Vorlesung
Code–Erzeugung
für moderne Mikroprozessoren

1 Motivation

Moderne Prozessoren werden immer komplexer: steigende Anzahl von Funktionseinheiten, größere Registerzahlen, parallelarbeitende Funktionseinheiten, komplexe Datenpfade. Gewisse Steuerungen werden durch die Hardware unterstützt: ‘cache’s, Instruktionsreihenfolge, Auswahl der Funktionseinheiten.

Gute Code–Erzeugung wird immer schwieriger. Der Compiler muß die Eigenschaften der Zielmaschine hinreichend kennen, um effizienten Code erzeugen zu können. Oft ist die Compilerstruktur so angelegt, daß zuerst Code für eine abstrakte Maschine, die möglichst universell sein soll, erzeugt wird. Dieser „abstrakte Maschinecode“ wird in einem zweiten Lauf erst in den Maschinencode des Zielprozessors transformiert. Die hier vorgestellten Zielprozessoren sollen helfen die abstrakte Maschine auch in Hinblick auf zu erzielende Effizienz zu definieren.

Teile der Code–Erzeugung sind $\mathcal{NP}$–vollständig. Es gibt Heuristiken, um trotzdem in angemessener Zeit zufriedenstellende Lösungen zu finden. Uns interessieren unter anderem solche Fragen: Welche Ideen stecken dahinter? Warum sind sie manchmal ineffizient? Warum kann man oft von Hand leicht Verbesserungen in von Compilern erzeugtem Code vornehmen?

1.1 Mikroprozessorentwicklung

Kurze Zusammenfassung aus [23] über die Entwicklung und den aktuellen Stand der Technik.

1.1.1 Leistung

Mikroprozessoren erfahren eine jährliche Leistungssteigerung von ca. 50%.

Zwei entscheidende Ursachen sind:
(1) schnellere on–chip Komponenten (Transistoren, Gatter), sowie
(2) eine größere Anzahl von Komponenten auf einem Chip.
Ersteres erlaubt höhere Schaltfrequenzen (300 MHz). Zweiteres kann für mehr Register und Speicher genutzt werden, womit ‘pipelining’ besser genutzt werden kann und Zugriffe auf den oft langsamen Hauptspeicher eingespart werden können.

Die Entwicklung der Chipfläche zeigt folgende Tabelle:

Jahr	Leiterbahnbreite	Fläche
1960	$50\mu m$	$4mm^2$
1994	$0.5\mu m$	$290mm^2$

Die Entwicklung der Transistorzahlen in CMOS–Technik zeigt folgende Tabelle:

Jahr	Prozessor	Anzahl Transistoren
1971	i4004	2.200
1994	DEC21164	9.300.000

(1) wirkt aber auch (2) entgegen, so sind GaAs–Chips zwar schneller, aber auch größer. Eine weitere Konsequenz der Verkleinerung liegt darin, daß komplexe bzw. viele Funktionseinheiten auf einem Chip realisiert werden können.

Die Ausführungszeit T eines Programms ist gegeben durch

$$T = I \cdot \hat{c}/f$$

wobei I die Anzahl der Instruktionen, $\hat{c}$ die mittlere Zyklenzahl pro Instruktion ('cycles per instruction' [**?**]) und f die Taktfrequenz ist. $\hat{c}$ kann durch parallele Instruktionsbearbeitung kleiner als 1 gemacht werden. I und $\hat{c}$ stehen im 'trade–off': CISC ('complex instruction set computer'), wenige komplexe Instruktionen; RISC ('reduced instruction set computer'), viele einfache Instruktionen.

Es besteht das Problem: ein Chip hat nur eine gewisse Menge Pins, über die nicht beliebig schnell Daten transportiert werden können. Dies bezeichnet man als VONNEUMANN–Flaschenhals: Operanden und Instruktionen müssen über Pins zum/vom Prozessor gebracht werden.

Die Auswirkungen des Flaschenhalses können durch Ausnutzen von Lokalität und on–chip Speicher gemildert werden. Programme zeigen (oft) die angenehmen Eigenschaften einer zeitlichen Lokalität, d. h. auf die gleichen Daten wird häufiger und in kurzen Abständen zugegriffen, sowie einer örtlichen Lokalität, d. h. es wird mit höherer Wahrscheinlichkeit auf benachbarte Daten zugegriffen.

Bemerkung: dies ist nicht immer so, und wird auch durch die jeweilige Implementierung bestimmt ('array'–Implementierung, 'list'–Implementierung). Man beachte, daß oft die eine Aussage die andere bedingt: weil es schneller geht mit Lokalität bei der Hardware, verwende ich sie; weil sie oft verwendet ist, wird sie von der Hardware unterstützt.

1.1.2 Speicherhierarchie

memhier.eps

Speichertyp	Größe
'register'	8 bis 128 Register
'first level cache'	1K bis 256K
'second level cache'	32K bis 4M
'main memory'	4M bis 10G
'hard disc'	„unendlich“

Der Datenaustausch zwischen den Speichern erfolgt blockweise, wobei je nach Stufe unterschiedliche Blockgrößen verwendet werden. Für den Compiler sieht dies meistens wie folgt aus. Die Register werden direkt im Instruktionswort adressiert. Der 'cache' ist nicht sichtbar, er spiegelt einen Teil des Hauptspeichers wider, so daß wiederholte Zugriffe schneller vonstatten gehen. Der Hauptspeicher wird mit den Adressierungsmechanismen der Hardware adressiert, wobei eine Umsetzung von logischen (für den Compiler sichtbaren Adressen) in physikalische (für die Hardware relevante) Adressen vorkommen kann. Der Zugriff

auf Platten wird ebenfalls für den Compiler nicht sichtbar durch das Laufzeitsystem des Betriebssystems geregelt.

Die ‘cache’ Struktur kann komplizierter sein: getrennter Instruktions- und Daten‘cache’ (ein Speicher: *harvard architecture* andernfalls *princeton architecture*).

‘cache’–Strategien werden später behandelt.

1.1.3 Pipelining

pipeline.eps

Vorteile: Parallelverarbeitung und implizit kleinere Taktraten
Nachteile: Löcher in der Pipeline, Wartezeiten, Kontrolle

RISC, reduced instruction set computer, einheitliche Instruktionsausführungszeit, hohes Maß an ‘pipelining’, verzögerte Reaktion auf Ereignisse (‘delayed load’, ‘stalling’)

‘superscalar’: mehr als eine Instruktion pro Takt
‘superpipeline’: viele ‘pipeline’–Stufen

1.1.4 Prozessorüberblick

Die folgende Tabelle listet einige charakteristischen Eigenschaften moderner Mikroprozessoren im Vergleich auf:

Merkmal	PowerPC601	Alpha 21064	MIPS R4400
CMOS μm	0.6	0.75	0.65
cm^2	1.09	2.33	1.86
MTrans.	2.8	1.68	2.3
MHz	50	200	150
Watt	9	30	15
SPEC int92	40	133	88
SPEC fp92	60	200	97
	princeton	harvard	harvard
	‘superscalar’	‘superscalar’	‘superpipelined’
‘cache asso.’	8	1	1
‘cache size’ Kbyte	32	8 + 8	16 + 16
‘fp-pipeline’	6	10	10
‘ld/st-pipeline’	5	7	8
‘int-pipeline’	4	7	8

Bemerkung: Ein DEC Alpha hat eine proportional zur Fläche 1.5–fache Wärmeentwicklung einer Schnellkochplatte mit Durchmesser 18cm.

Die Anforderungen an einen guten Compiler sind nun: Nutzen paralleler Einheiten, Kontrollieren von ‘pipeline’s sowie Handhaben von Speicherhierarchien.

2 Grundlagen

Wir folgen [65] Kapitel 11.

Compilerentwurf erfolgte in vorhergehenden Kapiteln des Buches für abstrakte Maschinen, die so definiert sind, daß eine Übersetzung möglichst einfach wird. Abstrakte Maschinen sind dazu gedacht, die Übersetzung zu vereinfachen, weniger dazu, effizienten Code zu ermöglichen.

2.1 Abstrakte Maschinen

Die Speicherorganisation ist so angelegt, daß sie die Lebensdauer verschiedener Klassen von Objekten (Variablen) widerspiegelt:

Program	Lebensdauer während des gesamten Programms (Konstanten)
Keller ('stack')	Lebensdauer entspricht geschachtelten Zeitintervallen (lokale Variablen, Aufrufparameter, temporäre Variablen)
Halde ('heap')	alle anderen Variablen (dynamisch allokierte Variablen)

Diese Speicherorganisation muß für parallele Maschinen erweitert werden.

Programmvariablen werden in [65] nicht angesprochen, je nach Architektur können diese jedoch in unterschiedlichen Speichern liegen.

Der Keller ist in sogenannte Rahmen ('stack frames') unterteilt, die eine statisch bestimmbare Struktur haben, dies bedeutet, der Compiler kennt die Struktur zur Compilezeit (hierzu später mehr).

Der Befehlsvorrat enthält komplexe Befehle, die teilweise Quellsprachenkonstrukte widerspiegeln, somit sind abstrakte Maschinen oft quellsprachenabhängig.

Beispiele sind: `make_stack_frame`, `return_from_call`

Moderne Mikroprozessoren enthalten Befehle, die nicht komplex sind (RISC–Architekturen), solche die komplex sind (CISC–, VLIW–Architekturen), und solche, die häufig nicht in der Quellsprache vorkommen (VECTOR–Architekturen).

Die Ausdrucksauswertung erfolgt in einem kleinen Abschnitt des Kellers (eine Alternative wäre eine Ausdrucksauswertung in einem festen Speicherbereich ('scratch'–Auswertung), sofern dies möglich ist).

2.2 Reale Maschinen

Speicherhierarchie: Register, Cache, Hauptspeicher, Hintergrundspeicher.

Cache und Hintergrundspeicher werden vernachlässigt.

Hintergrundspeicher ist für die Code–Erzeugung nicht von großem Interesse, sondern sollte eher vom Programmierer kontrolliert werden. Allerdings kann bei sehr großen Datenobjekten eine Kenntnis dessen Organisation von Interesse sein (Seitenstruktur und deren Handhabung). Der 'cache' sollte nicht vernachlässigt werden.

Der Zugriff auf Register ist schneller als der Zugriff auf den Hauptspeicher, somit ergibt sich ein wesentliches Ziel für den Compiler: gute Ausnutzung von Registern.

Der Hauptspeicher ist linear organisiert. Die Zugriffszeit ist—wegen des dazwischenliegenden ‘cache’s—nicht konstant, sondern hängt von der Adreßsequenz ab. Die ‘cache’–Struktur (und andere Architekturmerkmale) beeinflussen diese Zeit (hierzu später mehr).

Es kann mehrere Hauptspeicher geben.

Der Hauptspeicher ist üblicherweise in Worten organisiert, die kleinste Wortbreite ist meistens ein Byte, es gibt (fast immer: SB–PRAM z. B. nicht) auch andere Zugriffsbreiten (‘word’, ‘long’, ‘double’) auf diesen Speicher, manchmal auch weit komplexere Zugriffe mit Vektoren.

Es gibt verschiedene Zugriffsarten auf den Hauptspeicher, diese werden auch als Adressierungsarten bezeichnet (siehe später).

Mögliche Prozessorregister sind: Adreßregister, Datenregister, Gleitkommaregister (‘floating point register’), Basisregister, Indexregister, Bedingungscoderegister, Vektorregister, Schattenregister, Programmzähler, Maskenregister, und andere.

Moderne Mikroprozessoren besitzen oft einfach Universalregister (später mehr).

Der Befehlsvorrat gliedert sich in: Berechnungsbefehle (arithmetische und boolesche Operationen), Vergleichsbefehle, Transportbefehle (innerhalb von Registern oder zum Speicher), Sprungbefehle, Spezialbefehle (Betriebssystemaufrufe, ‘interrupts’).

Der Befehlsvorrat ergibt sich letztendlich aus dem Produkt der Operationen mit den zugelassenen Adressierungsarten.

Die meisten modernen Maschinen erlauben Berechnungsbefehle und somit eine Ausdrucksauswertung nur mit Operanden in Registern.

Wir bevorzugen keine der vorzufindenen Assemblersprachen, sondern nehmen das, was uns gerade einfällt, die Bedeutung wird an der jeweiligen Stelle erläutert.

Man kann folgende Unterschiede in der Code–Erzeugung für abstrakte und reale Maschinen erkennen.

Abstrakte Maschinen sind dazu gedacht, die Übersetzung zu vereinfachen, weniger um effizienten Code zu erzeugen.

Abstrakte Maschinen haben einen sehr eingeschränkten Vorrat an bedingten und unbedingten Sprüngen, reale Maschinen sind hier wesentlich umfangreicher ausgestattet und nutzen weiterhin zahlreiche Adressierungsarten bei diesen Sprüngen (kurze, lange Sprünge, mit konstanter oder variabler Distanz u.a.).

Die Verwaltung von Rekursion und Funktionsaufrufen wird in der abstrakten Maschine ähnlich wie in der realen Maschine vorgenommen, es werden allerdings einige der zur Verfügung stehenden Universalregister für diese speziellen Zwecke genutzt und stehen deshalb nicht mehr zur Ausdrucksauswertung zur Vefügung. (allerdings gibt es Optimierungen, die diesen Aufwand wiederum reduzieren, vergl. `-fomit-fram-pointer`–Schalter des `gcc` [Übung])

Bei realen Maschinen muß der Datentyp von Parametern und Operanden berücksichtigt werden, bei der abstrakten Maschine ging man davon aus, daß jeder Wert in ein Maschinenwort paßt. Dies war insbesondere für Zeichenketten der Fall, die heute von fast keinem Prozessor mehr als Operanden zugelassen sind. Aus diesem Grund müssen hier stets komplette Unterroutinen benutzt und dynamisch Speicherplatz verwaltet werden.

Die Ziele bei der Ausdrucksauswertung sind die Reduktion des Transportes von Werten zwischen Speicher und Registern, d. h. das Halten von Zwischenergebnissen in Registern; desweiteren die Übergabe von Parametern an Funktionen und die Funktionsergebnisse (‘return’–Werte) in Registern. Zwischendarstellungsoperationen sollen möglichst durch effiziente Maschinenbefehle ersetzen werden. Das Ausnutzen der parallelen Abarbeitungsmöglichkeiten eines Prozessors durch geschickte Instruktionsanordnung bildet ein weiteres Ziel.

Dabei entstehen ‘trade–off’s: der bessere Code benötigt mehr Register, womit zusätzliche Lade– und Speichere–Befehle notwendig werden. Man muß also die Transportkosten von Registerinhalten in die Kostenberechnung der Codeselektion einbeziehen; je nach ausgewähltem Code für eine Sequenz kann eventuell eine andere Sequenz parallel ausgeführt werden, d. h. die Codeselektion ist kontextabhängig.

Die CISC—RISC Problematik:

CISC (complex instruction set computer): viele, komplexe Adressierungsarten zur Unterstützung von Zugriffen auf Felder, Listen und Keller; mannigfaltige Varianten für Operationen für verschieden lange Operanden und Kombinationen von Operanden– und Resultatsorten; unterschiedliche Ausführungszeiten der Befehle; wenige Prozessorregister.

Vertreter sind: VAX, i80x86, M680x0, NSC 32000,

RISC (reduced instruction set computer): pro Maschinenzyklus eine Maschineninstruktion, länger dauernde Befehle (wie z. B. Speicheroperationen) werden ge‘pipeline’t betrieben; berechnende Operationen arbeiten nur auf Registern; es gibt nur wenige Adressierungsarten; oft sind viele Register vorhanden.

Vertreter sind: Sparc, MIPS,

VLIW (very long instruction word computer):

vliwproc.eps

Sie dadurch gekennzeichnet, daß mehrere Operationen auf mehreren Register in einer Instruktion spezifiert werden. Der Compiler hat während der Code–Erzeugung die Aufgabe, entsprechende einfache Maschinenbefehle (die im wesentlichen RISC–Charakter haben) zu solch komplexeren zusammenzusetzen. Dies hat erheblichen Einfluß auf die Registerallokation, da sich jetzt mehrere Funktionseinheiten die vorhandenen Register teilen müssen, sowie auf die Codeselektion, da verschiedene einfache Befehle unterschiedlich kombiniert werden können. Es gibt nur einen Befehlsstrom und somit nur einen Programmzähler.

Die Strategien, die für VLIW Maschinen entwickelt worden sind, sind durchaus auf moderne Prozessoren anwendbar, die zwar keine solch langen Instruktionsformate haben, deren Befehle jedoch überlappend ausgeführt werden.

3 Programmdarstellungen

Darstellung des Kontrollflusses eines Programms: Kontrollflußgraph, sowie Aufrufgraph.

Definition 1 (Kontrollflußgraph) *Der Kontrollflußgraph (cfg) einer Prozedur ist ein knotenmarkierter, kantengeordneter, gerichteter Graph* (N, E, s)*. Dabei gibt es zu jeder primitiven Anweisung* p *der Prozedur (d. h. Instruktion der Zwischendarstellung) einen Knoten* $n_p \in N$*, der mit dieser Anweisung markiert ist. Eine Kante verbindet zwei Knoten, wenn diese hintereinanderausgeführt werden können.* s *ist der Eingangsknoten des Graphen (erste auszuführende Instruktion der Prozedur).*

Der Kontrollflußgraph bildet die Grundlage der Datenflußanalyse. Die geeignete Wahl der Zwischendarstellung, die allen Anforderungen der Optimierungen des Compilers in möglichst allen Phasen der Übersetzung ermöglicht, ist ein wesentlicher Bestandteil des Compiler-Design. Normalerweise erzeut das 'front end' eine erste Version des zu übersetzenden Programms in der Zwischendarstellung. Alle nachfolgenden optimierenden Transformationen arbeiten auf dieser Zwischendarstellung. Jede der Tranformationen muß die Semantik des ursprünglichen Programms erhalten.

Knoten mit mehr als einem Vorgänger heißen Verschmelzungen, Knoten mit mehr als einem Nachfolger Verzweigungen.

Ein Kontrollflußgraph ist zusammenhängend (wäre er nicht zusammenhängend, so gäbe es sogenannten toten Code, der nie zur Ausführung käme, wir nehmen an, daß dies nicht der Fall ist).

Definition 2 (Grundblock) *Ein Grundblock ist ein maximal langer Pfad in einem Kontrollflußgraphen, der höchstens am Anfang eine Verschmelzung und höchstens am Ende eine Verzweigung hat.*

Bis auf das Auftreten von 'exception's (durch die Ausführung des Programms bestimmte Verzweigungen in besonderen Situationen z. B. Division durch 0) oder 'interrupt' (durch äußere Einflüsse bestimmte Verzweigungen) werden die Knoten in einem Grundblock immer sequentiell ausgeführt.

Definition 3 (Grundblockgraph) *Ist ein Kontrollflüsgraph eines Programms, dessen Grundblöcke durch einzelne, diese Blöcke representierenden Knoten ersetzt worden sind.*

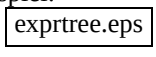
Definition 4 (Aufrufgraph) *Der Aufrufgraph ist ein gerichter Graph* (N, E)*. Die Knoten representieren die im Programm vorkommenden Prozeduren (einschließlich des Hauptprogramms). Eine Kante verbindet zwei Knoten* p *und* q*, wenn die entsprechende Prozedur in* p *die Prozedur* q *aufruft.*

Der Aufrufgraph ist besonders wichtig, um 'alias'–Effekte zu entdecken. Dabei versteht man unter 'alias' verschiedene Referenzen zu der gleichen Speicherstelle, aber unter verschiedenen Namen (insbesondere Pointer). 'Alias'e können dazu verwendet werden, Seiteneffekte zu programmieren, die für eine gute Code–Optimierung unerwünscht sind, da sie das Ausnutzen von Registervariablen einschränkt.

Mit Hilfe von Aufrufgraphen erkennt man rekursive und simultan–rekursive Funktionsaufrufe. Man benutzt ihn für interprozedurale Analyse von Programmen: Menge der benutzten und veränderten Programmvariablen

4 Optimale Ausdrucksauswertung

Wir betrachten Ausdrucksbäume. Beispiel:



Das Verfahren kann leicht auf Grundblöcke ausgeweitet werden.

Annahmen: Zwischendarstellungsoperationen entsprechen Maschinenoperationen, Universalregister, Zweiadreßbefehle

```
v may be register or constant
R_i    = M[v]         load
M[v]   = R_i          store
R_i    = R_i op M[v]  operate memory
R_i    = R_i op R_j   operate register
```

Auswertungsalgorithmus:

```
EXPRTREECODE()
  1    MARKTREE()
  2    GENCODE()
```

Bisheriger Algorithmus erlaubt nur eine Auswertung von Ausdrucksbäumen. Das folgende Verfahren gibt eine Registerverteilung für alle Berechnungen *innerhalb* einer Prozedur, also nicht über Prozedurgrenzen hinweg, an.

Die Eingabe ist eine Zwischendarstellung der Prozedur, bei der jeder Variablen ein symbolisches Register zugeteilt worden ist.

Ziel ist die Verteilung der endlich vielen realen Register an die potentiell beliebig vielen symbolischen Register, dabei darf nicht zwei symbolischen Registern das gleiche reale Register zugeordnet werden, wenn diese gleichzeitig „lebendig“ sind.

Sei S die Menge der symbolischen und R die Menge der realen Register.

Definition 5 (lebendig, Lebensspanne) *Ein Register $r \in S$ ist* lebendig *an Knoten p im Kontrollflußgraph cfg, wenn es einen Pfad von s nach p in cfg gibt, auf dem r gesetzt wird, und wenn es einen Pfad von p in cfg gibt, auf dem r benutzt (aber nicht gesetzt) wird. Die* Lebensspanne *von r ist die Menge aller p, an denen r lebendig ist.*

Die Kollisionen, die zwischen Register in S auftreten können werden durch den Registerkollisionsgraphen bestimmt:

Definition 6 (Kollision, Registerkollisionsgraph) *Zwei Lebensspannen zweier Register in S* kollidieren, *wenn eins von ihnen in der Lebensspanne des anderen gesetzt wird. Der* Registerkollisionsgraph *ist ein ungerichteter Graph, dessen Knotenmenge S ist. Es exitiert eine Kante zwischen zwei Knoten, wenn die Lebensspannen der beiden kollidieren.*

Das Problem der Registerverteilung reduziert sich nun auf: Färbe den Registerkollisionsgraph mit $|R|$ Farben, so daß keine zwei benachbarte Knoten gleiche Farbe haben.

Ist $k = |R| > 2$, so ist das Problem $\mathcal{NP}$–vollständig. In der Praxis werden deshalb Heuristiken verwendet. Auch Approximationsalgorithmen haben sehr schlechte Eigenschaften. Der beste bekannte Approximationsalgorithmus findet in polynomieller Zeit lediglich eine $\sqrt{n}$–Färbung eines drei–färbbaren Graphen.

Sei $G = (N, E)$ ein Graph. Hat $p \in N$ nur Grad $< k$, so kann p eine Farbe zugeordnet werden, die von allen seinen Nachbarn verschieden ist.

```
GRAPHCOLORING(G,R,M)
  1    find p node of G with grad(p) < k
  2    if p exists
```

```
3       then push p onto register stack R
4       else
5           select p heuristically
6           push p onto memory stack M
7       G' = without p and all adjacent edges
8       GRAPHCOLORING(G',R,M)

REGALLOC(G)
1       init empty register stack R
2       init empty memory stack M
3       GRAPHCOLORING(G,R,M)
4       while not empty R
5           pop p from R
6           color p appropriately
7       while not empty M
8           pop p from M
9           mark as load/store value
```

Für Zeile (5) in GRAPHCOLORING() sind folgende Heuristiken angebracht

- entferne Knoten mit größtem Grad
- schätze die Ladekosten der Variable, entferne Knoten mit geringsten Kosten

Als weitere Opimierungen kann man die Lebensspannen aufspalten bzw. die zweier Variablen verschmelzen.

Aufspalten bedeutet, daß diesem Register in S zwei oder mehrere Register in R zugeordnet werden. Dies ist besonders dann sinnvoll, wenn sich die Lebensspanne des Registers über einen langen Pfad erstreckt, auf dem auch eine Schleife liegt. Hier lohnt sich unter Umständen ein vorheriges Abspeichern und nachfolgendes Laden. In der Schleife steht dann ein Register mehr zur Verfügung.

Verschmelzen ist dann sinnvoll, wenn das gleiche Register von Beginn an für zwei nicht miteinander kollidierende Variablen verwendet wird. Hierdurch wird verhindert, daß den beiden Variablen verschiedene Register zugeordnet werden und reduziert die Anzahl der Knoten (und damit Kanten) im Registerkollisionsgraphen.

5 ‘Instruction scheduling’

Obwohl noch nicht die Gründe genauer erläutert worden sind, ist wohl folgendes klar:

Zwei Maschineninstruktionen eines Prozessors können voneinander hardwaremäßig abhängig sein, so daß ein Ausführen der einen ein Ausführen der anderen ausschließt. Der Anordnung der Instruktionen sind also Hardware–Beschränkungen auferlegt. (Diese können entweder durch die Hardware aufgelöst werden, „stalling“, oder müssen durch die Software vermieden werden, z. B. durch `nop` Einsetzen.)

Sei also eine Beschreibung der Hardware gegeben, die für jeden Instruktionscode festlegt, mit welch anderem Instruktionscode diese kollidiert und welche Anzahl von Maschinenzyklen vergehen muß, bevor der zweite ausgeführt werden kann. Wir nehmen an, es gibt eine Funktion MINDISTANCE(*p*,*q*), die für zwei Instruktionen p und q die minimale Distanz in

Instruktionszyklen angibt, die vergehen muß, wenn p jetzt und q später ausgeführt werden soll. Es wird also die (gesamte) Konfiguration der Maschine berücksichtigt. Man erkennt, daß MINDISTANCE(*p*,*q*) kontextabhängig ist.

Neben den Hardware–Abhängigkeiten gibt es als weitere Beschränkungen die Daten–Abhängigkeiten. Man unterscheidet folgende Abhängigkeiten zwischen Instruktionen i und j:

- 'output dependence': i verändert die gleiche Ressource wie j, z. B. Register, Condition–Code, Mode–Register;
- 'true dependence': i verwendet die von j veränderte Ressource;
- 'antidependence': i verändert eine Ressource, die vorher durch j noch benutzt wird.

Die Abhängigkeiten geben an, in welcher Reihenfolge die Befehle ausgeführt werden müssen.

Wie findet man diese Abhängigkeiten?

Für Werte in Register ist dies einfach. Da die Registerverteilung (eventuell) schon erfolgt ist, findet man die Abhängigkeiten leicht. Für Speicherzellen können sogenannte 'aliase' vorliegen, die nicht unbedingt erkannt werden können. Naive Lösung: der gesamte Speicher wird wie ein Register behandelt, damit können Lade– und Speichere–Operationen und Speichere–Operationen untereinander nicht miteinander vertauscht werden.

Betrachten wir als Ressourcen vorerst lediglich: Register, Hauptspeicher und Bedingungscoderegister.

Definition 7 (bestimmter Vorgänger) *Eine Instruktion p ist bestimmter Vorgänger einer Instruktion q, wenn einer der folgenden Fälle gilt:*

- *p ist der letzte Befehl, der eine Ressource setzt, bevor sie q nutzt;*
- *q ist der erste Befehl, der eine Ressource setzt, nachdem die p benutzt hat;*
- *p ist der letzte Befehl, der eine Ressource setzt, bevor sie q erneut setzt, ohne daß dazwischen ein Befehl r die Ressource benutzt.*

depedges.eps

Definition 8 (Abhängigkeitsgraph eines Grundblocks) *Sei B ein Grundblock. Sein Abhängigkeitsgraph $G = (N, E)$ ist gegeben durch: N ist die Menge aller Instruktionen i von B. (p, q) ist eine Kante in E, wenn p bestimmter Vorgänger von q ist.*

Bemerkung: man berechnet die transitive Hülle der direkten Abhängigkeiten zwischen je zwei Knoten, um alle Abhängigkeiten zu erhalten.

Das Bedingungscoderegister macht den ganzen Graphen recht unübersichtlich, da es viele Kanten verursacht. Desweiteren gibt es mehr Abhängigkeiten von Registern im Maschinencode als man dem Quellcode in der Hochsprache einfach ansieht. Neben den offensichtlichen gibt es noch

Setzungen Autoinkrement– und Autodekrement von Registern, Änderungen vom Bedingungscode, 'aliasing' im Speicher, Modifikationen des 'stack pointers',

Benutzungen Abfragen des Bedingungscodes.

In [65] wird ein $O(n^2)$–Algorithmus zur Berechnung des Abhängigkeitsgraphen angegeben. Man führt einen Rückwärtslauf über den Grundblock durch, wobei man sich folgende Datenstrukturen hält:

- Menge S der letzten Setzungen einer Ressource,
- Menge B der noch exponierten Benutzungen einer Ressource.

Jedes Element dieser Mengen ist mit der entsprechenden Instruktion markiert.

```
BBAGRAPH()
 1   b = last instruction of basis block
 2   update S and B
 3   while b = predecessor exists do
 4       forall resources r used or set in b do
 5           forall t ∈ S ∪ B do
 6               if b in conflict with t on r then
 7                   add edge (b, t)
 8       forall resources r set in b do
 9           S = S \ { all t setting r}
10           S = S ∪ {r}
11           B = B \ { all t using r}
12       forall resources r used in b do
13           B = B ∪ {r}
```

Jede topologische Sortierung des Abhängigkeitsgraphen liefert eine gültige Ausführungsreihenfolge der Instruktionen. Es stellt sich die Frage, welche nimmt man? Es gibt heuristische Varianten von topologischen Sortieralgorithmen, die auf dieses Problem speziell angepaßt sind. Hierzu später mehr.

Es ist leicht einsichtig, daß man durch Kantenreduktion im Graphen das Problem vereinfachen kann. Hierzu stehen folgende Möglichkeiten zur Verfügung:

- 'alias'–Analyse: man betrachtet den Speicher nicht mehr als ein einziges Register, sondern versucht, ihn in mehrere Register aufzuteilen, (hier kann auch auf Hochsprachenebene geholfen werden: der Programmierer sagt, daß auf eine bestimmte Variable nur über ihren Namen zugegriffen wird, d. h. die Benutzung dieser Variable ist seiteneffektfrei (hierzu später mehr).
- 'single assignment': man eliminiert die Abhängigkeiten der dritten Art durch Einführen neuer Variablen; genauer, man sorgt dafür, das eine Variable höchstens einmal auf der linken Seite steht (hierzu später mehr).

Reale Maschinen verursachen weitere Beschränkungen bei der Instruktionsabarbeitung.

Definition 9 (lebendige Maschinenressource) *Eine Maschinenressource m ist an einem Befehl b in einem Grundblock lebendig, wenn m später genutzt wird, ohne nochmals gesetzt zu werden.*

Durch eine Integration dieser Lebendigkeiten können weitere Kanten im Graphen eliminiert werden (vorallem Kanten der dritten Art bezüglich des Bedingungscoderegisters).

In realen Maschinen ist oft die Instruktionsausführungszeit nicht einheitlich. Demzufolge muß eine geschickte Reihenfolge für die Ausführung der Befehle gefunden werden, so daß Wartezeiten (oder bei Maschinen, die solche Konflikte nicht erkennen, sogar Fehler) vermieden werden. Es gibt zwei Ursachen für die Wartezeiten: Ressourcen sind blockiert oder das Ergebnis steckt noch in einer 'pipeline'.

Betrachten wir nochmals den Abhängigkeitsgraphen eines Grundblocks. Jede topologische Ordnung des Graphen liefert ein korrektes Programm. Das optimale Programm zu finden ist (sogar unter sehr einfachen Annahmen) $\mathcal{NP}$–vollständig (Reduktion auf 'scheduling problem'). Man ist also wieder auf Heuristiken angewiesen.

Definition 10 (blockierte Ressourcen) *Eine Maschinenressource m ist k–blockiert, wenn sie in den nächsten k Zyklen nicht benutzt werden kann. Eine Instruktion p ist k–blockiert, wenn sie auf eine Ressource noch k Zyklen warten muß.*

Bemerkungen: Es können nur 0–blockierte Instruktionen zu einem Zeitpunkt ausführt werden, die 0–blockierte Ressourcen verwenden.

Wir halten uns die Menge M der Maschinenressourcen, die angibt wie sie blockiert sind. Wir halten uns den Abhängigkeitsgraph G eines Grundblocks und annotieren die Knoten gemäß ihrer Blockiertheit. Wir nehmen an, daß die Register schon verteilt worden sind.

```
LISTSCHEDULING()
 1    calculate M at start of basic block
 2    while G not empty do
 3        forall p ∈ G being source node do
 4            UPDATEBLOCKING(p)
 5        select p to be scheduled
 6        schedule p
 7        G = G \ {p}
 8        update M
```

6 Trace–Scheduling

Trace–Scheduling wurde speziell zur Code–Erzeugung für VLIW Maschinen entwickelt. Die klassische Arbeit ist [26].

In wissenschaftlichen, numerischen Programmen ist die Parallelität oft bis zu einem Faktor von 90 ausgeprägt, wenn man die Einschränkungen durch Grundblöcke, die oft durch Randfälle und Sonderabfragen induziert ist, vernachlässigt.

In VLIW Maschinen stehen mehrere Funktionseinheiten zur Verfügung, die gemeinsam auf die gleichen Register zugreifen, parallel arbeiten können und von nur einem Instruktionswort gesteuert werden. Der Compiler muß also versuchen—um effizienten Code zu erzeugen—mehrere Instruktionen der Zwischendarstellung in ein Maschinenwort zu packen. Die Methoden des 'trace scheduling' sind auch bei Maschinen mit mehreren Funktionseinheiten anwendbar, die zwar nicht durch ein Instruktionswort gesteuert werden, aber lange Ausführungszeiten und viele 'pipeline'–Stufen haben, da dort mehrere Instruktionen als Block betrachtet werden können.

Nun zeigt sich, daß im allgemeinen die Anzahl und die Art der Befehle in einem Grundblock zu gering ist, um eine gute Auslastung der Funktionseinheiten zu gewährleisten. Aus diesem Grund werden mehrere Grundblöcke, im Extremfall alle Blöcke einer Prozedur, gleichzeitig betrachtet.

Die Grundidee des Verfahrens ist es, solche Grundblöcke miteinander auszuführen, die im Programmablauf normalerweise konsekutiv ausgeführt werden. Ferner werden speziell solche Grundblöcke ineinander „gemischt“, die oft ausgeführt werden, da dann der höchste Gewinn von der Parallelisierung erwartet werden kann.

```
PROBIF()
  1    if condition then pragma 15%
  2        block
  3    else block
```

Das Verfahren benötigt also Informationen über die Ausführungshäufigkeiten von Grundblöcken. Hierzu stehen die folgenden Methoden zur Verfügung:

- Angaben in der Hochsprache (sogenannte `pragmas`), wie hoch die Wahrscheinlichkeiten für eine Verzweigung sind, z. B. der **then**–Fall wird mit 15% ausgeführt, der **else**–Fall mit 85%.
- Heuristiken über Schleifen: es ist wahrscheinlicher am Ende einer Schleife zu deren Anfang zu springen als sie zu verlassen.
- Testläufe bringen vorallem für numerische Programme gute Statistiken. Der Bulldog–Compiler wurde speziell auf solchen Problemen getestet (FFT, LU, Matrizenoperationen).
- Es kann für mehrere Varianten der Ausführungsreihenfolge Code erzeugt werden, zu dem dann in Abhängigkeit von Programmvariablen (z. B. Größe von Schleifenzählern) gesprungen wird.

Betrachten wir den Grundblockgraph einer Prozedur. Nehmen wir an, wir haben eine Anordnung der Knoten im Graph gemäß ihrer erwarteten Ausführungshäufigkeit gegeben. Wir ordnen die Pfade innerhalb einer Schleife bezüglich der Häufigkeit der Ausführung der auf diesen Pfaden liegenden Grundblöcken an.

bbgtrace.eps

Auf dem Pfad, der mit der größten erwarteten Häufigkeit ausgeführt wird, werden die Instruktionen so verschoben, daß eine gute parallele Auswertung gewährleistet ist. Hierzu ist an anderer Stelle eventuell Kompensationscode erforderlich.

Hierzu einige Beispiele später.

Sind beliebig viele Funktionseinheiten vorhanden, kann der Abhängigkeitsgraph mit so vielen Instruktionen abgarbeitet werden, wie die Länge des kritischen Pfades angibt. Der Algorithmus arbeitet wie ‘list scheduling’ (also einer Variante des topologischen Sortierens), nur daß jetzt alle Instruktionen in den Kandidatenlisten gleichzeitig abgearbeitet werden. Folgender Algorithmus zeigt dies für einen Grundblock

```
TRACESCHEDULING()
  1    calculate M at start of basic block
  2    while G not empty do
```

```
3       forall p ∈ G being source node do
4           UPDATEBLOCKING(p)
5       select p to be scheduled
6       schedule p
7       G = G \ {p}
8       update M
```

Nun bestimmt die Hardware die genauen Möglichkeiten der parallelen Verarbeitung. So kann z. B. nur eine bedingte Verzweigung im Instruktionswort erlaubt sein.

Wir erweitern die Funktion BBAGRAPH() über Verschmelzungs– und Verzweigungspunkte hinweg um so einen verallgemeinerten Abhängigkeitsgraphen zu erhalten.

Schleifen, man erkennt sie an Rücksprüngen, werden zweimal betrachtet, bedingte Verzweigungen werden über alle Pfade analysiert, die Mengen B und S werden jeweils vereinigt

TBBAGRAPH()

Folgende semantikerhaltende Transformationen sind z. B. denkbar:

- Instruktionen vor Verzweigungen werden in beide Pfade nach der Verzweigung integriert, dabei darf es keinen Konflikt mit der Bedingung geben. Ist eine Variable in einem Pfad nicht lebendig, braucht der Code dort nicht eingeführt zu werden.
- Instruktionen innerhalb eines Pfades nach einer Verzweigung können in die Pfade vor die Verzweigung geschoben werden. Wenn eine entsprechende Variablen in einem Pfad nicht lebendig sind, braucht kein Code eingeführt zu werden.
- Verzweigungen können ebenfalls verschoben werden, sofern keine Abhängigkeiten mit den Bedingungen bestehen.
- Aufrollen von Schleifen erzeugt größere Blöcke.

codemove.eps

```
LOOP()
1   forever
2       if i > n then return
3       i++
4       block
5   return
```

```
LOOPUNROLLED()
1   forever
2       if i > n then return
3       i++
4       block
5       if i > n then return
6       i++
7       block
8       if i > n then return
9       i++
10      block
11  return
```

Reale VLIW sind allerdings nicht so ideal wie bis jetzt angenommen, sondern es gelten folgende Einschränkungen:

- Die Anzahl der Funktionseinheiten ist beschränkt. Es können also nicht alle Kandidaten des Abhängigkeitsgraphen gleichzeitig ausgeführt werden. Als Heuristik bevorzugt man die Instruktionen auf dem kritischen Pfad.
- Die Ausführungszeiten der Befehle sind oft unterschiedlich. Die Kanten müssen also wie oben bereits vorgeschlagen markiert werden. Es findet eine Blockiertheitsanalyse statt.
- Es kann sein, daß nicht jede Funktionseinheit auf jedes Register zugreifen kann, was die Code–Erzeugung sehr erschwert.

Hier endet [65].

Eine wesentliche Beschränkung von ‘trace scheduling’ liegt in den Speicherreferenzen. Eine ‘alias’–Analyse ist sehr wichtig, demzufolgende wird der Einsatz bei numerischen Programmen—sie haben statt Zeigerreferenzen oft analysierbare Feldzugriffe—und ‘single assignment’ Programmen erst besonders wirkungsvoll.

Die ‘Alias’–Analyse kann ebenfalls durch `pragmas` vereinfacht werden. Man braucht eine Funktion AMBIGOUSREFERENCE(), die entscheidet, ob zwei indirekte Referenzen zur gleichen Speicherzellen führen oder nicht.

7 Pipelining

Instruktions‘pipelining’:

Die verschiedenen Phasen der Instruktionsausführung werden mit einer ‘pipeline’ verarbeitet: Befehl laden (‘instruction fetch and decode’), Operanden Laden (‘operand fetch’), Instruktion ausführen (‘instruction execute’), Resultat schreiben (‘operand store’). Die Anzahl der ‘pipeline’–Stufen kann größer sein (werden wir bei der Besprechung der Prozessoren genauer sehen)

Damit ergibt sich folgendes Ausführungszeitdiagramm:

pipetime.eps

Treten Kollisionen auf, d. h. es werden Ergebnisse oder Ressourcen an zwei Stellen gleichzeitig gebraucht, treten sogenannte ‘hazards’ auf, die entweder durch Hardware (‘interlock logic’) gelöst werden, oder durch die Software vermieden werden müssen. Hierzu wieder mehr bei der Prozessorbeschreibung.

8 Adressierungsarten

[65] Kapitel 10.2

9 Mikroprozessoren

In diesem Abschnitt beschreiben wir knapp verschiedenen Mikroprozessoren soweit die Merkmale für einen Compiler von Interesse sind. Viele technische Besonderheiten bleiben dabei unerwähnt. So wird beispielsweise nicht auf die ‘interrupt’–Behandlung eingegangen.

Die meisten heutigen Prozessorbeschreibungen folgen einem Architekturkonzept, für die dann jeweilige Implementierungen (auch verschiedener Hersteller) implementiert wurden. Die Architekturbeschreibung folgt einer sogenannten ISA (‘instruction set architecture’), d. h. der Prozessor wird lediglich durch seine Instruktionen und für den Programmierer letztlich sichtbaren Register beschrieben.

Die Vorteile sind ...

9.1 PowerPC

9.2 RS6000

rsfig1p17.eps

Merkmale:

- RISC–Architektur
- Multiply–Add–Instruktion mit höherer Genauigkeit
- Ausführung von vier Instruktionen gleichzeitig: ‘branch instruction, condition register instruction, fixed–point instruction, floating–point instruction’
- Idee des ‘zero–cycle–branch’, die Instruktionseinheit versucht einen kontinuierlichen Operationsfluß den ‘fixed–point’ und ‘floating–point’ Einheiten zukommen zu lassen. Diese sollen unbeeinflußt von Verzweigungsoperationen arbeiten können.
- Der Nicht–Verzweigungspfad wird von der Hardware bevorzugt. Der Verzweigungspfad läuft nur dann verzögerungsfrei ab, wenn der Bedingungscode frühzeit vorhanden ist.
- Mehrfach vorhandenes (acht) Bedingungscoderegister erlaubt die parallele Verarbeitung mehrerer Bedingungen, d. h. Vergleichsoperationen spezifieren ein Bedingungscoderegister, Sprünge benutzen eins als Operand.
- Direktes Schreiben von Bedingungen durch Register ist möglich.
- ‘fixed–point’ Operationen: $+$ ein, $*$ drei bis fünf, $/$ 19–20 Zyklen
- ‘in–order–execution’ der Instruktionen
- ge‘pipeline’te ‘floating–point’–Einheit, mit zwei Stufen
- Umbenennung von Registern, so daß Ladeoperationen vorgezogen werden können.
- 32 ‘floating–point’ Register

9.3 Sparc

Die Angaben stammen aus [**?**], in dem die Sparc Architektur in der Version 8 beschrieben wird. Sparc steht für ‘Scalable Processor Architecture’.

Die Architekturbescheibung des Sparc basiert auf einer Instruktionsmenge, die von einem RISC Prozessor (der sogenannten Berkely RISC Architektur) abgeleitet wurde. Es existieren inzwischen eine Menge von verschiedenen Implementierungen der Architektur (SunSparc, SuperSparc, UltraSparc, HyperSparc).

Eine Besonderheit des Sparc Prozessors stellt sein gleitendes Registerfenster dar. Dieses erlaubt schnelle Parameterübergabe zwischen Prozeduraufrufen. Das Registerfenster kann aber neben `call` und `return` Instruktionen auch direkt manipuliert werden.

Die Architekturbescheibung schließt nicht in allen Teilen den ‘supervisor’ Modus ein. Man verlangt nur, daß der normale ‘user’ Code immer gleich abgearbeitet wird.

Charakteristiken

- 32-Bit linearer Adreßraum,
- 32-Bit Instruktionsformat,
- einfache Adressierungsarten (nur Register+Register oder Register+Konstante),
- Drei–Adreßbefehle,
- 40 bis 520 Register großes Registerfenster, welches allgemein benutzbare Register enthält,
- ein Register ist Null–Register, d. h. 0 wird gelesen, nichts wird geschrieben,
- `CALL` Instruktion schreibt automatisch ihre Adresse (PC) nach Register 15 (welches im Registerfenster liegt, siehe später),
- spezielle Register sind u. a.: Prozessorstatus–, Fenstermasken–, Multiplizier/Dividier–, Programmzähler–, ‘floating point’–Status–, Coprozessorstatus–Register,
- 32 separate 32–Bit ‘single–precision floating point’ Register, konfigurierbar ganz oder teilweise in ‘double–precision’ oder ‘quad–precision’ Register,
- implementiert den IEEE ‘floating point’ Standard,
- verzögerte Kontrollinstruktion, d. h. die nachfolgende Instruktion wird ausgeführt, kann aber annuliert werden, wenn der Sprung nicht ausgeführt wird, d. h. die Instruktion nach dem Sprung kann dem nachfolgenden Grundblock zugeordnet werden,
- praktisch jede ALU–Instruktion erlaubt ein optionales Ändern des Bedingungscoderegisters,
- Multiprozessor–Instruktion: atomare Lese–und–Setze–Operation, sowie atomare Wechsle–Register–mit–Speicher–Operation, sowie eine Definition in welcher Reihenfolge die Speicherzugriffe von mehreren Prozessoren ausgeführt werden dürfen. Wir gehen darauf nicht weiter ein.

Die Architekurbeschreibung umfaßt eine ‘integer’ Einheit (IU), eine ‘floating point’ Einheit, sowie eine optionale Coprozessor Einheit. Jede Einheit verfügt über eigene Registersätze.

Die Registerfenster

sparcfig4-1.eps

sparcfig4-2.eps

Die Registerfenster erlauben eine einfache und effiziente Parameterübergabe an Prozeduren (zumindest bis zu einer gewissen statischen Aufruftiefe, allerdings kann ein ‘trap’–Händler den Überlauf automatisch handhaben).

Der mögliche und einfache Nutzen des Registerfensters durch den Compiler zur Erweiterung der Anzahl der Register ist leider nur als Keller möglich, da der ‘trap’ und ‘interrupt’ Mechanismus des Prozessors „obere“ Register überschreiben kann.

sparcfigD-1.eps

sparcinst.eps

9.4 MIPS

Die Angaben stammen aus [**?**], in dem die MIPS Architektur beschrieben wird.

Die Beschreibung erfolgt wiederum als Architektur. Es werden auch die Implementierungen der R–Serie angesprochen. Die Architektur besiert auf der Stanford Mips Forschung.

mipsfig2-1.eps

Als Erweiterung der schon gesehen Architekturbeschreibungen ist bei MIPS explizit ein Coprozessor erwähnt. Die Schnittstellen dazu sind sehr einfach gefaßt, so daß viel Spielraum für eine Implementierung bleibt. Desweiteren exisitiert in der Beschreibung ein Kontrollprozessor, der das Implementieren von Betriebssystemaufgaben (‘traps’, ‘interrupts’, ‘exceptions’, ‘cache handling’) erleichtert, sowie eine Schnittstelle zum eigentlich ‘on–chip’ ‘floating point’ Prozessor.

Charakteristiken

- generelle 32–Bit Architektur (die neueren sind jedoch 64–Bit)
- 32 32–Bit allgemeine Register,
- Register 31 ist reserviert für Sprünge, Register 0 ist Null–Register,
- zweigeteiltes 64–Bit ‘multiply/divide’ Register
- Programmzähler, Statusregister
- ge‘pipeline’te Funktionseinheiten
- ‘delayed load’
- ‘delayed branch’
- IEEE ‘floating point’ Format, 64–Bit
- 16 32–Bit ‘floating point’ Register, als 16 64–Bit Register nutzbar,
- die 32 ‘floating point’ Register sind durch `move` Instruktionen auch vom „normalen“ Prozessor aus sichtbar,
- Drei–Register–Instruktionen

- Intruktionsanordnung ist bei manchen Operationen auf eine korrekte Reihenfolge achten, da die Hardware gewisser Implementierungen (vorallem die frühen) 'hazards' nicht erkennt und „unerwartet" für den Benutzer arbeitet.
- Beispiel R4000 (führt zu 'interlocks', d. h. die Hardware wartet automatisch):
  - Multiplizierer benutzt in seiner letzten Stufe den Addierer, der Compiler muß darauf achten, daß zu diesem Zeitpunkt keine andere Addition stattfindet, d. h. bei 'double precision' in der 4 und 5 Instruktion nach einer Multiplikation ist keine Addition erlaubt,
  - dies trifft auch für `cmp` zu,
  - Dividierer kann nicht ge'pipeline't betrieben werden,
  - Multiplizierer verkraftet höchstens zwei Operationen, sofern sie mindestens durch zwei 'idle' Instruktionen getrennt sind,
  - siehe [**?**] Seite 8–23 bis 8–25,

Scheduling für den MIPS ist nicht einfach, man muß praktisch für jedes Paar von Instruktionen die gebrauchten und eventuell blockierten Ressourcen untersuchen.

mipsinst.eps

mipsfig8-6.eps

9.5 DECalpha

10 Registerklassen

11 Codeselektion

12 Registerrenaming on the fly

13 Instruction scheduling on the fly

14 Minimaler Registerverbrauch

Register Allokation und Instruktionsscheduling sind zwei $\mathcal{NP}$–vollständige Probleme, man verwendet also Heuristiken bzw. für kleine DAGs auch Aufzählungsverfahren.

Register Allokation wurde mit Färbung des Registerkollisionsgraphen durchgeführt. Hierzu mußte ein Instruktionsschedulung bereits vorliegen. Nun kann man jedoch eine optimale Registerallokation bei vorliegendem Instruktions'schedule' mit einen relativ einfachen Algorithmus erzeugen.

Heutige moderne Mikroprozessoren verfügen über komplexe Funktionseinheiten, die superskalar und/oder ge'pipeline't angeordnet sind. Ferner stehen oft recht größe Registeranzahlen zur Verfügung.

Instruktionsanordnung ist schwierig da mit vielen Einschränkungen verbunden.

Der Zugriff auf den Hauptspeicher ist oftmals recht langsam, Registerretten ist eine teure Operation.

Man geht also zweckmäßigerweise wie folgt vor:

bestimme den ‘schedule’, der die wenigsten Register verbraucht.

Betrachten wir einen Ausdruck und den entsprechenden Grundblock der Drei–Adreßinstruktionen, der diesen Ausdruck auswertet.

Aufzählen aller möglichen Instruktions‘schedule’s benötigt Zeit $O(n!)$, ist also bereits für relativ kleine DAGs völlig unbrauchbar.

Wir geben einen $2^{O(n)}$ Algorithmus an, der denjenigen ‘schedule’ mit minimalem Registerverbrauch berechnet. Obwohl weiterhin exponentiell, zeigt der Algorithmus wegen seiner speziellen Aufzählungsreihenfolge der möglichen Anordnungen gute praktische Laufzeiten. Das Verfahren ist für ‘delay slot’s und mehrere Funktionseinheiten erweiterbar.

Nochmal die einfachen Fälle:

baumförmiger Graph mit einfachen Instruktionen: $O(n)$ Algorithmus von Sethi–Ullman liefert minimalen Registerverbrauch, sind ‘delay slot’s vorhanden ist das Bestimmen des optimalen ‘schedule’s bereits $\mathcal{NP}$–vollständig.

baumförmiger Graph und feste Anzahl von Register: $O(n)$ Algorithmus von Aho–Johnson liefert optimalen Instruktions‘schedule’.

für DAGs gibt es nur Heuristiken (selbst Approximationsalgorithmen haben nur sehr schlechte Laufzeiten).

Man kommt praktisch nicht darum herum beides, Register Allokation und Instruktionsscheduling miteinander zu erknüpfen, da sonst zu schlechter Code entsteht.

Eine einfache Heuristik ist z. B. das folgende: erst baumförmigen Abhängigkeitsgraph erzeugen, dann ‘schedule’ bestimmen (z. B. mit ‘list scheduling’), dann Registerverbrauch bestimmen (z. B. mit Sethi-Ulmann), dann dag konstruieren (bzw. rekonstruieren), dann gleiche Ordnung wie im ‘schedule’ verwenden und erneut Register Allokation durchführen.

Wir finden nun den optimalen ‘schedule’ und den damit verbundenen minimalen Registerverbrauch in einer bis jetzt nicht veröffentlichen schnellen Laufzeit, der in der Praxis für DAGs mit bis zu 50 Knoten, also für alle in der Praxis vorgefundenen Grundblocks, noch tolerierbare Reallaufzeiten aufzeigt.

Maschinenbeschreibung: drei Adreßbefehle, load/store–Architektur, n Register.

Grundblock und zugehöriger Datenabhängigkeitsgraph $G = (V, E)$ gegeben.

Definition 11 (‘schedule’) *‘schedule’ S von G ist eine bijektive Zuordung der Knotenmenge V zu einer geordneten Menge von Zeitpunkten $\{1, 2, ...\}$, so daß gilt: ist $v \in V$ Vorgänger von $w \in V$ in G, so ist $S(v) < S(w)$.*

‘schedule’ ist also topologische Sortierung des Graphen.

Definition 12 (Register Allokation) *Register Allokation r ist eine Zuordnung der Knotenmenge V zu einer geordneten Menge von Registern $\{1, 2, ...\}$ falls für $u, v, w \in V$ gilt: ist w Vater von v und $S(u) < S(v) < S(w)$, d. h. v wird zwischen u und w ge‘schedule’t, dann ist $r(u) \neq r(w)$ und $r(u) \neq r(v)$.*

Der Registerverbrauch $m(S)$ eines 'schedule's S ist die Anzahl der Register, die eine Register Allokation r benötigt.

Ein 'schedule' heißt optimal bezüglich Registerverbrauch, wenn er einen minimalen Registerverbrauch unter allen möglichen 'schedule's aufweist.

Berechnung eines minimalen Registerverbrauchs für einen gegebenen 'schedule' [30]:

Man hält sich Liste von benutzten Registern. Die Liste ist anfangs leer.

Wird jeden ge'schedule'ten Knoten nimmt man sich ein Register aus dieser Liste, ist die Liste leer, nimmt man sich ein neues Register.

Sind alle direkten Nachfolger eines Knotens im DAG abgearbeitet, gibt man das Register dieses Knotens wieder in die Liste. Man findet dies durch einen sogenannten Benutzungszähler heraus: der Zähler wird anfangs für jeden Knoten auf dessen outdegree gesetzt, Verarbeiten eines Knotens dekrementiert den Zähler aller seiner direkten Vorfahren.

Laufzeit ist offensichtlich $O(|E|)$, wegen Drei–Adreßbefehlen damit auch $O(|V|)$.

```
OPTREGALLOC(schedule,dag)
 1    list = empty list
 2    forall nodes in dag do
 3        use counter of node = outdegree
 4    while inst = next instruction in schedule
 5        if list is empty
 6            then reg = new register
 7            else reg = register from list
 8        use reg for inst
 9        forall children of inst in dag do
10            decrement use counter of child
11            if use counter of child equals zero
12                then put reg of child into list
```

Das Halten einer sortierten Liste der Register hilft, nur wenige 'load/store' Operationen einfügen zu müssen, falls Register ausgelagert werden müssen: man lagert die Register aus, die wenig verwendet worden sind. Man ändert Zeile (1): Erzeugen einer leeren sortierten Liste, Zeile (7): nimm immer kleinstes Element der Liste, Zeile (12): füge sortiert in die Liste ein. Man lagert schließlich die Register mit zu großer Nummer aus. Laufzeit steigt auf $O(|V| \log |V|)$.

Ein Aufzählungsalgorithmus aller möglichen 'schedule's ist z. B.

```
NCE(leaves,degrees,p)
 1    if p == n + 1 then print S
 2    else
 3        forall v ∈ leaves
 4            P = all parent nodes of v in G
 5            forall nodes w in G
 6                if w ∈ P
 7                    then degree'[w] = degree[w] − 1
 8                    else degree'[w] = degree[w]
 9            new leaves = all w in G with degree[w] == 0
10            leaves' = leaves −{v}∪ new leaves
11            S[p] = v
12            NCE(leaves',degrees',p + 1)
```

Die Rekursionstiefe von NCE() wird durch die maximale Tiefe des dags bestimmt. Die Größe der Schleife in Zeile (3) wird durch den maximalen 'outdegree' eines Knotens bestimmt. Da jedoch pro Rekursion ein Knoten weniger betrachtet werden muß, beträgt die Laufzeit $O(n! \cdot n)$. Man beachte, daß die innere Schleife stets linear in n ist; es wird ein 'schedule' gedruckt oder alle Knoten des dag werden besucht. Der Platzverbrauch ist $O(n^2)$. Man kann diesen Platz auf $O(n)$ reduzieren, indem neben S auch *leaves* und *degrees* global aufrechterhalten werden. Man fügt nach der Rückkehr aus der Funktion NCE() entsprechenden Restaurationscode ein.

Schnappschuß während der Abarbeitung eines dags:

snapnce.eps

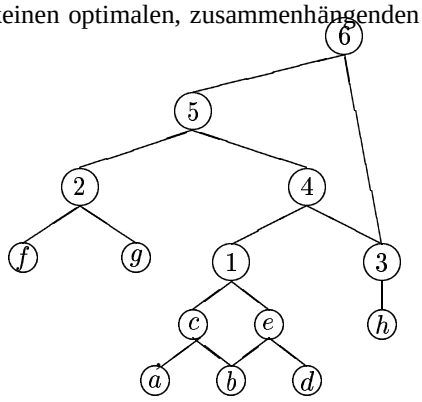
NCE() zählt also alle möglichen 'schedule's auf. Für jeden kann der Registerbedarf bestimmt werden und der optimale (je nach Kriterum) kann ausgewählt werden. Um nun die Suche zu beschleunigen, gibt es den Ansatz nur zusammenhängende 'schedules' zu betrachten.

Definition 13 (zusammenhängender 'schedule') *Ein 'schedule' S eines dags $G = (V, E)$ heißt* zusammenhängend, *wenn für jeden Knoten $w \in V$ mit Vorgängern u_1 und u_2 gilt: ist v_1 Vorgänger von u_1 und ist v_2 Vorgänger von u_2 und ist $S(u_1) < S(u_2)$, dann ist auch $S(v_1) \leq S(v_2)$.*

In einem zusammenhängenden 'schedule' wird also erst ein kompletter „Unterdag" eines direkten Vorgängers ge'schedule't, bevor ein anderer direkter Vorgänger ge'schedule't wird.

Zusammenhängende 'schedule's sind nicht unbedingt optimal bezüglich Registerverbrauchs (siehe Beispiel), man erzeugt solche 'schedule's durch Varianten von 'depth first search' (DFS). DFS erzeugt eine spezielle Menge von topologischen Sortierungen (aber eben nicht alle). Die Laufzeit zum Auffinden eines optimalen 'schedules' unter den zusammenhängenden 'schedule's beträgt $O(2^n)$ [43].

Beispiel eines dags, der keinen optimalen, zusammenhängenden 'schedule' erlaubt ([42] Bild 2).



Der dag kann nicht zusammenhängend in der Reihenfolge a, b, c, d, e, 1, f, g, 2, h, 3, 4, 5, 6 ge'schedule't werden. Es werden 4 Register für die Abarbeitung benötigt. Jeder optimale, zusammenhängende 'schedule' muß ebenfalls den Unter–dag mit Wurzel 1 zuerst abarbeiten, da dieser bereits 4 Register benötigt. Ein zusammenhängender 'schedule' kann dies allerdings nur dann tun, wenn er zuerst 1, dann 4, dann 5, dann 6 'schedule't. Um zusammenhängend zu bleiben, muß 3 vor 2 abgearbeitet werden, was in jedem Fall ein fünftes Register erfordert.

Definition 14 (Auswahl dag) *Ein* Auswahl dag $D = (Z, H)$ *ist ein azyklischer gerichteter Graph. Die Menge Z enthält alle verschiedenen Mengen z, die* NCE() *erzeugt. Es existiert eine Kante $h \in H$, die zwei Mengen $z_1, z_2 \in Z$ verbindet, wenn es in* NCE() *einen Schritt gibt, der z_2 direkt aus z_1 generiert.*

Beachte, ein Auswahl dag hat genau eine Wurzel, nämlich z_0, die Menge aller Blattknoten des Ausdruck–dags, und genau ein Blatt, nämlich $z_{\{\}}$, die leere Menge. Wir bezeichnen mit $s(z)$ die Menge aller dag–Knoten, die beim Aufruf von NCE() mit Blattmenge z bereits ge'schedule't worden sind.

Beispiel ([42] Bild 4)

Wir markieren jede Kante des Auswahl–dags mit dem dag–Knoten, der ausgewählt wurde. Eine Inschrift eines Pfades des Auswahl–dags ist die Aneinanderreihung aller Markierungen auf den Kanten des Pfades. Weiter bezeichnen wir mit G_z den durch $z \in Z$ induzierten Subgraphen in G. Es gilt: $G = G_{\{\}}$.

Lemma 1 *Jeder Pfad von der Wurzel zu einem Knoten $z \in Z$ im Auswahl–dag $D = (Z, H)$ entspricht eins–zu–eins einem 'schedule' von $s(z)$.*

Insbesondere folgt, daß verschiedene Teil'schedule's, die die gleiche Menge von Knoten enthalten, im gleichen Auswahlknoten z enden. Es gilt also:

Lemma 2 *Alle Pfade im Auswahl–dag von der Wurzel zu einem Knoten $z \in Z$ haben gleiche Länge $L(z)$.*

und somit auch

Lemma 3 *Der Auswahl–dag $D = (Z, H)$ kann in Schichten aufgeteilt werden, d. h. es gibt Mengen $L_1, L_2, \cdots, L_{L(z)}$ mit Eigenschaften:*

$$\begin{aligned} \bigcup_i L_i &= Z \\ L_i \cap L_j &= \{\} \quad \textit{für} \quad i \neq j \\ h = (z, z') \in H &\implies z \in L_i \quad \textit{und} \quad z' \in L_{i+1} \\ & \quad \textit{für geeignetes } i \end{aligned}$$

Führen wir weitere Bezeichnungen ein. Sei Σ_z die Menge aller 'schedule's für G_z, d. h. den durch z induzierten Unter–dag. Sei $S_z \in \Sigma_z$ mit minimalem Registerverbrauch in Σ_z. Und sei $m_z = m(S_z)$ diese minimale Anzahl von Registern. Damit ist $m_{\{\}}$ der Registerverbrauch eines optimalen 'schedule's für G.

Der Algorithmus NCC() berechnet statt eines Auswahl-Baums einen Auswahl-dag. Immer dann, wenn eine neue Blattmenge berechnet worden ist, wird überprüft, ob der entsprechende Knoten nicht schon vorhanden ist. Da der DAG in Schichten organisiert ist, brauchen lediglich Knoten in einer Schicht durchsucht zu werden. Ist ein solcher Knoten nicht vorhanden, erzeugt man ihn und initialisiert ihn mit S_z als Schedule mit minimalem Registerverbrauch. Im andern Fall überprüft man den Registerverbrauch und führt eventuell einen Aktualisierung des Schedules durch. Dies entspricht folgendem Algorithmus.

Übung

NCC()

Lemma 4 *Die Laufzeit von* NCC() *ist* $O(n2^{2n})$ *beschränkt.*

Die Anzahl der Knoten in D ist durch die Anzahl der Teilmengen von V beschränkt, also durch 2^n. In jedem Schritt von NCC() muß höchstens eine Schicht von Knoten in D betrachtet werden, welche höchsten Größe 2^n hat. Jeder Vergleich zweier Knoten kostet höchstens Zeit $O(n)$.

Weitere Verbesserungen des Algorithmus.

Eine Schicht L_i kann in Teilmengen $L_i^0, L_i^1, \ldots, L_i^K$ eingeteilt werden, wobei K eine obere Grenze für die minimale Anzahl von Registern ist. (Man kann K durch einen beliebigen Schedule vorberechnen). Wir speichern in L_i^k genau die Auswahlknoten mit $m_z = k$. Somit brauchen nicht mehr alle Knoten einer Schicht durchsucht zu werden. Was folgendes Lemma zeigt.

Lemma 5 *Der Vorgänger eines Auswahlknotens in* L_{i+1}^k *liegt entweder in* L_i^{k-1} *oder in* L_i^k.

Beweis: Das Anhängen eines dag Knotens v an einen Schedule S mit bisherigem Registerverbrauch $m(S)$ ergibt einen neuen Schedule S' mit: $m(S) \leq m(S') \leq m(S) + 1$, weil hoechstens ein Register mehr gebraucht wird und bestimmt nicht weniger.

Eine Aufteilung der Mengen L_i je nach Registerbedarf läßt folgenden Abhängigkeitsgraph zwischen den L_i^j entstehen: (Bild 5 in [42]).

ncc.eps

Der gezeigte Abhängigkeitsgraph erlaubt nun eine andere Auswertungsreihenfolge. Statt ‘breadth first search’ im Auswahl-dag kann man ‘depth first search’ vornehmen, womit man—sozusagen am linken Rand des Graphen—den ersten Schedule sucht, der die Schicht L_n betritt. Dieser hat notwendigerweise geringsten Registerverbrauch. Der Algorithmus nennen wir im folgenden NCV().

NCV()

Übung

Weitere Bemerkungen zum vorgestellten Verfahren:

NCV() kann auf verschiedenen Weisen parallelisiert werden:

- Parallelisierung über die L_i^k, die Operationen sind voneinander unabhängig.
- Abarbeiten eines Knoten L_i^k durch weitere Aufteilung, hier treten allerdings parallele Listen auf.
- Parallisierung der Suche nach einen vorhandenen dag Knoten

Experimentelle Ergebnisse siehe Übung.

Das Verfahren kann für ‘delay slots’ ausgebaut werden. Dies erweitert den Suchraum um eine weitere Achse, diese gibt die Anzahl der Instruktionszyklen an, die der gefundene

‘schedule’ beansprucht. Wegen des entstehenden ‘trade–offs’ zwischen Registerverbrauch und Instruktionsanzahl ist man jedoch gezwungen, sich alle optimalen ‘schedules’ bis zu einem gewissen Zeitpunkt zu halten, an dem erst entschieden werden kann, welcher Teil‘schedule’ genommen wird.

Dies erhöht die Laufzeit und den Platzbedarf des Algorithmus erheblich, weshalb nur noch dags bis zu 25 Knoten bearbeitet werden können. Mehrere Funktionseinheiten können ebenfalls durch die gleiche Methode eingebaut werden.

15 Zwischendarstellung

Abstraktionsebenen für Operatoren:

- Quellsprachenoperatoren: hoher Grad an Portabilität, jedoch werden einzeln Operatoren in viele Maschineninstruktionen übersetzt, d. h. Optimierungen auf dieser Ebene gestalten sich schwierig (haben wir in den Übungen gesehen).
- Unterhalb der Maschineninstruktion: hier verwendet man ein sehr niedriges Niveau, ‘register transfer level’ (RTL), (GCC) später muß ein Mustererkenner solche atomaren Operationen wieder zu Maschineninstruktionen zusammenbauen. Der Compiler ist ebenfalls portierbar, solange alle primitiven Operationen durch die Zielmaschine ausgedrückt werden können.
- Maschineninstruktion: diese sind selten leicht portierbar, erlauben aber den besten Ansatz für Optimierungen.

Problematisch ist stets der Informationsverlust (der sich insbesondere in Aliasen zeigt, die nicht mehr erkannt werden können).

Definition 15 (Erweiterter Grundblock) *Ein erweiterter Grundblock ist eine Folge von Instruktionen, die nur am Anfang betreten, aber an mehreren Stellen verlassen werden kann.*

16 Compilertricks

Definition 16 (Semantische Äquivalenz) *Zwei Programme sind semantisch äquivalent, wenn sie bei gleicher globaler Eingabe, die gleiche globale Ausgabe erzeugen und die gleiche Menge von Ausnahmensituationen anzeigen.*

Man beachte, daß man für die Ausnahmesituationen nur gleiche Mengen, nicht jedoch gleiche Reihenfolge, fordert. Dies kann jedoch für Realzeitanwendungen anders notwendig sein.

Ein Compiler darf nur semantik-erhaltende Transformationen durchführen, so daß das generierte Maschinenprogramm semantisch äquivalent zum Quellprogramm ist.

‘copy propagation’ Das Erstellen von Kopien von Variablen wird vermieden:

```
x=i+1; j=i; y=j+1;

x=i+1; j=i; y=i+1;
```

Damit kann später die Kopieraktion eliminiert werden. Das resultierende Programm kann leichter optimiert werden. Hierbei muß man beachten, daß keine Überschreibung des kopierten Wertes zwischenzeitlich erfolgt.

'register allocation' Zuordnung von Variablen an Register, so daß möglichst wenige Speicheroperationen ausgeführt werden müssen. Hat die 'hardware' nicht direkt sichtbare Register (Schattenregister, 'renaming' Technik), so sollte deren Einsatz optimiert werden. Dies kann insbesondere mit spekulativer Auswertung kombiniert werden.

'instruction selection' Zwischendarstellungsoperationen müssen Maschinenbefehle zugeordnet werden. Hier stehen oftmals verschiedene Möglichkeiten zur Auswahl, die kontextabhängig zu unterschiedlichen Laufzeiten führen. Dies führt bei 'superscalaren' Prozessoren auch zu Konflikten. So kann eine Multiplikation mit einer Zweierpotenz durch eine 'shift' Operation ersetzt werden. Sollen jedoch zwei parallel ausgeführt werden, kann man vor die Wahl gestellt sein. Desweiteren gibt es Prozessoren (z. B. PRAM Prozessor), wo diese Optimierung keinen Vorteil bringt.

'instruction scheduling' Anorden der Instruktion in möglichst optimaler Reihenfolge. Insbesondere sollen 'pipeline interlock', 'pipeline hazards' vermeiden, 'delay slots' gefüllt und parallele Einheiten genutzt werden. Korrektes 'chaining' von 'pipelines' soll ermöglicht werden. Vorhandene nop Instruktionen können gegebenenfalls mit spekulativen Auswertungen ersetzt werden.

'redundancy elimination' Nutzen von gemeinsamen Unterausdrücken,

'strength reduction' Ersetzen von Multiplikationen in Schleifen durch Additionen (vorallem bei Adressierungsberechnungen)

'procedure inlining' Es werden der Eingangs– und Endcode einer Procedur eingespart. Desweiteren kann durch den entstehenden größeren Grundblockgraph besser optimiert werden. 'Inlining' erhöht jedoch die Codelänge und kann somit auch zu einer Erhöhung der Laufzeit Führen (insbesondere wegen des 'cache'–Verhaltens der Maschine). Man kann Heuristiken verwenden (z. B. kleine nicht rekursive Prozeduren werden eingebunden) oder es in die Hochsprache integrieren (Schlüsselwort `inline`).

'constant propagation' and 'constant folding' Sind alle Operanden eines Ausdrucks (bzw. Teilausdrucks) konstant, so ist auch der Ausdruck konstant und kann vorberechnet werden. Dies kann Auswirkungen auf Schleifenbedingungen haben, so daß ganze Code-Blöcke elimiert werden können. Dies kann verstärkt nach 'inlining' (Übergabe von Konstanten) ausgenutzt werden.

'common subexpression elimination' Gleiche Teilausdrücke werden nur einmal berechnet. Hier besteht aber ein 'trade-off' zwischen Wiederberechnen und Zwischenspeichern. Eine einfache Instruktion kann sich als günstiger als ein Speichern und Laden erweisen. Einiges kann hier ebenfalls in der Hochsprache erfolgen, für viele Zwischenwerte (vorallem bei Feldzugriffen) ist dies jedoch dort nicht sichtbar.

'dead check removal' Überflüssige Laufzeitchecks können elimiert werden, z. B. bei mathematischen Routinen, wenn dies schon vom Programmierer vorgesehen wird, oder in Programmiersprachen mit Grenzenüberprüfung (Pascal), wenn die Bedingung als sicher gilt.

‘unreachable code elimination’ Dieser Optimierungsschritt elimiert Instruktionen, die nie erreicht werden können. Scheinbar sollten diese in der Hochsprache nicht auftreten, sie ergeben sich jedoch oft nach ‘procedure inlining’ oder ‘constant propagation’.

‘loop optimization’ Herausziehen von unnötigen Instruktionen aus Schleifen,

‘code motion’ Verschieben von Instruktionen aus Bereichen (vorallem Schleifen), die häufig ausgeführt werden in solche die weniger oft ausgeführt werden. Diese Optimierung kann auch häufig bereits in der Hochsprache durchgeführt werden. Durch ‘inlining’ kann dies jedoch in einigen Fällen erst anwendbar werden.

‘loop unrolling’ Der Schleifenblock wird mehrfach hintereinander gesetzt. Hierdurch wird einerseits der Grundblock (oder ‘trace’) vergrößert, und andererseits die Anzahl der Schleifenbedingungstests reduziert.

‘unswitching’ Erzeugen mehrfacher Versionen einer Schleife falls nicht veränderliche Bedingungen innerhalb der Schleife, laufzeitabhängig auftreten.

‘speculative execution’ Operationen werden vorgezogen ausgeführt, obwohl noch nicht sicher fest steht, daß deren Ergebnisse auch verwendet werden. Dies ist besonders zum Füllen von ‘delay slots’ und ‘pipeline delays’ sinnvoll. ‘speculative execution’ wird von einigen Prozessoren in Hardware durchgeführt (Benutzung von Schattenregistern), kann aber auch vom Compiler in bestimmten Fällen ausgenutzt werden, da nur das Speichern gegebenenfalls unterdrückt werden muß. Man beachte jedoch, daß die Menge der erzeugten Ausnahmen eventuell Probleme bereiten kann:

```
if(...)  y=1.0/r;
z=y+1.0
```

Wird der Ausdruck für `y` vor die Bedingung gezogen, kann eine Ausnahme „Division durch Null“ unter Umständen unerwünscht auftreten.

‘branch optimization’ Es kann sein, daß beide Verzweigungsrichtungen eines bedingten Sprunges unterschiedlich Charakteristiken aufweisen. Man sollte also berücksichtigen, daß hier die Hardware bei Sprüngen richtig ausgenutzt wird. Dies ist bei Prozessoren mit aufwendiger ‘branch prediction’ eine anspruchsvolle Aufgabe. Hilfestellungen in der Hochsprache können dem Compiler Informationen liefern, die er zu einer statischen Sprungvorhersage ausnutzen kann.

‘pipeline optimization’ Obwohl der Compiler beim ‘scheduling’ versucht guten Code für eine ge‘pipeline’te Architektur zu erzeugen, kann eine Nachoptimierung (z. B. durch Einführen von nops) in einem ‘peep hole pass’ weitere Verbesserungen bringen.

‘cache optimization’ Daten und auch Programmcode werden so im Speicher angeordnet, daß die ‘caches’ gut ausgenutzt werden. So kann man z. B. Schleifenanfänge auch auf ‘cache’-Zeilenanfänge legen. Konsekutiv gebrauchte Daten können zum Beispiel in Datenstrukturen auch konsekutiv abgelegt werden.

‘peephole optimization’ Der Compiler schiebt ein kleines Fenster über bereits erzeugten Code und versucht weitere lokale Optimierungen einzubauen. Dieser Schritt kann insbesondere auch von einem nachgeschalteten Assembler durchgeführt werden.

‘local optimization’ innerhalb Grundblocks, bzw. innerhalb eines erweiterten Grundblocks,

‘global optimization‘ innerhalb einer Prozedur

'inter–procedural optimization' Diese arbeitet über Prozedurgrenzen hinweg, allerdings müssen die Prozeduren innerhalb eines Moduls übersetzt werden, was der Modularität der Programmentwicklung Grenzen setzt. Gibt es gängige Compiler, die diese Art der Optimierung durchführen?

Bemerkungen für das Schreiben von effizienten C–Routinen:

Zeiger–Felder Häufig sind indizierte Felder innerhalb von Schleifen effizienter als Zeiger, da dem Compiler ermöglicht wird, die Laufvariable zu elimieren und die Adreßvergleiche zur Schleifenbedingung zu verwenden.

Variablenbenutzung Guten Compilern kann man durch 'single assignment' Strategien helfen, d. h. man benutzt eine Variable nur einmal auf der linken Seite einer Zuweisung. Dies ist insbesondere in Schleifenblöcken sinnvoll, da dann leichter der Sprung vom Ende zum Anfang unter Beibehaltung der Registerallokation durchgeführt werden kann.

Registervariablen erübrigen sich.

Schleifentyp Ist man sich sicher, daß mindestens ein Schleifendurchlauf gemacht wird, sind `do`–Schleifen effizienter als `while`–Schleifen.

Schleifenrichtung Abwärtszählende Schleifen sind effizienter als aufwärtszählende, da Prozessoren häufig einfacher auf 0 vergleichen können als auf andere Werte.

Schleifenbedingung Diese sollte so gewählt werden, daß dem Compiler möglichst wenige Einschränkungen auferlegt werden. Eine einfache Abfrage auf 0 hat oft Vorteile gegenüber Vergleichen.

Parameterübergabe Moderne Compiler übergeben Variablen in Registern des Prozessor. Kenntnis der Anzahl und der Typen (insbesondere der Wortbreiten) dieser Parameterregister ermöglich ein „richtiges" übergeben von Variablen. So sollten z. B. 'double precision' Werte bei bestimmten Processoren (Sparc) in Registerpaaren übergeben werden.

Konstante Schleifenlänge Oft können Schleifen mit konstanter Schleifenbedingung vom Compiler besser gehandhabt werden, so daß es sich eventuell lohnt eine variable Schleifenlänge in der Hochsprache bereits mit konstanten Blöcken zu verschachteln.

Folgen von Parameterübergaben Wird ein Parameter an mehrere Funktionen in einer Aufruffolge weitergereicht, so sollte der Parameter möglichst an gleicher Stelle in der Parameterliste stehen. In vielen Prozessoren mit entsprechendem Parameterübergabeprotokoll erübrigen sich so Kopieroperationen.

Funktionsdefinition Damit der Compiler von 'inline'-Code Gebrauch machen kann, müssen entsprechende Funktionen in einem Modul untergebracht werden. 'Inlining' ist jedoch in C nicht sehr gut gehandhabt. Deklariert man eine Funktion als 'inline' wird sie immer so verwendet. Eine flexible Handhabung an der Aufrufstelle ist nicht vorgesehen. Hier hilft nur klassisches 'inlining' mit dem Präprozessor.

Lokalität In der Definition von Datenstrukturen ('struct') lohnt sich eine genauere Analyse, in welcher Reihenfolge die Komponenten deklariert werden. Einerseits sollte man auf 'alignment' der Datentypen achten ('double precision' Werte in 32-Bit Maschinen), andererseits können die 'cache'-Eigenschaften des Prozessors berücksichtigt werden, da meistens 'cache'-Zeilen geladen werden und man so dafür sorgen kann, daß Lokalität beachtet wird.

Code-Plazierung Schleifen Blöcke sollten ohne überflüssigen Code geschrieben werden. Insbesondere sollte man Bedingungen in den Schleifenblöcken weitgehend vermeiden.

Man stellte fest:

- Compiler nutzen keine komplexen Operationen

17 Cache Strategien

MIPS 'cache' Implementierungen

18 Weitere Literatur

Literatur

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal ACM*, 23(3):488–501, 1976.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[3] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. *SIGPLAN*, 23(7):308–317, July 1988.

[4] W. Ambrosch, M. Ertl, F. Beer, and A. Krall. Dependence–conscious global register allocation. In *Proceedings of the Conference on Programming Languages and System Architectures*. Springer Lecture Notes in CS 782, March 1994.

[5] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proceedings ACM SIGPLAN PLDI'89*, pages 258–263, 1989.

[6] D. Bernstein, J. M. Jaffe, and M. Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM Journal on Computing*, pages 1098–1127, 1989.

[7] David Bernstein, Hanran Boral, and Ron Pinter. Optimal chaining in expression trees. *IEEE Transactions on Computer*, 37(11):1366–1374, November 1988.

[8] David Bernstein and R. Y. Pinter et al. Optimal scheduling of arithmetic operations in parallel with memory access. In *ACM*, 1984.

[9] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay on one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–67, January 1989.

[10] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN*, 26(6):241–255, June 1991.

[11] David Bernstein, Michael Rodek, and Izidor Gertner. On the complexity of scheduling porblems for parallel/pipelined machines. *IEEE Transactions on Computer*, 38(9):1308–1314, September 1989.

[12] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. *SIGPLAN notes*, 26(6):122–131, June 1991.

[13] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *SIGPLAN*, 26(6):229–240, June 1991.

[14] Marc M. Brandis. *Optimizing Compilers for Structured Programming Languages*. PhD thesis, ETH Zürich, 1995.

[15] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings ACM SIGPLAN PLDI'89*, pages 47–57, 1989.

[16] G. J. Chaitin. Register allocation & spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):201–207, 1982.

[17] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[18] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *ACM SIGPLAN Notices*, 19(6):222–232, 1984.

[19] Fred C. Chow and John L. Hennessy. The priority–based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[20] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallet. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C–30(7):460–477, July 1981.

[21] J. J. Dongarra and A. R. Jinds. Unrolling loops in fortran. *Software Practice and Experience*, 9(3):219–226, 1979.

[22] Kemal Ebcioglu and Alexandru Nicolau. A global resource–constrained parallelization technique. In *Proceedings of the 3rd International Conference on Supercomputing*. ACM Press, 1989.

[23] Hans Eberle. Aktuelle techniken zur leistungssteigerung von mikroprozessoren. Technical Report 237–1995, ETH Zürich, July 1995.

[24] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceeding of 5th Workshop on Compilers for Parallel Computers*, pages 503–516, 1995.

[25] John R. Ellis. *Bulldog: A Compiler for VLIW Architechtures*. PhD thesis, Yale University, 1984.

[26] John R. Ellis. *Bulldog: A Compiler for VLSI Architectures*. The MIT Press, Cambridge, Massachusetts, 1985.

[27] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[28] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *SIGPLAN*, 19(6):37–47, June 1984.

[29] Michael St.O. Franz. *Code–Generation On–the–Fly: A Key to Portable Software*. PhD thesis, ETH Zürich, 1994.

[30] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11), 1974.

[31] Gibbons and Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Compiler Construction*, 21(7):11–16, July 1986.

[32] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 442–452, July 1988.

[33] K. J. Gough. Register allocation in the gardens point compilers. In *ACSC18*, January 1995. Paper: ftp://pluto.fit.qut.edu.au/pub/papers/regalloc.ps.Z.

[34] Robert Griesemer. Scheduling instructions by direct placement. In *Proceedings of the 4th, International Conf. on Compiler Construction*, pages 229–235, October 1992.

[35] Thomas Gross. Code optimization of pipeline constraints. Technical Report 83–255, Computer Systems Lab, EECS, Stanford University, December 1983.

[36] John L. Hennessy and Thomas Gross. Postpass code optimization of pipelined constraints. *ACM Transactions of Programming Languages and Systems*, 5:422–448, July 1983.

[37] Wei-Chung Hsu. *Register Allocation and Code Scheduling for Load/Store Architectures*. PhD thesis, University of Wisconsin – Madison, 1987.

[38] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions of Software Engeneering*, 15(10):1252–1262, October 1989.

[39] Suneel Jain. Circular scheduling: A new technique to perform software pipelining. *SIGPLAN*, 26(6):219–228, June 1991.

[40] Wolfgang Karl. *Parallele Prozessor–Architekturen*, volume 93 of *Reihe Informatik*. BI Wissenschaftsverlag, 1993. ISBN 3-411-16451-4.

[41] Christoph W. Keßler. Code–optimierung quasiskalarer vektorieller grundblöcke. Master's thesis, University of Saarbrücken, Germany, 1990. in german.

[42] Christoph W. Keßler. Scheduling expression DAGs for minimal register need. In *submitted to: ISTCS'96 Jerusalem*, 1996.

[43] Christoph W. Keßler and Thomas Rauber. Generating optimal contiguous evaluations for expression DAGs. *Computer Languages*, 21(2):113–127, 1995.

[44] Christoph W. Keßler, Thomas Rauber, and Wolfgang J. Paul. A randomized heuristic approach to register allocation. In *Proceedings of PLILP'91 3rd Int. Symp. on Programming Language Implem. and Logic Programming*, number 528 in LNCS, pages 195–206. Springer, August 1991.

[45] Christoph W. Keßler, Thomas Rauber, and Wolfgang J. Paul. Scheduling vector straight line code on vector processors. In R. Giegerich and S. L. Graham, editors, *Code Generation — Concepts, Tools, Techniques*, WICS. Springer, 1992.

[46] Christoph W. Keßler and Helmut Seidl. Integrating synchronous and asynchronous paradigms: he fork95 parallel programming language. In *Proc. MPPM-95 Int. Conf. on Massively Parallel Programming Models*. IEEE CS press, October 1995.

[47] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings, 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 726–737, 1990.

[48] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN*, 23(7):318–328, July 1988.

[49] Mark Leone and Peter Lee. Lightweight run time code generation. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantic–Based Program Manipulation*, June 1994.

[50] J. Llosa, M. Valero, and E. Ayguade. Bidirectional scheduling to minimize register requirements. In *Proceeding of 5th Workshop on Compilers for Parallel Computers*, pages 534–554, 1995.

[51] Wen mai W. Hwu and Pohua P. Chang. Efficient instruction sequencing with inline target insertion. *IEEE Transactions on Computing*, 1992 ??

[52] Rajeev Motvani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Report 698, Courant Institute, July 1995.

[53] Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua (Editors). *Advances in Languages and Compilers for Parallel Processing*. MIT Press (Research Monographs in Parallel and Distributed Computing), 1991.

[54] Krishna V. Palem and Barbara B. Simons. Scheduling time–critical instructions on RISC machines. *ACM Transactions of Programming Languages and Systems*, 15(4), September 1993.

[55] S. Pinter. Register allocation with instruction scheduling: a new approach. In *PLDI*, 1993.

[56] Todd A. Proebsting and Charles N. Fischer. Linear–time, optimal code scheduling for delayed–load architectures. *SIGPLAN*, 26(6):256–267, June 1991.

[57] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4:226–248, 1975.

[58] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal ACM*, 17:718–728, 1970.

[59] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the 3rd International Conference on Achitectural Support for Programming Languages and Operating Systems*, pages 290–302, April 1989.

[60] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on MIcroarchitecture*, pages 93–102, November 1991.

[61] D. A. Spuler. *C++ and C Efficiency*. Prentice Hall, 1992.

[62] D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Technical Report 94/3, James Cook University, Department of Computer Science, 1994. Abstract: http: //coral. cs. jcu. edu.au/ftp/web/research/techreports/ftp94.html#94-3, Paper: http: //coral.cs.jcu.edu.au/ftp/pub/techreports/94-3.ps.gz.

[63] D. A. Spuler and A. S. M. Sajeev. Compiler detection of function call side effects. Technical Report 94/1, James Cook University, Department of Computer Science, 1994. Abstract: http: //coral.cs. jcu. edu.au/ftp/web/research/techreports/ftp94.html#94-1, Paper: http: //coral.cs.jcu.edu.au/ftp/pub/techreports/94-1.ps.gz.

[64] R. Venugopal and Y. N. Srikant. Scheduling expression trees with reusable registers on delayed-load architectures. *Computer Languages*, 21(1):49–65, 1995.

[65] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau*. Springer Verlag, 1992. ISBN 3-540-55704-0.