus einem oder mehreren physikalischen Registern bestehen. Anstelle eines physikalischen Registers wird einem Registeroperanden nun stets ein logisches Register zugeteilt.

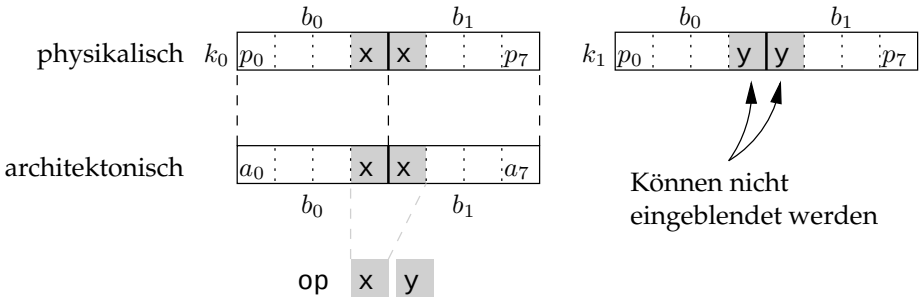

Abbildung 3.7: Einblendungsproblem durch Operanden, die sich in mehreren physikalischen Blöcken befinden.

Bei logischen Registern, die mehr als ein physikalisches Register repräsentieren, wird für das Registerzuteilungsverfahren gefordert, dass diese physikalischen Register *im gleichen physikalischen Block* liegen. Ohne diese Einschränkung könnte es sonst vorkommen, dass wie in Abbildung 3.7 für einen Operanden mehrere physikalische Blöcke eingeblendet werden müssten. In dem Beispiel müssten also vier physikalische Blöcke eingeblendet werden, um auf beide Operanden zugreifen zu können.

Von dieser Einschränkung sind ohnehin nur wenige Prozessoren betroffen, weil sich bei den meisten ein Operand stets in nur einem physikalischen Register befindet.

3.3 Exemplarische Registerarchitekturen

In diesem Abschnitt werden einige Architekturausprägungen aufgeführt, um die Eigenschaften aus dem vorherigen Abschnitt zu verdeutlichen. Begonnen wird dabei mit einer einfachen Ausprägung, die in den darauf folgenden Beispielen schrittweise erweitert wird.

3.3.1 Mehrere Registerbänke

Bei dieser Registerarchitektur bestehen die Registerbänke, wie in Abbildung 3.8 dargestellt, aus nur einem Registerblock. Es kann also nur Block b_0 von der Registerbank k_0 oder k_1 eingeblendet werden. Die zulässigen Einblendungen werden in der Abbildung durch die beiden Pfeile dargestellt. Das Einblenden kann in diesem Fall auch als Umschalten zwischen den Registerbänken angesehen werden. Diese Ausprägung ist der Registerarchitektur von Ravindran aus 2.4.1 ähnlich. Weitere Konsequenzen, die sich daraus ergeben, wurden bereits in Abschnitt 3.2.3 genannt.

In der folgenden Spezifikation der abgebildeten Registerarchitektur werden die Symbole aus der Begriffsbildung auf Seite 17 verwendet. Dabei ist $\mathcal{R}_A$ die Menge der architektonischen Register, $\mathcal{R}_P$ die der physikalischen, $\mathcal{B}$ die Menge der Registerblöcke pro Registerbank, $\mathcal{K}$ die Menge der Registerbänke, CPU die Menge der Prozessoren und $\mathcal{C}$ die Signatur der zulässigen Einblen-

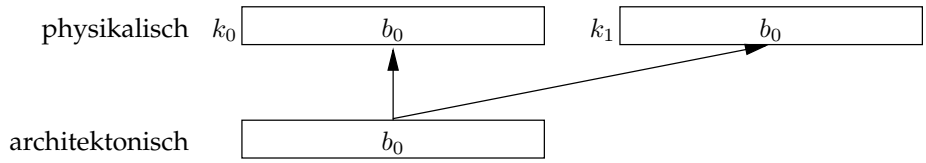

Abbildung 3.8: Registerarchitektur mit nur einem Registerblock.

dungen.

$$\mathcal{R}_A = \{a_0, \ldots, a_{15}\}, \mathcal{R}_P = \{p_0, \ldots, p_{15}\},$$
$$\mathcal{B} = \{b_0\}, \mathcal{K} = \{k_0, k_1\}, CPU = \{CPU_0\},$$
$$\mathcal{C} : \emptyset \rightarrow \mathcal{K}$$

Die Funktionen zur Signatur $\mathcal{C}$ sind konstant. Die Menge der zulässigen Einblendungen entspricht daher der Menge der Registerbänke $\mathcal{K}$.

3.3.2 Eingeschränkte Einblendungen

Im Vergleich zum vorherigen Beispiel verfügt diese Ausprägung über mehrere Registerblöcke. Die Einblendungen sind jedoch eingeschränkt. Ein Block einer physikalischen Registerbank kann daher nur in den architektonischen Block mit dem gleichen Index eingeblendet werden. In den architektonischen Block b_0 können also, wie durch die Pfeile angedeutet, nur die b_0-Blöcke der beiden physikalischen Bänke eingeblendet werden.

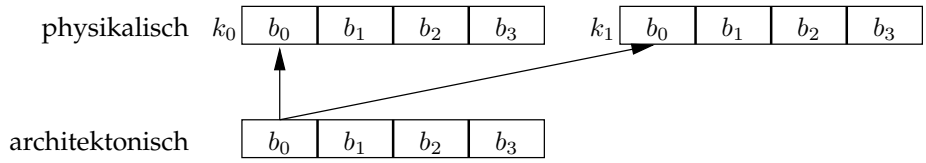

Abbildung 3.9: Registerarchitektur mit eingeschränkten Einblendungen.

Die Probleme, die sich daraus ergeben, wurden in Abschnitt 3.2.4 beschrieben. Diese Registerarchitektur ist zwar mächtiger, als die vorherige, dafür sind mehrere Rekonfigurationsanweisungen notwendig, um das Umschalten einer Bank nachzubilden.

Die Registerarchitektur aus der Abbildung ist wie folgt spezifiziert:

$$\mathcal{R}_A = \{a_0, \ldots, a_{15}\}, \mathcal{R}_P = \{p_0, \ldots, p_{15}\},$$
$$\mathcal{B} = \{b_0, b_1, b_2, b_3\}, \mathcal{K} = \{k_0, k_1\}, CPU = \{CPU_0\},$$
$$\mathcal{C} : \mathcal{B} \rightarrow \mathcal{K}$$

Dabei wird durch die Signatur $\mathcal{C}$ beschrieben, dass in einen architektonischen Block $\mathcal{B}$ nur ein physikalischer Block mit dem gleichen Index aus einer Registerbank $\mathcal{K}$ eingeblendet werden kann.

3.3.3 Beliebige Einblendungen

Bei dieser Ausprägung können, nicht nur eingeschränkte, sondern beliebige Einblendungen vorgenommen werden. Es treten also keine Konflikte mehr

zwischen physikalischen Blöcken auf, weil $|\mathcal{B}_A|$ beliebige physikalische Blöcke gleichzeitig eingeblendet werden können.

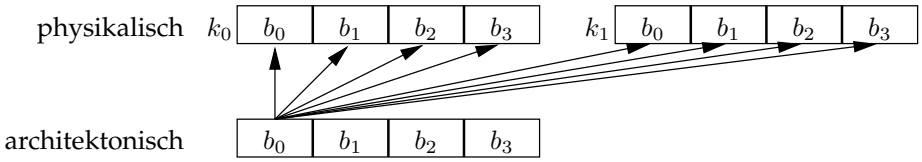

Abbildung 3.10: Registerarchitektur mit beliebigen Einblendungen.

Die zulässigen Einblendungen für den architektonischen Block b_0 werden in Abbildung 3.10 durch die Pfeile angegeben.

$$\mathcal{R}_A = \{a_0, \ldots, a_{15}\}, \mathcal{R}_P = \{p_0, \ldots, p_{15}\},$$
$$\mathcal{B} = \{b_0, b_1, b_2, b_3\}, \mathcal{K} = \{k_0, k_1\}, CPU = \{CPU_0\},$$
$$\mathcal{C} : \mathcal{B} \rightarrow \mathcal{B} \times \mathcal{K}$$

Die Signatur $\mathcal{C}$ in der Definition beschreibt, dass in einen architektonischen Block $\mathcal{B}$ ein beliebiger physikalischer Block $\mathcal{B}$ aus einer Registerbank $\mathcal{K}$ eingeblendet werden kann.

Diese Registerarchitektur ist mächtiger, als die mit eingeschränkten Einblendungen. Dafür muss in der Rekonfigurationsanweisung angegeben werden, in welchen architektonischen Block ein physikalischer eingeblendet werden soll, wodurch das Instruktionsformat der Rekonfigurationsanweisung größer wird.

3.3.4 Unterscheidung zwischen Lese- und Schreibzugriffen

Bei dieser Registerarchitektur wird zusätzlich zwischen Lese- und Schreibzugriffen unterschieden. Wie bei dem vorherigen Beispiel können beliebige Einblendungen vorgenommen werden. Es ist aber ebenso möglich, bei der Registerarchitektur eingeschränkte Einblendungen zu verwenden.

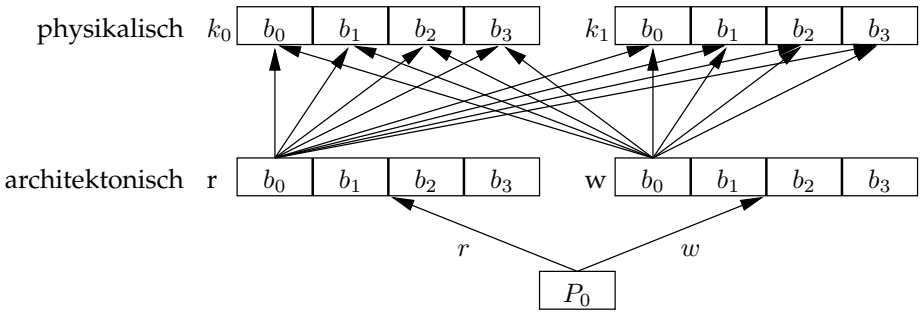

Abbildung 3.11: Registerarchitektur mit Unterscheidung zwischen Lese- und Schreibzugriffen.

Für die Lese- und Schreib-Einblendungen ist in Abbildung 3.11 jeweils eine architektonische Bank dargestellt. Für Lesezugriffe werden die Einblendungen

der linken architektonischen Bank verwendet und für Schreibzugriffe die der rechten. Für einen Block b_0 können also zwei Einblendungen hergestellt werden, wobei für jede ein beliebiger physikalischer Block eingeblendet werden kann.

Die Effekte, die sich durch die Unterscheidung ergeben, wurden bereits in Abschnitt 3.2.5 beschrieben.

Die abgebildete Registerarchitektur kann wie folgt spezifiziert werden:

$$\mathcal{R}_A = \{a_0, \ldots, a_{15}\}, \mathcal{R}_P = \{p_0, \ldots, p_{15}\},$$
$$\mathcal{B} = \{b_0, b_1, b_2, b_3\}, \mathcal{K} = \{k_0\}, CPU = \{CPU_0\}, \mathcal{M} = \{r, w\},$$
$$\mathcal{C} : \mathcal{B} \times \mathcal{M} \rightarrow \mathcal{B} \times \mathcal{K}$$

Die Signatur $\mathcal{C}$ wurde, im Vergleich zur vorherigen Ausprägung, um die Menge $\mathcal{M}$ erweitert. Dadurch wird ausgedrückt, dass die Einblendung nun auch von der Zugriffsart abhängt.

3.3.5 Mehrere Prozessoren

Das folgende Beispiel beschreibt eine Registerarchitektur mit mehreren Prozessoren, bei der jeder Prozessor über eine physikalische Registerbank k_0 verfügt. Eine solche Registerbank kann von allen Prozessoren in gleicher Weise genutzt werden.

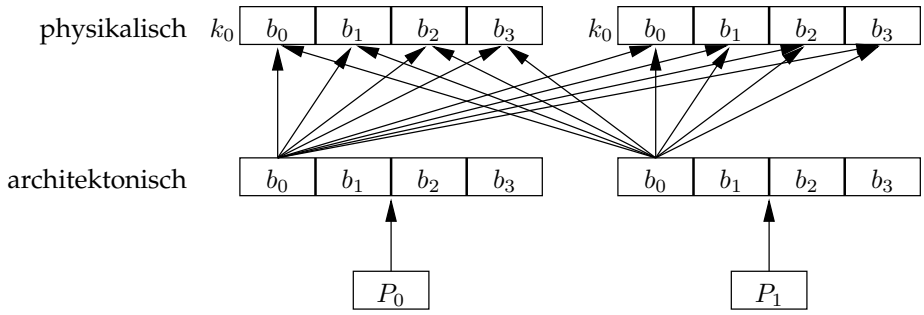

Abbildung 3.12: Registerarchitektur mit mehreren Prozessoren.

In der Abbildung 3.12 wird dies durch die Pfeile verdeutlicht, die für beide Prozessoren angeben, welche physikalischen Blöcke in den architektonischen Block b_0 eingeblendet werden können.

Die Registerarchitektur kann wie folgt spezifiziert werden:

$$\mathcal{R}_A = \{a_0, \ldots, a_{15}\}, \mathcal{R}_P = \{p_0, \ldots, p_{15}\},$$
$$\mathcal{B} = \{b_0, b_1, b_2, b_3\}, \mathcal{K} = \{k_0\}, \mathcal{C}PU = \{CPU_0, CPU_1\},$$
$$\mathcal{C} : \mathcal{B} \times CPU \rightarrow \mathcal{B} \times \mathcal{K} \times CPU$$

Die Signatur $\mathcal{C}$ gibt dabei die Menge der zulässigen Einblendungen *aller* Prozessoren an. Es kann also in einen architektonischen Block $\mathcal{B}$ eines Prozessors CPU ein physikalischer Block $\mathcal{B}$ einer beliebigen Bank $\mathcal{K}$ eines beliebigen Prozessors CPU eingeblendet werden.

Es ist auch möglich, diese Registerarchitektur mit der Unterscheidung zwischen Lese- und Schreibzugriffen zu kombinieren. Weil die Darstellung der Registerarchitektur dadurch aber recht unübersichtlich wird, wurde darauf verzichtet.

3.4 Betrachtete Registerarchitektur

Für das Registerzuteilungsverfahren, das in Kapitel 5 behandelt wird, wird die Registerarchitektur mit beliebigen Einblendungen aus Abschnitt 3.3.3 betrachtet.

Die Anzahl der physikalischen Registerbänke wird dabei jedoch nicht festgelegt. Damit die Rekonfiguration beim Einprozessor-Fall Sinn ergibt, sollten allerdings mindestens zwei physikalische Registerbänke vorhanden sein.

Die Probleme beim gleichzeitigen Einblenden aller Registeroperanden einer Instruktion aus Abschnitt 3.2.3 werden dadurch verhindert, dass für eine N-Adress-Maschine mindestens N architektonische Blöcke gefordert werden.

Durch das Registerzuteilungsverfahren werden die Einblendungen zur *Übersetzungszeit* vollständig festgelegt. Der Instruktionssatz der Architektur bietet deshalb nur eine Rekonfigurationsanweisung mit unmittelbaren Operanden. Das heißt, dass der architektonische und der physikalische Block jeweils als *Wert* in der Instruktion codiert ist. Der physikalische, bzw. architektonische Block kann also nicht in einem Register angegeben werden. Das Speichern einer Einblendung in einem Register ist daher ebenfalls nicht nötig.

Kapitel 4

Analyse der n-Lebendigkeit

In diesem Kapitel wird eine Datenflussanalyse vorgestellt, die eine Aussage über das zukünftige Programmverhalten liefert. Dabei werden für jede Stelle des Programms jeweils Informationen über die nächsten Registerzugriffe ermittelt.

Diese Informationen werden im Registerzuteilungsverfahren benötigt, um festzulegen, welche virtuellen Register einem gemeinsamen physikalischen Block zugeteilt werden. Des Weiteren werden sie auch beim Einfügen von Rekonfigurationsanweisungen benötigt, um Einblendungen *vorzeitig* herstellen zu können. Die Eigenschaft[1], die die Datenflussanalyse berechnet, ist die Menge der n verschiedenen Register, auf die als nächstes zugegriffen wird. Werden also nach einer Programmstelle s die Registerzugriffe $1, 2, 1, 3, 2, 4, 5$ durchgeführt und ist $n = 4$, hat die Eigenschaft den Wert $1, 2, 3, 4$.

Ist ein Register in der Eigenschaft enthalten, wird es im Folgenden als *n-lebendig* bezeichnet. Die Register $1, 2, 3, 4$ sind in dem Beispiel also 4-lebendig. Ein Register ist an einer Programmstelle n-lebendig, wenn es sich unter den n Registern befindet, auf die als nächstes zugegriffen wird. Die Eigenschaft, die durch die Datenflussanalyse ermittelt wird, ist somit die Menge der n-lebendigen Register. Dieser Lebendigkeitsbegriff ist nicht mit dem aus der Datenflussanalyse "lebendige Variablen" zu verwechseln.

In dem obigen Beispiel wurde angegeben, dass an der Position s als nächstes auf die Register $1, 2, 1, 3, 2, 4, 5$ zugegriffen wird. Um eine solche Aussage *zur Übersetzungszeit* treffen zu können, darf der Kontrollfluss zwischen s und den Zugriffen nicht verzweigen[2]. Folglich kann die Liste nur die Zugriffe zwischen s und dem nächsten bedingten Sprung enthalten. Somit kann die Menge der n-lebendigen Register ebenfalls nur die Register beinhalten, auf die noch vor dem nächsten Sprung zugegriffen wird.

Um dennoch die Registerzugriffe nach einem Sprung zu berücksichtigen, wird nun statt einer Liste ein *Baum* verwendet.

In der Abbildung 4.1a ist ein Teil eines Kontrollflussgrafen mit Zugriffen auf die Register $1, \ldots, 5$ dargestellt. Der Baum in 4.1b besteht aus den beiden Zugriffsfolgen, die von der Position s in 4.1a ausgehen können. Dieser Baum

[1] Der Begriff "Eigenschaft" wird hier im Sinne des Abschnitts 2.2 verwendet.

[2] Eine Verzweigung ist nicht mit dem Ende eines Grundblocks gleichzusetzen, weil ein Grundblock auch den Ausgangsgrad 1 haben kann.

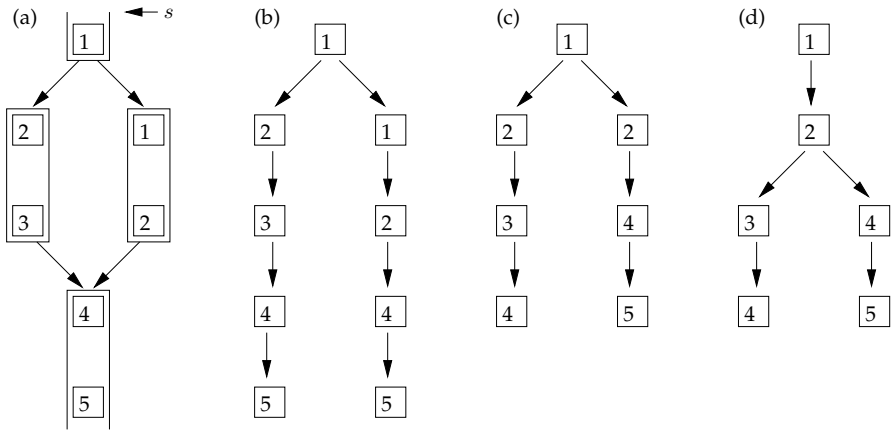

Abbildung 4.1: (a) Kontrollflussgraf (b) Baum der Zugriffsfolgen (c) Zugriffsbaum (d) Reduzierter Zugriffsbaum

ersetzt die Liste aus dem vorherigen Beispiel. Den Baum der n-lebendigen Register in Abbildung 4.1c erhält man, indem nur die jeweils ersten Zugriffe auf ein Register berücksichtigt werden und der Baum auf die Tiefe $n = 4$ begrenzt wird. Um den Baum zu vereinfachen, werden abschließend die beiden Knoten des Registers 2 zusammengefasst, wodurch sich der Baum aus 4.1d ergibt.

Dieser letzte Baum ist die Eigenschaft, die durch die Datenflussanalyse berechnet wird. Der Baum wird im Folgenden als *Zugriffsbaum* bezeichnet.

Dem Zugriffsbaum aus 4.1d lässt sich entnehmen, dass die Register $1, 2, 4$ echt 4-lebendig sind, weil sie auf jedem einfachen Pfad liegen, der von der Wurzel des Baumes ausgeht. Über die Register $3, 5$ kann keine *sichere* Aussage gemacht werden. Wenn die Sprungwahrscheinlichkeit in dem Kontrollflussgrafen bei der Verzweigung jeweils $\frac{1}{2}$ ist, kann jedoch die Aussage getätigt werden, dass die Register $3, 5$ an der Stelle s jeweils mit der Wahrscheinlichkeit $\frac{1}{2}$ auch 4-lebendig sind. Aus diesem Grund werden die Knoten des Baums zusätzlich mit Wahrscheinlichkeiten markiert. Die Sprungwahrscheinlichkeiten werden in dieser Arbeit als gleichverteilt angenommen. Die Datenflussanalyse ist aber auch für beliebige Wahrscheinlichkeitsverteilungen ausgelegt.

Die beiden Knoten des Registers 4 werden in 4.1d zwar mit der Wahrscheinlichkeit $\frac{1}{2}$ markiert, in der Summe ergibt sich jedoch die Wahrscheinlichkeit 1, das sichere Ereignis. Das Register 4 wird deshalb, wie auch die Register $1, 2$, als *echt n-lebendig* bezeichnet. Die beiden Register $3, 5$ sind hingegen nicht echt n-lebendig, sondern nur n-lebendig.

4.1 Einordnung des Datenflussproblems

Durch das Datenflussproblem wird ein Zugriffsbaum berechnet, der die n-lebendigen Register enthält. Die Knoten des Baumes sind dabei mit Wahrscheinlichkeiten markiert, die in den folgenden Abschnitten aber zunächst noch nicht beachtet werden.

Das Datenflussproblem gehört zur Klasse der Rückwärts-Vereinigungs Pro-

bleme. Die Zugriffsbäume werden also gegen den Kontrollfluss propagiert und an Blockgrenzen durch eine Vereinigungsfunktion vereinigt. Weiterhin wird durch die Datenflussanalyse ein minimaler Fixpunkt berechnet, weshalb die Eigenschaften mit $\perp$ initialisiert werden und die Transferfunktion monoton steigen muss.

In dem nun folgenden Abschnitt werden die Eigenschaften des Zugriffsbaums genauer Beschrieben. In den beiden darauf folgenden Abschnitten wird auf die Transferfunktion und die Vereinigungsfunktion der Datenflussanalyse eingegangen.

4.1.1 Zugriffsbäume

Ein Zugriffsbaum enthält, wie bereits erwähnt, alle Register, die n-lebendig sind. Die Anzahl der Register kann dabei auch größer sein als n, wenn einige Register nicht echt n-lebendig sind.

Jeder einfache Pfad von der Wurzel zu einem Blatt entspricht einer Reihenfolge in der zur Laufzeit auf die Register zugegriffen werden kann. Der Pfad enthält allerdings nur die jeweils ersten Zugriffe auf die Register. Ein Register kann also höchstens einmal auf einem solchen Pfad vorkommen. Diese Eigenschaft ist wichtig, damit der Zugriffsbaum nicht tiefer sein braucht als n.

Besitzt ein Knoten im Zugriffsbaum mehrere Kinder des gleichen Registers, werden diese wie im vorherigen Beispiel 4.1d zusammengefasst, wodurch sich ein *reduzierter Zugriffsbaum* ergibt. Dieser Schritt wird durchgeführt, um die Anzahl der Knoten zu verringern. Dadurch benötigt der Baum weniger Speicherplatz und Algorithmen können effizienter ausgeführt werden.

Zusammenfassend gelten für den Zugriffsbaum folgende Sätze:

Satz 1 *In einem Zugriffsbaum kommt auf jedem Pfad von der Wurzel zu einem Blatt jedes Register höchstens einmal vor.*

Satz 2 *Ein Zugriffsbaum hat maximal die Tiefe n.*

Satz 3 *In einem reduzierten Zugriffsbaum hat ein Knoten nie zwei Kindknoten des gleichen Registers.*

Damit keine Bäume mit mehreren Wurzeln auftreten können und um die Operationen auf dem Baum zu vereinfachen, wird jedem Baum eine *künstliche Wurzel* vorangestellt. Diese Wurzel wird mit keinem Register assoziiert und hat auch sonst keine weitere Bedeutung.

4.1.2 Transferfunktion

Die Transferfunktion bildet die OUT Eigenschaft eines Grundblocks auf die IN Eigenschaft des Grundblocks ab. Es wird also der OUT Zugriffsbaum wie in Abbildung 4.2 auf den IN Zugriffsbaum abgebildet.

Dabei wird dem OUT Zugriffsbaum die Liste der Register, auf die im Grundblock zugegriffen wird, vorangestellt. Diese Liste enthält die Register in der Reihenfolge der Zugriffe und wird im Folgenden als *GEN Liste* bezeichnet. Die GEN Liste beschreibt die Auswirkung des Grundblocks auf die propagierte Eigenschaft.

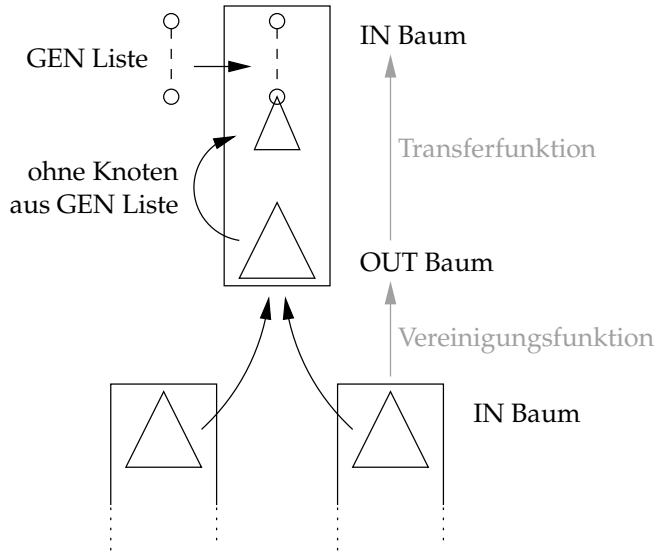

Abbildung 4.2: Propagierung der Zugriffsbäume.

Damit kein Knoten auf einem einfachen Pfad von der Wurzel zu einem Blatt doppelt vorkommt, werden beim Zusammenfügen der GEN Liste und des OUT Baums alle Knoten aus dem OUT Baum entfernt, die auch in der GEN Liste enthalten sind. Zudem werden noch alle Knoten entfernt, deren Tiefe größer ist als n. Der resultierende IN Baum erfüllt damit die Sätze 1,2 und enthält nun alle Register, die am Anfang des Grundblocks n-lebendig sind.

4.1.3 Vereinigungsfunktion

Die Vereinigungsfunktion berechnet die OUT Eigenschaft eines Grundblocks durch die Vereinigung der IN Eigenschaften der Nachfolger-Grundblöcke. In Abbildung 4.2 wird angedeutet, wie die beiden IN Bäume der Nachfolger-Grundblöcke zu dem OUT Baum vereinigt werden.

Dabei werden die beiden künstlichen Wurzeln der IN Bäume vereinigt. Weil die resultierende künstliche Wurzel dadurch zwei Kindknoten des gleichen Registers haben kann, muss der Baum gegebenenfalls noch reduziert werden.

4.2 Baumoperationen

Die Transferfunktion und die Vereinigungsfunktion führen Operationen auf den Zugriffsbäumen durch. Die Transferfunktion *stellt einem Baum eine Liste von Knoten voran*, die zudem aus dem Baum *gelöscht* werden müssen. Die Vereinigungsfunktion *vereinigt* mehrere Bäume, was wiederum eine anschließende *Reduktion* notwendig machen kann.

Diese vier hervorgehobenen Operationen werden nun definiert. Das Voranstellen einer Liste von Knoten wird dabei durch das sukzessive *voranstellen eines Knotens* ersetzt.

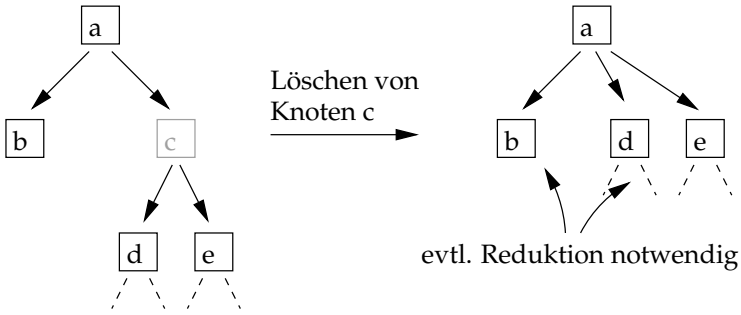

Abbildung 4.3: Löschen von Knoten c aus dem Baum.

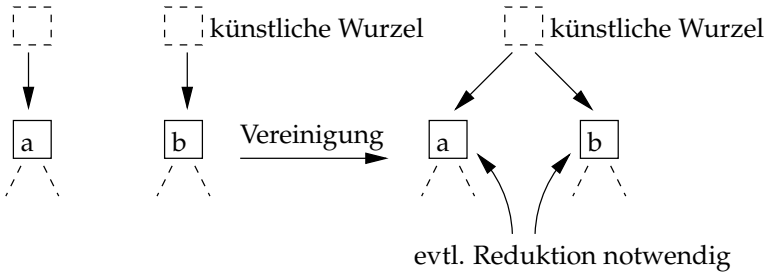

Abbildung 4.4: Vereinigen von zwei Bäumen.

4.2.1 Löschen eines Knotens

Beim Löschen eines Knotens wird dieser durch seine Kinder ersetzt. In dem Beispiel in Abbildung 4.3 wird der Knoten c gelöscht und deshalb durch seine Kindknoten d und e ersetzt. In diesem veränderten Baum kann der Vater des gelöschten Knotens nun gleiche Kinder haben, sodass evtl. eine Reduktion notwendig ist. Gilt in dem Beispiel $b = d$, müssen diese beiden Knoten durch eine Reduktion vereinigt werden.

Die Unterbäume des gelöschten Knotens rücken bei der Ersetzung eine Ebene nach oben, wodurch sich die Tiefe der Blätter in diesen Teilbäumen um 1 verringert. Eine anschließende Reduktion hat keinen weiteren Einfluss auf die Tiefe der Blätter.

4.2.2 Vereinigen von Bäumen

Beim Vereinigen von zwei Bäumen werden die künstlichen Wurzeln wie in Abbildung 4.4 zusammengefasst. Dadurch kann evtl. eine anschließende Reduktion notwendig sein. Gilt in dem Beispiel $a = b$, müssen diese beiden Knoten durch eine Reduktion verschmolzen werden.

4.2.3 Reduzieren von Bäumen

Besitzt ein Knoten zwei gleiche Kinder, werden diese wie in Abbildung 4.5 durch die Reduktion zu einem Knoten zusammengefasst. Dabei vereinigen

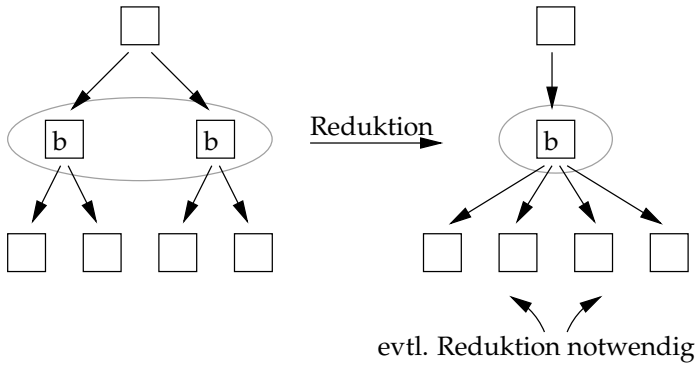

Abbildung 4.5: Reduzieren von zwei gleichen Kindknoten.

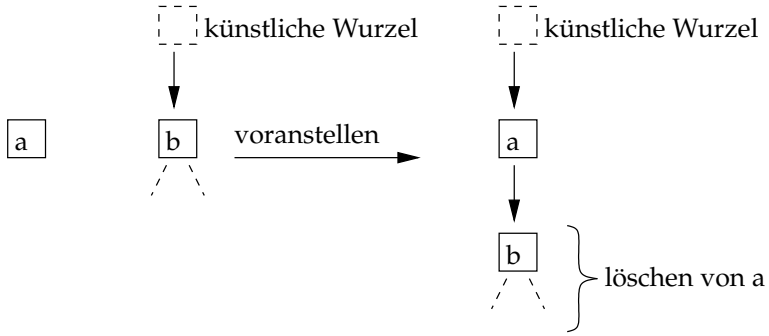

Abbildung 4.6: Voranstellen eines Knotens.

sich die Mengen der Kinder der zusammengefassten Knoten, sodass evtl. weitere Reduktionen rekursiv durchgeführt werden müssen.

4.2.4 Voranstellen von Knoten

Beim Voranstellen wird ein Knoten in einem Baum zwischen der künstlichen Wurzel und deren Kindern eingefügt. Weiterhin werden dabei aus dem Baum alle Knoten gelöscht, die dem gleichen Register angehören, wie der eingefügte Knoten. In dem Beispiel 4.5 wird der Baum der Knoten a vorangestellt.

Das Voranstellen einer Liste von Knoten kann durch das wiederholte Voranstellen der Elemente der Liste, beginnend mit dem letzten, nachgebildet werden.

4.3 Erweiterung um Wahrscheinlichkeiten

In der Einleitung dieses Kapitels wurde bereits erwähnt, dass durch die Verwendung von Wahrscheinlichkeiten auch eine Aussage über die Register gemacht werden kann, auf die nicht mit Sicherheit zugegriffen wird. Dabei wird angegeben, wie wahrscheinlich ein Zugriff auf das Register ist. Zu diesem Zweck wird nun jeder Knoten des Zugriffsbaums mit der Wahrscheinlichkeit

eines Zugriffs markiert. Der künstlichen Wurzel wird dabei eine Wahrscheinlichkeit von 1 zugeordnet.

Im Folgenden wird angenommen, dass die Wahrscheinlichkeit, bei einem Sprung ein Ziel X anzuspringen, gleichverteilt ist. Es können aber auch Profiling Daten oder die Schleifenheuristik aus 5.1.1 für die Verteilung verwendet werden.

Weil innerhalb eines Grundblocks keine Sprünge vorkommen, ist die Wahrscheinlichkeit eines Zugriffs auf ein Register der GEN Liste gleich 1. Die Auswirkung der Baumoperationen auf die Wahrscheinlichkeiten sind in Abbildung 4.7 zu sehen. Beim Löschen ändert sich dabei nichts an den mitgeführten Wahrscheinlichkeiten. Beim Vereinigen werden die Wahrscheinlichkeiten in den beiden Bäumen jeweils mit $\frac{1}{2}$ multipliziert, bzw. beim Vereinigen von n Bäumen mit $\frac{1}{n}$. Werden zwei Knoten reduziert, addieren sich die Wahrscheinlichkeiten der beiden Knoten.

4.3.1 Sätze über Wahrscheinlichkeiten im Baum

Ist die Summe der Wahrscheinlichkeiten im Baum gleich n, wird der Baum als *voll besetzt* bezeichnet. Ein Baum ist nicht voll besetzt, wenn von der Programmstelle s ein Pfad zum Ende des Programms existiert, auf dem auf weniger als n verschiedene Register zugegriffen wird. Ein *voll besetzter* Baum ist nicht mit einem *vollständigen* Baum zu verwechseln.

Satz 4 *In jedem Baum ist die Wahrscheinlichkeit eines Knoten nicht kleiner als die Summe der Wahrscheinlichkeiten der Kinder. Ist der Baum voll besetzt, ist die Wahrscheinlichkeit gleich der Summe. Der Satz gilt auch für die künstliche Wurzel.*

$$\forall x : p_x \geq \sum_{c \in children(x)} p_c$$

Es wird nun gezeigt, dass der Satz im leeren Baum und nach jeder der vier Baumoperationen gilt. Weil bei der Datenflussanalyse jeder Zugriffsbaum durch die wiederholte Anwendung der vier Baumoperationen aus dem leeren Baum entsteht, gilt der Satz somit für jeden Zugriffsbaum.

Leerer Baum Der leere Baum besteht nur aus der künstlichen Wurzel, somit existieren keine Kinder und der Satz gilt.

Löschen Beim Löschen wird ein Knoten durch seine Kinder ersetzt. Die Summe der Wahrscheinlichkeiten der Kinder ist dabei nach der Induktionsvoraussetzung nicht größer als die Wahrscheinlichkeit des gelöschten Knotens. Damit ist die Wahrscheinlichkeit des Vaters des gelöschten Knotens weiterhin nicht kleiner als die Summe der Wahrscheinlichkeiten seiner Kinder.

Vereinigen Beim Vereinigen werden alle Knoten eines Baumes mit demselben Faktor f multipliziert. Es gilt daher

$$p_x \geq \sum_{c \in children(x)} p_c \Rightarrow f * p_x \geq \sum_{c \in children(x)} f * p_c$$

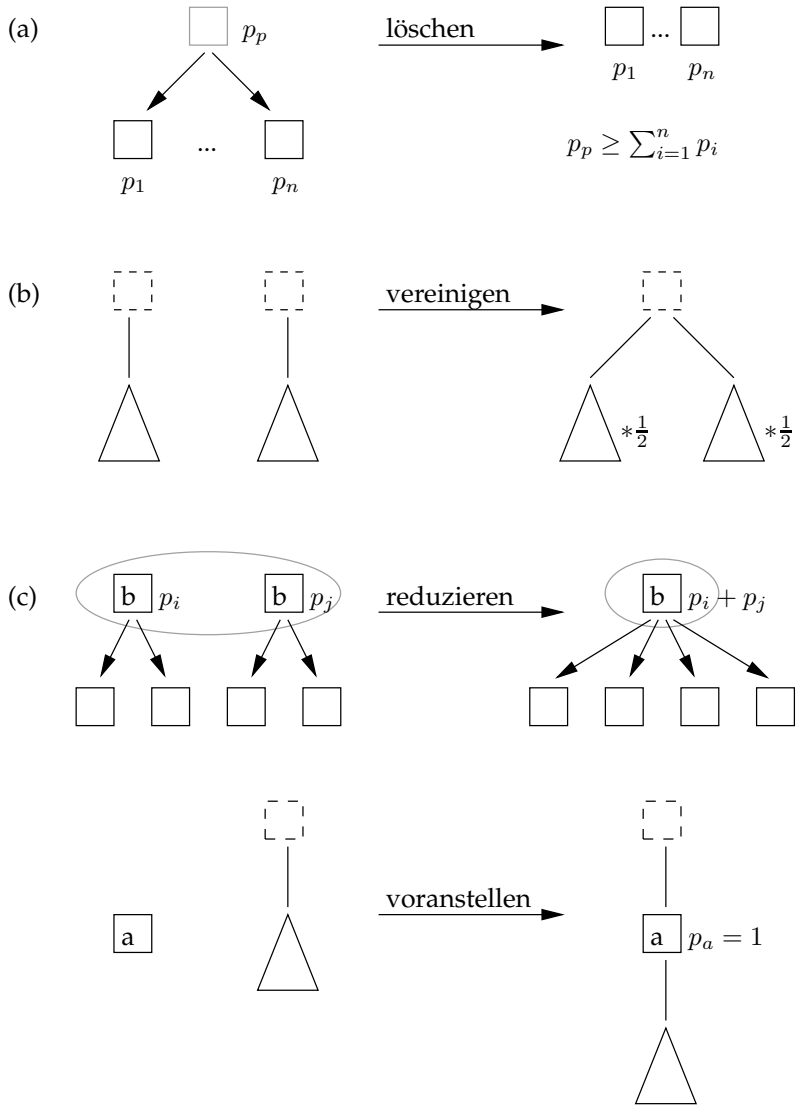

Abbildung 4.7: Erweiterung der Baumoperationen um Wahrscheinlichkeiten.

Reduzieren Gilt der Satz in zwei zu reduzierenden Teilbäumen mit den Wurzeln x und y, gilt er auch in dem reduzierten Teilbaum:

$$\left(p_x \geq \sum_{c \in children(x)} p_c\right) \wedge \left(p_y \geq \sum_{c \in children(y)} p_c\right)$$
$$\Rightarrow \quad p_x + p_y \geq \left(\sum_{c \in children(x)} p_c\right) + \left(\sum_{c \in children(y)} p_c\right)$$
$$\Rightarrow \quad p_x + p_y \geq \left(\sum_{c \in children(x) \cup children(y)} p_c\right)$$

Voranstellen Die künstliche Wurzel w eines Baumes und der vorangestellte Knoten a haben stets die Wahrscheinlichkeit 1.

$$p_w \geq \sum_{c \in children(w)} p_c \wedge p_w = p_a \Rightarrow p_a \geq \sum_{c \in children(a)} p_c$$

Satz 5 *Die Summe der Wahrscheinlichkeiten auf einer Ebene des Baums ist nicht größer als 1. Die Summe ist gleich 1, wenn der Baum voll besetzt ist.*

$$\forall d : \sum_{x, depth(x)=d} p_x \leq 1$$

Nach Satz **4** ist die Summe der Wahrscheinlichkeiten auf Ebene $n+1$ nicht größer ist als die Summe der Wahrscheinlichkeiten auf Ebene n. Weil die Wahrscheinlichkeit der künstlichen Wurzel gleich 1 ist, gilt Satz 5 somit für alle Ebenen.

Satz 6 *Die Summe der Wahrscheinlichkeiten der Knoten eines Registers ist nicht größer als 1.*

$$\forall r : \left(\sum_{x, register(x)=r} p_x\right) \leq 1$$

Die Summe q_w der Wahrscheinlichkeiten der Knoten eines Registers r in einem Teilbaum ist stets kleiner als die Wahrscheinlichkeit p_w der Wurzel des Teilbaums. Wenn w ein Knoten des Registers r ist, kommt nach Satz 1 in dem Teilbaum von w ansonsten kein Knoten von r mehr vor. Es gilt also $q_w = p_w$

Ansonsten gilt durch Induktion:

$$p_w \geq \sum_{i=0}^{n} p_i \wedge p_i \geq q_i \wedge q_w = \sum_{i=0}^{n} q_i$$
$$\Rightarrow \quad q_w = \sum_{i=0}^{n} q_i \leq \sum_{i=0}^{n} p_i \leq p_w$$
$$\Rightarrow \quad q_w \leq p_w$$

Der Induktionsanfang ist für die Blätter durch

$$q_b = \begin{cases} p_b & \text{wenn b ein Knoten des Registers r ist} \\ 0 & \text{sonst} \end{cases}$$

gegeben.

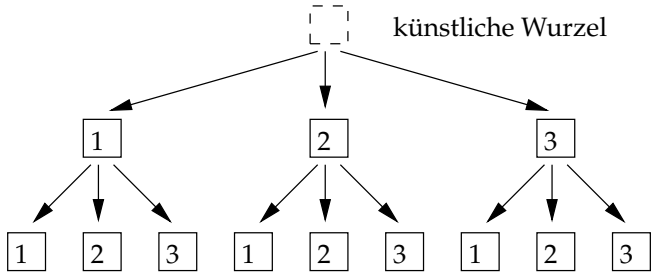

Abbildung 4.8: Beispiel eines vollständigen Zugriffsbaums.

4.4 Terminierung der Datenflussanalyse

Um die Terminierung der Datenflussanalyse zu gewährleisten, werden bei der generischen Datenflussanalyse zwei Forderungen gestellt. Die erste Forderung ist, dass die Transferfunktion monoton steigen muss. Es muss also

$$a \subseteq b \Rightarrow f(a) \subseteq f(b)$$

gelten. Die zweite Bedingung ist, dass die Vereinigungsfunktion die Eigenschaften einer Join-Operation haben muss. Wie sich später zeigen wird, erfüllt die hier definierte Vereinigungsfunktion *nicht* diese Bedingung. Trotzdem kann gezeigt werden, dass die Datenflussanalyse konvergiert.

Im nächsten Abschnitt wird aus beweistechnischen Gründen der Zugriffsbaum zu einem *vollständigen Zugriffsbaum* erweitert.

4.4.1 Vollständiger Zugriffsbaum

Ein vollständiger Zugriffsbaum ist der größte Zugriffsbaum, der bei einer Datenflussanalyse auftreten kann. Ein Zugriffsbaum ist also stets ein Teilbaum des vollständigen Zugriffsbaums.

Der vollständige Zugriffsbaum ist ein k-närer Baum der Tiefe n, wobei k die Anzahl der Register angibt, die in der Datenflussanalyse auftreten. Jeder innere Knoten des vollständigen Zugriffsbaumes hat für jedes Register je einen Kindknoten. Ein Beispiel eines vollständigen Zugriffsbaums für die Register $\{1, 2, 3\}$ und $n = 2$ ist in Abbildung 4.8 dargestellt. Die Wahrscheinlichkeiten der Knoten wurden in der Abbildung weggelassen.

Ein Zugriffsbaum kann in einen gleichwertigen, vollständigen Zugriffsbaum überführt werden. Dazu werden die fehlenden Knoten hinzugefügt und mit der Wahrscheinlichkeit 0 markiert. Die Wahrscheinlichkeit 0, das unmögliche Ereignis, drückt dabei aus, dass auf den Knoten nicht zugegriffen wird. Ein solcher Knoten kann also als neutral bezeichnet werden. Die Semantik des Zugriffsbaumes ändert sich durch diese Vervollständigung also nicht. Dafür werden zahlreiche Sonderfälle in den Beweisen vermieden, weil nicht unterschieden werden muss, ob ein Knoten existiert.

In den folgenden Beweisen müssen die Knoten eindeutig benannt werden können. Zu diesem Zweck wird die *Position* eines Knotens eingeführt, die den Knoten identifiziert. Jeder Knoten im Baum hat also eine eindeutige Position. Die Position eines Knotens wird durch den einfachen Pfad von der Wurzel zum

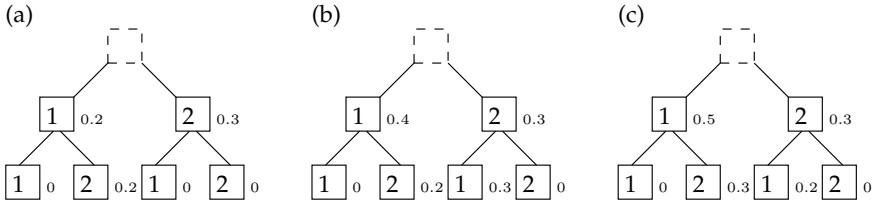

Abbildung 4.9: Beispiele für vollständige Zugriffsbäume.

Knoten angegeben. Eine Position x hat also die Form $x = (\rho_1, \ldots, \rho_d)$. Dabei ist ρ_d der Knoten, dessen Position angegeben wird. Der Index d gibt somit die Tiefe des Knotens an. Die Menge aller Knotenpositionen wird im Folgenden mit POS bezeichnet und ist durch

$$POS := \{Register^i | i \leq n\}$$

gegeben.

In der Abbildung 4.8 haben die Knoten unter der künstlichen Wurzel die Positionen (1),(2),(3) und die Knoten auf der darunterliegenden Ebene die Positionen $(1,1)$, $(1,2)$, $(1,3)$, $(2,1)$, usw.

4.4.2 Halbordnung der Zugriffsbäume

Als Grundlage für die folgenden Beweise muss zunächst die Halbordnung auf den Zugriffsbäumen definiert werden. Dafür werden vollständige Zugriffsbäume betrachtet, damit die verglichenen Bäume die gleiche Struktur haben. Die Halbordnung ordnet die Zugriffsbäume nach den Wahrscheinlichkeiten der Knoten.

Ein vollständiger Zugriffsbaum a ist dabei kleiner als ein vollständiger Zugriffsbaum b, wenn für alle Knotenpositionen die Wahrscheinlichkeit in a nicht größer ist, als die in b. Formal wird dies durch

$$a \subseteq b \Leftrightarrow \forall x \in POS : p_a(x) \leq p_b(x)$$

ausgedrückt. Dabei ist $p_a(x)$ die Wahrscheinlichkeit des Knotens an der Position x im Baum a.

In Abbildung 4.9 sind drei vollständige Zugriffsbäume für die Registermenge $\{1,2\}$ und $n = 2$ gegeben. Die Wahrscheinlichkeiten stehen rechts neben den Knoten. Der Baum (a) ist gemäß der Halbordnung nun kleiner als der Baum (b) und der Baum (c). Der Baum (b) ist jedoch nicht kleiner als der Baum (c), weil der Knoten $(2,1)$ in (b) eine Wahrscheinlichkeit von 0.3 hat, in (c) jedoch nur eine von 0.2. Es gilt also nicht $p_b(2,1) \leq p_c(2,1)$. Umgekehrt ist aber auch der Baum (b) nicht größer als der Baum (c), weil die Wahrscheinlichkeit des Knotens (1) und des Knotens $(1,2)$ im Baum (b) kleiner ist.

4.4.3 Monotonie der Transferfunktion

In diesem Abschnitt wird gezeigt, dass die Transferfunktion f monoton steigt, also

$$a \subseteq b \Rightarrow f(a) \subseteq f(b)$$

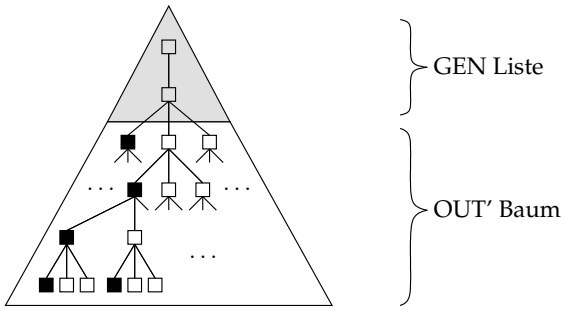

Abbildung 4.10: Beispiel eines IN Baums.

gilt.

In Abschnitt 4.1.2 wurde beschrieben, wie die Transferfunktion einen OUT Baum auf einen IN Baum abbildet. Dabei ergibt sich der IN Baum durch das Aneinanderhängen der GEN Liste und des OUT' Baums. Der OUT' Baum ist dabei der OUT Baum ohne die Register der GEN Liste.

In Abbildung 4.10 ist ein solcher IN Baum zu sehen. Der obere Teil des Baums besteht aus der GEN Liste. Weil die GEN Liste für einen Grundblock konstant ist, ist dieser Teil des IN Baums ebenfalls konstant. Insbesondere haben alle Knoten in diesem Teil stets die Wahrscheinlichkeit 1.

Der untere Teil des IN Baums besteht aus dem vollständigen OUT' Baum. Die Knoten der GEN Liste haben im OUT' Baum die Wahrscheinlichkeit 0 und sind schwarz markiert. Weil die GEN Liste konstant ist, sind also stets die gleichen Knoten schwarz markiert. Die Knoten im oberen Teil des Baumes und die schwarzen Knoten des OUT' Baums sind also eingabeunabhängig und besitzen daher in allen IN Bäumen die gleichen Wahrscheinlichkeiten. Es stellt sich nun die Frage, wie die Wahrscheinlichkeiten der übrigen Knoten von den Wahrscheinlichkeiten des OUT Baums abhängen.

Um dies zu beantworten, wird zunächst die del Funktion eingeführt. Diese Funktion gibt für einen Grundblock an, wie sich die Knoten in einem Baum verschieben, wenn die Register der GEN-Liste aus dem Baum gelöscht werden. Die Register der GEN-Liste werden im Weiteren auch GEN-Register genannt.

Die del Funktion bildet einen Knoten von seiner jetzigen Position auf die Position ab, die er nach dem Löschvorgang hat. Dabei werden aus der Position, die als Liste der Elternknoten gegeben ist, alle GEN-Register gelöscht. Für Knoten der GEN Liste ist die Funktion jedoch nicht definiert.

Die formale Definition von del lautet:

$$
\begin{aligned}
del(\epsilon) &= \epsilon \\
del(\rho_1, \rho_2, \ldots, \rho_d) &= \begin{cases} del(\rho_2, \ldots, \rho_n) & \text{wenn } \rho_1 \in GEN \\ \rho_1 \circ del(\rho_2, \ldots, \rho_n) & \text{sonst} \end{cases}
\end{aligned}
$$

In dem Beispiel in Abbildung 4.11 besteht die GEN Liste nur aus dem Register 1. Es werden also alle Knoten des Registers 1 aus dem Baum gelöscht. Dadurch wird der Knoten an der Position $(1,2)$ mit dem Knoten an der Position (2) vereinigt. Die Position $(1,2)$ wird deshalb von del auf die Position (2) abgebildet. Der Knoten an der Position (2) wird hingegen auf sich selbst abgebildet, weil er seine Position durch den Löschvorgang nicht ändert. Für

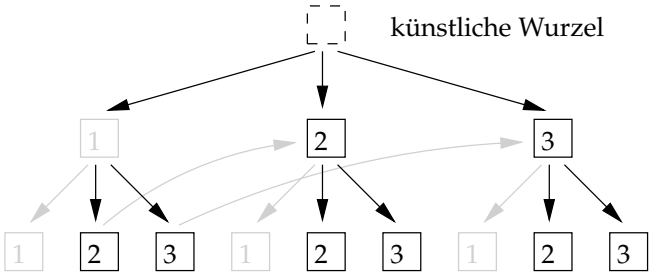

Abbildung 4.11: Löschen der GEN-Register aus dem vollständigen Baum.

den Knoten 1 an der Position $(2, 1)$ ist die del Funktion nicht definiert, weil der Knoten aus dem Baum gelöscht wird.

Die Definition der del Funktion hängt nur von der GEN Liste ab. Weil die GEN Liste eines Grundblocks konstant ist, ist somit auch die Definition der del Funktion eines Grundblocks konstant.

Zudem wird im Folgenden die $register$ Funktion benötigt, die angibt, welches Register sich an einer gegebenen Position befindet. Die Funktion ist durch

$$register(\rho_1, \rho_2, \ldots, \rho_d) \quad = \quad \rho_d$$

definiert.

Sei weiterhin die Menge $Q(x)$ die Menge aller Positionen, die durch die del Funktion auf eine Position x abgebildet werden. Positionen, an denen Knoten gelöscht werden, sind jedoch nicht in dieser Menge enthalten.

$$Q(x) := \{y | del(y) = x \wedge register(y) \notin GEN\}$$

Weil die Definitionen der del und $register$ Funktionen konstant sind, ist somit auch die Definition von $Q(x)$ konstant. Es wird also stets die gleiche Menge von Positionen $Q(x)$ durch die del Funktion auf eine Position x abgebildet. Bei dem Beispiel in Abbildung 4.11 gilt zum Beispiel $Q(2) = \{(1, 2), (2)\}$

Die Wahrscheinlichkeit des Knotens an der Position x im OUT' Baum ist nun gleich der Summe der Wahrscheinlichkeiten an den Positionen $Q(x)$ im OUT Baum.

$$p_{OUT'}(x) = \sum_{y \in Q(x)} p_{OUT}(y)$$

Diese Gleichung gibt also an, wie sich die Wahrscheinlichkeit eines Knotens im OUT' Baum berechnet. Es werden dabei stets die Wahrscheinlichkeiten der gleichen Knoten des OUT Baums summiert.

Mit Hilfe dieser Aussagen folgt nun die Monotonie der Transferfunktion. Seien dazu $OUT1$,$OUT2$ zwei vollständige Bäume für die

$$OUT1 \subseteq OUT2$$

gilt. Der Baum OUT1 ist also kleiner als der Baum OUT2.

Mit Hilfe der Definition der Halbordnung kann auch eine äquivalente Aussage über die Knotenwahrscheinlichkeiten gemacht werden.

$$\forall x : p_{OUT1}(x) \leq p_{OUT2}(x)$$

Die Wahrscheinlichkeit an einer Position x ist also im OUT1 Baum nicht größer als im OUT2 Baum. Diese Ungleichheit gilt auch für die Summe über eine Menge von Knoten in beiden Bäumen.

$$\sum_{y \in Q(x)} p_{OUT1}(y) \leq \sum_{y \in Q(x)} p_{OUT2}(y)$$

Die beiden Summen geben nun die Wahrscheinlichkeit eines Knotens x im OUT1', bzw. OUT2' Baum an. Es folgt somit, dass die Wahrscheinlichkeit eines Knotens x im OUT1' Baum kleiner ist, als im OUT2' Baum.

$$p_{OUT1'}(x) \leq p_{OUT2'}(x)$$

Diese Beziehung gilt auch für einen Knoten $x \in GEN$, weil die Wahrscheinlichkeit eines solchen Knotens in beiden Bäumen gleich 0 ist. Da die Knoten der GEN Liste konstant sind, folgt somit für die IN Bäume, dass die Wahrscheinlichkeit eines Knotens x im IN1 Baum nicht größer ist als im IN2 Baum.

$$\forall x : p_{IN1}(x) \leq p_{IN2}(x)$$

Wendet man hierauf wiederum die Definition der Halbordnung an, erhält man eine äquivalente Aussage über die Bäume.

$$IN1 \subseteq IN2$$

Es wurde somit gezeigt, dass die Aussage

$$OUT1 \subseteq OUT2 \Rightarrow IN1 \subseteq IN2$$

und folglich die Monotonie der Transferfunktion gilt.

4.4.4 Monotonie der Vereinigungsfunktion

Wie bereits erwähnt, hat die Vereinigungsfunktion nicht die Eigenschaft einer Join-Operation. Es kann aber gezeigt werden, dass die Vereinigungsfunktion monoton ist, was für die Terminierung der Datenflussanalyse genügt. Die Monotonie ist dabei durch

$$a \leq a' \wedge b \leq b' \Rightarrow a \cup b \leq a' \cup b'$$

definiert. Jede Join-Operation besitzt übrigens diese Eigenschaft.

Die Wahrscheinlichkeit an einer Position x im OUT Baum ergibt sich als Mittelwert der Wahrscheinlichkeiten an der Position x in den IN Bäumen. Die Wahrscheinlichkeit im OUT Baum steigt daher, wenn die Wahrscheinlichkeiten in den IN Bäumen steigen.

Beim Vereinigen der Bäume $IN_1, \ldots, IN_m$ zu einem OUT Baum ist die Wahrscheinlichkeit $p_{OUT}(x)$ also wie folgt durch die Wahrscheinlichkeiten $p_{IN_i}(x)$ gegeben:

$$p_{OUT} = \frac{\sum_{i=0}^{m} p_{IN_i}(x)}{m}$$

Es ist nun zu zeigen, dass der OUT Baum höchstens wächst, wenn alle IN Bäume gleich bleiben oder wachsen.

$$(\forall i \leq m : IN1_i \subseteq IN2_i) \Rightarrow (OUT1 \subseteq OUT2)$$

Diese Aussage wird nun formal bewiesen. Dabei werden für die Aussagen über Bäume im Beweis die äquivalenten Aussagen über Knotenwahrscheinlichkeiten betrachtet. Die Erläuterungen zu dem Beweis folgen anschließend.

$$
\begin{aligned}
& \forall i \leq m : IN1_i \subseteq IN2_i \\
\Rightarrow \quad & \forall i \leq m, x \in POS : p_{IN1_i}(x) \leq p_{IN2_i}(x) \\
\Rightarrow \quad & \forall x \in POS : \frac{\sum_{i \leq m} p_{IN1_i}(x)}{n} \leq \frac{\sum_{i \leq m} p_{IN2_i}(x)}{n} \\
\Rightarrow \quad & \forall x \in POS : p_{OUT1}(x) \leq p_{OUT2}(x) \\
\Rightarrow \quad & OUT1 \subseteq OUT2
\end{aligned}
$$

In dem Beweis wird zunächst die Definition der Halbordnung verwendet, um eine Aussage über alle Knoten zu erhalten. Wenn für alle i die Wahrscheinlichkeit $p_{IN1_i}(x)$ kleiner ist, als $p_{IN2_i}(x)$, ist auch die Summe darüber kleiner. Mit der Definition der Vereinigungsfunktion folgt daraus, dass die Wahrscheinlichkeit an einer Position x im Baum $OUT1$ nicht größer ist, als im Baum $OUT2$. Mit der Definition der Halbordnung folgt daraus letztlich, dass der OUT1 Baum kleiner ist, als der OUT2 Baum.

4.4.5 Terminierung der Datenflussanalyse

In den vorherigen beiden Abschnitten wurde gezeigt, dass die Transferfunktion und die Vereinigungsfunktion monoton sind. Nun soll gezeigt werden, dass dadurch auch die IN und OUT Eigenschaften mit der Anzahl der Iterationen der Datenflussanalyse monoton steigen.

Zu Beginn der Datenflussanalyse werden alle Wahrscheinlichkeiten in den IN und OUT Bäumen mit 0 initialisiert. Damit können die Wahrscheinlichkeiten in der ersten Iteration nur steigen.

Weil dadurch die OUT Bäume monoton gewachsen sind, wachsen auch die IN Bäume durch die Monotonie der Transferfunktion monoton. Gleiches gilt für die OUT Bäume mit der Monotonie der Vereinigungsfunktion.

Weil die Wahrscheinlichkeiten in einem Baum durch $p_x \leq 1$ begrenz sind, folgt daraus die Konvergenz der Wahrscheinlichkeiten in den IN und OUT Bäumen und damit auch die Konvergenz der Datenflussanalyse. Dennoch kann es vorkommen, dass der Fixpunkt nicht nach einer endlichen Anzahl von Iterationen gefunden wird.

In dem Beispiel in Abbildung 4.12 nähert sich die Wahrscheinlichkeit von y gemäß $1 - 2^{-}k$ dem Wert 1 beliebig nah an, erreicht diesen aber nicht. Setzt man also als Bedingung für die Terminierung die Gleichheit aller Wahrscheinlichkeiten in zwei aufeinander folgenden Iterationen an, terminiert die Datenflussanalyse nicht.

Eine Voraussetzung dafür ist, dass die Wertemenge der Wahrscheinlichkeiten nicht abzählbar ist — eine Bedingung, die bei der Verwendung von Datentypen mit endlichem Speicherplatz nicht erfüllt ist.

4.4.6 Eigenschaften der Vereinigungsfunktion

Die hier beschriebene Vereinigungsfunktion erfüllt nicht die Bedingung einer Join-Operation. Dafür müsste das Ergebnis der Vereinigungsfunktion unter anderem stets größer sein, als die jeweiligen Operanden.

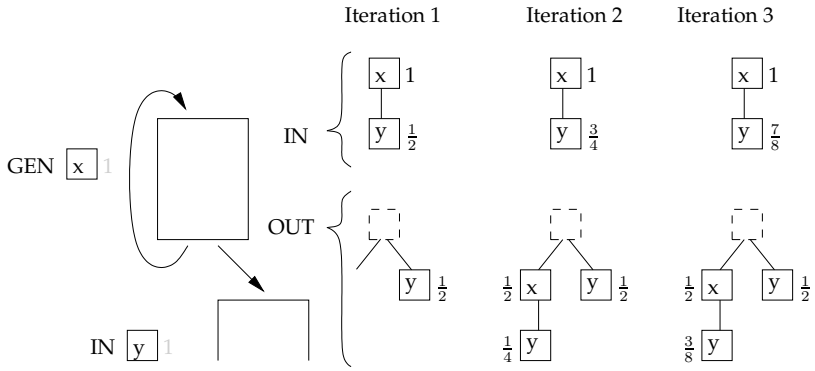

Abbildung 4.12: Entwicklung der Wahrscheinlichkeiten der IN und OUT Bäume über mehrere Iterationen der Datenflussanalyse.

Diese Bedingung kann durch einen kleinen Kunstgriff erfüllt werden, indem in der Vereinigungsfunktion nicht ein Mittelwert, sondern eine Summe gebildet wird. Das Ergebnis ist zwar in diesem Fall größer, als die jeweiligen Operanden, allerdings bildet die Vereinigungsfunktion nicht das Supremum der beiden Operanden. Dabei ist das Supremum das kleinste Element der Menge, das nicht kleiner ist als die beiden Operanden. Eine solche Funktion ist also auch idempotent, was für die Summenbildung in der Vereinigungsfunktion nicht gilt.

Zudem besitzt die Menge der zulässigen Zugriffsbäume kein Top Element. Dieses müsste größer sein, als alle zulässigen Zugriffsbäume. Es müsste sich also um einen vollständigen Baum mit einer Wahrscheinlichkeit von 1 an jedem Knoten handeln. Ein solcher Zugriffsbaum ist jedoch nicht zulässig, weil Satz 1 nicht gilt.

Kapitel 5

Registerzuteilung

Im Compiler wird ein Programm in der Code-Auswahl Phase zunächst in Assembler-Instruktionen übersetzt, die noch virtuelle Registeroperanden haben. Diese virtuellen Registeroperanden stehen in beliebiger Anzahl zur Verfügung und werden in der Registerzuteilungsphase auf die Register des Prozessors abgebildet.

Bei der Registerzuteilung in dieser Arbeit werden die virtuellen Register zunächst den *physikalischen Registern* zugeteilt. Die Instruktionen haben dann keine virtuellen, sondern *physikalische Registeroperanden*. Nach der Registerzuteilung steht also fest, in welchem physikalischen Register ein Wert gespeichert wird. Es steht allerdings noch nicht fest, über welches architektonische Register auf das physikalische Register zugegriffen wird.

Diese Entscheidung wird erst in einem zweiten Schritt getroffen, in dem *Rekonfigurationsanweisungen* eingefügt werden. Die Rekonfigurationsanweisungen blenden einen physikalischen Block in einen architektonischen Block ein. Nach dem Einfügen stehen also die Einblendungen für jede Stelle des Programms fest. Es ist daher bekannt, in welchen architektonischen Block ein physikalischer Block eingeblendet wird. Folglich ist auch bekannt, in welches architektonische Register ein physikalisches Register eingeblendet wird. Somit kann ein physikalischer Registeroperand durch seinen *architektonischen* ersetzt werden. Dieser wird schließlich als Operand in der Maschineninstruktion codiert.

Beide Schritte zusammen bilden die Registerzuteilung in dieser Arbeit. Der erste Schritt wird dabei als *Zuteilung der physikalischen Register* und der zweite als *Einfügen der Rekonfigurationsanweisungen* bezeichnet.

5.1 Zuteilung der physikalischen Register

Für die Zuteilung der physikalischen Register kommt eine Erweiterung des globalen Registerzuteilungsverfahrens von Chaitin zur Anwendung, das in Abschnitt 2.3 beschrieben wurde. Durch dieses Verfahren werden die physikalischen Register effizient genutzt und die Häufigkeit von Spillcode gering gehalten.

Von der Durchführung der Registerzuteilung hängt jedoch auch die Anzahl der Rekonfigurationsanweisungen ab, die im zweiten Schritt in den Code ein-

gefügt werden. Wird auf die virtuellen Register $v_1, \ldots, v_n$ im Programm häufig zeitnah zugegriffen, sollten diese möglichst den Registern eines einzigen physikalischen Blocks zugeteilt werden. Dadurch ist nur eine Rekonfigurationsanweisung notwendig, um die Register einzublenden.

Um festzustellen, welche Register möglichst einem gemeinsamen physikalischen Block zugeteilt werden sollten, wird ein Affinitätsgraf berechnet. Dieser Graf drückt aus, wie häufig auf zwei Register zeitnah zugegriffen wird. Die Affinitäten aus diesem Grafen sind bei der Zuteilung der physikalischen Register ein *sekundäres Entscheidungskriterium*.

Wenn nach dem Zuteilungsverfahren von Chaitin also mehrere alternative physikalische Register zugeteilt werden können, wird anhand der Affinität entschieden, welches physikalische Register zugeteilt wird. Das Verfahren von Chaitin wird durch diese Erweiterung also nicht eingeschränkt.

Wie später gezeigt wird, fließen in den Affinitätsgrafen auch die bereits durchgeführten Zuteilungen ein. Deshalb wird der Affinitätsgraf nach der Zuteilung eines physikalischen Registers berechnet. Dieser Schritt ist zwar nicht notwendig, verbessert jedoch die Aussagekraft des Affinitätsgrafen.

5.1.1 Affinitätsanalyse

In der Affinitätsanalyse wird auf der Grundlage des Assemblercodes einer Funktion ein Affinitätsgraf erstellt. Der Affinitätsgraf dient bei der Registerzuteilung als Hilfe, um die virtuellen Register so zu färben, dass am Ende möglichst wenige Rekonfigurationsanweisungen eingefügt werden müssen.

Die Knoten des Affinitätsgrafen sind virtuelle Register. Die Kanten des Grafen sind mit Affinitäten gewichtet. Dabei gibt die Affinität zwischen zwei virtuellen Registern an, wie vorteilhaft es ist, beide virtuellen Register einem gemeinsamen physikalischen Block zuzuordnen.

Um die Affinität zwischen zwei virtuellen Registern definieren zu können, ist zunächst ein Vorgriff auf die Verdrängungsstrategie beim Einblenden von physikalischen Blöcken notwendig.

Verdrängungsstrategie

Die Verdrängungsstrategie legt fest, welcher physikalische Block verdrängt wird, wenn eine Einblendung notwendig ist und in alle architektonischen Blöcke bereits physikalische Blöcke eingeblendet sind.

Bei der hier verwendeten Verdrängungsstrategie wird der physikalische Block verdrängt, der erst am spätesten in der Zukunft erneut verwendet wird. Im Folgenden wird diese Strategie als “latest used again” (LUA) bezeichnet. Das Verfahren wurde ursprünglich von Belady [Bel66] als optimaler paging Algorithmus vorgestellt und findet unter anderem bei der lokalen Registerzuteilung auf Grundblockebene Verwendung.

Es kann folgende Aussage darüber getroffen werde, wann ein physikalischer Block nach dieser Strategie *nicht* verdrängt wird.

Satz 7 *Ein physikalischer Block x wird zwischen zwei Zugriffen auf x nicht nach LUA verdrängt, wenn zwischen den beiden Zugriffen auf weniger als $|\mathcal{B}_A|$ andere Blöcke zugegriffen wird.*

Bei der Begründung sind zwei Fälle zu unterscheiden:

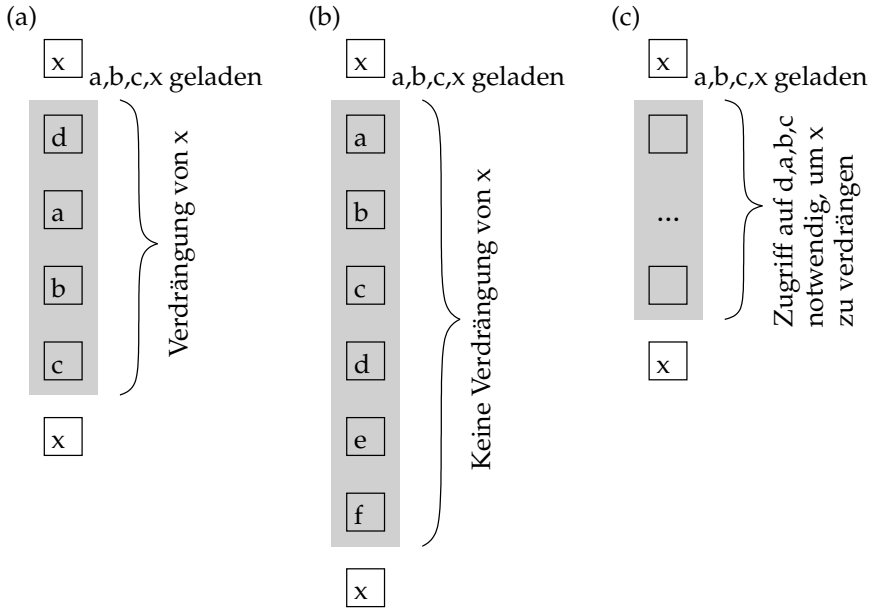

Abbildung 5.1: Verdrängung eines physikalischen Blocks zwischen zwei Zugriffen auf den Block.

Fall 1: Nach dem ersten Zugriff auf x sind nicht alle architektonischen Blöcke mit physikalischen belegt. In diesem Fall wird kein physikalischer Block verdrängt, sondern ein freier architektonischer Block verwendet.

Fall 2: Alle architektonischen Blöcke sind belegt. Angenommen der physikalische Block x wird zwischen zwei Zugriffen nach LUA verdrängt. Dann muss auf alle anderen $|\mathcal{B}_A| - 1$ physikalischen Blöcke, die zum Zeitpunkt des Verdrängens eingeblendet sind noch vor dem nächsten Zugriff auf x zugegriffen werden. Zusätzlich muss auf einen nicht eingeblendeten Block zugegriffen werden, womit die Anzahl der Blöcke auf die zugegriffen wird mindestens $|\mathcal{B}_A|$ beträgt. Dies ist ein Widerspruch zu der Bedingung, dass auf nicht mehr als $|\mathcal{B}_A - 1|$ Blöcke zugegriffen wird.

In Abbildung 5.1c ist ein Beispiel für eine Registerarchitektur mit $|\mathcal{B}_A| = 4$ architektonischen Blöcken zu sehen. Angenommen nach dem ersten Zugriff auf x sind die physikalischen Blöcke a, b, c, x eingeblendet. Damit x verdrängt wird, muss zwischen den beiden Zugriffen auf x auf einen Block d und auf die Blöcke a, b, c zugegriffen werden, damit x der am spätesten wiederverwendete Block ist. Es muss also auf mindestens 4 Blöcke zugegriffen werden. Wird auf weniger als 4 Blöcke zugegriffen, kann x zwischen den beiden Zugriffen folglich nicht verdrängt werden. In Abbildung 5.1a sind 4 solche Zugriffe dargestellt, durch die x verdrängt wird.

Umgekehrt gilt jedoch nicht, dass ein Block zwingend verdrängt wird, wenn auf mehr als $|\mathcal{B}_A|$ Blöcke zugegriffen wird. Ein Gegenbeispiel ist in Abbildung 5.1b zu sehen. Darin tritt die erste Verdrängung bei dem Zugriff auf d auf. Danach wird jedoch nicht mehr auf die Blöcke a, b, c zugegriffen, wodurch

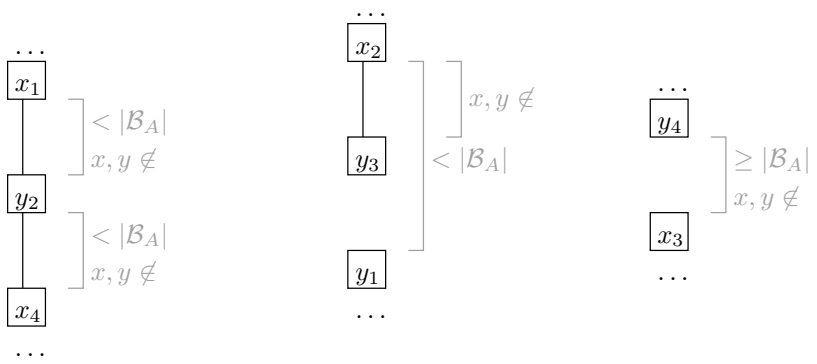

Abbildung 5.2: Beispiele von Zugriffspaaren.

x nicht der am spätesten wiederverwendete physikalische Block ist. Somit wird nicht x, sondern einer der Blöcke a, b, c verdrängt.

Definition der Affinität

Für die Verdrängungsstrategie LUA kann nun die Affinität definiert werden.

Die Affinität zwischen zwei virtuellen Registern x und y drückt aus, wie viele potenzielle Rekonfigurationsanweisungen dadurch vermieden werden können, dass x und y demselben physikalischen Block zugeordnet werden.

Im Folgenden bezeichne x_i einen Zugriff auf das virtuelle Register x und y_j einen Zugriff auf das virtuelle Register y. Die Affinität zwischen x und y ist nun gleich der Anzahl der ungeordneten Zugriffspaare (x_i, y_j), für die Folgendes gilt:

1. Zwischen den beiden Zugriffen wird auf weniger als $|\mathcal{B}_A|$ andere virtuelle Register zugegriffen.

2. Zwischen den beiden Zugriffen wird nicht auf x oder y zugegriffen.

Angenommen die virtuellen Register x und y werden demselben physikalischen Block P zugeteilt. Dann kann für ein Zugriffspaar (x_i, y_j) der physikalische Block P zwischen den beiden Zugriffen x_i und y_i nicht verdrängt werden. Der Grund dafür ist, dass zwischen x_i und y_j auf weniger als $|\mathcal{B}_A|$ andere virtuelle Register zugegriffen wird. Dadurch wird auch auf weniger als $|\mathcal{B}_A|$ andere physikalische Blöcke zugegriffen. Nach Satz 7 kann der physikalische Block P deshalb zwischen den Zugriffen nicht verdrängt werden.

Findet im Programm erst der Zugriff x_i und dann der Zugriff y_i statt, wird für den Zugriff auf y_i folglich keine Rekonfigurationsanweisung eingefügt. Werden die Register x und y hingegen zwei unterschiedlichen physikalischen Blöcken zugeordnet, kann es notwendig sein, jeweils eine Rekonfigurationsanweisung für die Zugriffe x_i und y_j einzufügen.

Die zweite Bedingung gewährleistet, dass für jedes Zugriffspaar auch eine zusätzliche Rekonfiguration auftreten kann, wenn x und y zwei unterschiedlichen physikalischen Blöcken zugeteilt werden. Es wird also vermieden, dass die Affinität überschätzt wird.

In Abbildung 5.2 ist ein Beispiel für die beiden virtuellen Register x und y dargestellt. Dabei wird, wie in grau angegeben, zwischen den Zugriffen x_1 und y_2 auf weniger als $|\mathcal{B}_A|$ verschiedene virtuelle Register zugegriffen. Zudem befindet sich zwischen x_1 und y_2 kein Zugriff auf x oder y. Folglich ist nach der obigen Definition (x_1, y_2) ein Zugriffspaar. Gleiches gilt für die Paare (y_2, x_4) und (x_2, y_3).

Die Zugriffe y_4 und x_3 bilden hingegen kein Zugriffspaar, weil dazwischen nicht auf weniger als $|\mathcal{B}_A|$ andere virtuelle Register zugegriffen wird. Dadurch wird die erste Bedingung verletzt. Die Zugriffe x_2 und y_1 bilden ebenfalls kein Zugriffspaar, weil sich dazwischen der Zugriff y_3 befindet. Dadurch wird die zweite Bedingung verletzt. Die Affinität zwischen x und y beträgt im Beispiel somit 3.

Ausführungshäufigkeiten

Damit zur Laufzeit möglichst wenige Rekonfigurationen durchgeführt werden, sollten Rekonfigurationsanweisungen möglichst nicht in häufig ausgeführten Grundblöcken auftreten. Die Affinitäten drücken bisher jedoch nur aus, wie viele Rekonfigurationsanweisungen im Code vermieden werden, wenn zwei virtuelle Register dem gleichen physikalischen Block zugeteilt werden. Durch die Verwendung von Ausführungshäufigkeiten sollen die Affinitäten nun eine Aussage darüber liefern, wie viele Rekonfigurationen zur Laufzeit eingespart werden können.

Die Ausführungshäufigkeiten der Grundblöcke können durch ein Profiling des Programms bestimmt werden. Wenn diese Daten nicht zur Verfügung stehen, kann zur Übersetzungszeit alternativ durch eine *Schleifenheuristik* abgeschätzt werden, wie häufig ein Grundblock ausgeführt wird. Dabei wird für jede Schleife eine konstante Anzahl von c Iterationen angenommen. Sind zwei Schleifen ineinander geschachtelt, wird somit für die Grundblöcke der inneren Schleife eine Ausführungshäufigkeit von c^2 angenommen.

Um Rekonfigurationsanweisungen in häufig ausgeführten Grundblöcken zu vermeiden, werden die Zugriffspaare (x_i, y_j) nun mit den Ausführungshäufigkeiten gewichtet. Befindet sich ein Zugriffspaar in einem Grundblock mit der Ausführungshäufigkeit i, wird es also mit i gewichtet. Die Affinität zwischen zwei virtuellen Register ergibt sich somit als gewichtete Summe der Zugriffspaare.

Die Affinität zwischen x und y schätzt nun ab, wie viele Rekonfigurationen zur Laufzeit vermieden werden, wenn x und y einem gemeinsamen physikalischen Block zugeordnet werden.

Affinitäten zu physikalischen Blöcken

Es wurde bisher gezeigt, wie die Affinitäten zwischen den virtuellen Registern berechnet werden. In diesem Abschnitt werden nun auch Affinitäten zwischen virtuellen Registern und physikalischen Blöcken eingeführt. Diese Affinitäten ergeben sich, wenn durch Chaitins Verfahren bereits einigen virtuellen Registern die physikalischen Register zugeteilt wurden. Die virtuellen Register werden dann in der Affinitätsanalyse mit dem zugehörigen physikalischen Block assoziiert. Die Assoziation hat für die Affinitätsanalyse die gleiche Wirkung, wie das Ersetzen der virtuellen Register durch die jeweiligen physikalischen

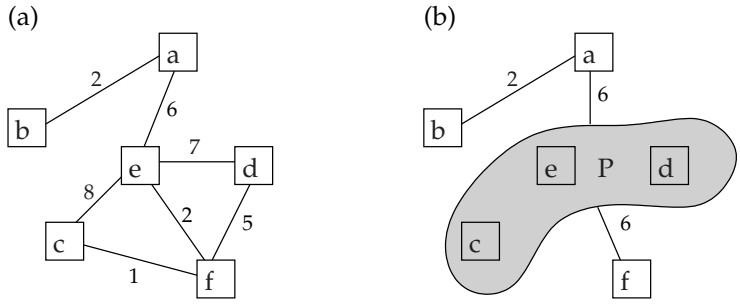

Abbildung 5.3: Assoziation von virtuellen Registern mit physikalischen Blöcken im Affinitätsgrafen.

Blöcke. Die physikalischen Blöcke werden von der Affinitätsanalyse also transparent wie virtuelle Register behandelt. In den weiteren Beschreibungen werden deshalb auch nur die virtuellen Register erwähnt, weil stets Gleiches für die physikalischen Blöcke gilt.

In Abbildung 5.3a ist ein Affinitätsgraf dargestellt, dessen virtuelle Register noch nicht zugeteilt sind. Werden nun die virtuellen Register c, e, d dem physikalischen Block P zugeordnet, ergibt sich daraus der Affinitätsgraf 5.3b. Darin haben die virtuellen Register a, f nun keine Affinität mehr zu den virtuellen Registern c, e, d, sondern stattdessen zu dem physikalischen Block P. Bemerkenswert ist, dass die Affinität zwischen f und P nicht gleich der Summe der Affinitäten zwischen f und c, e, d ist. Ein solcher Effekt kommt daher, dass die virtuellen Register c, e, d nun von der Affinitätsanalyse als physikalischer Block P gesehen werden und doppelte Affinitäten dadurch wegfallen.

Die Bedeutung einer Affinität zu einem physikalischen Block ist recht intuitiv. Eine Affinität zwischen einem virtuellen Register x und einem physikalischen Block P drückt aus, wie viele Rekonfigurationsanweisungen dadurch vermieden werden können, dass x einem physikalischen Register in P zugeteilt wird.

Dadurch, dass die virtuellen Register mit dem zugeordneten physikalischen Block assoziiert werden, verbessert sich auch die Aussagekraft der Affinitätsanalyse. Bei der Definition der Affinität auf Seite 50 wurde davon ausgegangen, dass nicht bekannt ist, welche virtuellen Register einem gemeinsamen physikalischen Block zugeordnet werden. Es musste deshalb bei der Bestimmung der Zugriffspaare die pessimistische Abschätzung gemacht werden, dass alle virtuellen Register verschiedenen physikalischen Blöcken zugeordnet werden. Daraus ergab sich, dass zwei Zugriffe x_i, y_j nur ein Zugriffspaar bildeten, wenn dazwischen auf weniger als $|\mathcal{B}_A|$ *virtuelle Register* zugegriffen wurde. Durch die Assoziation der virtuellen Register mit den physikalischen Blöcken fallen nun virtuelle Register zusammen und es entstehen neue Zugriffspaare.

In Abbildung 5.4a werden den beiden virtuellen Registern a, c physikalische Register aus dem physikalischen Block P zugeteilt. Folglich werden in 5.4b die beiden Zugriffe nun mit P assoziiert. Weil zwischen den Zugriffen x_1 und y_1 jetzt auf weniger als $|\mathcal{B}_A|$ physikalische Blöcke zugegriffen wird, entsteht das neu Zugriffspaar (x_1, y_1).

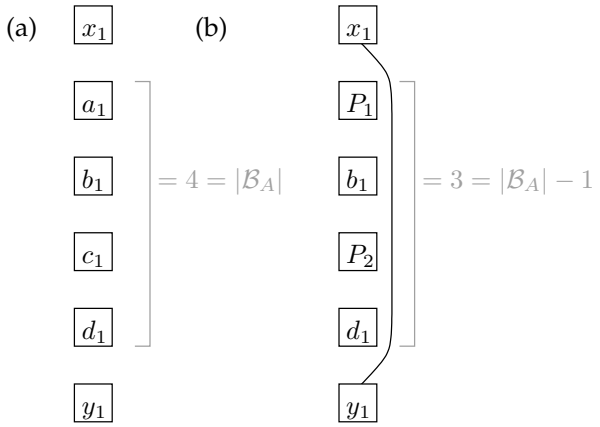

Abbildung 5.4: Entstehung neuer Zugriffspaare.

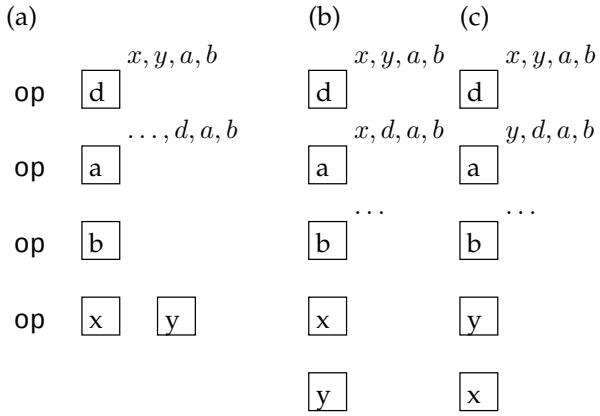

Abbildung 5.5: Abhängigkeit der Verdrängung von der totalen Ordnung der Zugriffe.

Ordnung der Zugriffe

Bisher wurden nur *total geordnete* Folgen von Zugriffen betrachtet. Bei Instruktionen mit zwei oder mehr Operanden finden die Zugriffe jedoch zeitgleich statt. Die Zugriffe innerhalb einer Instruktion sind folglich nicht geordnet.

Für die Verdrängungsstrategie LUA ist bei solchen gleichzeitigen Registerzugriffen nicht eindeutig definiert, welcher Zugriff der Späteste ist. Durch die Festlegung einer *speziellen* totalen Ordnung zu dieser Halbordnung wird dieser Konflikt aufgelöst. Dabei kann die Entscheidung über eine Verdrängung wie in Abbildung 5.5 von der gewählten Reihenfolge der Operanden innerhalb einer Instruktion abhängen. Abhängig von der Reihenfolge, in der die beiden Zugriffe auf x und y betrachtet werden, ergibt sich die total geordnete Zugriffsfolge in (b) oder (c). Es wird deshalb durch den Zugriff auf d in der Liste (b) der Block y verdrängt und in (c) der Block x. Die Anzahl der Rekonfigurationsanweisungen hängt jedoch nicht von der Wahl der Reihenfolge ab. Wie im Beispiel zu sehen ist, muss in beiden Fällen eine Rekonfigurationsanweisung

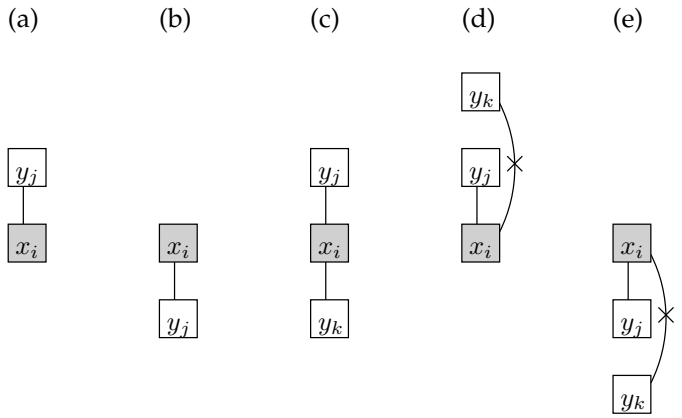

Abbildung 5.6: Beispiele für Zugriffspaare.

vor der Instruktion `op x y` durchgeführt werden.

Berechnung des Affinitätsgrafen

Bisher wurde beschrieben, wie eine Affinität definiert ist und welche Bedeutung sie hat. In diesem Abschnitt wird nun ein Algorithmus angegeben, um die Affinitäten effizient zu berechnen.

Laut Definition muss die Anzahl der Zugriffspaare (x_i, y_j) bestimmt werden, für die gilt:

1. Zwischen den beiden Zugriffen wird auf weniger als $|\mathcal{B}_A|$ andere virtuelle Register zugegriffen.

2. Zwischen den beiden Zugriffen wird nicht auf x oder y zugegriffen.

Für diese Aufgabe wird die Datenflussanalyse "n-Lebendigkeit" aus Kapitel 4 verwendet. Bei der Analyse werden die die n-lebendigen virtuellen Register bestimmt. Dadurch ist für jede Stelle des Programms bekannt, auf welche $|\mathcal{B}_A|$ virtuellen Register als nächstes zugegriffen wird.

Die erste Bedingung für ein Zugriffspaar kann somit durch die n-Lebendigkeit ausgedrückt werden. Ein Zugriffspaar (x_i, y_j) liegt nun genau dann vor, wenn y beim Zugriff x_i n-lebendig ist, bzw. wenn x beim Zugriff y_j n-lebendig ist. Weil beim Zugriff auf x_i nur gegeben ist, ob y n-lebendig ist, ist nicht bekannt mit welchen $y_{...}$ der Zugriff x_i das Zugriffspaar bildet. Diese Information ist auch nicht notwendig, weil stets nur ein Zugriffspaar $(x_i, y_{...})$ existiert.

Weil bei der n-Lebendigkeit auch festgehalten wird, in welcher Reihenfolge die Zugriffe erfolgen, kann festgestellt werden, ob zwischen dem Zugriff x_i und dem nächsten Zugriff auf y ein Zugriff auf x erfolgt. Damit kann auch die zweite Bedingung erfüllt werden.

In Abbildung 5.6 sind einige Beispiele für Zugriffspaare der virtuellen Register x und y dargestellt. Bilden zwei Zugriffe ein Zugriffspaar, sind sie in der Abbildung durch eine Kante verbunden. In (a) bilden (y_j, x_i) ein Zugriffspaar, weil x beim Zugriff y_j lebendig ist. Analog dazu besteht in (b) ein Zugriffspaar,

```
 1  FUNCTION analysiere_affinität()
 2    nlebendigkeitsanalyse();
 3    FORALL grundblöcke DO
 4      nlebendige_register=nlebendig_out(grundblock)
 5      FORALL instruktionen DO   /* letzte zur ersten */
 6        FORALL operanden DO
 7          FORALL nlebendige_register DO
 8            IF nlebendiges_register == operand THEN
 9              BREAK
10            FI
11            affinität[operand,nlebendiges_register]
                              +=ausführungen(grundblock)
12          DONE
13          voranstellen(nlebendige_register,operand)
14        DONE
15      DONE
16    DONE
17  END
```

Abbildung 5.7: Algorithmus zur Berechnung der Affinitäten.

weil y beim Zugriff x_i lebendig ist. In (c) bestehen folglich zwei Zugriffspaare (y_j, x_i) und (x_i, y_k). Hingegen liegt in (d) kein Zugriffspaar (y_k, x_i) vor, weil sich dazwischen der Zugriff y_j befindet. Dies wird dadurch festgestellt, dass für die Stelle y_k im Zugriffsbaum y oberhalb von x liegt. Es ist also bekannt, dass vor dem nächsten Zugriff auf x noch ein Zugriff auf y erfolgt. In (e) bilden (x_i, y_k) ebenfalls kein Zugriffspaar, weil nur ein Zugriffspaar der Form $(x_i, y_{...})$ gebildet wird.

Nachdem die Grundlage des Algorithmus skizziert wurde, wird er im Folgenden genau beschrieben. Dazu wird der Pseudocode des Algorithmus in Abbildung 5.7 erläutert.

Zu Beginn werden durch die n-Lebendigkeitsanalyse die n-lebendigen virtuellen Register für den Anfang und das Ende eines jeden Grundblocks bestimmt.

Durch die Schleifen in den Zeilen 3 und 5 werden nun in allen Grundblöcken die Instruktionen von der letzten zur ersten durchlaufen. Dabei wird der Zugriffsbaum am Ende des Grundblocks mit dem OUT Zugriffsbaum aus der n-Lebendigkeitsanalyse initialisiert. Durch das sukzessive Voranstellen der Zugriffe dieses Grundblocks in Zeile 13 wird so der Zugriffsbaum für jede Position im Grundblock bestimmt.

In der Zeile 6 wird über alle Zugriffe einer Instruktion iteriert. Für jeden Zugriff x_i wird nun in der Zeile 11 die Affinität zu den n-Lebendigen Registern erhöht. Trifft der Algorithmus beim Iterieren der Register auf das Register x, geht der Algorithmus zum nächsten Zugriff der Instruktion über, um die zweite Bedingung für die Zugriffspaare zu gewährleisten.

Es bleibt noch zu erwähnen, dass die Zugriffspaare in Zeile 11 mit der Ausführungshäufigkeit des Grundblocks gewichtet werden.

5.1.2 Erweiterung des Zuteilungsverfahrens

In dem vorherigen Abschnitt wurde die Berechnung des Affinitätsgrafen behandelt. In diesem Abschnitt wird nun beschrieben, wie mit Hilfe des Affinitätsgrafen den virtuellen Registern die physikalischen zugeteilt werden. Die Zuteilung wird auch als Färbung eines virtuellen Registers mit einem physikalischen bezeichnet.

Das Registerzuteilungsverfahren basiert auf dem Verfahren von Chaitin. Dabei werden nacheinander die virtuellen Register mit den physikalischen gefärbt. Die Reihenfolge wird dabei so gewählt, dass möglichst jedem virtuellen Register ein freies physikalisches Register zugeteilt werden kann. Das Verfahren legt jedoch nicht fest, welches physikalische Register zugeteilt wird, wenn mehrere Alternativen bestehen.

An dieser Stelle wird nun der Affinitätsgraf verwendet, um zu entscheiden, welches physikalische Register zugeteilt werden soll. Es gibt mehrere Heuristiken, nach denen ein physikalisches Register ausgewählt werden kann.

Bei einer nahe liegenden Heuristik wird dem virtuellen Register das physikalische Register zugeteilt, zu dessen physikalischen Block die größte Affinität besteht. Dabei wird jedoch nicht beachtet, wie viele Register noch in dem physikalischen Block frei sind. Folglich würden die virtuellen Register einem physikalischen Block nach dem anderen zugeordnet. Das liegt daran, dass die Affinität zwischen einem virtuellen Register und einem physikalischen Block, dem noch kein virtuelles Register zugeteilt wurde, gleich null ist. Ein virtuelles Register wird also tendenziell dem physikalischen Block zugeordnet, dem bisher die meisten virtuellen Register zugeordnet wurden.

Um die Partitionierung der virtuellen Register auf die physikalischen Blöcke zu balancieren, muss deshalb zusätzlich die Anzahl der belegten, bzw. freien physikalischen Register berücksichtigt werden. Ein solche Heuristik kann darin bestehen, die Affinität durch die Anzahl der belegten physikalischen Register zu teilen. Dabei werden allerdings die virtuellen Register recht gleichmäßig auf alle physikalischen Blöcke verteilt, was nachteilig ist, wenn die Register aus wenigen physikalischen Blöcken für die gesamte Färbung ausreichen würden.

Deshalb wird eine Heuristik verwendet, die zusätzlich die Affinitäten zu anderen nicht zugeordneten virtuellen Registern berücksichtigt. Die Bewertungsfunktion dieser Heuristik lautet:

$$rate = a_{max} - \frac{a_{max} - a_P}{1 + f * a_{sum} * c}$$

Die Symbole der Formel sind in Abbildung 5.8 aufgeführt und haben die folgende Bedeutung:

a_{max} ist die maximale Affinität, die zwischen dem virtuellen Register x und einem physikalischen Block P bestehen kann. Durch jeden Zugriff x_i können maximal zwei Zugriffspaare für x_i und den physikalischen Block P entstehen. Die maximale Affinität ergibt sich somit zu

$$a_{max}(x) = 2 * \sum_i \text{ausführungshäufigkeit}(x_i)$$

a_P ist die bisherige Affinität zwischen x und dem physikalische Block P.

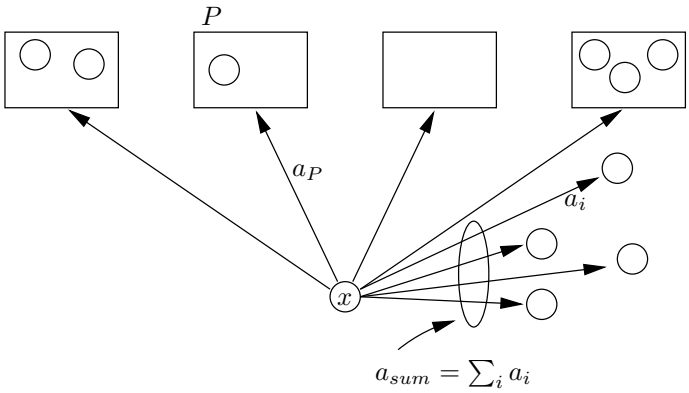

Abbildung 5.8: Gewichtung von leeren physikalischen Blöcken um eine gleichmäßige Verteilung zu erzielen.

f ist die Anzahl der freien physikalischen Register in dem physikalischen Block P.

a_{sum} ist die Summe der Affinitäten zwischen dem virtuellen Register x und allen anderen noch nicht zugeteilten virtuellen Registern.

c ist eine Konstante, durch die das Verhalten der Bewertung beeinflusst werden kann. Für $c = 0$ ist die Bewertung gleich der Affinität zwischen x und P. Es ergibt sich also die Heuristik, die als erstes vorgestellt wurde. Für große c hängt die Bewertung stark von der Anzahl der freien Register und den Affinitäten zu nicht zugeordneten virtuellen Registern ab.

In Abbildung 5.9 ist der Graf der Bewertungsfunktion für $a_P = 30, a_{max} = 100, c = 0.1$ zu sehen. Man kann an dem Grafen sehr gut erkennen, wie sich die Bewertung mit steigendem f, bzw. a_{sum} der oberen Grenze a_{max} nähert. Für $f = 0$ oder $a_{sum} = 0$ liefert die Bewertungsfunktion die untere Grenze a_P.

Die Bewertungsfunktion gewährleistet also, dass die Bewertung im Intervall $[a_P, a_{max}]$ liegt. Wenn fast alle virtuellen Register gefärbt sind, ist a_{sum} relativ klein. Wie man dem Grafen entnehmen kann, liegt die Bewertung in diesem Fall in der Nähe der aktuellen Affinität a_P. Wenn ein physikalischer Block kaum noch freie Register hat, liegt die Affinität ebenfalls in der Nähe von a_P. Der Grund für diese beiden Effekte ist, dass in beiden Fällen dem Block P kaum noch virtuelle Register zugeteilt werden können, die eine hohe Affinität zu x haben.

5.2 Rekonfiguration

Im vorherigen Abschnitt wurde beschrieben, wie den virtuellen Registern die physikalischen Register zugeteilt werden. Nach dieser Zuteilung ist festgelegt, in welchem physikalischen Register ein Wert gespeichert wird. Es steht aber noch nicht fest, über welches architektonische Register auf das physikalische zugegriffen wird.

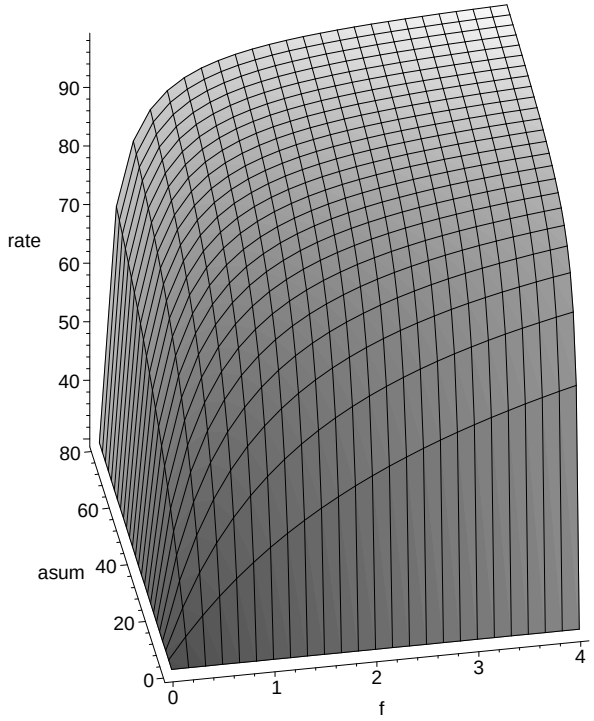

Abbildung 5.9: Graf der Bewertungsfunktion.

In der Phase, die nun beschrieben wird, werden Rekonfigurationsanweisungen in den Code eingefügt, um physikalische Blöcke einzublenden. In der Rekonfigurationsanweisung wird angegeben, in welchen architektonischen Block ein physikalischer Block eingeblendet werden soll. Somit ist auch bekannt, in welches architektonische Register ein physikalisches eingeblendet wird. Die physikalischen Registeroperanden können deshalb in dieser Phase auch durch die architektonischen ersetzt werden.

Zunächst wird im nächsten Abschnitt beschrieben, wie Rekonfigurationsanweisungen auf Grundblockebene eingefügt werden. Die dadurch hergestellten Einblendungen können jedoch noch nicht über Grundblockgrenzen hinaus verwendet werden. Um auch dies zu ermöglichen, wird im darauf folgenden Abschnitt die Festlegung von Einblendungen zwischen Grundblöcken behandelt. Zum Festlegen der Einblendungen ist es wiederum notwendig, einen maximalen Schnitt von Lebensspannen zu bestimmen. Eine Datenstruktur und Algorithmen zu diesem Zweck werden anschließend in dem Abschnitt 5.2.3 beschrieben.

5.2.1 Rekonfiguration in einem Grundblock

Zum Einfügen der Rekonfigurationsanweisungen werden die Instruktionen eines Grundblocks von der ersten zur letzten durchlaufen und dabei Informationen über die aktuell eingeblendeten physikalischen Blöcke mitgeführt. Weil am Anfang eines Grundblocks zunächst nicht bekannt ist, welche physikalischen

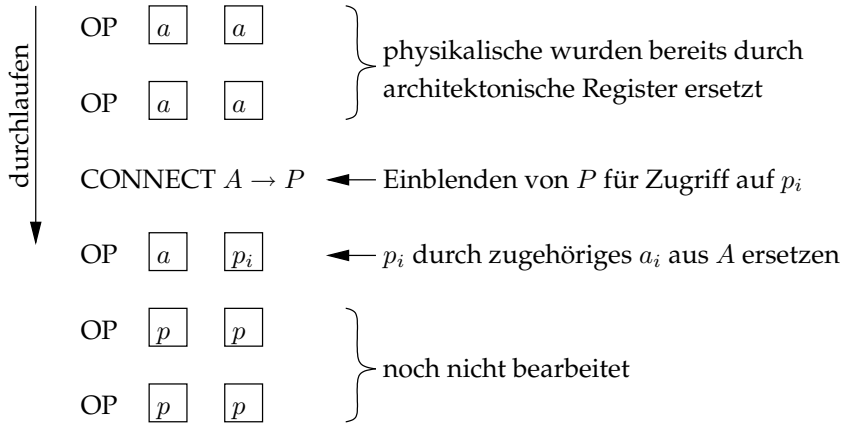

Abbildung 5.10: Einfügen von Rekonfigurationsanweisungen und Ersetzen der physikalischen durch die architektonischen Register.

Blöcke bereits durch die Vorgänger-Grundblöcke eingeblendet wurden, wird deshalb die pessimistische Annahme gemacht, dass am Anfang eines Grundblocks kein physikalischer Block eingeblendet ist.

Greift beim Durchlaufen des Grundblocks eine Instruktion auf ein Register eines physikalischen Blocks zu, der nicht eingeblendet ist, wird vor der Instruktion eine Rekonfigurationsanweisung eingefügt, die den physikalischen Block einblendet. Einblendungen werden also *so spät wie möglich* vorgenommen.

Der physikalische Block wird dabei möglichst in einen freien architektonischen Block eingeblendet. Ist dies nicht möglich, wird nach LUA der physikalische Block verdrängt, auf den am spätesten in der Zukunft erneut zugegriffen wird.

In Abbildung 5.10 ist ein Ausschnitt eines Grundblocks zu sehen. Darin wurden bereits die physikalischen Registeroperanden der ersten Instruktionen durch architektonische ersetzt. Dazu mussten keine Rekonfigurationsanweisungen eingefügt werden, weil die zugehörigen physikalischen Blöcke bereits eingeblendet waren. Für das physikalische Register p_i ist jedoch noch nicht der zugehörige physikalische Block P eingeblendet. Deshalb muss vor der dritten Instruktion eine Rekonfigurationsanweisung eingefügt werden, die P in den architektonischen Block A einblendet. Durch die Wahl von A ergibt sich das architektonische Register a_i, in das p_i eingeblendet wird. Somit kann p_i in der Instruktion durch a_i ersetzt werden.

5.2.2 Einblendungen zwischen Grundblöcken

Dadurch, dass am Anfang eines Grundblocks die architektonischen Blöcke bisher als leer angenommen werden müssen, wird in einem Grundblock für *jeden* physikalischen Block, auf den zugegriffen wird, *mindestens* eine Rekonfigurationsanweisung eingefügt. Es wird nun ein Verfahren vorgestellt, durch das Einblendungen auch über das Ende eines Grundblocks hinaus verwendet werden können.

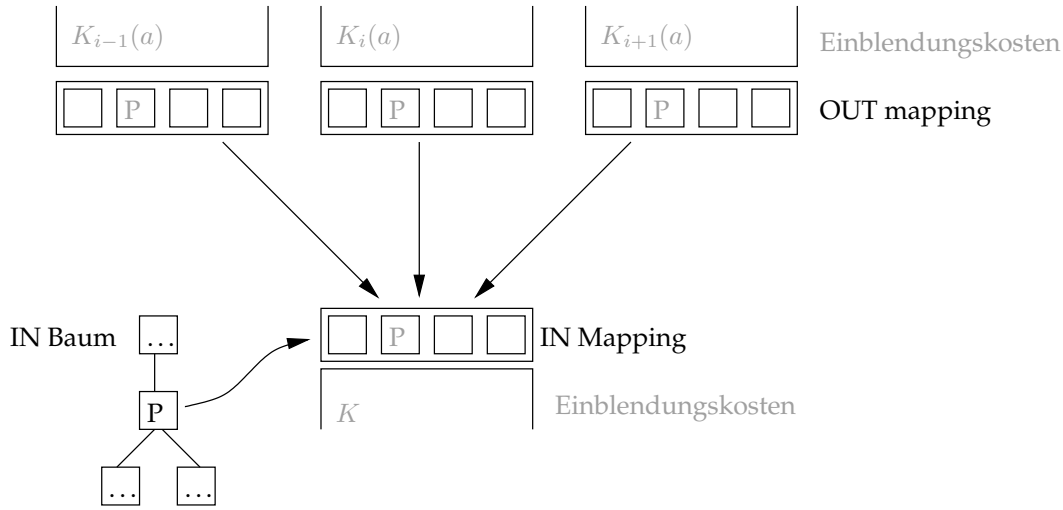

Abbildung 5.11: Festlegen von Einblendungen zwischen Grundblöcken.

Damit ein physikalischer Block P am Anfang eines Grundblocks als eingeblendet angenommen werden kann, muss P am Ende *aller* Vorgänger-Grundblöcke in *denselben* architektonischen Block eingeblendet sein. Damit dies gelingt, werden vor dem Einfügen der Rekonfigurationsanweisungen die Einblendungen zwischen den Grundblöcken durch IN- und OUT-Mappings festgelegt.

IN-/OUT-Mapping

Jeder Grundblock verfügt, wie in Abbildung 5.11 dargestellt, über ein IN-Mapping und ein OUT-Mapping. Das IN-Mapping beinhaltet alle Einblendungen, die am Anfang eines Grundblocks bestehen, also durch die Vorgänger-Grundblöcke zur Verfügung gestellt werden. Das OUT-Mapping beinhaltet alle Einblendungen, die am Ende eines Grundblocks bestehen müssen, weil sie ein Nachfolger-Grundblock erwartet. Enthält also das IN-Mapping eines Grundblocks die Einblendung von P in A, befindet sich diese Einblendung auch in allen OUT-Mappings der Vorgänger-Grundblöcke.

In der Abbildung enthält das IN-Mapping, die Einblendung des physikalischen Blocks P in einen architektonischen Block. Folglich muss diese Einblendung in den gleichen architektonischen Block auch in den OUT-Mappings der Vorgänger-Grundblöcke enthalten sein. Andernfalls währe nicht gewährleistet, dass am Anfang des Grundblocks der physikalische Block P stets in dem architektonischen Block eingeblendet ist.

Problem

Es stellt sich nun die Frage, welche Einblendungen zwischen den Grundblöcken festgelegt werden müssen, damit möglichst wenige Rekonfigurationsanweisungen zur Laufzeit durchgeführt werden.

Durch die Analyse der n-Lebendigkeit der physikalischen Blöcke kann für

den Anfang eines Grundblocks festgestellt werden, welche physikalischen Blöcke zukünftig verwendet werden. Wird also eine Einblendung für einen physikalischen Block P festgelegt, der am Grundblockanfang n-lebendig ist, wird P zwischen dem Anfang des Grundblocks und dem ersten Zugriff auf P nicht verdrängt. Durch die Festlegung der Einblendung wird in dem Grundblock also eine Rekonfigurationsanweisung eingespart. Muss für die Einblendung in den Vorgänger-Grundblöcken keine zusätzliche Rekonfigurationsanweisung eingefügt werden oder werden die Vorgänger-Grundblöcke seltener ausgeführt, werden somit zur Laufzeit weniger Rekonfigurationen durchgeführt. In einem solchen Fall ist es also sinnvoll, den physikalischen Block in das Mapping aufzunehmen.

Es können jedoch nicht beliebig viele physikalische Blöcke in das Mapping aufgenommen werden. Hat ein Grundblock mehr als einen Nachfolger-Grundblock, kann es zu einem Konflikt kommen. Wenn für die beiden Nachfolger-Grundblöcke zwei unterschiedliche physikalische Blöcke in den gleichen architektonischen Block eingeblendet werden sollen, kann nur für einen Nachfolger diese Einblendung bereitgestellt werden.

Verfahren

Es wird nun ein Verfahren beschrieben, dass die Ideen aus dem vorherigen Abschnitt verwendet, um die Einblendungen festzulegen.

Dabei werden die Grundblöcke beginnend mit dem entry Block topologisch sortiert durchlaufen. In jedem Grundblock werden für jeden physikalischen Block P des IN Baums und jeden architektonischen Block A die Kosten ermittelt, die entstehen, wenn P durch alle Vorgänger-Grundblöcke in den architektonischen Block A eingeblendet wird. Sind die so ermittelten Kosten geringer als die Kosten einer Rekonfigurationsanweisung am Anfang des Grundblocks, wird die Einblendung von P in A festgelegt, also in dem IN-Mapping des Grundblocks und den OUT-Mappings der Vorgänger-Grunblöcke eingetragen.

Wie die Kosten dabei definiert sind, wir in dem nächsten Abschnitt beschrieben.

Kosten

Die Kosten schätzen ab, wie viele Rekonfigurationsanweisungen zur Laufzeit ausgeführt werden. Die Kosten einer Rekonfigurationsanweisung am Anfang eines Grundblocks entsprechen somit der Ausführungshäufigkeit des Grundblocks. Die Kosten, die durch das Festlegen einer Einblendung entstehen, sind gleich der Summe der Kosten, die in den Vorgänger-Grundblöcken entstehen. Die Kosten in einem Vorgänger-Grundblock sind dabei…

- …gleich null, wenn die Einblendung von P in A bereits in dem OUT-Mapping enthalten ist.
- …gleich null, wenn im Grundblock auf P zugegriffen wird und die Einblendung von P bis zum Grundblockende nicht verdrängt wird.
- …gleich null, wenn sich die Einblendung von P in A im IN-Mapping befindet und P im Grundblock nicht verdrängt wird.

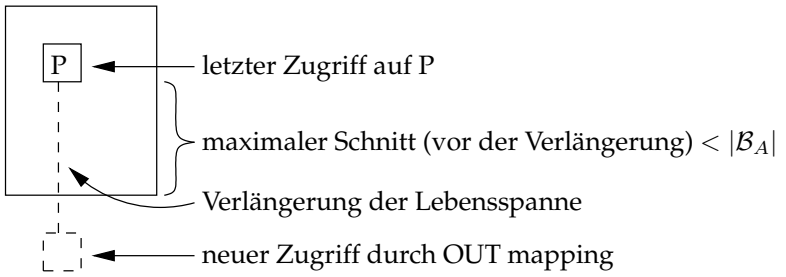

Abbildung 5.12: Verlängerung einer Lebensspanne.

- ...gleich der Ausführungshäufigkeit des Grundblocks, wenn eine Rekonfigurationsanweisung eingefügt werden muss.

Ob eine Einblendung bis zum Grundblockende verdrängt wird, wird durch eine Analyse des maximalen Schnitts bestimmt. Diese Analyse wird im folgenden Abschnitt beschrieben.

5.2.3 Maximale Schnittweite bis zum Grundblockende

Um bei der Bestimmung der Kosten entscheiden zu können, ob ein physikalischer Block P von seiner letzten Verwendung bis zum Blockende verdrängt wird, werden Informationen über den Schnitt der Lebensspannen der physikalischen Blöcke benötigt.

Die Lebensspanne eines physikalischen Blocks P innerhalb eines Grundblocks erstreckt sich vom ersten Zugriff auf P bis zum letzten Zugriff auf P. Eine solche lokale Lebensspanne kann also als Intervall dargestellt werden. Das Vorhandensein von P im IN bzw. OUT-Mapping wird wie ein Zugriff auf P vor, bzw. nach dem Grundblock behandelt. Befindet sich also ein physikalischer Block im IN und OUT-Mapping, beginnt die Lebensspanne vor dem Grundblock und endet nach dem Grundblock.

Ist der maximale Schnitt zwischen der letzten Verwendung von P und dem Grundblockende, wie in Abbildung 5.12 angedeutet, kleiner als $|\mathcal{B}_A|$, kann P in das OUT-Mapping aufgenommen werden, ohne dass P zwischen der bisher letzten Verwendung und der Verwendung im OUT-Mapping nach LUA verdrängt wird. Das Aufnehmen von P in das OUT-Mapping entspricht dem Verlängern der Lebensspanne von P bis hinter das Blockende. Der verlängerte Teil der Lebensspanne ist in der Abbildung gestrichelt dargestellt.

Weil der neue Zugriff durch das OUT-Mapping *hinter* dem Grundblockende erfolgt, werden dadurch keine physikalischen Blöcke verdrängt, auf die im Grundblock nochmals zugegriffen wird. Es sind folglich keine zusätzlichen Rekonfigurationsanweisungen notwendig.

Weil sich durch das Einfügen von P in das OUT-Mapping die Lebensspanne verlängert, erhöht sich der Schnitt von der bisher letzten Verwendung bis zum Grundblockende um eins. Weil der Schnitt in diesem Intervall zuvor kleiner war als $|\mathcal{B}_A|$, kann der Schnitt nun nicht größer sein als $|\mathcal{B}_A|$. Dies ist eine weitere Begründung dafür, dass durch die Verlängerung der Lebensspanne keine zusätzlichen Rekonfigurationsanweisungen notwendig sind.

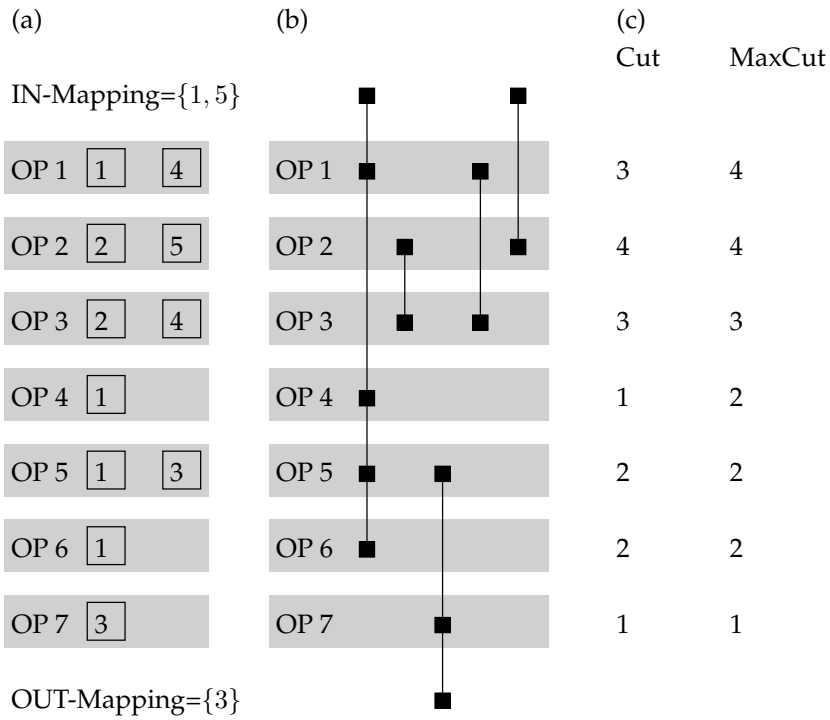

Abbildung 5.13: (a) Zugriffe auf physikalische Blöcke (b) Lebensspannen (c) Schnitt und maximaler Schnitt

In Abbildung 5.13 ist ein Beispiel für die Bestimmung der Lebensspannen und des maximalen Schnitts gegeben. In (a) sind die Instruktionen sowie das IN- und OUT-Mapping eines Grundblocks gegeben. In (b) sind die daraus resultierenden Lebensspannen der physikalischen Blöcke und in (c) der Schnitt sowie der maximale Schnitt dargestellt. Weil sich die physikalischen Blöcke 1 und 5 in dem IN-Mapping befinden, beginnen die zugehörigen Lebensspannen vor dem Grundblock. Ähnliches gilt für den physikalischen Block 3, dessen Lebensspanne hinter dem Grundblock endet. Der Schnitt in (c) ergibt sich als horizontaler Schnitt durch die Lebensspannen in (b). Der Anfang und das Ende einer Lebensspanne zählen dabei ebenfalls zum Schnitt. In Operation `OP 2` ist die Schnittbreite deshalb 4. Der Grund dafür ist, dass innerhalb einer Instruktion eine Einblendung nicht geändert werden kann. Es muss also vor der Instruktion `OP 2` der Block 2 und der Block 5 eingeblendet werden, wie auch die Blöcke 1 und 4 eingeblendet bleiben.

Der maximale Schnitt für eine Instruktion s gibt nun das Maximum des Schnitts zwischen s und dem Grundblockende an. Der maximale Schnitt steigt deshalb in der Abbildung vom Ende zum Anfang des Grundblocks monoton.

Datenstruktur

Es wird nun eine Datenstruktur vorgestellt, die die maximale Schnittweite zwischen einer Instruktion und dem Ende des Grundblocks liefert und zudem bei einer Verlängerung von Lebensspannen bis zum Grundblockende einfach aktualisiert werden kann.

Im Folgenden gebe die Funktion $MC(x)$ die maximale Schnittweite von der Instruktion x bis zum Ende des Grundblocks an. Dabei fällt MC vom Anfang zum Ende des Grundblocks monoton. Anstatt bei der Funktion für jede Instruktion die maximale Schnittweite zu speichern, wird stattdessen festgehalten, bei welchen Instruktionen sich die maximale Schnittweite ändert. Eine solche Stelle wird im Folgenden als Schnittgrenze bezeichnet. Zu diesem Zweck wird die Funktion $MCB(w)$ (MaxCutBoundary) eingeführt, die die späteste Instruktion x liefert, für die $MC(x) \geq w$ gilt. Es gilt also:

$$MC(MCB(w)) \geq w \wedge MC(MCB(w)+1) < w$$

Für den Grenzfall, dass w größer ist als der maximale Schnitt des gesamten Blocks, gelte $MCB(w) = -\infty$ und für den Fall $w = 0$ gelte $MCB(0) = \infty$. Im Folgenden wird angenommen, dass MC auf MCB reduziert wird, damit eine Änderung der Lebensspannen nur eine Änderung von MCB notwendig macht.

In Abbildung 5.16a sind die Lebensspannen aus dem Beispiel 5.13 und die daraus resultierenden Schnittgrenzen dargestellt. An der Schnittgrenze $MCB[3]$ ist der Schnitt nicht kleiner als 3 und unterhalb der Schnittgrenze kleiner als 3. Gleiches gilt für die anderen Schnittgrenzen.

Initialisierung

In diesem Abschnitt wird der Algorithmus zur Initialisierung der MCB Funktion angegeben. Die MCB Funktion wird dabei als Array implementiert.

Der Pseudocode des Algorithmus ist in Abbildung 5.14 gegeben. Darin werden in den Zeilen 1–8 zunächst für jeden physikalischen Block der erste

```
 1  FOR instr=first TO last DO
 2    FORALL physblocks OF instr DO
 3      IF first_use[physblock]==UNDEFINED THEN
 4        first_use[physblock]=instr
 5      FI
 6      last_use[physblock]=instr
 7    DONE
 8  DONE
 9
10  cut=0
11  maxcut=0
12  FORALL w DO
13    MCB[w]=-INFINITY
14  DONE
15  MCB[0]=INFINITY
16  FOR instr=last TO first DO
17    FORALL physblocks OF instr DO
18      IF last_use[physblock]==instr THEN
19        cut++
20      FI
21    DONE
22    WHILE maxcut < cut DO
23      maxcut++
24      MCB[maxcut]=instr
25    DONE
26    FORALL physblocks OF instr DO
27      IF first_use[physblock]==instr THEN
28        cut--
29      FI
30    DONE
31  DONE
```

Abbildung 5.14: Algorithmus zur Bestimmung der MCB Funktion.

```
1 w = MC(x)
2 FOR i=w DOWNTO 1 DO
3   MCB[i+1]=MCB[i]
4 DONE
```

Abbildung 5.15: Algorithmus zur Aktualisierung der MCB Funktion.

und der letzte Zugriff auf den Block bestimmt. In den Zeilen 12–14 wird die MCB Funktion mit dem Grenzfall $-\infty$ initialisiert. In der Schleife in Zeile 16 werden die Instruktionen nun von der letzten zur ersten durchlaufen. Dabei werden die Informationen der momentanen Schnittweite und der bisherigen maximalen Schnittweite mitgeführt.

Tritt beim Durchlaufen der Instruktionen in einer Instruktion das Ende einer oder mehrerer Lebensspanne auf, wird in Zeile 19 der Schnitt dementsprechend erhöht. Ist der Schnitt nun größer, als der bisherige maximale Schnitt, werden in den Zeilen 22–25 eine oder mehrere Schnittgrenzen eingeführt und der maximale Schnitt angepasst. Sollten in der Instruktion Lebensspannen beginnen, wird der Schnitt in den Zeilen 26–30 dementsprechend verringert.

Durch die Reihenfolge "Schnitt erhöhen", "Schnitt vergleichen", "Schnitt verringern" werden sowohl die Anfänge, als auch die Enden von Lebensspannen bei der Bildung des Schnitts einbezogen.

Verlängerung

Wenn eine Lebensspanne verlängert wird, braucht die MCB Funktion nicht neu berechnet werden. Stattdessen kann sie sehr effizient aktualisiert werden.

In Abbildung 5.15 ist der Algorithmus zum Aktualisieren der MCB Funktion angegeben. Dabei ist x die erste Instruktion nach der unverlängerten Lebesspanne, also die erste Instruktion, für die sich der Schnitt um 1 erhöht.

In dem Algorithmus ist w die erste Schnittgrenze oberhalb oder an der Position von x. Durch die Zeilen 2–4 werden alle Schnittgrenzen, die kleiner oder gleich w sind, um eine Position nach unten gerückt. Die Schnittgrenze w befindet sich also anschließend an der Position der ehemaligen Schnittgrenze $w-1$.

In Abbildung 5.16b wird gezeigt, wie sich die Verlängerung einer Lebensspanne auf die MCB Funktion auswirkt.

5.2.4 Einfügen von Rekonfigurationsanweisungen

Das Einfügen von Rekonfigurationsanweisungen unterscheidet sich nun in einigen wenigen Punkten von dem Einfügen aus 5.2.1.

Am Anfang des Grundblocks wird die Menge der Einblendungen nicht als leer angenommen, sondern ist gleich der Menge der IN-Mappings. Für die physikalischen Blöcke des IN-Mappings müssen also keine Rekonfigurationsanweisungen mehr eingefügt werden.

Ist durch einen Zugriff im Grundblock die Einblendung von P notwendig, wird bei der Wahl des architektonischen Blocks A sekundär darauf geachtet, möglichst den physikalischen Block gemäß dem OUT-Mapping einzublenden. Die LUA Verdrängungsstrategie wird dabei jedoch beibehalten. Diese Heuristik wird also nur verwendet, wenn dadurch kein physikalischer Block

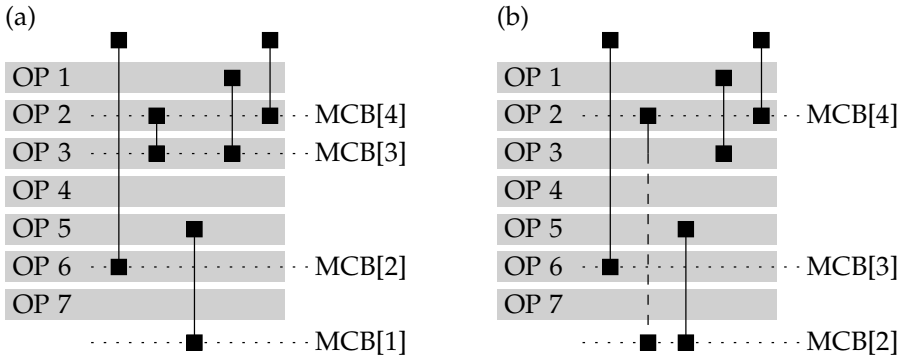

Abbildung 5.16: Auswirkung der Verlängerung einer Lebensspanne auf die Schnittgrenzen.

verdrängt wird, auf den im Grundblock nochmals zugegriffen wird. Im Folgenden sei $free$ die Menge der architektonischen Blöcke, in die *kein* physikalischer Block eingeblendet ist, auf den im Grundblock nochmals zugegriffen wird. Ist $free$ nicht leer, erfolgt die Wahl des architektonischen Blocks in dieser Reihenfolge:

1. Wenn P im OUT-Mapping in A eingeblendet ist und $A \in free$ gilt, wird A gewählt.

2. Wenn A nicht im OUT-Mapping enthalten ist und $A \in free$ gilt, wird A gewählt.

3. Ansonsten wird ein beliebiges $A \in free$ gewählt.

Am Ende des Grundblocks werden schließlich alle Einblendungen hergestellt, die im OUT-Mapping enthalten sind, aber noch nicht bestehen. Wird zum Beispiel auf einen physikalischen Block nicht zugegriffen und ist er auch nicht im IN-Mapping enthalten, wird er am Ende des Grundblocks eingeblendet.

5.3 Interprozedurale Einblendungen

Bisher wurde das Einfügen von Rekonfigurationsanweisungen auf Funktionsebene betrachtet. Welche Auswirkungen Funktionsaufrufe auf die Einblendungen haben, wurde dabei außer acht gelassen.

In diesem Abschnitt werden deshalb Einblendungen zwischen Funktionsaufrufen betrachtet. Es stellen sich nun die Fragen,

- welche Einblendungen für eine Funktion bestehen, wenn sie aufgerufen wird und

- welche Einblendungen nach einem Aufruf einer anderen Funktion bestehen.

Das Problem ist dem Retten von Registerinhalten bei Funktionsaufrufen ähnlich. Allerdings müssen Einblendungen im Aufrufer nicht vor einem Funktionsaufruf gesichert werden, weil zur Übersetzungszeit bekannt ist, welches physikalische Register in ein architektonisches eingeblendet ist.

Für einen Funktionsaufruf muss zudem festgelegt werden, in welchen Registern sich Parameter sowie Stackpointer und Rücksprungadresse befinden.

Diese Vereinbarungen könnten prinzipiell für jede aufgerufene Funktion unterschiedlich sein und von den Aufrufern abhängen. Dafür ist jedoch eine Aufrufanalyse notwendig, die nur bedingt im Compiler durchgeführt werden kann. Deshalb werden die Regeln für Funktionsaufrufe durch eine *Aufrufkonvention* festgelegt, die für alle Funktionen gilt.

Für die Aufrufkonvention stellen sich nun die folgenden Fragen, auf die im Anschluss eingegangen wird:

- Definiert die Aufrufkonvention, dass sich ein Parameter in einem speziellen *physikalischen Register* befindet oder wird festgelegt, dass sich ein Parameter in einem speziellen *architektonischen Register* befindet.
- Wie werden *Stackpointer* und *Linkregister* behandelt?
- Welche Einblendungen sind *zu Beginn einer Funktion* gegeben?
- In welchem Zustand befinden sich die Einblendungen *nach einem Funktionsaufruf*?

5.3.1 Wertübergabe

Die Aufrufkonvention muss vorgeben, in welchen *physikalischen* Registern sich die Parameter, sowie Stackpointer und Linkregister befinden. Andernfalls wäre es nicht möglich, eine Registerzuteilung für die Funktion vorzunehmen, weil zur Übersetzungszeit nicht bekannt wäre, welche physikalischen Register bereits durch die Parameter belegt sind. Außerdem könnten die physikalischen Blöcke mit den Parametern nicht verdrängen und zu einem späteren Zeitpunkt wieder einblenden werden. Dafür müsste statisch bekannt sein, um welche physikalischen Blöcke es sich handelt oder eine Instruktion zur Abfrage einer Einblendung im Instruktionssatz vorgesehen sein. Letzteres ist bei der hier betrachteten Architektur, die in Abschnitt 3.4 beschrieben wird, nicht der Fall.

Weil sich Stackpointer und Linkregister bei einem Prozessor ohne rekonfigurierbare Registerbänke stets in den gleichen Registern befinden, liegt der Gedanke nah, diese beiden Register auch stets in die gleichen architektonischen Register oder sogar permanent einzublenden. Beides ist jedoch weder notwendig, noch vorteilhaft. Wird zum Beispiel der Stackpointer häufig benutzt, bleibt er ohnehin permanent eingeblendet. An welcher Position der Stackpointer eingeblendet wird, ist dabei egal, weil er in den Operanden durch das zugehörige architektonische Register ersetzt wird. Stackpointer und Linkregister, wie auch Parameterregister, werden deshalb beim Einfügen der Rekonfigurationsanweisungen wie gewöhnliche physikalische Register behandelt. Eine Unterscheidung und besondere Behandlung dieser Register im Übersetzer ist somit nicht notwendig.

Greift eine Instruktion jedoch implizit auf den Stackpointer zu, ohne dass dieser als Operand in der Instruktion kodiert wird, kann dieser auch nicht

durch das zugehörige architektonische Register ersetzt werden. Diesem Problem widmet sich der nächste Abschnitt.

5.3.2 Implizite Registerzugriffe

Beim Einfügen von Rekonfigurationsanweisungen wurde davon ausgegangen, dass eine Instruktion nur auf die Register zugreift, die in den Operanden codiert sind. In der Praxis werden aber auch implizite Zugriffe auf Register durchgeführt. Bei einigen Prozessoren wird zum Beispiel wird bei einem Funktionsaufruf automatisch die Rücksprungadresse in das Linkregister geschrieben. Weil das Linkregister zum Beispiel mit `r15` fest vorgegeben ist, wird es nicht als Operand in den Maschinenbefehl codiert. Wenn der Sprungbefehl nun über `a15` auf das Linkregister zugreifen würde, müsste zuvor `r15` in `a15` eingeblendet werden.

Das Problem lässt sich allerdings weitaus einfacher lösen, indem der Prozessor die Rücksprungadresse nicht in `a15` schreibt, sondern den Einblendungsmechanismus umgeht und direkt auf das physikalische Register `r15` zugreift. Dabei braucht `r15` nicht eingeblendet zu sein.

Gleiches gilt auch für alle anderen impliziten Zugriffe auf physikalische Register.

Wenn im Prozessor die Einblendung der Registerbänke durch eine Tabelle implementiert wird, die architektonische auf physikalische Blöcke abbildet, dürfte diese Behandlung von impliziten Zugriffen keine Probleme bereiten.

5.3.3 Einblendungskonvention

Die Einblendungskonvention gibt an, welche Einblendungen durch den Aufrufer hergestellt werden und welche Einblendungen somit zu Beginn einer aufgerufenen Funktion gegeben sind. Ebenso wird durch die Einblendungskonvention festgelegt, welche Einblendungen nach einem Funktionsaufruf bestehen.

Zum Beispiel wäre es denkbar, dass der Aufrufer die *identische Einblendung*[1] für den Aufgerufenen herstellt oder die Parameter und den Stackpointer in definierte architektonische Blöcke einblendet. Wenn ein Zugriff auf einen dieser physikalischen Blöcke sowohl im Aufrufer, als auch im Aufgerufenen in der Nähe des Aufrufs erfolgt, wäre somit nur eine Einblendung notwendig. Auf der anderen Seiten können dadurch auch unnötige Rekonfigurationen im Aufrufer ausgeführt werden, wenn weder Aufrufer noch Aufgerufener in der Nähe des Aufrufs auf den physikalischen Block zugreifen.

Für den Rücksprung könnte die Einblendungskonvention festgelegt werden, dass die identische Einblendung hergestellt werden muss. Das Wiederherstellen der Einblendungen vom Aufrufzeitpunkt lässt sich jedoch nicht ohne erheblichen Mehraufwand realisieren.

Um ungenutzte Rekonfigurationen zu verhindern, wird deshalb im Folgenden eine Aufrufkonvention verwendet, bei der *keine* bestimmten Einblendungen beim Aufruf und beim Rücksprung bestehen müssen. Für den Aufgerufenen befinden sich die Einblendungen zu Beginn der Funktion also in einem nicht definierten Zustand. Die Einblendung der physikalischen Blöcke, also

[1] In den architektonischen Block A_i wird der physikalische Block P_i eingeblendet

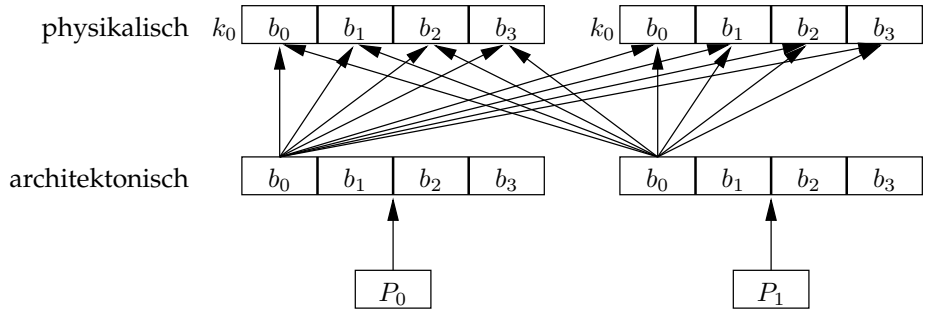

Abbildung 5.17: Registerarchitektur mit mehreren Prozessoren.

auch der Parameterregister und des Stackpointers, muss daher bei Bedarf in der aufgerufenen Funktion vorgenommen werden. Dafür braucht der Aufrufer keine zusätzlichen Einblendungen vor einem Funktionsaufruf durchführen. Nach einem Funktionsaufruf befinden sich die Einblendungen für den Aufrufer ebenfalls in einem nicht definierten Zustand. Dafür braucht die aufgerufene Funktion jedoch keine zusätzlichen Einblendungen für den Aufrufer vornehmen. Der Aufrufer ist jedoch nicht gezwungen, nach dem Funktionsaufruf die Einblendungen wiederherzustellen. Stattdessen kann der Aufrufer den Funktionsaufruf als Invalidierung der Einblendungen ansehen. Einblendungen werden dann erst bei den nachfolgenden Zugriffen auf nicht eingeblendete physikalische Register durchgeführt.

5.3.4 Implizite Einblendung bei Aufruf und Rücksprung

Wenn der Prozessor für den Funktionsaufruf und den Rücksprung separate Befehle verwendet, also keine die auch für Sprünge *innerhalb* der Funktion genutzt werden, kann der Prozessor in beiden Fällen implizit Einblendungen vornehmen. Zum Beispiel könnte beim Aufruf und beim Rücksprung die *identische Einblendung* hergestellt werden.

Die aufgerufene Funktion kann dadurch auf einen Teil der physikalischen Register zugreifen, ohne dass dafür Rekonfigurationsanweisungen ausgeführt werden müssen. Ebenso kann nach einem Aufruf auf die implizit eingeblendeten Register zugegriffen werden, ohne das dafür eine Rekonfigurationsanweisung notwendig ist.

5.4 Erweiterung für synchrone Prozessor-Cluster

In der Arbeit wurde für das Registerzuteilungsverfahren bisher nur eine Architektur mit einem Prozessor betrachtet. In diesem Abschnitt wird nun gezeigt, wie sich das Verfahren verwenden lässt, um eine Registerzuteilung für einen synchronen Prozessor-Cluster durchzuführen.

Auf die Eigenschaften des Prozessor-Clusters in Abbildung 5.17 wurden bereits in Abschnitt 3.3.5 eingegangen. Jeder Prozessor greift bei dieser Registerarchitektur über eine eigene architektonische Bank auf eine Menge von gemeinsamen physikalischen Registern zu. Zusätzlich wird vorausgesetzt, dass

die Prozessoren, wie bei einer VLIW-Architektur, die Instruktionen synchron ausführen.

Die vier Schritte der Registerzuteilung, die nun erweitert werden müssen, lauten:

1. Erstellung des Konflikgrafen
2. Erstellung des Affinitätsgrafen
3. Färbung des Konfliktgrafen
4. Einfügen der Rekonfigurationsanweisungen

Für die Erstellung des Konfliktgrafen werden parallel ausgeführten Instruktionen konzeptionell zu einer Instruktion zusammengefasst. Diese Instruktion besitzt die Operanden aller zusammengefassten Instruktionen. Aus diesem resultierenden Programm wird nun der Konfliktgraf wie bei einem Prozessor erzeugt. Dadurch werden für eine Lebensspanne die Zugriffe aller Prozessoren auf ein virtuelles Register berücksichtigt. Zudem entsteht nur ein einziger Konfliktgraf, der alle virtuellen Register enthält und später mit den gemeinsamen physikalischen Registern gefärbt wird.

Es wird folglich ebenfalls nur ein einziger Affinitätsgraf erzeugt, weil die Färbung des Konfliktgrafen für alle Prozessoren gemeinsam durchgeführt wird. Weil jeder Prozessor jedoch über eine eigene architektonische Bank zum Einblenden der physikalischen Blöcke verfügt, hat eine Rekonfigurationsanweisung auf einem Prozessor keine Auswirkung auf die Einblendungen der anderen Prozessoren. Folglich bilden Zugriffe auf unterschiedlichen Prozessoren keine Zugriffspaare. Die Affinität zwischen zwei virtuellen Registern x und y wird daher für jeden Prozessor separat bestimmt und anschließend addiert.

Die Färbung des Konfliktgrafen erfolgt, wie beim Einprozessor-Fall, mit Hilfe des Affinitätsgrafen. Änderungen ergeben sich an dieser Stelle nicht.

Weil jeder Prozessor die physikalischen Blöcke separat einblenden kann, werden die Rekonfigurationsanweisungen für jeden Prozessor gesondert eingefügt. Es wird für jeden Prozessor also der Algorithmus aus 5.2 ausgeführt. Abhängigkeiten zwischen den Prozessoren bestehen dabei nicht.

Kapitel 6

Implementierung

Neben der Diplomarbeit wurde der bestehende, in `C` geschriebene, Übersetzer des Score-Prozessors um eine Registerzuteilung für rekonfigurierbare Registerbänke erweitert. Die Registerzuteilung wurde nicht vollständig neu geschrieben, sondern an den nötigen Stellen erweitert, sodass der Übersetzer weiterhin Code für normale Registerbänke erzeugen kann. Die Erweiterungen wurden in einem *Rekonfigurationsmodul* zusammengefasst und können daher einfach auf einen anderen Übersetzer übertragen werden.

Die Arbeitsweise der Registerzuteilung kann beim Aufruf durch *Kommandozeilenparameter* beeinflusst werden. Soll Code für rekonfigurierbare Registerbänke erzeugt werden, kann über einen Parameter die Anzahl der architektonischen Blöcke und die Anzahl der Register pro Block angegeben werden.

Die Anzahl der physikalischen Register lässt sich hingegen nicht derart einfach variieren, weil davon die Definition der Registerklassen in der UPSLA-Spezifikation abhängt. Stattdessen kann `make` über einen Parameter mitgeteilt werden, ob ein Übersetzer für 8, 16 oder 32 physikalische Register gebaut werden soll.

6.1 Aufbau des Rekonfigurationsmoduls

In dem Rekonfigurationsmodul wird die *Affinitätsanalyse* für die Registerzuteilung und das *Einfügen der Rekonfigurationsanweisungen* durchgeführt. Beide Teile benötigen bestimmte Informationen über die zu übersetzende Funktion, die durch verschiede Analysen gewonnen werden.

Damit Funktionen und globale Variablen den Analysen zugeordnet werden können, werden sie schematisch benannt.

6.1.1 Benennungsschema

Jeder Funktion oder globalen Variablen wird eine Abkürzung oder kurze Bezeichnung vorangestellt, um sie zum einen zu gliedern und zum anderen Konflikte im globalen Namensraum zu vermeiden. Die verwendeten Abkürzungen sind in Tabelle 6.1 aufgelistet.

Abkürzung	Beschreibung
`rb`	Highlevel Funktionen (gehören teilweise zur Schnittstelle des Moduls)
`pbu`	Analyse der n-Lebendigkeit
`at`	Erzeugung des Affinitätsgrafen
`cut`	Maximaler Schnitt bis zum Ende eines Grundblocks
`pb`	Zugriffe auf physikalische Blöcke
`ab`	Herstellen von Einblendungen
`bbdata`	Daten an Grundblöcke binden

Tabelle 6.1: Abkürzungen im RB Modul

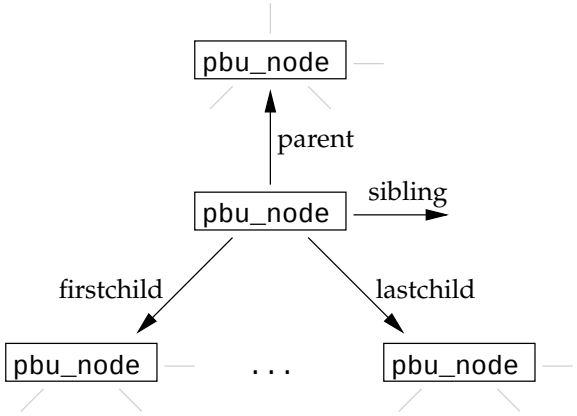

Abbildung 6.1: Verknüpfung der Baumknoten in der PBU Datenstruktur

6.1.2 Datenstrukturen

Folgende Datenstrukturen, einschließlich der Funktionen um darauf zu arbeiten, wurden im RB Modul implementiert.

Affinitätsgraf

Der Affinitätsgraf besteht aus `at_node` und `at_edge` Objekten. Jedes `at_node` Objekt besitzt eine *eindeutige* $nodeid$. Daneben existieren zwei Hashes ($nodeid \rightarrow at\_node$ bzw. $nodeid \times nodeid \rightarrow at\_edge$) um einen Knoten bzw. eine Kante aufzufinden.

Zugriffsbäume

Die Datenstruktur des Baums, in dem die n-lebendigen Register gespeichert werden, ist in Abbildung 6.1 zu sehen. Durch die Verwendung der vier Zeiger pro Knoten, können die Algorithmen effizient im Baum navigieren und Manipulationen am Baum vornehmen. Weiterhin enthält jeder Knoten des Baums die Nummer des Registers und die Zugriffswahrscheinlichkeit.

Die Baumoperationen (löschen, vereinigen, reduzieren und voranstellen) werden auf die Funktion `pbu_merge` reduziert. Diese Funktion vereinigt zwei

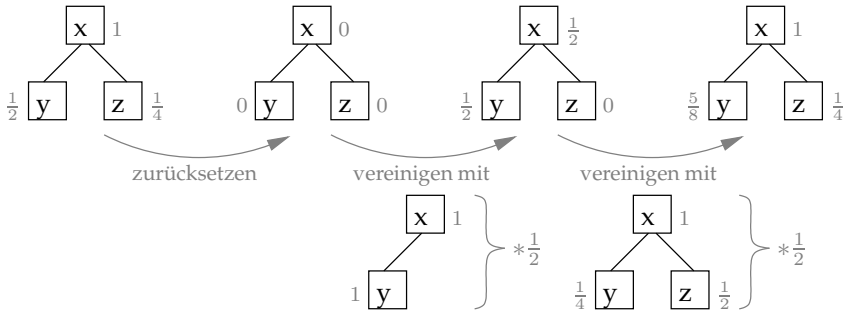

Abbildung 6.2: Berechnung des OUT Zugriffsbaums aus den IN Zugriffsbäumen der Nachfolger-Grundblöcke.

Bäume x und y, wobei eine Liste von physikalischen Blöcken angegeben werden kann, die dabei in y ignoriert[1] werden. Das Ergebnis der Vereinigung befindet sich anschließend in x. Über einen weiteren Parameter wird ein Faktor angegeben, mit dem die Wahrscheinlichkeiten in y multipliziert werden. Durch die geschickte Implementierung von `pbu_merge` werden keine Knoten in den IN und OUT Bäumen unnötig zerstört und wieder erzeugt. Während der Datenflussanalyse werden also insbesondere keine temporären Bäume erzeugt oder gelöscht und wieder neu aufgebaut. Stattdessen wird der bestehende Baum aktualisiert und gegebenenfalls um Knoten erweitert.

In Abbildung 6.2 ist ein Beispiel zu sehen, in dem die `pbu_merge` Funktion verwendet wird, um einen OUT Zugriffsbaum aus den IN Zugriffsbäumen der Nachfolger-Grundblöcke zu berechnen. Dabei werden zunächst im OUT Baum die Wahrscheinlichkeiten aus der letzten Iteration zurückgesetzt. Anschließend werden in jedem IN Baum die Wahrscheinlichkeiten durch die Anzahl der Nachfolger-Grundblöcke geteilt auf die Wahrscheinlichkeiten des OUT Baums addiert. In dem Beispiel sind im oberen Teil die Zwischenergebnisse und im unteren Teil die beiden IN Bäume dargestellt. Die Monotonie der OUT Bäume gewährleistet dabei, dass die Wahrscheinlichkeit in jedem Knoten steigt und keine Knoten aus dem Baum entfernt werden müssen.

In Abbildung 6.3 ist ein Beispiel zu sehen, in dem durch `pbu_merge` ein IN Zugriffsbaum aus dem OUT Zugriffsbaum neu berechnet wird. Dabei werden zunächst im IN Baum unterhalb der GEN Liste die Wahrscheinlichkeiten zurückgesetzt. Im Beispiel werden also den Knoten c und d die Wahrscheinlichkeit 0 zugewiesen. Anschließend wird der Teilbaum unterhalb von b mit dem OUT Baum vereinigt. Dabei werden die Knoten der GEN Liste a, b im OUT Baum ignoriert. Das ignorieren entspricht dem Löschen der beiden Knoten im OUT Baum. Weil der Knoten e des OUT Baums noch nicht im IN Baum enthalten war, wird er in den IN Baum eingefügt.

Zugriffe auf physikalische Blöcke

In dieser Datenstruktur wird für jeden physikalischen Block die Liste seiner Zugriffe innerhalb des aktuellen Grundblocks gespeichert. Die Zugriffe sind

[1] entspricht dem vorherigen Löschen der Knoten

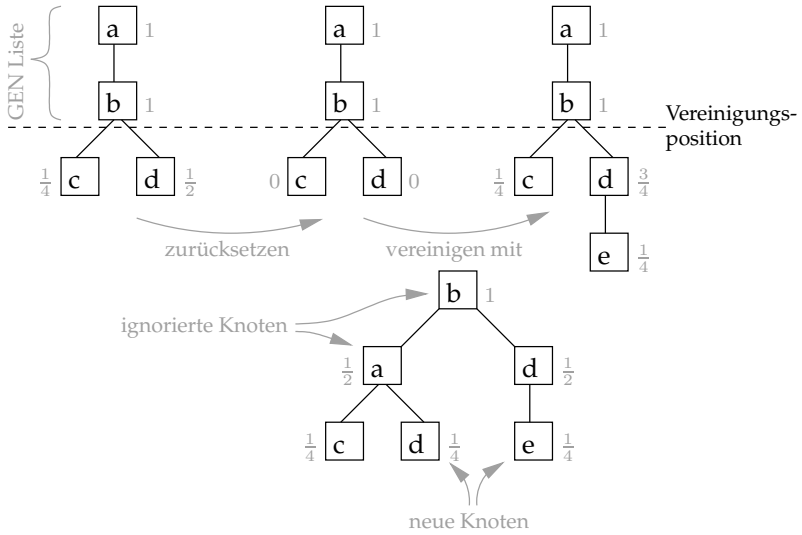

Abbildung 6.3: Berechnung des IN Zugriffsbaums aus dem OUT Baum.

dabei nach den Zugriffspositionen, beginnend mit dem ersten, geordnet. Jeder physikalische Block existiert zudem einen Zeiger auf den ersten und den letzten Zugriff in der Liste.

Der `next` Zeiger wird beim Einfügen der Rekonfigurationsanweisungen für die LUA Strategie verwendet und zeigt auf den nächsten Zugriff, der nach der jeweiligen Position im Grundblock folgt. Ist der `next` Zeiger gleich `NULL`, existiert im Grundblock kein weiterer Zugriff mehr.

Herstellen von Einblendungen

Mit den `ab` Funktionen können die Einblendungen einer architektonischen Bank verändert werden. Wenn ein physikalischer Block eingeblendet werden muss, liefert die Funktion `ab_latestused` den architektonischen Block, in

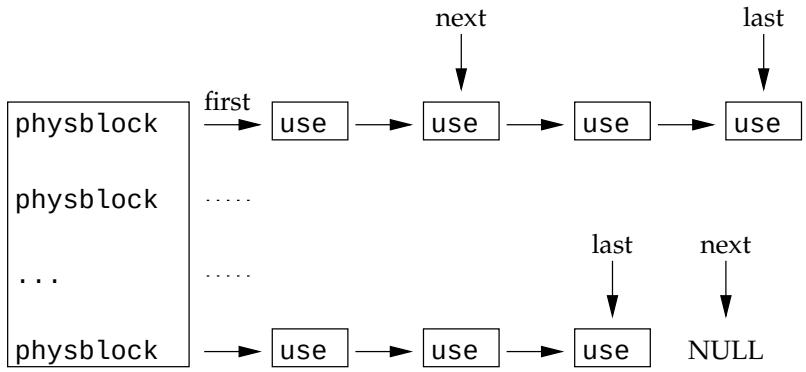

Abbildung 6.4: Zugriffe auf die physikalischen Blöcke. Die Zeiger `first`, `next` und `last` gehören zu der `physblock` Struktur.

den eingeblendet werden soll.

Grundblockdaten

Die `BBData` Funktionen ordnen jeden Grundblock eine Datenstruktur `struct rb_bbdata` zu. Um einen Zeiger auf die Datenstruktur an den Grundblock zu binden, werden die Schnittstellenfunktionen `rb_get_bbdata` und `rb_set_bbdata` verwendet. Der Typ der Datenstruktur ist dem Übersetzer nicht bekannt, sondern nur dem RB Modul. In der Funktion `bbdata_init` wird die Datenstruktur erzeugt, initialisiert und ein Zeiger darauf an den Grundblock gebunden.

Verwendung der GLib

In dem RB Modul werden die Datenstrukturen `MemoryChunk` und `HashTable` der GLib verwendet.

Ein MemoryChunk ist eine Fabrik für Speicherblöcke einer zuvor festgelegten Größe. Im Unterschied zu `malloc`, können jedoch durch einen Funktionsaufruf von `g_mem_chunk_reset` alle jemals durch den MemoryChunk erzeugten Speicherblöcke freigegeben werden. Das hat den Vorteil, dass in dem Programm nicht jeder Speicherblock durch einen Aufruf von `free` wieder freigegeben werden braucht.

Eine `HashTable` speichert eine Menge von Schlüssel-Wert Paaren. Für den Schlüssel muss bei der Initialisierung der HashTable eine Hash- und eine Vergleichsfunktion angegeben werden. Weil die HashTable nur mit Zeigern arbeitet, muss der Speicher für den Schlüssel und den Wert, zum Beispiel durch einen MemoryChunk, reserviert werden.

Beim Affinitätsgrafen wird je ein MemoryChunk verwendet, um Knoten und Kanten Objekte zu erzeugen. Um Knoten und Kanten anhand der Knotennummern nachzuschlagen, werden zwei HashTables verwendet.

6.2 Schnittstelle

Die *gesamte* Schnittstelle zwischen dem Übersetzer und dem Rekonfigurationsmodul wird in der Header-Datei `rb.h` spezifiziert. Darin befinden sich Prototypdeklarationen der Funktionen des RB Moduls, die der Übersetzer aufrufen kann. Um die Schnittstelle in der anderen Richtung zu realisieren, werden zudem Signaturen der Funktionen definiert, die der Übersetzer implementieren muss und die das RB Modul über einen Callback-Mechanismus aufrufen kann. Die wesentlichen Teile der Schnittstelle werden in Abbildung 6.5 mit Hilfe von Konzepten der Objektorientierten Programmierung dargestellt.

Vor der Initialisierung des RB Moduls muss der Übersetzer ein Callback-Objekt der `rb_callback` Struktur erstellen und den darin enthaltenen Funktionsvariablen die Zeiger auf die Callback Funktionen zuweisen. Ein Zeiger auf das Callback-Objekt wird dann beim Aufruf von `rb_init` an das RB Modul übergeben.

Der Callback Mechanismus wird in Abbildung 6.6 verdeutlicht. Dabei weist der Übersetzer zunächst den beiden Zeigern in der Callback-Struktur die beiden Funktionen zu und übergibt einen Zeiger an die Struktur an das RB Modul.

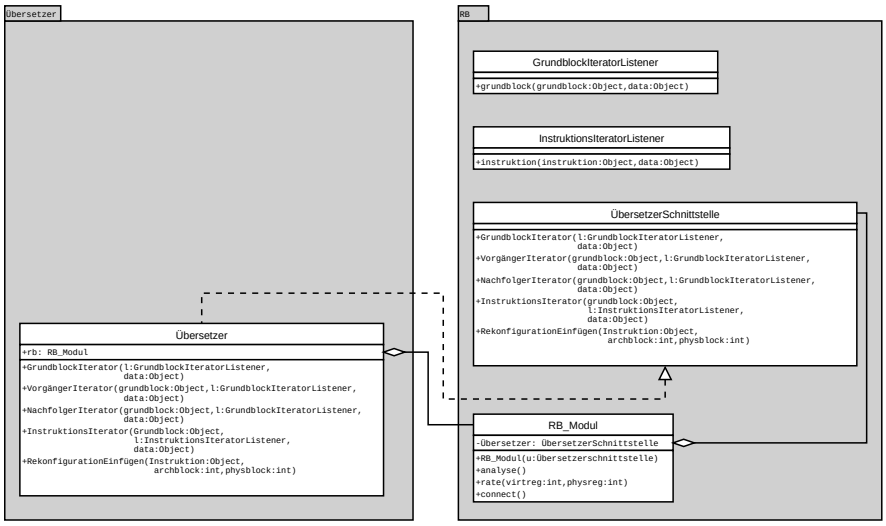

Abbildung 6.5: Grundlegender Teil der Schnittstelle zwischen dem Übersetzer und dem RB Modul, ausgedrückt durch Konzepte der Objektorientierten Programmierung.

Iterator	Iteriert über
basicblock_iterator	Grundblöcke
instr_iterator	Instruktionen eines Grundblocks
successor_iterator	Vorgänger-Grundblöcke eines Grundblocks
predecessor_iterator	Nachfolger-Grundblöcke eines Grundblocks

Tabelle 6.2: Auflistung der Iteratoren.

Das RB Modul ruft dann über den Funktionszeiger `zeiger_b` die Funktion `funktion_b` des Übersetzers auf.

6.2.1 Iteratoren

Das RB Modul hat *keine Kenntnis* über die Datenstrukturen des Übersetzers. Auf Grundblöcke und Instruktionen kann deshalb nicht direkt zugegriffen werden. Stattdessen können diese mit Iteratoren, die vom Übersetzer implementiert werden, durchlaufen werden. Die Iteratoren sind mit einer kurzen Beschreibung in Tabelle 6.2 aufgelistet.

Der Grundblockiterator[2] ruft für jeden Grundblock eine Funktion auf, die ihm als aktueller Parameter übergeben wurde. An diese Funktion übergibt der Iterator, wie in Abbildung 6.7 dargestellt, wiederum ein undurchsichtiges Handle, mit dem der Grundblock in anderen Callback-Funktionen identifiziert werden kann.

Neben der aufzurufenden Funktion erhalten die Iteratoren noch einen Zeiger (ZeigerA bzw. ZeigerB in Abbildung 6.7) als Parameter, den sie an die aufzurufende Funktion weiterreichen. Damit können auf der Seite des RB Moduls

[2] für den Instruktionsoperator gilt ähnliches

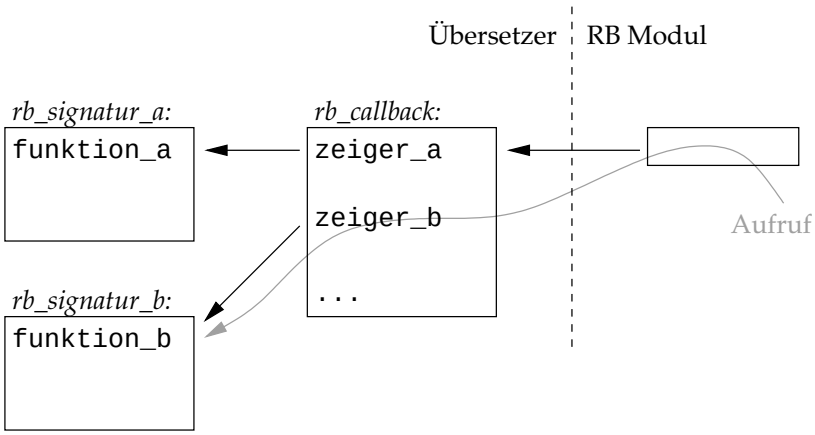

Abbildung 6.6: Callback Mechanismus. Typbezeichner sind kursiv dargestellt.

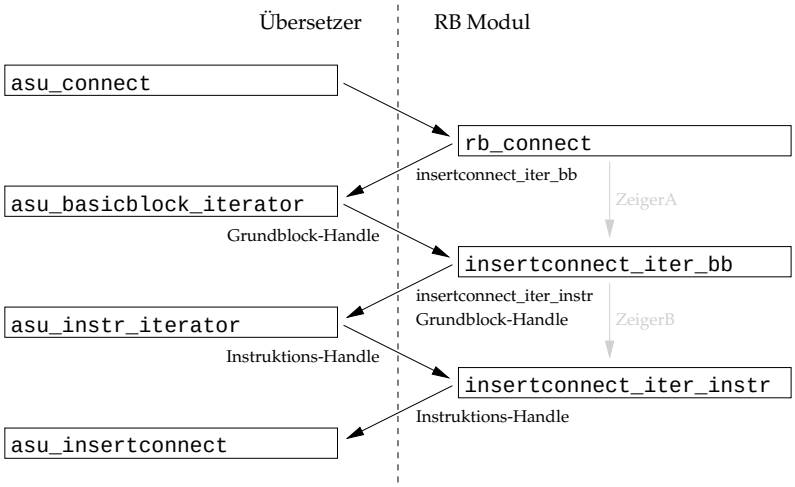

Abbildung 6.7: Aufrufkeller bei der Verwendung des Grundblock- und Instruktionsiterators.

Bits	Bedeutung
0	Rekonfiguration aktivieren
1	Analyse einschalten
2-3	Rating Methode
4	Rekonfigurationsanweisungen einfügen
5	Physikalische durch architektonische Register ersetzen
6	Einblendungen zwischen den Grundblöcken festlegen
7	Implizite Einblendungen am Funktionsanfang annehmen
8-11	Gewichtung von leeren physikalischen Registern
12-13	Register pro Block (logarithmisch)
14-16	Anzahl der architektonischen Blöcke (logarithmisch)

Tabelle 6.3: Parameter des RB Moduls.

Parameter über den Iterator hinaus übergeben werden.

6.2.2 Aufruf des Rekonfigurationsmoduls

Bevor das Rekonfigurationsmodul verwendet werden kann, muss es durch einen Aufruf von `rb_init` initialisiert werden. Dabei wird die Struktur mit den Callback-Funktionen an das RB Modul übergeben.

Während der Registerzuteilung ruft der Übersetzer dann vor jeder Färbung eines Registers die Funktionen `rb_analyse` und `rb_rate` auf, um den Affinitätsgrafen zu erzeugen und die Zuteilung eines virtuellen Registers zu einem physikalischen zu beurteilen. Anhand der Beurteilung nimmt der Übersetzer nun die Färbung des virtuellen Registers vor.

Nachdem alle virtuellen Register gefärbt sind, werden alle physikalischen Register durch einen Aufruf von `rb_connect` durch architektonische ersetzt. Abschließend werden die Datenstrukturen des RB Moduls durch einen Aufruf von `rb_destroy` zerstört.

6.2.3 Abhängigkeiten

Wenn man von optionalen Analyseausgaben durch das `visu` Modul des Übersetzers absieht, besitzt das RB Modul keine Abhängigkeiten zum Übersetzer. Es ist also schwach gekoppelt und lässt sich somit als eigenständiger Baustein einfach in andere Übersetzer integrieren.

6.3 Kommandozeilenparameter

Das RB Modul kann durch den Parameter `-b<number>` des Übersetzers beeinflusst werden. Dabei ist `<number>` ein Bitfeld mit der in Tabelle 6.3 angegebenen Struktur. Durch die Verwendung von nur einem Parameter vereinfacht sich die Schnittstelle zwischen dem Übersetzer und dem RB Modul und ein neuer Schalter kann wesentlich schneller in das RB Modul aufgenommen werden.

Kapitel 7

Evaluierung

In diesem Kapitel werden abschließend einige Simulationsergebnisse eines Prozessors mit rekonfigurierbaren Registerbänken vorgestellt.

Für die Ergebnisse wurden einige Benchmarks[1] durch den erweiterten Übersetzer in Maschinencode übersetzt und in einem Software Simulator ausgeführt. Dabei hat der Simulator die Ausführung der Programme zyklengenau simuliert, um die genaue Anzahl der Taktzyklen zu ermitteln, die ein Hardware-Prozessor für die Ausführung benötigen würde. Für eine Rekonfigurationsanweisung wurde dabei 1 Taktzyklus und für einen Speicherzugriff wurden 2 Taktzyklen angenommen.

Neben der Anzahl der Simulationszyklen wird in einigen Diagrammen auch die Anzahl der eingefügten Rekonfigurationsanweisungen dargestellt.
Während die Taktzyklen für ein gesamtes Programm angegeben werden, wird die Anzahl der eingefügten Rekonfigurationsanweisungen für jede Funktion separat angegeben.

Um eine Aussage über den Registerbedarf eines Benchmarks zu machen, wird häufig in einem weiteren Diagramm der Registerdruck angegeben. Der Registerdruck ist dabei als der maximale Schnitt der Lebensspannen in einer Funktion definiert. Bei einem Registerdruck von k enthält der Konfliktgraf somit eine k-Clique. Folglich werden mindestens k physikalische Register benötigt, um die virtuellen Register zu färben. Auch wenn der Registerdruck nur eine untere Schranke für den Registerbedarf angibt, kann damit zumindest ein Eindruck vom Registerbedarf gewonnen werden.

In den Diagrammen werden die Ergebnisse für unterschiedliche Einstellungen des Übersetzers und verschiedene Registerarchitekturen gegenübergestellt. Dabei wird zum Beispiel die Anzahl der architektonischen und physikalischen Blöcke variiert oder das Festlegen der Einblendungen aktiviert, bzw. deaktiviert. Die Einstellungen des Übersetzers werden zusammen mit den Eigenschaften der rekonfigurierbaren Registerbank im Folgenden als *Konfiguration* bezeichnet. Eine Konfiguration wird in den Diagrammen durch einen einheitlichen Schlüssel der Form

```
xA xP xR Sx [F] [C]
```

[1] Ein Testprogramm, durch dessen Ausführungsdauer die Leistungsfähigkeit einer Architektur gemessen wird.

beschrieben. Dabei steht `x` für eine Zahl. Die Buchstaben in den eckigen Klammern treten optional auf. Der Schlüssel beschreibt damit wie folgt eine Konfiguration:

xA gibt die Anzahl der architektonischen Blöcke an.

xP gibt die Anzahl der physikalischen Blöcke an.

xR gibt die Anzahl der Register pro architektonischem, bzw. physikalischem Block an.

Sx gibt die Heuristik bei der Färbung der virtuellen Register an. Dabei steht 0 für die Zuteilung gemäß der Affinitätsanalyse, 1 für eine Zuteilung in einer festen Reihenfolge und 2 für die Zuteilung eines zufälligen freien physikalischen Registers.

F gibt an, ob die Einblendungen zwischen den Grundblöcken festgelegt wurden.

C gibt an, ob beim Funktionsaufruf und Rücksprung die identische Einblendung hergestellt wird.

Ein Schlüssel `4A 8P 4R S0 F C` beschreibt somit eine Konfiguration einer Registerarchitektur mit 4 architektonischen Blöcken (`A`), 8 physikalischen Blöcken (`P`) und 4 Registern pro Block (`R`). Die Registerarchitektur besitzt also 16 architektonische und 32 physikalische Register. Beim Übersetzen des Benchmarks wurde eine Affinitätsanalyse durchgeführt (`S0`), die Einblendungen zwischen den Grundblöcken festgelegt (`F`) und implizite Einblendungen bei Funktionsaufrufen und Rücksprüngen angenommen (`C`).

Werden hingegen keine rekonfigurierbaren Registerbänke verwendet, wird statt dem Schlüssel nur die Anzahl der Register in der Form "x Register" angegeben. Für einen Prozessor mit 16 Registern wird also in der Legende "16 Register" aufgeführt.

In den folgenden Abschnitten wird nun untersucht, welche Auswirkungen jeweils

- die Anzahl der Register pro Grundblock,
- die Anzahl der physikalischen Register,
- die Affinitätsanalyse und
- das Festlegen der Einblendungen

auf die Ausführungsgeschwindigkeit eines Programms hat.

7.1 Blockgröße

In diesem Abschnitt wird untersucht, wie sich die Anzahl der Register pro Block auf die Ausführungsgeschwindigkeit auswirkt. Es werden dabei stets 16 architektonische und 32 physikalische Register verwendet.

Wie man dem Diagramm 7.1 entnehmen kann, steigt bei fast allen Benchmarks die Anzahl der Simulationszyklen mit kleiner werdenden Blöcken.

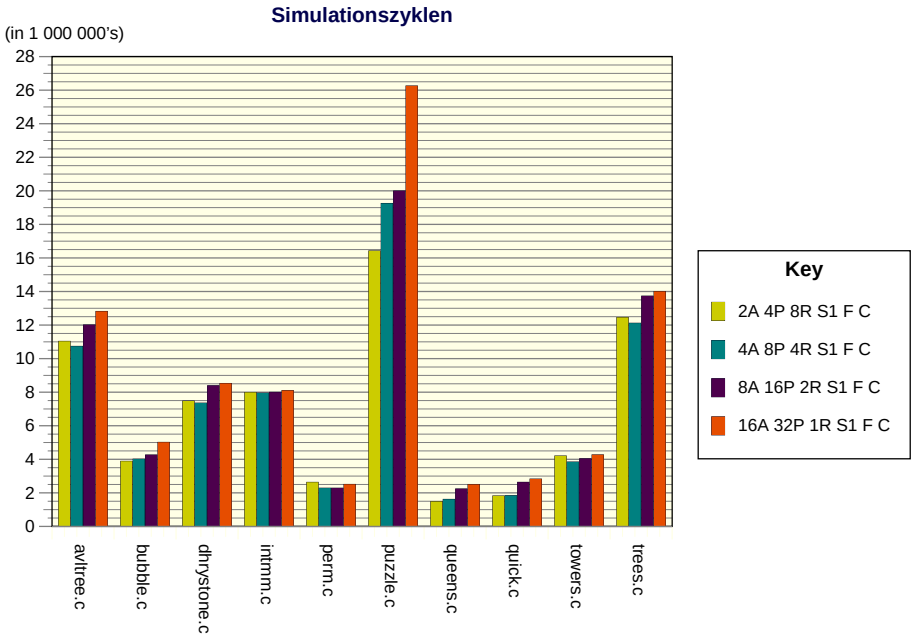

Abbildung 7.1: Anzahl der Simulationszyklen bei verschiedenen Blockgrößen.

Dieses Verhalten kann damit erklärt werden, dass mehr Rekonfigurationsinstruktionen notwendig sind, um eine bestimmte Anzahl von benötigten Registern einzublenden. Einige Benchmarks benötigen bei 4 Registern pro Block aber auch weniger Simulationszyklen, als bei der Verwendung von 8 Registern pro Block. Diese Benchmarks profitieren stärker von den flexibleren Einblendungen bei 4 architektonischen Blöcken. Am schlechtesten schneidet durchgehend die Verwendung von nur einem Register pro Block ab, weil dabei sehr viele Rekonfigurationsanweisungen eingefügt werden müssen.

7.2 Anzahl der physikalischen Register

In diesem Abschnitt wird untersucht, wie sich der Einsatz einer rekonfigurierbaren Registerbank auswirkt. Dabei wird eine normale Registerarchitektur mit 16 Registern mit einer rekonfigurierbaren Registerbank mit 16 architektonischen und 32 physikalischen Registern verglichen.

Wie man im Diagramm 7.2 sehen kann, steigt bei allen angegebenen Benchmarks die Anzahl der Simulationszyklen an, wenn rekonfigurierbare Registerbänke verwendet werden.

Der Grund dafür ist, dass die Benchmarks einen geringen Bedarf an Registern haben. Aus dem Diagramm 7.3 geht hervor, dass die meisten Funktionen der Benchmarks einen Registerdruck von weniger als 8 Registern haben. Folglich können die Programme nicht von der höheren Registerzahl profitieren. Es fallen deshalb nur die zusätzlichen Kosten durch Rekonfigurationsanweisungen ins Gewicht.

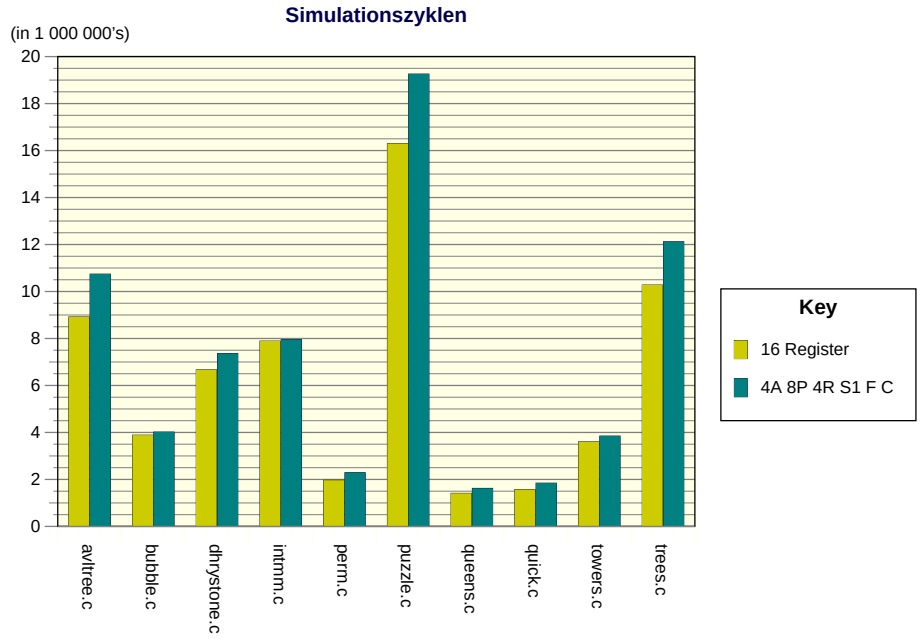

Abbildung 7.2: Anzahl der Simulationszyklen bei unterschiedlicher Anzahl von physikalischen Registern.

Die meisten Rekonfigurationsanweisungen entstehen dadurch, dass sich die nichtflüchtigen Register im Bereich 16–31 liegen. Diese Register stehen bei einem Funktionsaufruf oder Rücksprung nicht automatisch durch die *identische Einblendung* zur Verfügung. Die nichtflüchtigen Register müssen also nach jedem Funktionsaufruf wieder durch eine Rekonfigurationsanweisung eingeblendet werden. Dieses Problem könnte wahrscheinlich dadurch verringert werden, dass bei einem Funktionsaufruf und Rücksprung nicht die identische Einblendung hergestellt wird, sondern statt dessen sowohl ein Teil der nichtflüchtigen, wie auch der flüchtigen Register eingeblendet wird.

Um zu zeigen, dass durch eine Erhöhung der physikalischen Register die Ausführungszeit verringert werden kann, wurde eine Matrixmultiplikation für eine Architektur mit nur 8 architektonischen Registern simuliert. Die Ergebnisse der Simulation sind im Diagramm 7.4 dargestellt. Der linke bzw. der recht Balken gibt die Anzahl der Simulationszyklen für nicht rekonfigurierbare Registerbänke mit 8, bzw. 16 Registern an. Der Balken in der Mitte gibt die Taktzyklen für eine rekonfigurierbare Registerbank mit 8 architektonischen Registern und 16 physikalischen Registern an. Für 8 adressierbare Register konnte durch die Verdoppelung der physikalischen Register also die Ausführungsdauer um ca. 10% gesenkt werden.

Wie durch den Vergleich des linken und des rechten Balkens gesehen werden kann, wirkt sich die geringere Anzahl der Register bei diesem Benchmark nur schwach auf die Ausführungsdauer aus. Der Grund dafür ist, dass zum einen der Registerdruck, wie im Diagramm 7.5 dargestellt, nicht sehr hoch ist und zum anderen in der innersten Schleife der Matrix Multiplikation kein

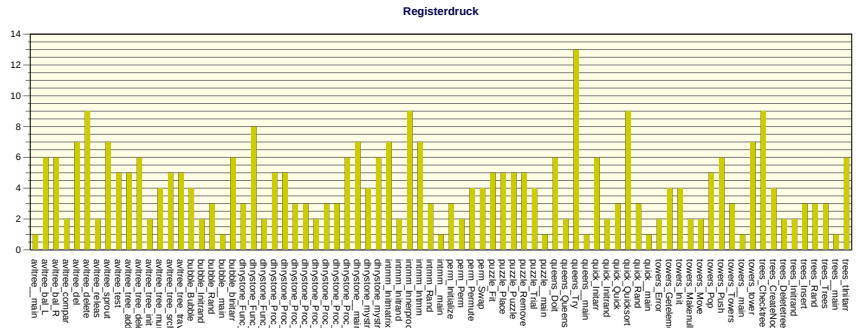

Abbildung 7.3: Registerdruck der Beispielprogramme

Spillcode auftritt.

Der Nutzen der größeren Menge an physikalischen Registern zeigt sich ebenfalls in dem folgenden Benchmark, bei dem eine Faltung berechnet wird. Bei diesem Programm wird ein Array `a` der Größe 32 mit einem Array `b` der Größe 100 gefaltet und das Ergebnis in ein Array `c` geschrieben. Es wird also die Berechnung

```
1  FOR i=31 TO 99
2    c[i]=0
3    FOR j=0 TO 31
4      c[i]+=a[j]*b[i-j]
5    DONE
6  DONE
```

durchgeführt. Für das Beispiel liegt nahe, dass die Elemente des Arrays `a` in Registern gehalten werden. Obwohl genügend physikalische Register zur Verfügung stehen, erzeugt der Übersetzer dennoch für jeden Zugriff auf `a` einen Speicherzugriff. Damit der Inhalt von `a` dennoch in Registern gehalten wird, mussten statt dem Array 32 lokale Variablen `a00,...,a31` verwendet werden.

Die Ergebnisse der Simulation sind im Diagramm 7.6 gegeben. Die Anzahl der benötigten Zyklen ist bei der rekonfigurierbaren Registerarchitektur mit 32 physikalischen Registern nun wesentlich geringer, als bei der Registerarchitektur mit nur 16 Registern. Die Architektur mit 16 Registern benötigt ca. 37000 Zyklen, wogegen die rekonfigurierbare Registerarchitektur mit ca. 28000 Zyklen um 25% schneller ist. Der relative Geschwindigkeitsunterschied würde noch stärker ausfallen, wenn die Speicherzugriffe auf die Arrays `b` und `c` nicht durchgeführt würden. Es zeigt sich bei der Architektur mit 32 Registern zudem wieder, dass die Anzahl der Simulationszyklen steigt, wenn die Anzahl der Register pro Block fällt. Bei der Faltung liegt das daran, dass bei wenigen Registern pro Block in der inneren Schleife mehr Rekonfigurationsanweisungen notwendig sind.

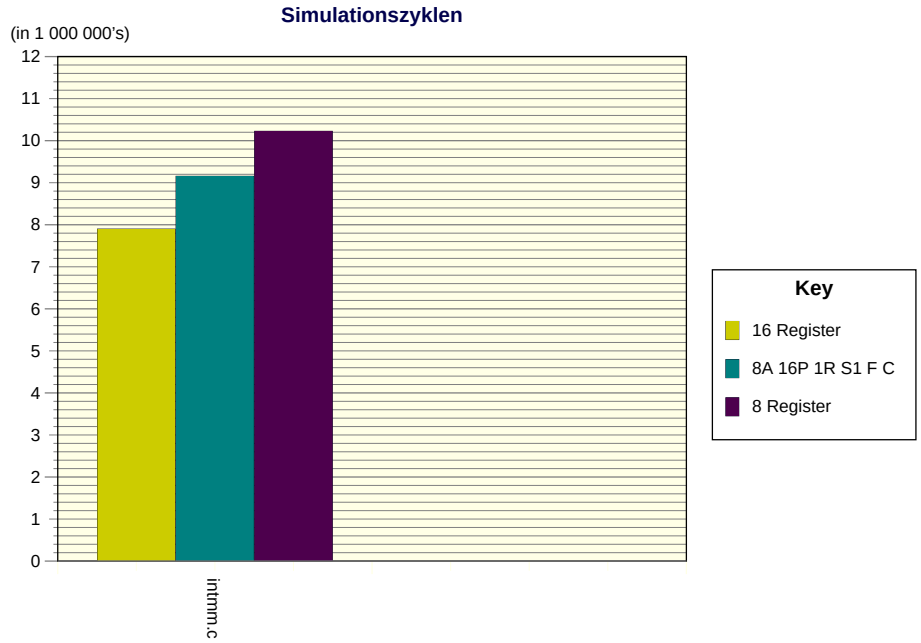

Abbildung 7.4: Simulationszyklen der Matrixmultiplikation.

7.3 Affinitätsanalyse

Um die Affinitätsanalyse zu demonstrieren, wird in einem Benchmark zunächst der Reihe nach auf 32 lokale Variablen $a_0, \ldots, a_{31}$ zugegriffen. Anschließend wird mehrfach in der Bitreversal-Permutation auf die Variablen zugegriffen. Die Zugriffsreihenfolge der Bitreversal-Permutation lautet

$$a_{rev(0)}, a_{rev(1)}, a_{rev(2)}, a_{rev(3)} \ldots$$

woraus sich für 32 Variablen die Reihenfolge $a_0, a_{16}, a_8, a_{24}, \ldots$ ergibt.

Durch diese unregelmäßige Zugriffsreihenfolge wird ein Programm mit nicht lokalen Zugriffen repräsentiert. Eine einfache Heuristik, die die Zugriffe nicht analysiert, liefert für derartige Zugriffsmuster nicht mehr zufällig eine gute Registerzuteilung.

Die Ergebnisse der zyklengenauen Simulation sind im Diagramm 7.7 zu sehen. Weil wiederholt auf die 32 lokalen Variablen zugegriffen wird, stehen diese im Konflikt miteinander. Der zugehörige Konfliktgraf enthält also eine Clique der Größe 32. Durch diesen hohen Registerdruck ergibt sich für die Architektur mit 16 Registern die hohe Anzahl von Simulationszyklen im ersten Balken.

Der zweite Balken im Diagramm gibt die Anzahl der Simulationszyklen für eine Heuristik S1 an, bei der die virtuellen Register in der Reihenfolge der Definitionen zugeteilt werden. Zudem wird dabei einem virtuellen Register stets das erste freie physikalische Register zugeteilt. Der dritte Balken gibt die Anzahl der Simulationszyklen für die Zuteilung mit Hilfe des Affinitätsgrafen an (S0). Der vierte Balken gibt die Zyklen für eine Strategie S2 an, bei der zufällig

Abbildung 7.5: Registerdruck der Matrixmultiplikation.

ein freies physikalisches Register für die Färbung gewählt wird.

Wie zu sehen ist, konnte bei diesem Benchmark die Anzahl der Rekonfigurationsanweisungen durch die Affinitätsanalyse verringert werden. Bei den übrigen Benchmarks wirkt sich die Affinitätsanalyse leider nicht so deutlich aus, weil die Zugriffe recht lokal erfolgen.

7.4 Festlegen der Einblendungen

In diesem Abschnitt wird das Festlegen von Einblendungen zwischen Grundblöcken und das implizite Herstellen von Einblendungen bei Funktionsaufrufen untersucht. Die Auswirkungen auf die Ausführungsdauer können dem Diagramm 7.8 entnommen werden.

Durch das Festlegen der Einblendungen ergibt sich bei allen Benchmarks eine deutlich kürzere Ausführungsdauer. Durch die impliziten Einblendungen bei Funktionsaufrufen verringert sich die Anzahl der Simulationszyklen ebenfalls bei den meisten Benchmarks. Bei dem Benchmark “Puzzle” steigt hingegen die Anzahl der Simulationszyklen von $18 * 10^6$ auf über $19 * 10^6$. Die Anzahl der Rekonfigurationsanweisungen verringert sich bei diesem Benchmark jedoch in allen Funktionen oder bleibt annähernd konstant. Die längere Ausführungsdauer wird deshalb vermutlich durch wenige Anweisungen in einer häufig ausgeführten Schleife verursacht.

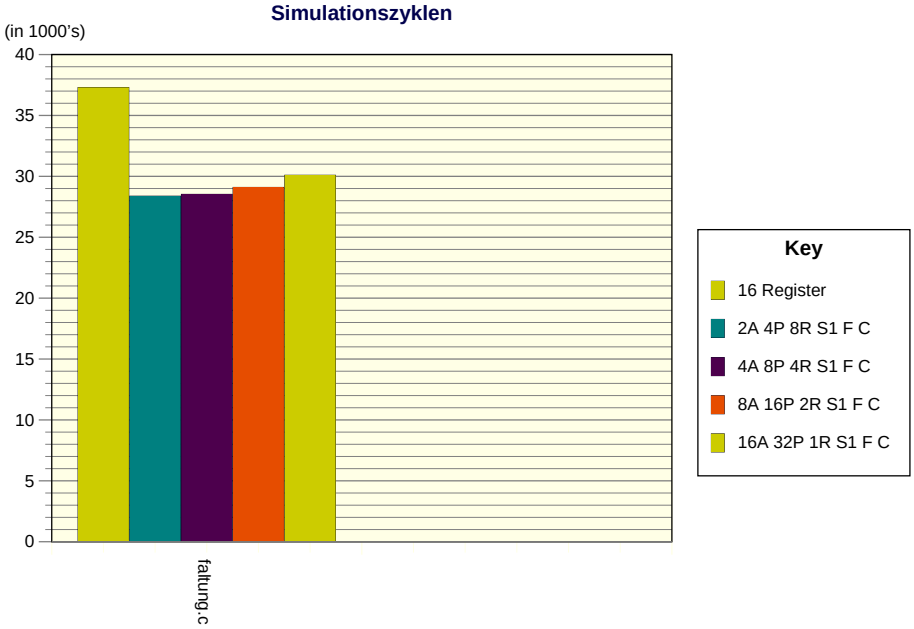

Abbildung 7.6: Anzahl der Simulationszyklen für eine Faltung.

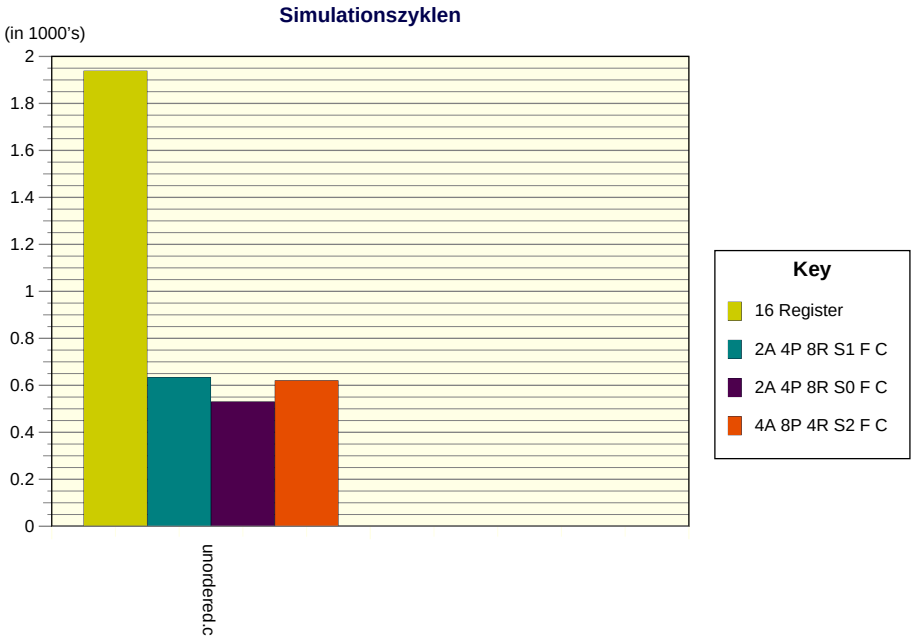

Abbildung 7.7: Anzahl der Simulationszyklen für Zugriffe in Bitreversal-Permutation.

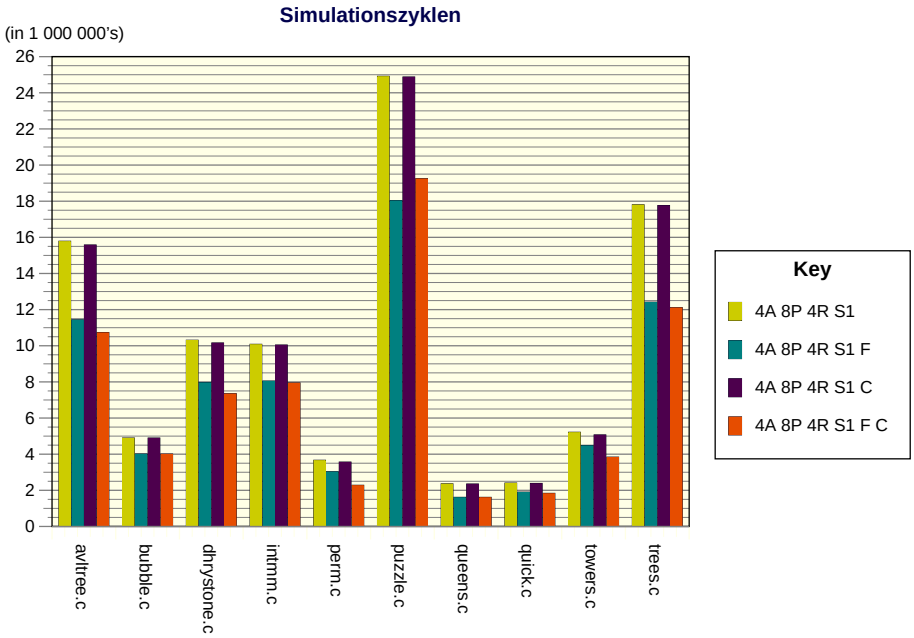

Abbildung 7.8: Anzahl der Simulationszyklen beim Festlegen von Einblendungen und impliziten Einblendungen.

Kapitel 8

Zusammenfassung

Wie sich in der Evaluierung gezeigt hat, kann die Leistungsfähigkeit eines Prozessors durch eine rekonfigurierbare Registerbank gesteigert werden. Die Laufzeit verbessert sich jedoch nur, wenn der Registerbedarf der Programme so hoch ist, dass die zusätzlichen Register genutzt werden. Dieser hohe Registerbedarf kann auch durch andere Optimierungen des Übersetzers, die die Laufzeit verringern, verursacht werden. Es sollte der Effekt der rekonfigurierbaren Registerbank daher noch einmal in Verbindung mit solchen Optimierungen untersucht werden.

Wird eine rekonfigurierbare Registerbank eingesetzt, werden durch die Analysen der Registerzuteilung deutliche Geschwindigkeitsvorteile erzielt. Durch das Festlegen der Einblendungen zwischen den Grundblöcken konnte bei den Benchmarks die Ausführungsgeschwindigkeit um 20-30% gesteigert werden.

Bei der Bitreversal-Permutation konnte die Ausführungsdauer durch die Affinitätsanalyse ebenfalls um ca. 15% gesenkt werden. Bei anderen Benchmarks fällt die Affinitätsanalyse hingegen weniger gut aus, weil die Analyse mit anderen Prozessoroptimierungen im Konflikt steht. So besitzt der Score-Prozessor zum Beispiel einen speziellen Befehl, um mehrere Register auf dem Stack zu sichern. Die Register sind jedoch nicht frei wählbar, weshalb im Übersetzer bisher die Zuteilungsreihenfolge fest vorgegeben war. Durch die Affinitätsanalyse wird diese Zuteilungsreihenfolge nun nicht mehr beibehalten und der Befehl kann deshalb kaum noch verwendet werden.

Die Verwendung von impliziten Einblendungen verbessert ebenfalls die Laufzeit der Benchmarks. Der Effekt könnte aber vermutlich noch gesteigert werden, indem statt der identischen Einblendung eine andere Einblendung verwendet wird, die auf die Registerklassen des Übersetzers abgestimmt ist.

Auch wenn an einigen Stellen noch Verbesserungen durchgeführt werden können, vermeidet das vorgestellte Registerzuteilungsverfahren schon jetzt viele Rekonfigurationsanweisungen, wodurch die Ausführung des übersetzen Programms deutlich beschleunigt wird.

Literaturverzeichnis

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[Kas90] Uwe Kastens. *Übersetzerbau*, volume 3.3 of *Handbuch der Informatik*. R. Oldenbourg Verlag, München, 1990.

[KMC+93] Tokuzo Kiyohara, Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Sadun Anik, and Wen mei W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *ISCA*, pages 247–256, 1993.

[RSM+03] Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, and Richard B. Brown. Increasing the number of effective registers in a low-power processor using a windowed register file. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 125–136, New York, NY, USA, 2003. ACM Press.