Universität Siegen

Naturwissenschaftlich-Technische Fakultät

Registerallocation

Seminararbeit

von Marcell Fischbach

Inhaltsverzeichnis

1 Einleitung **3**

2 Lokale Registervergabe **4**
2.1 Basicblock 4
2.2 Lokale Registervergabe im Detail 6

3 Global Registervergabe **8**

4 Der Graph-Coloring-Algorithmus **9**
4.1 Ablauf des Graph-Coloring-Algorithmus 9
4.2 Der Interferenzgraph 9
4.3 Färbung des Graphen 11

5 Der Linear-Scan-Algorithmus **13**
5.1 Im Detail 13
5.2 Beispiel 15
5.3 Komplexität 16

6 Vergleich der beiden Verfahren **18**
6.1 Laufzeitverhalten des Allocators 18
6.1.1 Patologischer Fall 20
6.2 Laufzeitverhalten des generierten Codes 21

7 Fazit **22**

Literatur **23**

1 Einleitung

Eine wesentliche Aufgabe eines Compilers besteht darin effizienten Code zu erzeugen. Um dies zu erreichen muss dafür gesorgt werden, dass alle Möglichkeiten, die Hardware bieten kann genutzt werden. So sind arithmetische Operationen, die mit Registern arbeiten um Größenordnungen schneller, als Operationen, die mit Speicheroperanden arbeiten. Für die Erzeugung von effizientem Code ist es daher unabdingbar, dass die vorhandenen Register sinnvoll genutzt werden. Je nach Architektur müssen Operanden in Registern vorliegen, oder die Operationen sind zumindest kürzer bzw. schneller auf Registern.

Da Register jedoch nur in begrenztem Maße vorhanden sind, muss man Verfahren anwenden, die eine möglichst optimale Zuweisung von 'Programmvariablen' zu Registern suchen. Dieser Prozess wird in zwei Schritten durchgeführt: Zum Einen wird entschieden, welcher Wert zu welcher Zeit in einem Register stehen muss und zum Anderen müssen diese vergebenen Register an physikalisch vorhandene Register gebunden werden. Zu jeder Zeit darf in jedem Register nur ein Wert gespeichert sein. Da es aber im Normalfall weit mehr Werte als Register gibt gestaltet sich eine optimale Vergabe der Register sehr aufwändig. Tatsächlich ist eine optimale Vergabe ein NP-vollständiges Problem und muss somit mit heuristischen Verfahren gelöst werden. Zwei dieser Verfahren werden in den Abschnitten 4 und 5 beschrieben.

In den folgenden Abschnitten werden die zwei Registervergabegebiete 'lokale Registervergabe' und 'globale Registervergabe' näher erläutert und die Unterschiede und deren Zusammenspiel beschrieben. Es werden zwei Verfahren für Registerallocation genauer beschrieben. Das Graph-Coloring-Verfahren, welches in vielen Compilern, die heutzutage im Einsatz sind, zur Nativecodeerzeugung verwendet wird, und das Linear-Scan-Verfahren, welches durch sein lineares und damit sehr schnelles Laufzeitverhalten auf dem Gebiet der Just-In-Time-Compiler (z.B. Java HotSpot™Client Compiler) zum Einsatz kommt.

2 Lokale Registervergabe

Die Vergabe von Registern ist eine der letzten Phasen innerhalb der Codegenerierung. Der Orignalcode wurde in einem vorangegangen Schritt in eine Zwischensprache `IL` (*Intermediate Language*) übersetzt und als Eingabe an die Registerallocation übergeben. Die `IL` besteht aus einer Liste von *Pseudoinstruktionen*, die hardwarenahe Operationen darstellen ohne auf die Feinheiten der Zielarchitektur einzugehen (dies geschieht in dem letzten Schritt der Codegenerierung, dort werden aus den *Pseudoinstruktionen* spezifische *Opcodes* für die Zielarchitektur erzeugt). Operanden bzw. Rückgabewerte werden durch *Pseudoregister* dargestellt. Dies ist eine Zwischendarstellung, bei der die Entscheidung wo sie im Arbeitsspeicher abgelegt werden oder welchem Register sie zugewiesen werden noch nicht getroffen wurde. Dies stellt exakt die Aufgabe der Registervergabe dar. Es wird theoretisch zwischen zwei unterschiedlichen Gebieten der Registervergabe unterschieden, der *lokalen* und der *globalen* Registervergabe. In der Praxis wird diese Unterscheidung jedoch meist nicht gemacht, sondern die *lokale* wird als Teil der *globalen* durchgeführt.

Die *lokale* Registervergabe wird nur innerhalb der kleinsten *Einheiten*, den sogenannten *Basicblocks*, durchgeführt. Obwohl ein solcher Block aus dieser Sicht das gesamte Programm darstellt, sind die Boundaries des Blocks beeinflussbar. So kann mittels *Boundary Constraint* an den Grenzen des Blocks bestimmt werden, welche Variablen beim Eintritt und beim Verlassen *live* sind und natürlich auch wo diese zu finden sind, in welchem Stackslot oder in welchem Register. Dies wirkt sich stark auf die Arbeitsweise des in Abschnitt 2.2 beschriebenen Algorithmus aus.

2.1 Basicblock

Da *lokale* Registervergabe nur innerhalb eines *Basicblocks* stattfindet, soll an dieser Stelle näher auf diese eingegangen werden. Unter einem *Basicblock* versteht man eine Folge von Instruktionen, die zu keinem Zeitpunkt durch einen Sprungbefehl unterbrochen werden können. Eine Ausnahme bildet das Ende eines Blocks, dort darf ein Sprungbefehl (bedingt oder auch unbedingt) auftreten. Des Weiteren darf ein *Basicblock* nur maximal eine *Sprungmarke* besitzen, zu der aus anderen Blöcken gesprungen wird. Diese Marke muss auf den ersten Befehl in dem Block verweisen. Ein Basicblock wird, wenn er ausgeführt wird, komplett ausgeführt. Weder darf die Ausführung während

```
1: i = 1
2: j = j
3: t1 = t0 * i
4: t2 = t1 + j
5: t3 = 8 * t2
6: t4 = t3 - 88
7: a[t4] = 0.0
8: j = j + 1
9: if j <= 10 goto 3
10: i = i + 1
11: if i <= 10 goto 2
12: i = 1
13: t5 = i - 1
14: t6 = 88 * t5
15: a[t6] = 1.0
16: i = i + 1
17: if i <= 10 goto 13
```

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
for i from 1 to 10 do
  a[i, i] = 1.0
```

Abbildung 1: Program zur Befüllung einer 10 $\times$ 10 Matrix als Instruktionsfolge und als Pseudocodedarstellung

des Ablaufs durch einen Sprung unterbrochen werden, noch darf durch einen Sprung in die Mitte des Blocks verwiesen werden.

In Abbildung 1 ist ein Beispiel gegeben welches auf der rechten Seite in einem *Pseudocode* eine 10×10 Identitätsmatrix erzeugt. Die linke Seite zeigt das gleiche Programm jedoch als Abfolge von einzelnen Instruktionen.

Auf die Instruktionszeilen 2, 3 und 13 wird in den Zeilen 9, 11 und 17 verwiesen, was in den Zeilen 2, 3 und 13 jeweils den Beginn eines neuen Basicblocks erfordert. Da Sprungbefehle immer die letzten Instruktionen innerhalb eines Basicblocks sind, sind in den Instruktionszeilen 10 und 12 jeweils neue Blöcke erforderlich. Dies führt zu den *Basicblocks* wie in Abbildung 2 dargestellt.

```
B1:     1: i = 1                      B2:     2: j = 1

B3:     3: t1 = t0 * i                B4:    10: i = i + 1
        4: t2 = t1 + j                       11: if i <= 10 goto 2
        5: t3 = 8 * t2
        6: t4 = t3 - 88               B6:    13: t5 = i - 1
        7: a[t4] = 0.0                       14: t6 = 88 * t5
        8: j = j + 1                         15: a[t6] = 1.0
        9: if j <= 10 goto 3                 16: i = i + 1
                                             17: if i <= 10 goto 13
B5:    12: i = 1
```

Abbildung 2: Instruktionsfolge als Aufteilung in einzelne Basicblocks

2.2 Lokale Registervergabe im Detail

Es gibt diverse Algorithmen zur *lokalen* Registervergabe, doch im Folgenden soll ein naives Verfahren [ALSU08] beschrieben werden, welches lediglich das Prinzip erläutern soll.

Während der Registervergabe muss zu jedem Zeitpunkt bekannt sein welche Variablen wo gespeichert sind. Für jedes Register beschreibt ein *Registerdeskriptor* welche Variable(n) in diesem Register gespeichert sind und für jede Variable beschreibt ein *Adressdeskriptor* an welcher Stelle im Speicher der Wert der Variable steht und ob der dort gespeicherte Wert aktuell ist, d.h. kein neuerer Wert der Variable in einem Register gespeichert ist.

Für jede Instruktion des Basicblocks müssen mögliche zusätzliche Instruktionen eingefügt werden, welche die Operanden in die entsprechenden Register laden. Betrachte man z.B. die Instruktion `x = y + z`. Für die resultierende `ADD`-Operation müssen die Variablen `y` und `z` in Register geladen werden, falls sie nicht bereits dort gespeichert sind. Dies resultiert zu einer Instruktionsfolge

```
LD Ry, y
LD Rz, z
ADD Rx, Ry, Rz
```

Das Ergebnis der Operation befindet sich nach der Operation in dem Register `Rx`.

Da Register jedoch nicht in beliebiger Anzahl vorhanden sind, passiert es das Register gewählt werden, in denen bereits ein Wert gespeichert ist. Um diese Informationen nicht zu verlieren, müssen wiederum zusätzliche Instruktionen eingefügt werden, die den Wert, vor der Überschreibung, auf den Stack auslagern. Dieser Vorgang wird als *Spilling* bezeichnet.

Die Aufgabe der *lokalen* Registervergabe besteht darin die Register so auszuwählen, dass möglichst wenig *Spill*-Instruktionen nötig sind. Betrachte man weiter die Instruktion `x = y + z`. Die Auswahl des Registers für `y` soll getroffen werden. Die Auswahl für `z` erfolgt analog.

1. Wenn `y` bereits in einem Register steht, muss weder eine Lade- noch eine Speicherinstruktion eingefügt werden. Das Register in dem `y` steht kann als Operand genutzt werden.

2. Steht `y` nicht in einem Register, es ist jedoch noch ein *freies* Register verfügbar, so kann `y` in dieses Register geladen werden und als Operand verwendet werden.

3. Dies ist der Problemfall. Weder steht `y` in einem Register noch ist ein weiteres freies Register verfügbar. Es muss somit ein potentielles Register `R` gewählt werden, welches den aktuellen Inhalt `v` enthällt.

    - Besagt der Adressdeskriptor von `v`, dass `v` noch in einem anderen Register als `R` steht, kann `R` problemlos überschrieben werden.
    - Wenn `v` der zu berechnende Wert ist (in unserem Beispiel also `x`) und nicht gleichzeitig einer der anderen Operanden (in unserem Beispiel also `z`), so wird der aktuelle Wert von `v` nicht benötigt und kann überschrieben werden.
    - Wird `v` nicht weiter verwendet, so kann auch in diesem Fall `R` mit `y` überschrieben werden.
    - Trifft keiner der drei ersten Fälle zu, so muss der Wert von `v` gespillt werden; sofern der Adressdeskriptor von `v` besagt, dass der Wert in `R` aktueller ist. Da das Register `R` durch Verschiebeoperation mehrere Werte beinhalten kann, muss dieser Wert für alle `v` gespeichert werden.

Diese Prüfung wird für alle möglichen Register durchgeführt. Das Register, für welches die wenigsten zusätzlichen Instruktionen eingefügt werden müssen, wird für die Operation ausgewählt.

3 Global Registervergabe

Ein großes Problem bei der *lokalen* Registervergabe besteht darin, dass bei der Bearbeitung eines Basicblocks nichts über dessen Grenzen hinaus bekannt ist. Wie bereits in Abschnitt 2 erwähnt, stellt der Basicblock für die lokale Registervergabe das gesamte Programm dar. Würden keine Boundary Constraints angegeben, müssten Variablen am Ende eines Basicblocks in den Arbeitsspeicher geschrieben werden. Werden genau diese Variablen zu beginn des nächsten Blocks wieder benötigt müssen die Werte wieder aus dem Arbeitsspeicher gelesen werden. Dies führt zu unnötigen *Store-Load*-Kombinationen. Betrachtet man nun nicht nur einen kleinen Ausschnitt des Programms (den Basicblock) sonder versucht eine Sicht auf eine etwas höhere Ebene zu bekommen, so führt dies zu *globaler* Registervergabe. Aus dieser Sicht ist es möglich die Boundary Constraints der lokalen Registervergabe sinnvoll zu setzen.

Wenn man den Flussgraph, also die Struktur der Abläufe der Basicblocks, kennt, so können günstigere Entscheidungen getroffen werden. Es werden Analysen über die *Liveness* der Variablen durchgeführt. Im Einzelnen wird ermittelt welche Variablen `V` am Ende eines Basicblocks `A` *live* sind und welche Variablen `W` zu Beginn eines weiteren Blocks `B` *live* sein müssen. Ist nun der Block `B` Nachfolger von `A` so ist es nicht nützlich die Variablen $V \bigcap W$ in den Arbeitsspeicher zu sichern, da diese dort sofort wieder geladen werden müssen. Man versucht also Variablen über Blockgrenzen hinweg (*global*) Registern zuzuweisen.

Aufgrund des Fakts [ALSU08], dass Programme die meiste Zeit ihres Ablaufs auf innere Schleifen verwenden, liegt ein Ansatz nahe, Variablen die über den gesamten Lauf der Schleife und somit evtl. über mehrere Basicblöcke hinweg *live* sind, in Registern zu halten. Auf diese Situation wird in Abschnitt 4.3 bei der Auswahl der zu spillenden Variablen erneut eingegangen.

4 Der Graph-Coloring-Algorithmus

Das Verfahren der Graphfärbung geht auf den Versuch zurück, alle Grafschaften der englischen Landkarte derart zu färben, dass keine zwei Grafschaften, die sich eine gemeinsame Grenze teilen, in der gleichen Farbe erscheinen. Im Jahr 1852 postulierte Francis Guthrie den Vier-Farben-Satz (in dem er eben diese Vermutung aufstellte), welcher jedoch 1890 durch John Percy Heawood widerlegt wurde, welcher seinerseits den Fünf-Farben-Satz bewies. Erst im Jahre 1976 bewiesen Kenneth Appel und Wolfgang Haken endgültig den Vier-Farben-Satz. Bemerkenswert ist, dass dieser Beweis der erste große Computergestützte Beweis war. [WPG]

4.1 Ablauf des Graph-Coloring-Algorithmus

Im Jahr 1982 entwickelte Gregory J. Chaitin [Cha04] einen Algorithmus zur Registerallocation, der auf Graphfärbung basiert. Der Graph-Coloring-Algorithmus nutzt als Basiskonstrukt einen Interferenzgraph, welcher die gegenseitigen Abhängigkeiten der Pseudoinstruktionen enthält. Dann versucht man diesen Interferenzgraph mittels Graphreduktion zu färben. Jede *Farbe* entspricht letztlich einem Register, dem die entsprechende Variable - der Knoten im Interferenzgraph - zugewiesen wird. Ist keine Färbung möglich, muss an der richtigen Stelle in der Intermediaterepräsentation entsprechender Spillcode eingefügt werden, der Variablen nicht in Registern sondern im Arbeitsspeicher hält. Abbildung 3 enthält ein detailiertes Ablaufdiagramm des Graph-Coloring-Algorithmus. Die zwei großen Teilschritte, der Aufbau des Interferenzgraphen und die Färbung des Graphen, sollen nun näher erläutert werden.

4.2 Der Interferenzgraph

Der Interferenzgraph ist die Kernstruktur für den Graph-Coloring-Algorithmus und enthält für jedes Pseudoregister des Programms einen Knoten. Da dies für komplexe Programme zu einer großen und *unhandlichen* Datenstruktur führt wird auf eine duale Repräsentation zurückgegriffen: Eine Bitmatrix und ein Nachbarschaftsvektor.

Die Bitmatrix für einen Graphen mit `N` Knoten ist eine symmetrische Matrix mit `N` × `N` Bits. Das Bit in Zeile `I` und Spalte `J` ist genau dann `1`

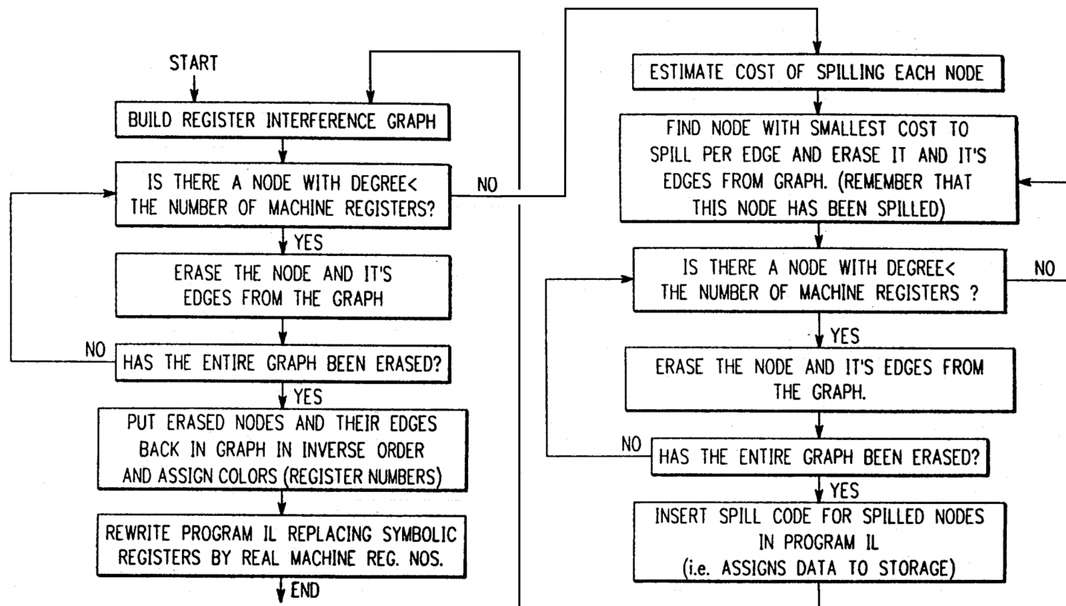

Abbildung 3: Flussdiagram des Graph-Coloring-Algorithmus

wenn es eine Beziehung zwischen den Knoten `I` und `J` gibt. Die Bitmatrix ist exzellent für einen schnellen, wahlfreien Zugriff auf den Interferenzgraphen, für eine schnelle, sequentielle Bearbeitung des Graphen ist sie jedoch zu zeitaufwändig. Zu diesem Zweck wird für jeden Knoten ein Vektor mit den Knoten gespeichert, zu denen es eine Nachbarschaftsbeziehung gibt. Die Länge dieses Vektors gibt zudem den Grad des Knoten an.

Der Algorithmus für den Aufbau des *Interferenzgraphen* erfolgt in zwei Schritten. Zuerst wird sequenziell über alle *Pseudoinstruktionen* der *IL* iteriert und es werden für alle Nachbarschaftsbeziehungen Einträge in die Bitmatrix gemacht. Nun können an allen `N` Knoten die Nachbarschaftsvektoren erzeugt und im Speicher angelegt werden. In einem zweiten Schritt wird erneut über alle *Pseudoinstruktionen* iteriert und es können an den Knoten die Nachbarschaftsvektoren befüllt werden.

4.3 Färbung des Graphen

Die Ermittlung einer Färbung des Graphen erfolgt über ein Graphenreduktionsverfahren. Es werden schrittweise Knoten aus dem Graphen entfernt bis dieser leer ist. Im Anschluss werden in umgekehrter Reihenfolge Register zugewiesen. Wie bereits zuvor erwähnt sind dies theoretisch zwei unterschiedliche Schritte. Bei der Entfernung der Knoten aus dem Graphen werden Register vergeben (1. Schritt) und im Anschluss werden diesen physikalische Register zugewiesen (2. Schritt). In einer konkreten Implementierung werden diesen beiden Schritte jedoch nicht voneinander getrennt.

Befinden sich Knoten im Graph, deren *Grad* geringer ist als die Anzahl der zur Verfügung stehenden Register, so kann dieser Knoten und alle Verbindungen zu diesem Knoten aus dem *Interferenzgraphen* entfernt werden. Der entfernte Knoten wird zusammen mit den Nachbarschaftsbeziehungen in einer Liste gespeichert, in der er später bei der Graphrekonstruktion wieder gefunden werden kann. Dieser Vorgang wird fortgesetzt solange die Bedingung gilt, dass ein Knoten im Graph existiert, dessen Grad geringer als die Anzahl der Register ist. Auf diese Weise nimmt die Größe des Graphen ab bis dieser komplett geleert wurde.

Kann zu einem Zeitpunkt jedoch kein weiterer Knoten gefunden werden auf den diese Bedingung zutrifft, würde der Algorithmus stecken bleiben. Dies wird verhindert indem Knoten aus dem Graphen entfernt werden, die im Anschluss nicht einem Register zugewiesen werden, sondern im Arbeitsspeicher abgelegt werden. Genauer gesagt werden diese Werte auf dem Stack abgelegt. Durch die Funktionsweise des Stacks ist dies insbesondere auch bei rekursiven Funktionsaufrufen möglich, da neue Werte (in unterschiedlichen Rekursionstiefen) immer wieder auf dem Stack oben aufgelegt und dort wieder abgeräumt werden können.

Die Entscheidung welcher Knoten *gespilled* wird entscheidet sich durch die *Kosten* die das Spillen eines Knotens verursacht. Bei jeder Nutzung eines Wertes, der keinem Register zugewiesen ist, müssen in dem entstehenden Code zusätzliche *Load*- und *Store*- Befehle eingefügt werden, die den Wert aus dem Speicher lesen und ihn auch wieder zurückschreiben. Da dies im Besonderen in Schleifen stark ins Gewicht fällt werden zuerst Knoten *gespilled* die möglichst wenig zusätzlichen Overhead im resultierenden Code verursa-

chen. Nachdem ein Knoten ausgewählt wurde, wird dieser ebenfalls zusammen mit allen Verbindung aus dem *Graphen* entfernt. Dieser Knoten wird als gespilled markiert. Nun ist der Graph wieder um einen Knoten veringert worden und das Verfahren kann generell fortgesetzt werden wie zuvor. Wird auf diese Weise, also mit gespillten Werten, der Graph eliminiert muss nun im Anschluss die *IL* neu geschrieben werden, denn es muss der entsprechende Spillcode erzeugt und eingefügt werden. *Pseudoregister* die in Operanden auftreten, müssen zuvor aus dem Speicher gelesen werden; d.h. es müssen vor den Operationen *Load*-Operationen eingefügt werden. Für *Pseudoregister* die als Resultat einer Operation auftreten müssen nach der Operation zusätzliche *Store*-Operationen eingefügt werden. Diese neu entstandene *IL* wird nun erneut versucht zu färben; d.h. der Algorithmus wird wieder komplett von vorne durchlaufen, beginnend mit dem Aufbau des *Interferenzgraphen*. Dies wird solange wiederholt, bis eine Färbung ohne zusätzliches Hinzufügen von Spillcode möglich ist.

Wurde nun der Graph vollständig in einem Durchlauf eliminiert, so dass kein weiterer Spillcode mehr eingefügt werden musste, kann im Anschluss die eigentliche Färbung der *Pseudoregister*, d.h. die Zuweisung zu physikalisch vorhandenen Register, erfolgen. Die zuvor aus dem Graph entfernten Knoten werden dazu nun in umgekehrter Reihenfolge wieder in den Graph eingefügt. Jedem Knoten, der nicht gespilled wurde, wird dabei ein Register zugewiesen. Dazu wird ein *freies* Register gewählt, welches keinem der Knoten aus den Nachbarschaftsbeziehungen zugewiesen ist.

5 Der Linear-Scan-Algorithmus

Der Linear-Scan-Algorithmus [PS99] benutzt eine andere Strategie als der Graph-Coloring-Algorithmus. Es wird nicht mit einem *unhandlichen* und *komplexen* Graphen gearbeitet sonder mit einer Liste von *zeitlichen Intervallen*. Diese Intervalle besagen zu welchem Zeitpunkt, bzw. innerhalb welchem Zeitraum ein *Pseudoregister live* ist. Konflikte, ähnlich den Nachbarschaftsbeziehungen bei dem Graph-Coloring-Algorithmus, ergeben sich hier wenn Intervalle überlappen. Ziel des Linear-Scan-Algorithmus besteht darin so vielen Intervallen wie möglich Register zuzuweisen. Sollte zu einer gegebenen Anzahl von `R` Register zu einem Zeitpunkt `n` Intervalle überlappen so müssen mindestens $n - R$ davon im Arbeitsspeicher abgelegt werden.

5.1 Im Detail

Der Algorithmus iteriert sequenziell über alle Live-Intervalle und versucht jedem dieser Intervalle ein Register zuzuweisen. Da dieses Verfahren verlangt, dass dies in chronologischer Reihenfolge erfolgt, liegt die Liste der Live-Intervalle in sortierter Form vor. Der Algorithmus betrachtet immer nur den Zeitpunkt zu dem ein Intervall beginnt. Es ist somit erforderlich, dass die Liste nach dem Startzeitpunkt sortiert ist.

Zu jedem Schritt existiert eine Liste `active`, die die Live-Intervalle beinhaltet, denen bereits ein Register zugewiesen wurde. Diese Liste ist chronologisch nach dem Endzeitpunkt der Intervalle sortiert. Bei der Bearbeitung eines jeden Intervalls wird zuerst die Liste `active` nach bereits *abgelaufenen* Intervallen geprüft. Alle Intervalle, deren Endzeitpunkt vor dem Startzeitpunkt des aktuellen Intervalls liegen, gelten als abgelaufen. Diese werden aus der Liste `active` entfernt, um Platz für weitere Registervergaben an andere Intervalle zu ermöglichen. Durch den Umstand, dass die Liste `active` nach dem Endzeitpunkt sortiert ist, genügt es, die Liste soweit zu durchsuchen, bis entweder das Ende der Liste erreicht ist (in diesem Falle bleibt `active` leer) oder der Endzeitpunkt eines enthaltenen Intervalls nach dem Startzeitpunkt des aktuellen Intervalls liegt. Dieses Verfahren ist unter anderem für die hohe Geschwindigkeit des Algorithmus verantwortlich.

```
linear_scan_register_allocation
  active ← {}
  foreach live interval i, in order of increasing start point
    expire_old_intervals(i)
    if length(active) = R then
      spill_at_interval(i)
    else
      register[i] ← a register removed from pool of free registers
      add i to active, sorted by increasing end point

expire_old_invervals(i)
  foreach inverval j in active, in order of increasing end point
    if endpoint[j] ≥ startpoint[i] then
      return
    remove j from active
    add register[j] to pool of free registers

split_at_interval(i)
  spill ← last interval in active
  if endpoint[spill] > endpoint[i] then
    register[i] ← register[spill]
    location[spill] ← new stack location
    remove spill from active
    add i to active, sorted by increasing end point
  else
    location[i] ← new stack location
```

Abbildung 4: Linear-Scan-Register-Allocation in Pseudocodedarstellung

Die Anzahl der in `active` enthaltenen Intervalle darf zu keinem Zeitpunkt `R` (also die Anzahl der zur Verfügung stehenden Register) überschreiten. Der *Worstcase* tritt ein, wenn zu Beginn eines Intervalls die Länge von `active = R` ist und keines der in `active` enthaltenen Intervalle *abgelaufen* ist.

In dieser Situation muss eines der *Live-Intervalle* (aus `active` oder das aktuelle Intervall) *gespilled* werden. Mehrere mögliche Heuristiken bzgl. der Auswahl des zu spillenden Intervalls sind denkbar. Hier soll an dieser Stelle jedoch auf die Basisimplementierung eingegangen werden. Es wird angenommen dass das Intervall, welches zuletzt endet, das größte Potential bietet in

Konflikt mit anderen Intervallen zu stehen. Durch die Auswahl dieses Intervall erhofft man sich somit das größte Konfliktpotential zu eliminieren. Aufgrund der Sortierung nach dem Endzeitpunkt in `active` ist dies entweder das letzte Intervall in `active` oder das aktuelle Intervall. Abbildung 4 Zeigt den Ablauf des Algorithmus in einer Pseudocodedarstellung.

5.2 Beispiel

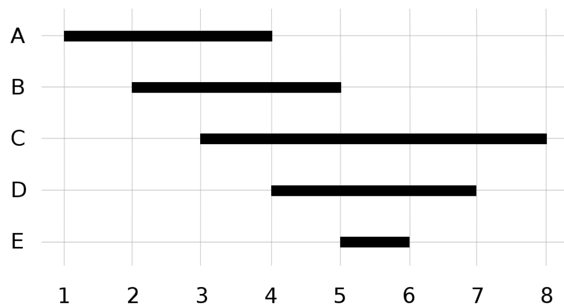

Abbildung 5: Livness der Beispielintervalle

Um die Funktionsweise des Algorithmus zu verdeutlichen soll an dieser Stelle ein kurzes Beispiel angeführt werden. Seien die folgenden fünf Intervalle `V` = {`A, B, C, D, E`} gegeben, wie Abbildung 5 darstellt. Sei weiter angenommen, dass die Anzahl der Register `R` = `2` ist. Zu Beginn des Algorithmus ist `active` = {}. Im ersten Schritt wird Interval `A` bearbeitet. Da `active` leer ist kann `A` aufgenommen werden. Im zweiten Schritt wird Intervall `B` bearbeitet. Da in `active` noch Platz für `B` ist, kann auch `B` aufgenommen werden. Am Ende von Schritt zwei ist `active` = {`A,B`} und sowohl `A` und `B` wurden Registern zugewiesen. Im dritten Schritt wird Intervall `C` bearbeitet. Da in `active` jedoch kein Platz mehr ist und keines der enthaltenen Intervalle abgelaufen ist, muss ein Intervall `gespilled` werden. Da Intervall `C` erst nach `B` endet kann `C` nicht in `active` aufgenommen werden und muss ausgelagert werden. Am Ende von Schritt drei bleibt `active` = {`A,B`} unverändert. Im vierten Schritt wird `A` aus `active` entfernt und macht Platz für weitere Allocations. Das Intervall `D` kann somit nun in `active` aufgenommen werden. Am Ende von Schritt vier ergibt sich `active` = {`B, D`}. Im fünften

Schritt wird `B` aus `active` entfernt. Das Intervall `E` kann aufgenommen werden. Dies ergibt zum Schluss `active` = {`E, D`}. Auf diese Weise wurden den Intervallen `A, B, D` und `E` Register zugewiesen. Nur das Intervall `C` wurde ausgelagert.

Hätte der Algorithmus nicht das Intervall, welches zuletzt endet, sondern welches zuerst endet, gespillt wären nur `C` und `D` Registern zugewiesen. In den ersten beiden Schritten hätten `A` und `B` in `active` aufgenommen werden können. Für `C` hätte im dritten Schritt `A` weichen müssen. Für `D` hätte im vierten Schritt dann `B` gespillt werden müssen. `E` hätte gar nicht aufgenommen werden können und hätte sofort gespillt werden müssen. Das Ergebnis der Registerallocation ist somit stark von der gewählten Heuristik abhängig, mit der das zu spillende Intervall gewählt wird.

5.3 Komplexität

Eine der Daseinsberechtigungen für den Linear-Scan-Algorithmus besteht darin, dass er eine möglichst geringe Compilezeit gewährleistet. Aus diesem Grund soll hier noch auf die Komplexität des Algorithmus eingegangen werden.

Sei V die Anzahl der Variablen, also die Anzahl der Live-Intervalle, die für Registerallocation vorgesehen sind und `R` die Anzahl der verfügbaren Register. Wie man leicht an dem obigen *Pseudocode* sehen kann hängt die maximale Länge von `active` von der Größe von `R` ab. So ergibt sich eine Komplexität von *O(V)*, unter der Annahme, dass `R` konstant ist.

Da `R` in manchen aktuellen, oder auch in zukünftigen Prozessoren erheblich größer werden kann ist es leicht zu erkennen, wie die Komplexität ebenfalls von `R` abhängt. Wenn man sich den Fakt vor Augen führt, dass die Intervalle in `active` nach dem Endzeitpunkt sortiert eingefügt werden, ist es leicht zu erkennen, dass das *Worstcase*-Verhalten des Linear-Scan-Algorithmus von der Komplexität der Einfügeoperation in `active` vorgegeben wird. Wird `active` als balancierter Binärbaum implementiert, so ergibt sich für die Suche nach dem Einfügepunkt eine Komplexität von *O(log R)*, was zu einer Gesamtkomplexität von $O(V \times \log R)$ führt. Alternativ kann `active` auch als einfache Liste implementiert werden. In diesem Fall muss linear über `active` nach dem Einfügepunkt gesucht werden. Dies führt zu einer Komplexität von *O(R)* was wiederum zu einer Gesamtkomplexität von

$O(V \times R)$ führt. Obwohl dies ein unverkennbar schlechteres Laufzeitverhalten ist, als die Nutzung eines Binärbaums, ist sie für genügend kleines `R` die bessere Wahl. Dies lässt sich einfach erklären, da die Nutzung eines Binärbaums bei kleiner Größe unnötigen, zusätzlichen Overhead mitbringt, der den Algorithmus wiederum ausbremst.

6 Vergleich der beiden Verfahren

In den vorangegangen zwei Abschnitten wurden zwei Algorithmen näher beschrieben, die mit dem gleichen Ziel arbeiten, dies jedoch auf zwei unterschiedliche Weisen tun. Der Graph-Coloring-Algorithmus baut aus dem *Pseudocode* einen Graph und versucht für diesen eine Färbung zu finden, sodass möglichst vielen Knoten ein Register zugewiesen werden kann. Aufgrund der Komplexheit dieses Graphen sind auch mehrere Durchläufe und Umschreibungen des Codes nötig. Der Liner-Scan-Algorithmus erzeugt aus dem *Pseudocode* eine Liste mit Intervallen und weist diesen in einem linearen Lauf Register zu.

Grundsätzlich müssen zwei unterschiedliche Bereiche auf Performance und Qualität geprüft werden.

1. Laufzeitverhalten des Allocators (Compilezeit)

2. Laufzeitverhalten des generierten Codes

Alle im Folgenden aufgeführten Auswertungen und Testberichte basieren auf den Untersuchungsergebnissen der Entwickler des Linear-Scan-Algorithmus *Massimiliano Poletto* und *Vivek Sarkar* [PS99].

6.1 Laufzeitverhalten des Allocators

Das wichtigste Kriterium des Linear-Scan-Algorithmus besteht in seiner schnellen, linearen Laufzeit. Im Folgenden soll das Laufzeitverhalten des Linear-Scan-Algorithmus mit dem des Graph-Coloring-Algorithmus verglichen werden. Zusätzlich beinhaltet dieser Vergleich einen weiteren Registerallocator, der auf Use-Case-Verhalten beruht. D.h. die Variablen, die am häufigsten benutzt werden, werden in Registern gehalten, alle anderen nicht.

Das Diagramm aus Abbildung 6 zeigt das Laufzeitverhalten gemessen mit unterschiedlichen Benchmarks. Der `tcc` [tcc] Compiler wurde zu diesem Zweck um die Nutzung des Linear-Scan-Algorithmus erweitert. Die kompilierten Programme sind diverse Varianten von Matrixmultiplikationen, Sortierung von Listen, etc.

Die vertikale Achse spiegelt den Kompilieraufwand gemessen in Zyklen pro generierter Instruktion wider. Je größer der Balken, desto größer ist auch

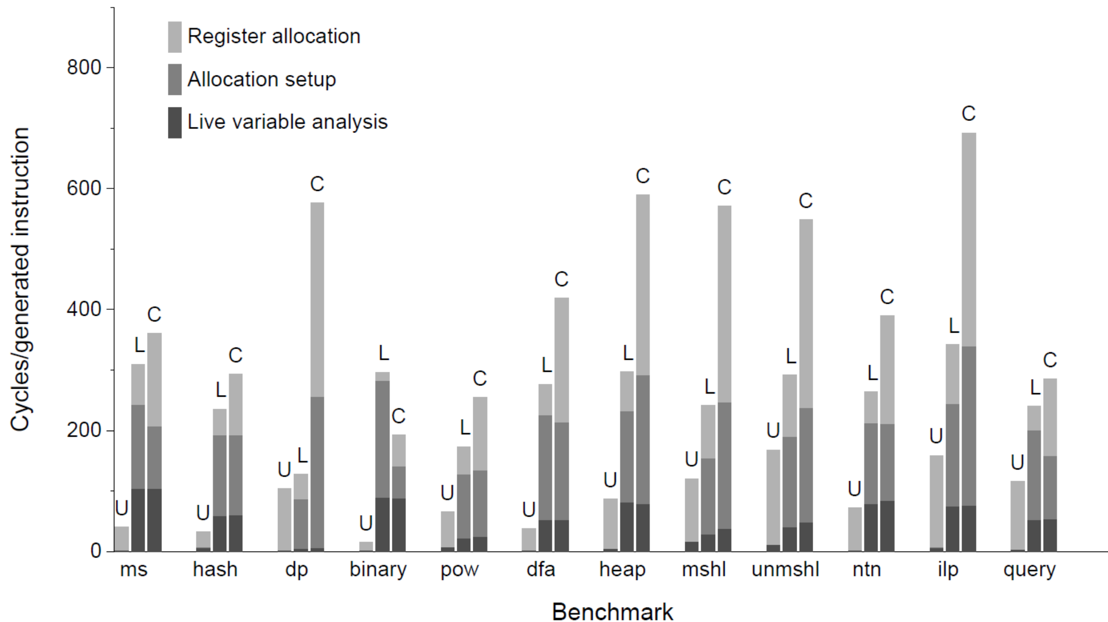

Abbildung 6: Registerallocation Overhead gemessen mit unterschiedlichen Registerallocation Algorithmen

die Compilezeit. Für jeden Benchmark sind drei Balken angegeben, die jeweils unterschiedliche Strategien der Registerallocation repräsentieren. `U` steht für den *Use-Case-Algorithmus*, `L` steht für den *Linear-Scan-Algorithmus* und `C` steht für den *Graph-Coloring-Algorithmus*. Des Weiteren ist jeder Balken in drei Segmente unterteilt, die sich wie folgt zusammensetzen:

1. **Live variable analysis**: Analyse der Liveness der Variablen. Wird in **U** nicht benötigt und taucht dort auch nicht auf.

2. **Allocation setup**: Vorarbeit, die geleistet werden muss bevor die Registerallocation durchgeführt werden kann. Im Falle von `L` bezieht sich das auf den Aufbau der Intervalle und im Falle von `C` bezieht sich das auf den Aufbau des Interferenzgraphen. Auch dieser Teil betrifft `U` nicht.

3. **Register allocation**: Im Falle von `U` bezieht sich dies auf Sortierung der Variablen nach Häufigkeit der Nutzung und der Registerzuweisung.

Abbildung 7: Patologischer Fall mit `n` zeitgleichen Intervallen

Im Falle von `L` bezieht sich das auf die Auswertung der Intervalle und bei `C` bezieht es sich auf die Färbung des Graphen.

An Abbildung 6 kann man erkennen, dass der Linear-Scan-Algorithmus in allen Fällen, bis auf eine Ausnahme, erheblich schneller ist als der Graph-Coloring-Algorithmus. Obwohl die Vorarbeit, der Aufbau der Intervalle bzw. des Interferenzgraphen, in beiden Algorithmen in etwa gleiche Zeit benötigt, ist der tatsächliche Aufwand der Registerallocation in allen Fällen bei dem Linear-Scan-Algorithmus erheblich schneller. Bei der einen Ausnahme handelt es sich um den *binary* Benchmark, der mit sehr wenigen Variablen, jedoch einer großen Menge an Basicblöcken arbeitet. In diesem Fall ist es schneller den relativ kleinen Interferenzgraphen zu erzeugen als die Liveintervalle aus allen Basicblöcken zu extrahieren. Aber auch hier ist die Durchführung der Registerallocation im Linear-Scan-Algorithmus wieder erheblich schneller als im Graph-Coloring-Algorithmus.

6.1.1 Patologischer Fall

Folgender Test wurde ebenfalls als Stresstest mit dem Linear-Scan und dem Graph-Coloring-Algorithmus durchgeführt. Die Idee liegt nicht darin ein Programm zu kompilieren, dass etwas Sinnvolle tut, sondern ein Programm zu konstruieren, das den Registerallocator belastet. Man versucht ein Programm zu kompilieren, welches `n` überlappende Intervalle beinhaltet wie Abbildung 7 zeigt.

Die Abbildung 8 zeigt das Laufzeitverhalten von beiden Algorithmen. Beide Achsen sind logarithmisch aufgebaut, die horizontale Achse stellt die Anzahl der gleichzeitigen Live-Variablen dar, während die vertikale Achse die Bearbeitungszeit darstellt. Obwohl beide Algorithmen sich bei genügend kleinem `n` annähernd gleich verhalten skaliert der Linear-Scan-Algorithmus er-

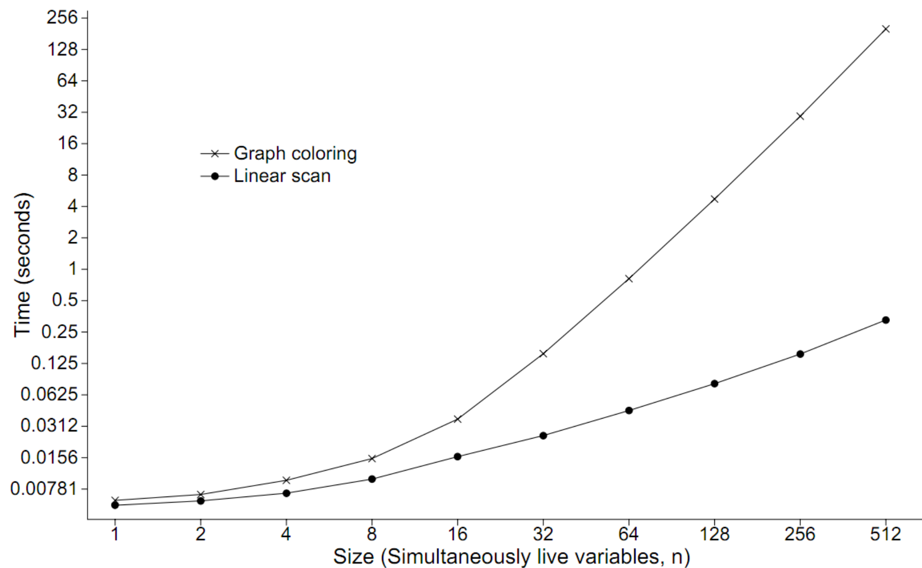

Abbildung 8: Overhead des Graph-Coloring-Algorithmus und des Linear-Scan-Algorithmus als Funktion der Anzahl von gleichzeitigen Livevariablen

heblich besser. Ab einer Größe von `n = 512` ist der Linear-Scan-Algorithmus über 600 mal schneller als der Graph-Coloring-Algorithmus.

6.2 Laufzeitverhalten des generierten Codes

Die Abbildung 9 zeigt das Laufzeitverhalten des generierten Codes, getestet mit mehreren Benchmarks. Auch hier ist die vertikale Achse logarithmisch aufgebaut und zeigt die Laufzeit der Programme in Sekunden. Hier kann man gut den Vergleich zwischen den drei hier vorgestellten Verfahren anstellen. Während die Programme, die mit dem Use-Case-Algorithmus kompiliert wurden erheblich langsamer laufen als die Anderen, können die mit Liner-Scan-Algorithmus kompilierten Programme nahezu mit denen, mit dem Graph-Coloring-Algorithmus kompilierten Programmen gleichziehen. Im Durchschnitt sind die Linear-Scan-Programme ca. 10% langsamer als die Graph-Coloring-Programme.

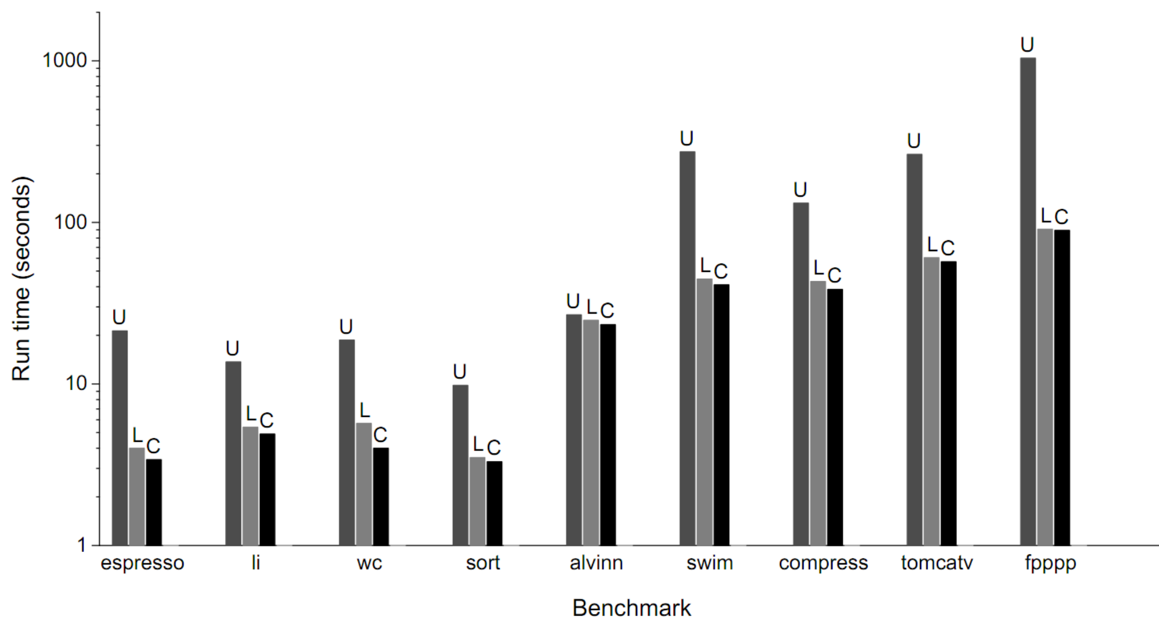

Abbildung 9: Laufzeitverhalten von Benchmarks, kompiliert mit unterschiedlichen Registerallocation Algorithmen

7 Fazit

Die Entscheidung welcher Algorithmus in einem Compiler eingesetzt werden soll ist natürlich stark von dem Einsatzgebiet abhängig. Ist die oberste Prämisse die Erzeugung von optimalem und extrem schnellem Code kann man um die Nutzung eines Graph-Coloring-Algorithmus nicht herumkommen. Der resultierende Code ist in allen Benchmarks schneller.

Wird der Compiler in einer Echtzeitumgebung eingesetzt wo zur Laufzeit von Programmen weitere Programme kompiliert werden müssen, wie dies z.B. in einem Just-In-Time-Compiler der Fall ist, ist der Linear-Scan-Algorithmus vermutlich eine bessere Alternative da der resultierende Code nicht viel langsamer ist als der des Graph-Coloring-Algorithmus, die Compilezeit jedoch erheblich besser ist. Tatsächlich wird der Linear-Scan-Algorithmus in dem Java HotSpot™Client Compiler eingesetzt [Wim04].

Literatur

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.

[Cha04] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39:66–74, April 2004.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, September 1999.

[tcc] tcc, `http://bellard.org/tcc/`; 2011-01-23.

[Wim04] Christian Wimmer. Linear scan register allocation for the java hotspot client compiler, 2004.

[WPG] Graph coloring, `http://en.wikipedia.org/wiki/Graph_coloring`; 2011-01-23.