# Phasen der Codegenerierung

1. Vorverarbeitung
   - Semantik-basierte Optimierungen
   - Basis: Zwischencode (abstrakter Syntaxbaum)
2. eigentliche (proper) Codegenerierung
   - Erzeugung des Zielcodes (mit Optimierung)
   - Bsp.: Befehlsauswahl durch BURS
3. Nachbearbeitung
   - Maschinen-spezifische Optimierungen
   - Basis: Sequenz von Maschinenbefehlen (in Assemblerdarstellung)

# Vorverarbeitung (1)

- constant folding (Vereinfachung konstanter Teile)
  - Bsp.: 2*pi*x/360  x*1.745e-2
- strength reduction (Reduktion von Operatorkosten)
  - Bsp.: `8*x` → `x<<3`
  - in BURS integrierbar
  - Problem: identisches Verhalten?
    (a) Rechnung modulo einer Zweierpotenz (`x<<3`)
    (b) mglw. Exception bei arithmetischem Überlauf (`8*x`)
- dead code elimination (Löschen unerreichbaren Codes)
  - if-Bedingung kann nur einen Wahrheitswert annehmen: ein Zweig unerreichbar
  - Code zwischen goto mit immer ausgeführtem Sprung und nächstem textuellen Sprungziel unerreichbar

# Vorverarbeitung (2)

## Beispiele für arithmetische Vereinfachungen und strength reduction

| operation | ⇒ | replacement |
|---|---|---|
| `E * 2 ** n` | ⇒ | `E << n` |
| `2 * V` | ⇒ | `V + V` |
| `3 * V` | ⇒ | `(V << 1) + V` |
| `V ** 2` | ⇒ | `V * V` |
| `E + 0` | ⇒ | `E` |
| `E * 1` | ⇒ | `E` |
| `E ** 1` | ⇒ | `E` |
| `1 ** E` | ⇒ | `1` |

**Figure 4.88** Some transformations for arithmetic simplification.

# Vorverarbeitung (3), Schleifen

- loop-invariant code motion: Verschieben von Iterations unabhängigen Ausdrücken vor die Schleife
- (partial) loop unrolling: Expansion des Rumpfes
  - Verringerung des Aufwands für Sprünge
  - Optimierungen zwischen Iterationen (Bsp. Fibonacci)
- loop fusion: Zusammenfassen einer Sequenz von Schleifen zu einer Schleife
- loop tiling:
  Aufspaltung einer Schleife in zwei geschachtelte, z.B. via div/mod
  - Optimierung des Cache-Verhaltens
  - Parallelisierung
- affine Transformationen (Vorlesung Schleifenparallelisierung)

# Vorverarbeitung (4), Prozedur-Cloning

## Duplikation eines Prozedur-Rumpfes mit Spezialisierung

- Bsp.: Spezialisierung von $\sum_{p=0}^{n} a_p x^p$ für $x = 1.0$:

```
double power_series_x_1(int n, double a[]) {
   double result = 0.0;
   for(int p=0; p<=n; p++)
      result += a[p];
   return result;
}
```

# Vorverarbeitung (5), Prozedur-Inlining (a)

- Expansion/Einkopieren eines Rumpfes an der Aufrufstelle
  - Baumteile für Parameterübergabe, Rumpf, Ergebnisrückgabe
  - Expansion der Parameterberechnung (call-by-name) gewöhnlich unerwünscht
  - ermöglicht oft zusätzliche Vereinfachungen,
    sei n tot (nicht lebendig)
    dann wird print_square(3) zu:
    printf(“square = %d\n”, 9);

```
void S {
        ...
        print_square(i++);
        ...
}

void print_square(int n) {
        printf("square = %d\n", n*n);;
}
```

**Figure 4.89** C code with a routine to be in-lined.

```
void S {
    ...
    {int n = i++; printf("square = %d\n", n*n);}
    ...
}

void print_square(int n) {
    printf("square = %d\n", n*n);
}
```

**Figure 4.90** C code with the routine in-lined.

# Nachbearbeitung (1)

Generelles Prinzip:
Gucklochoptimierung (peephole optimization)

Aspekt 1:
Generierung von Ersetzungsmustern

Aspekt 2:
Lokalisierung und Ersetzung von Instruktionssequenzen

# Nachbearbeitung (2)

- Aspekt 1: Generierung von Ersetzungsmustern
  - Tripel: zu ersetzen | Einschränkungen → Ersetzung

    `Load_Reg` $R_a$`,`$R_b$`; Load_Reg` $R_c$`,`$R_d$ | $R_a=R_d$`,`$R_b=R_c$

    → `Load_Reg` $R_a$`,`$R_b$
  - Allzweck-Gucklochoptimierer für einen gegebenen Befehlssatz
    - erfordert kompromisslose Einhaltung der Semantik
    - Ersetzung muss auch auf den Bedingungsregistern den gleichen Effekt haben (Shift/`*2`, Increment/`+1`)
    - viele Muster werden selten oder nie zur Anwendung kommen
  - spezieller Gucklochoptimierer
    - Ersetzungen können die Semantik gezielt optimieren
    - `Load_Const 1,`$R_a$`; Add_Reg` $R_b$`,`$R_c$ | $R_a=R_b$`,`

      `is_last_use(`$R_b$`)` → `Increment` $R_c$

      (unterschiedlicher Inhalt von $R_b$)
    - etwa 100 meist einfache Muster sind ausreichend

# Nachbearbeitung (3)

- Aspekt 2: Lokalisierung und Ersetzung von Instruktionssequenzen
  - i.d.R. auf Code für Basisblöcke beschränkt
  - lineare Mustersuche und -ersetzung
  - wichtig: den Suchzeiger genügend weit zurücksetzen!
  - Optimierungsproblem: NP-vollständig
  - Heuristik : längster Match (Automat ähnlich einem Scanner)

# Erzeugung von Maschinencode

- Codesequenz liegt in Compiler-interner Darstellung vor
- Ziel: ausführbare Objektdatei
- Möglichkeiten
  - C als portable Zielsprache
    - Ausnutzung der Optimierungsmöglichkeiten existierender Compiler
    - Projekt “C--”: C mit reduziertem Sprachumfang, welches effiziente Optimierung ermöglicht
  - Umsetzung in Assemblercode
    - Mnemonics (Namenskürzel) statt Bitmuster für Befehle
    - symbolische Namen für Variablen und Sprungziele

# Komponenten eines Objektprogramms (1)

1. Codesegment
   - enthält auszuführende Maschinenbefehle
   - feste Größe
   - wird üblicherweise zur Laufzeit nicht geändert
2. Datensegment
   - Speicherplatz für statische globale Datenobjekte
   - feste Größe
   - wird zur Laufzeit normalerweise oft gelesen und beschrieben

# Komponenten eines Objektprogramms (2)

1. Codesegment

2. Datensegment
   - Speicherplatz für statische globale Datenobjekte
   - feste Größe
   - wird zur Laufzeit normalerweise oft gelesen und beschrieben

3. Stacksegment
   - Speicherplatz für statische Datenobjekte, die nur während der Abarbeitung eines Blocks existieren
   - Größe von dynamischer Aufrufstruktur abhängig
   - wird zur Laufzeit normalerweise oft gelesen und beschrieben

# Komponenten eines Objektprogramms (3)

1. Codesegment
2. Datensegment
3. Stacksegment
4. Register
   - extrem kurzfristiger Speicherplatz für Adressen und Operanden/Ergebnis von Maschinenbefehlen
   - werden erst während der Abarbeitung mit Daten des Programms belegt

# Assembler, Linker und Loader

- Assembler erzeugt Bitmuster für
  - Mnemonics (Befehlskürzel)
  - benannte Speicherstellen und Sprungziele
  - Registernamen, Adressierungsmodi, etc.
- Linker
  - verbindet mehrere Objektcodes (auch in Libraries)
  - ersetzt symbolische Funktionsnamen durch Sprungziele
- Loader
  - unter Kontrolle des Betriebssystems
  - reserviert und belegt die Segmente für das Programm
  - übergibt die Kontrolle an eine Aufrufstelle

# Linken von Objektprogrammen (1)

- statisches Linken
  - Verbinden mehrerer Objektprogramme (und Libraries) zu einem neuen Objektprogramm
  - Vorteil: kein Aufwand zur Laufzeit
    - Adressen von Funktionen sind aufgelöst
    - Code liegt im Speicher
  - Nachteile
    - Programme können groß werden
    - auch nicht benötigte Teile belegen Speicherplatz
    - kein Sharing zwischen mehreren Prozessen
- Alternative: dynamisches Linken
  (während der Programmausführung)

# Linken von Objektprogrammen (2)

- Ausgangspunkt: Adressen jedes Objektprogramms sind relativ zu 0
- beim Linken wird der Speicherbereich verschoben
- Adressen in Befehlsfeldern (Sprungziele) müssen angepasst werden
- relocation information
  - welche Befehle enthalten Adressen, welches Segment?
  - Form: Bitmap oder Liste
- external entry point
  - Anfangsadresse einer Routine in einem anderen Objektprogramm
  - entry points gespeichert in einer Symboltabelle

# Bestandteile eines Objektprogramms

1. Codesegment
2. Datensegment
3. Relocation Bitmap
4. Externe Symboltabelle

# Linken, Beispiel

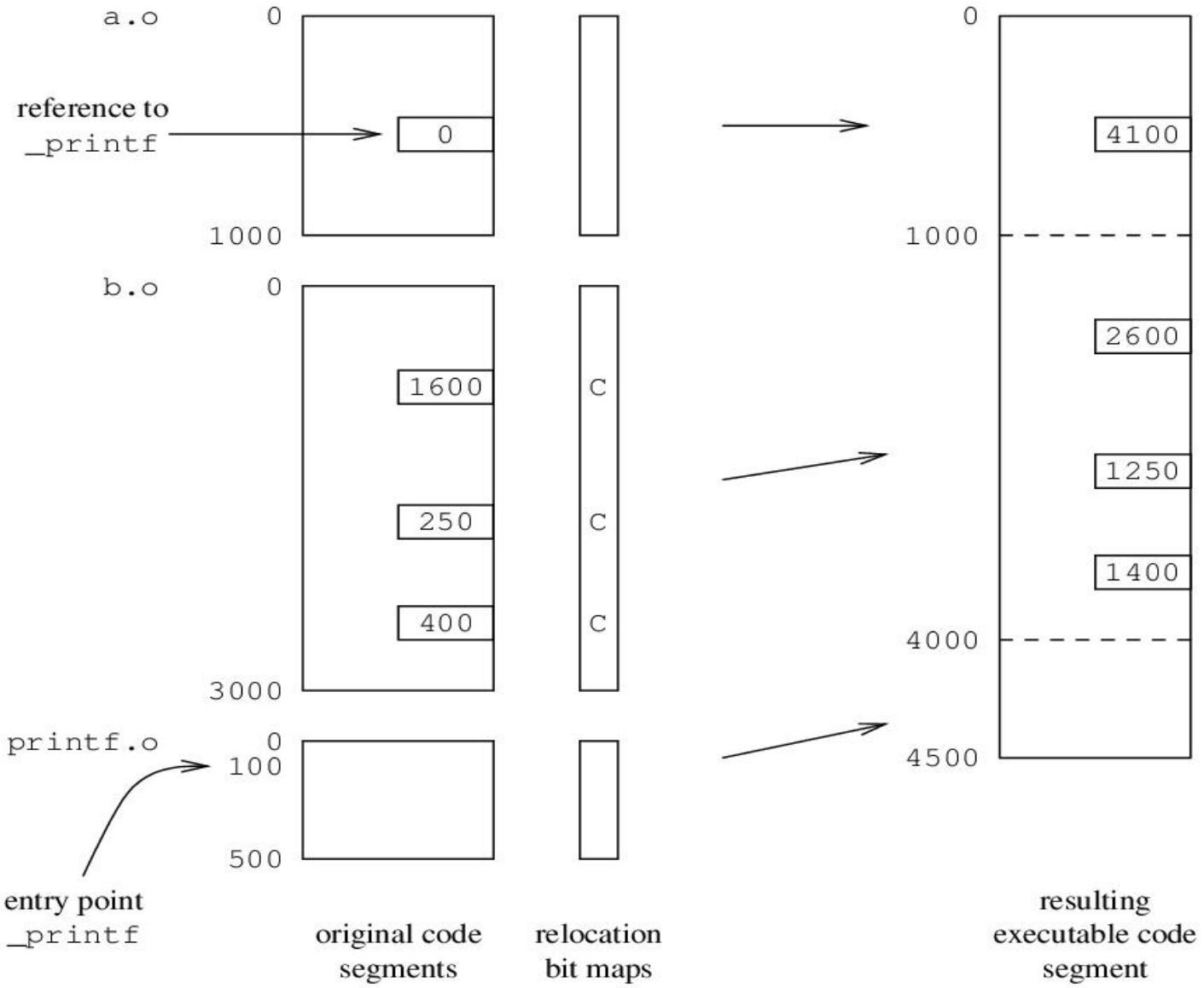


**Figure 4.91** Linking three code segments.

# Assembler (1)

- Umwandlung von Namen in Bitmuster
  - für Befehle
  - für Daten
- Einfügen von noop-Befehlen
  - zum Erreichen des korrekten Alignments
- Unterstützung für Sprungbefehle
  - Berechnung von Sprungdistanz oder Adresse
  - Entscheidung des Modus:

    Pentium-Prozessor: short / near / far
- Unterstützung von structs
- Makros
  - Parameter
  - Fallunterscheidungen
  - Wiederholungsanweisungen

# Assembler (2a)

Datensegment

Codesegment

```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

**Figure 4.92** Assembly code fragment with internal symbols.

# Assembler (2b)

```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

4-Bytes im Datensegment

Inhalt: hexadezimal 9A020000

Gewicht little endian: 0 256 ...

relocatable

**Figure 4.92** Assembly code fragment with internal symbols.

# Behandlung von Vorwärtsverweisen

- Möglichkeit 1: Two-Scans Assembly
  1. Durchlauf: simulierte Erzeugung von Maschinenbefehlen zum Feststellen der Lage aller Label
  2. Durchlauf: tatsächliche Erzeugung der Befehle mit Einsetzen der Werte für die Label
- Möglichkeit 2: Backpatching
  - Erzeugung der Maschinenbefehle bis auf Adressfelder
  - Einfügen von Befehlen mit Adressfeldern in Backpatchliste
  - Abarbeitung der Backpatchliste und Belegung der Adressfelder

# Backpatching

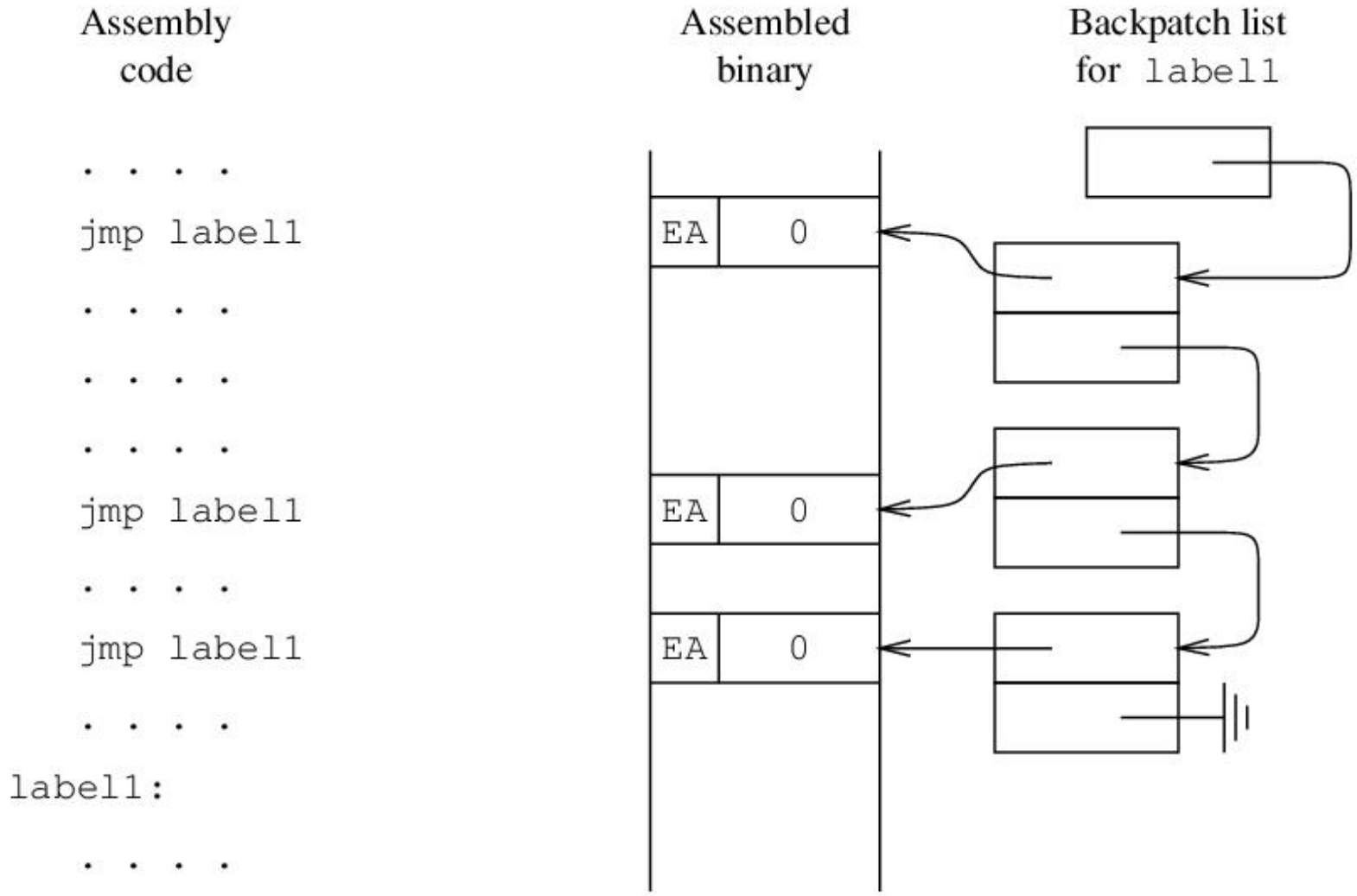


**Figure 4.93** A backpatch list for labels.

# Externe Symboltabelle

- Zwei Arten von Einträgen:
  - Entry Point: von aussen gebrauchte Stelle
    - Routine
    - Datenobjekt
  - Referenz: Stelle, die nach außen verweist
    - Routinenaufruf
    - Verwendung einer externen Variablen
- gespeicherte Information
  - Art des Segments: Code / Daten
  - Adresse