Backend

Kernaufgaben

- Befehlsauswahl
- Befehlsanordnung
- Registerzuteilung
- beeinflussen sich gegenseitig
  - Phasenkopplungsproblem

Zusätzlich:

- Spezial-Optimierungen für die jeweilige Zielarchitektur
- Implementierung der Konventionen der Laufzeitumgebung
- Implementierung komplexer Hochsprachen-Konstrukte (z.B. Vererbung)

SSA basiertes Backend

Erhalte SSA im Backend

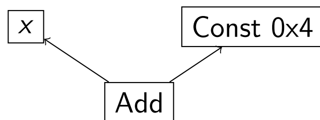
- Einheitliche Programmrepräsentation im Übersetzer
- Optimierungen teils wiederverwendbar

Ablauf des Backends

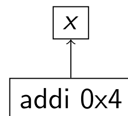
- Befehlsauswahl
- Anordnung
- Registerzuteilung
- Nach-Anordnung im Rahmen der Registerzuteilung
- SSA-Abbau

Befehlsauswahl

- Graph-Transformation des Zwischensprachen-Graphs in einen Maschinen-Graph
- Muster aus Zwischensprachen-Knoten werden in Knotenmenge aus Maschinen-Knoten überführt



wird zu

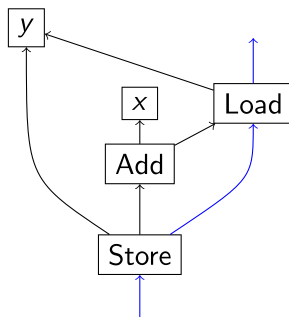


Befehlsauswahl

- Datenabhängigkeiten in SSA-Grundblöcken sind DAGs, keine Bäume
- Daher: Termersetzungsverfahren nur bedingt anwendbar
- Es existieren Erweiterungen von TES für DAGs
  - CGGG von Boesler, IPD
  - PBQP-Verfahren von Eckstein, König und Scholz, TU Wien
  - BURS Code-Generator des Java Hotspot Compilers (SUN)
- Brachial-Methode: Aufbrechen der DAGs in Bäume
- Generative Verfahren haupsächlich zur Mustersuche
- Nebenbedingungen schwer berücksichtigbar

Befehlsauswahl – Nebenbedingungen

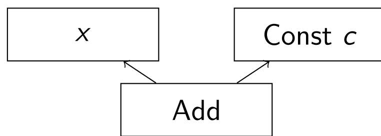
Load/Op/Store-Befehle bei ia32



- kann für ia32-Prozessoren in einem Befehl implementiert werden
- Allerdings darf das Load von keinem anderen Befehl verwendet werden
- Mit generativen Verfahren nicht möglich
- Ersetzung muss von Hand implementiert werden

Befehlsauswahl – Nebenbedingungen

Immediates bei Arm



- Um Konstante in den Befehl „reinzuziehen“ muss sie bestimmte Eigenschaften erfüllen:

$$c = x \texttt{ ror } (2k + 1)$$

  wobei x eine 8-bit Zahl ist.
- Laden von beliebigen Konstanten mit generativen Verfahren nicht möglich
- Ersetzung muss von Hand implementiert werden

Befehlsauswahl

Fazit

- Generische Verfahren ermöglichen bequeme Spezifikation der Muster
- Eigenheiten der Zielmaschinen aber nur ungenügend darstellbar
- Daher oft Hybrid-Verfahren (Softwaretechnisch „bedenklich“)
- Ausprogrammieren von Hand nicht unbedingt von Nachteil

Konventionen der Laufzeitumgebung

Application Binary Interface

- Wissenschaftlich irrelevant
- In der Praxis essentiell
- Weitere Parameterisierung des Backends durch Laufzeitumgebung
- Bei den meisten Sprachen/Betriebssystemen nicht standardisiert, deswegen auch oft übersetzerabhängig

Konventionen der Laufzeitumgebung
C/C++

- Bestimmt hauptsächlich Modalitäten beim Funktionsaufruf
  - Welche Parameter werden in Registern, welche auf dem Keller übergeben
  - Wer bereinigt den Keller? (Aufrufer oder Aufgerufener)
  - Welche Register müssen von einer aufgerufenen Funktion gesichert werden?
- Aber auch
  - „name mangling“
  - Exception Handling
  - Ausrichtung der Datentypen (alignment)
  - Gebrauch der Registersätze
- Zentral für das Funktionieren des Gesamtsystems (Bibliotheken)
- Bei C/C++ Überstetzungsziel: Prozessor, Betriebssystem, Compiler

segment word size	calling convention, operating system, compiler	parameters in registers	parameter order on stack	stack cleanup by	comments
16 bit	cdecl		C	caller	
	pascal		Pascal	function	
	fastcall Microsoft (non-member)	ax, dx, bx	Pascal	function	return pointer in bx
	fastcall Microsoft (member function)	ax, dx	Pascal	function	`this` on stack low address. return pointer in ax
	fastcall Borland	ax, dx, bx	Pascal	function	`this` on stack low address. return ptr on stack high addr.
	Watcom	ax, dx, bx, cx	C	function	return pointer in si
32 bit	cdecl		C	caller	
	stdcall		C	function	
	pascal		Pascal	function	
	Gnu		C	hybrid	
	fastcall Microsoft	ecx, edx	C	function	return pointer on stack if not member function
	fastcall Gnu	ecx, edx	C	function	
	fastcall Borland	eax, edx, ecx	Pascal	function	
	thiscall Microsoft	ecx	C	function	default for member functions
	Watcom	eax, edx, ebx, ecx	C	function	return pointer in esi
64 bit	Windows (Microsoft, Intel)	rcx/xmm0, rdx/xmm1, r8/xmm2, r9/xmm3	C	caller	Stack aligned by 16. 32 bytes backing space on stack for register parameters. The specified registers can only be used for parameter number 1, 2, 3 and 4, respectively.
	Linux, BSD (Gnu, Intel)	rdi, rsi, rdx, rcx, r8, r9, xmm0-7	C	caller	Stack aligned by 16. Red zone below stack.

Quelle: `http://www.agner.org`

Spezialoptimierungen

Viele Prozessoren haben Eigenschaften, die durch Befehlsauswahl nicht ausnutzbar sind.

- Spezielle Registerbänke
- Vektor-Einheiten
- Bedingte Befehle (Predication)
- etc.

Spezialoptimierungen

Beispiel – Bedingungsregister des PowerPC

- PowerPC besitzt acht 4-bit Register
  um Vergleichsergebnisse zu speichern.
- Es sind Rechenoperationen auf für diese Register vorhanden
  (and, or, xor, etc.)
- Können direkt von Sprungoperationen verwendet werden
- Ideal, um boole'sche Terme (Bedingungen) darin zu berechnen
- Das spart Allzweck-Register
- Allerdings sind die Bedingungsregister nicht auslagerbar
- Deshalb muss darauf geachtet werden, dass die Terme niemals mehr als diese acht Register in Anspruch nehmen

Implementierung von Hochsprachenkonstrukten

Viele Sprachen bieten Konstrukte, die nicht direkt mit Prozessorbefehlen implementierbar sind

- Vererbug, Mehrfachvererbung, Polymorphe Aufrufe
- Ausnahmebehandlung
- Spezielle Arithmetik
  - Dezimal-Rechung in Cobol
  - 64-bit Arithmetik auf 32-bit Maschinen

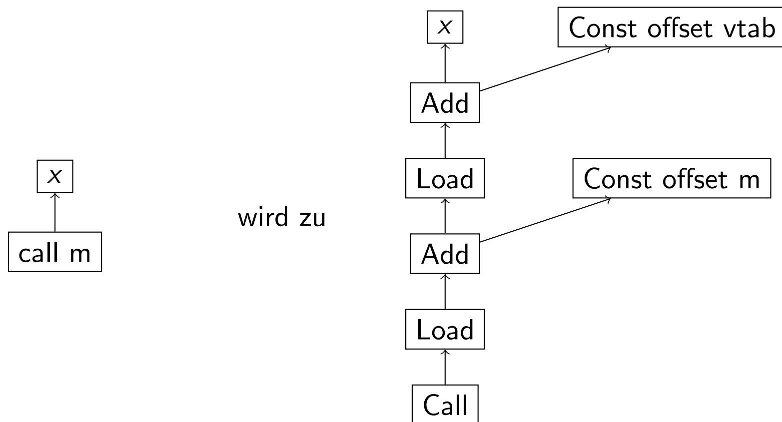
Polymorphe Aufrufe

- Aufgerufene Methode abhängig vom dynamischen Typ
- Der ist i.A. ist erst zur Laufzeit bekannt
- Lege pro Klasse eine Tabelle `vtab` mit Zeigern auf alle polymorph rufbaren Methoden an
- Füge jeder Klasse ein Feld `vtab_ptr` mit Zeiger auf diese Tabelle hinzu
- Alle polymorphen Aufrufe müssen nun wie folgt implementiert werden

Polymorphe Aufrufe

Graph-Transformation

Der Knoten *obj* beschreibt die Adresse eines Objektes der Klasse T mit einer Methode m



Ausnahmebehandlung

Auch vom Betriebssystem abhängig, wegen Ausnahmen,
die vom Prozessor ausgelöst werden

- Teilen durch 0
- Speicherzugriffsfehler
- evtl. noch andere

Verschiedene Mechanismen:

- Signale unter Unix
- Structured Exception Handling bei Win32

Ausnahmebehandlung

Prinzipielles Vorgehen

- Halte einen zusätzlichen Keller
- Bei jedem try-Block wird eine Datenstruktur auf den Keller gelegt, die anzeigt, welche Ausnahmen gefangen werden
  - Ermöglicht finden des innersten catch-Blocks bei Auftreten der Ausnahme
- catch-Block wird aber im Kontext der enthaltenden Funktion abgearbeitet
- Das heisst: Register und Keller müssen den selben Zustand haben wie bei Aufruf der Funktion, die die Ausnahme ausgelöst hat
- Alle Register inklusive Kellerpegel und Schachtelzeiger müssen auf dem Ausnahmenkeller gesichert werden
- Das kostet Laufzeit

Ergo

Der Übersetzer sollte so viele Ausnahmen wie möglich wegoptimieren!

Schlussfolgerungen

- Übersetzer muss eine Menge an Zusatzaufgaben erledigen, die über simple Code-Erzeugung hinausgehen
- Generative Verfahren leisten oft nur Bruchteil dessen
- Viel von Hand auszuprogrammieren
- Moderne Hochsprachen-Konstrukte kostspielig und aufwendig zu übersetzen