

**IA039**

**Překlad a překladače**

# Opakování – RICS procesory

- limitovaný počet instrukcí, jednotná délka
- jednoduché adresní módy, load/store, hodně registrů
- delay branches, brach prediction, out-of-order execution
- superskalární (MIPS – 2xFPU, 2xALU, adresace)
- superpipeline (ANDES)
- vyrovnávací paměti

# Optimalizující překladač

- překlad do mezijazyka
- optimalizace
  - meziprocedurální analýza
  - optimalizace cyklů
  - globální optimalizace
- generování kódu
  - využití všech jednotek

# Mezijazyk

- Čtveřice (obecně  $n$ -tice)
  - Instrukce: operátor, dva operandy, výsledek
  - Příklad
    - \* Přiřazení:  $X := Y \text{ op } Z$
  - Paměť: přes dočasné proměnné  $t_n$
  - Skoky: podmínka počítána samostatně
  - Skoky: na absolutní adresy

# Základní překlád

```
while ( j < n ) {  
    k = k + j*2  
    m = j*2  
    j++  
}
```

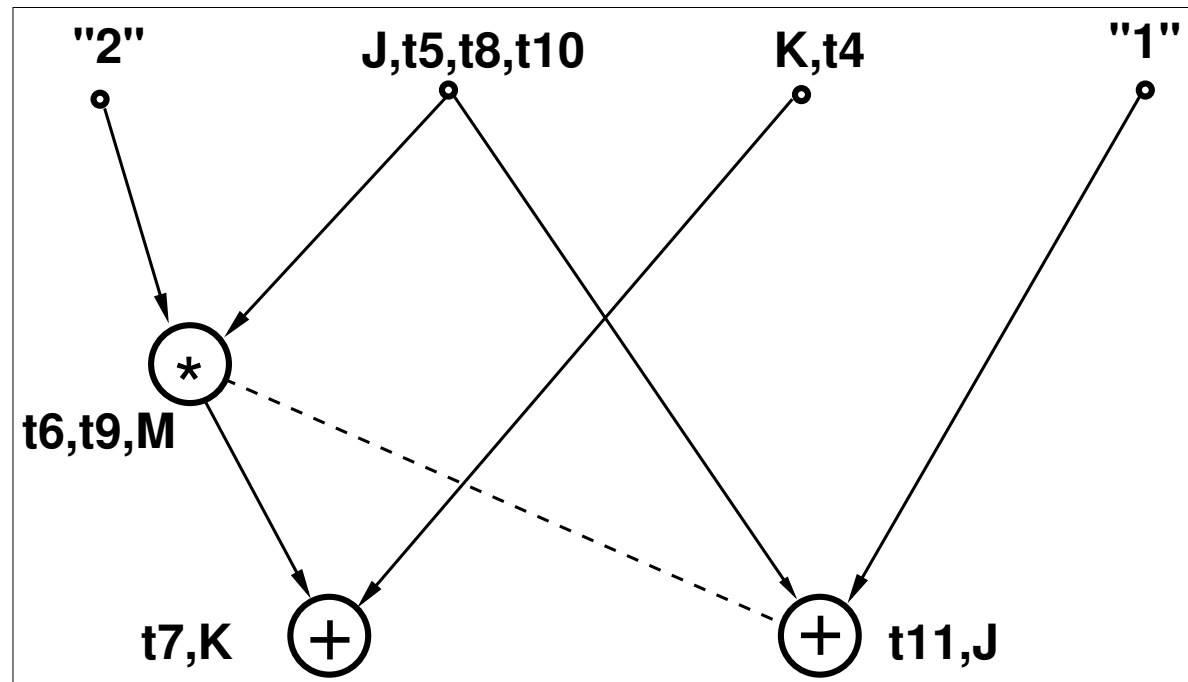
```
A:: t1 := j  
    t2 := n  
    t3 := t1 < t2  
    jmp (B) t3  
    jmp (C) TRUE  
B:: t4 := k  
    t5 := j  
    t6 := t5*2  
    t7 := t4+t6  
    k := t7  
    t8 := j  
    t9 := t8*2  
    m := t9  
    t10 := j  
    t11 := t10+1  
    j := t11  
    jmp (A) TRUE  
C::
```

# Základní bloky

- Program je pak reprezentován *grafem toku* (flow graph)
- Blok –část programu bez skoků
  - Jeden vstupní a jeden výstupní bod
  - Blok jako DAG (Directed Acyclic Graph)
- Optimalizace uvnitř bloků
  - Odstranění opakovaných (pod)výrazů
  - Odstranění přebytečných proměnných

# Directed Acyclic Graph

```
B:: t4 := k
    t5 := j
    t6 := t5*2
    t7 := t4+t6
    k := t7
    t8 := j
    t9 := t8*2
    m := t9
    t10 := j
    t11 := t10+1
    j := t11
    jmp (A) TRUE
```



# Modifikovaný překlad

```
B:: t4 := k
    t5 := j
    t6 := t5*2
    t7 := t4+t6
    k := t7
    t8 := j
    t9 := t8*2
    m := t9
    t10 := j
    t11 := t10+1
    j := t11
    jmp (A) TRUE
```

```
B:: t4 := k
    t5 := j
    t6 := t5*2
    m := t6
    t7 := t6+t4
    k := t7
    t11 := t5+1
    j := t11
    jmp (A) TRUE
```



# Další pojmy

- *Proměnné*
  - Definice a místa použití
- *Cykly*
- Proces *generování cílového kódu*
  - Součástí tzv. peephole optimalizace

# Optimalizovaný překlad

```
A:: t1 := j
    t2 := n
    t3 := t1 < t2
    jmp (B) t3
    jmp (C) TRUE
```

```
B:: t4 := k
    t5 := j
    t6 := t5*2
    t7 := t4+t6
    k := t7
    t8 := j
    t9 := t8*2
    m := t9
    t10 := j
    t11 := t10+1
    j := t11
    jmp (A) TRUE
```

```
C::
```

```
A:: t1 := j
    t2 := n
    t4 := k
    t9 := m
    t12 := t1+t1
    t3 := t1 >= t2
    jmp (B1) t3
```

```
B:: t4 := t4+t12
    t9 := t12
    t1 := t1+1
    t12 := t12+2
    t3 := t1 < t2
    jmp (B) t3
```

```
B1:: k := t4
    m := t9
```

```
C::
```

# Klasické optimalizace

- Propagace kopírováním

- Příklad:

$$\begin{array}{l} X = Y \\ Z = 1. + X \end{array} \implies \begin{array}{l} X = Y \\ Z = 1. + Y \end{array}$$

- Zpracování konstant

- propagace konstant

- Odstranění mrtvého kódu

- nedosažitelný kód

- šetření cache pro instrukce

# Klasické optimalizace II

- Strength reduction

- Příklad:  $K**2 \implies K*K$

- Přejmenování proměnných

- Příklad

$x = y*z;$

$x0 = y*z;$

$q = r+x+x; \implies$

$q = r+x0+x0;$

$x = a+b$

$x = a+b$

- Odstraňování společných podvýrazů (podstatné zejména pro výpočet adres prvků polí)

# Klasické optimalizace III

- Přesun invariantního kódu z cyklů
- Zjednodušení indukčních proměnných (výrazů  $s$ )
  - $A(I)$  je většinou počítáno jako:  
$$\text{address} = \text{base\_address}(A) + (I-1) * \text{sizeof\_datatype}(A)$$
což lze snadno v lineárním cyklu převést na  
*mimo cyklus:*  
$$\text{address} = \text{base\_address}(A) - \text{sizeof\_datatype}(A)$$
*v cyklu:*  
$$\text{address} = \text{address} + \text{sizeof\_datatype}(A)$$
- Přiřazení registrů proměnným

# Odstraňování smetí

- Podprogramy, makra
  - Inlining
- Podmíněné výrazy
  - Reorganizace složitých výrazů
  - Nadbytečné testy (`if` versus `case`)
- Podmíněné výrazy uprostřed cyklů
  - Nezávislé na cyklu
  - Závislé na cyklu
    - \* Nezávislé na iteraci
    - \* Závislé mezi iteracemi

# Podmíněné výrazy – příklad

```
DO I=1,K
  IF (N .EQ 0) THEN
    A(I)=A(I)+B(I)*C
  ELSE
    A(I)=0
  ENDIF
```



```
IF (N .EQ 0) THEN
  DO I=1,K
    A(I)=A(I)+B(I)*C
  CONTINUE
ELSE
  DO I=1,K
    A(I)=0
  CONTINUE
ENDIF
```

# Odstraňování smetí II

- Redukce

- min (nebo max):

```
for(i=0;i<n;i++)
```

```
    z=(a[i] > z) ? a[i] : z;
```

- jak obejít rekurzivní závislost:

```
for(i=0;i<n-1;i+=2) {
```

```
    z0=(a[i] > z0) ? a[i] : z0;
```

```
    z1=(a[i+1] > z1) ? a[i+1] : z1;
```

```
}
```

```
z=(z0 < z1) ? z1 : z0;
```



# Redukce – Asociativní transformace

- Numerická nepřesnost:

$$\begin{aligned}(X + Y) + Z &= (.00005 + .00005) + 1.0000 \\ & .00010 + \quad \quad \quad + 1.0000 = 1.0001\end{aligned}$$

ale

$$\begin{aligned}X + (Y + Z) &= .00005 + (.00005 + 1.0000) = \\ & .00005 \quad \quad \quad + 1.0000 = 1.0000\end{aligned}$$

- Redukce

DO I=1,N

SUM=SUM+A(I)\*B(I)

redukce s rekurzivní závislostí - můžeme použít stejný trik jako u min  
redukce?

# Odstraňování smetí III

- Skoky

- Konverze typů

```
REAL*8 A(1000)
```

```
REAL*4 B(1000)
```

```
DO I 1=1,1000
```

```
    A(I)=A(I)*B(I)
```

- Ruční optimalizace

- Společné podvýrazy

- Přesun kódu

- Zpracování polí (inteligentní překladač, C a ukazatele)

# Optimalizace cyklů

- Cíle:
  - Redukce režie
  - Zlepšení přístupu k paměti (cache)
  - Zvýšení paralelismu

# Datové závislosti I

- **Flow Dependencies** (backward dependencies)

- Příklad:

DO I=2,N A(I)=A(I-1)+B(I)	⇒	DO I=2,N,2 A(I)=A(I-1)+B(I) A(I+1)=A(I-1)+B(I)+B(I+1)
------------------------------	---	---

- **Anti-Dependencies**

- Standardně přejmenování proměnných

- Příklad

DO I=1:N A(I) = B(I) * E B(I) = A(I+2) * C	⇒	DO I=1:N A'(I) = B(I) * E B(I) = A(I+2) * C A(I) = A'(I)
--	---	---

# Datové závislosti II

## ■ Output Dependencies

### ■ Příklad:

$$A(I) = C(I) * 2$$

$$A(I+2) = D(I) + E$$

- V cyklu je vypočteno několik hodnot konkrétní proměnné, zapsat však lze pouze tu „poslední“
- Může být problém zjistit, která je ta „poslední“

# Rozvoj cyklů (loop unrolling) I

- Tělo cyklu se několikrát zkopíruje:

```
DO I=1,N
```

```
  A(I)=A(I)+B(I)*C
```

⇓

```
DO I=1,N,4
```

```
  A(I)=A(I)+B(I)*C
```

```
  A(I+1)=A(I+1)+B(I+1)*C
```

```
  A(I+2)=A(I+2)+B(I+2)*C
```

```
  A(I+3)=A(I+3)+B(I+3)*C
```

# Rozvoj cyklů (loop unrolling) II

- Hlavní smysl
  - Snížení režie
    - \* Snížení počtu průchodů cyklem
  - Zvýšení paralelizace (i v rámci jednoho procesoru)
    - \* Software pipelining
- Pre- a postconditioning loops
  - Adaptace na skutečný počet průchodů

# Rozvoj cyklů (loop unrolling) III

- Nevhodné cykly
  - Malý počet iterací → úplný rozvoj cyklů
  - „Tlusté“ (=velké) cykly: samy obsahují dostatek příležitostí k paralelizaci
  - Cykly s voláním procedur: souvislost s rozvojem procedur (inlining)
  - Cykly s podmíněnými výrazy: spíše starší typy procesorů
  - „Rekurzivní“ cykly: cykly s vnitřními závislostmi ( $a[i]=a[i]+a[i-1]*b$ )



# Problémy s rozvojem cyklů

- Rozvoj špatným počtem iterací
- Zahlcení registrů
- Výpadky vyrovnávací paměti instrukcí (příliš velký cyklus)
- Hardwarové problémy
  - Především na multiprocесorech se sdílenou pamětí (cache koherence, přetížení sběrnice)
- Speciální případy: rozvoj vnějších cyklů, spojování cyklů

# Spojování cyklů

- opakované použití dat
- větší tělo cyklu
- kompilátor zvládne sám, pokud mezi cykly není jiný kód

```
for(i=0;i<n;i++)
```

```
    a[i]=b[i]+1
```

```
for(i=0;i<n;i++)
```

```
    c[i]=a[i]/2
```

```
for(i=0;i<n;i++)
```

```
    d[i]=1/c[i+1]
```

⇒

```
a[0]=b[0]+1
```

```
c[0]=a[0]/2
```

```
for(i=1;i<n;i++) {
```

```
    a[i]=b[i]+1
```

```
    c[i]=a[i]/2
```

```
    d[i-1]=1/c[i]
```

```
}
```

```
d[n-1]=1/c[n+1]
```

# Rozvoj vnějších cyklů

- Příklad

```
DO I=1,N
  DO J=1,N
    A(I)=A(I)+B(I,J)*C(J)
```

- $A(I)$  je v vnitřním cyklu konstanta,  $C(J)$  se prochází správně
- $B(I,J)$  se prochází opačně!

```
DO I=1,N,4
  DO J=1,N
    A(I+0)=A(I+0)+B(I+0,J)*C(J)
    A(I+1)=A(I+1)+B(I+1,J)*C(J)
    A(I+2)=A(I+2)+B(I+2,J)*C(J)
    A(I+3)=A(I+3)+B(I+3,J)*C(J)
```

# Optimalizace přístupů k paměti

- Optimální: nejmenší krok (práce s cache)
- Práce s maticemi – C vs. Fortran
  - C: ukládá po řádcích, nejrychleji se mění pravý index
  - Fortran: ukládá pa sloupcích, nejrychleji se mění levý index
- Obrácení indexu

- Příklad:

```
DO I=1,N
```

```
  DO 10 J=1,N
```

```
    A(I,J)=B(I,J)+C(I)*D
```

⇒

```
DO J=1,N
```

```
  DO 10 I=1,N
```

```
    A(I,J)=B(I,J)+C(J)*D
```

# Optimalizace přístupů k paměti II

- Blokování

- Příklad:

```
DO I=1,N
```

```
  DO 10 J=1,N
```

```
    A(J,I)=A(J,I)+B(I,J)
```

⇓

```
DO I=1,N,2
```

```
  DO 10 J=1,N,2
```

```
    A(J,I)=A(J,I)+B(I,J)
```

```
    A(J+1,I)=A(J+1,I)+B(I,J+1)
```

```
    A(J,I+1)=A(J,I+1)+B(I+1,J)
```

```
    A(J+1,I+1)=A(J+1,I+1)+B(I+1,J+1)
```

# Optimalizace přístupů k paměti III

- Nepřímá adresace

- Příklad:

$b[i] = a[i+k] * c$ , hodnota  $k$  neznáma při překladu

$a[k[i]] += b[i] * c$

- Použití ukazatelů

- Nedostatečná paměť

- „Ruční“ zpracování

- Virtuální paměť