

Limbaje formale și tehnici de compilare

Curs 13: optimizare

Obiectivele optimizării

- ▶ Optimizarea urmărește transformarea unui program, astfel încât programul rezultat:
 - Să performeze mai bine conform anumitor criterii
 - Să fie semantic echivalent cu cel original
- ▶ Criterii de optimizare:
 - Viteză mai mare de execuție
 - Consum mai mic de memorie
 - Comportament mai bun în anumite configurații sau arhitecturi (ex: un program poate fi optimizat pentru a folosi GPU)

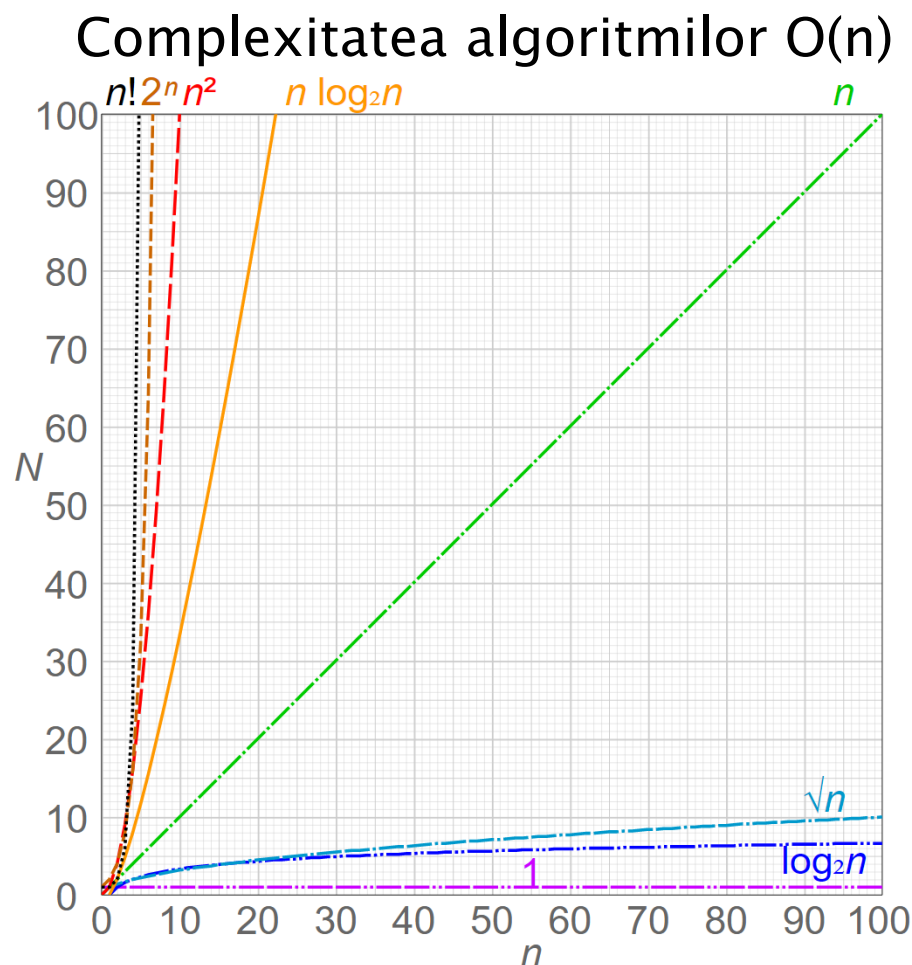
Echilibrarea diverselor optimizări

- ▶ Anumite optimizări ale unui criteriu pot duce la degradarea performanțelor conform altui criteriu
- ▶ Din acest motiv, compilatoarele au mai multe opțiuni de optimizare, astfel încât programatorul să le aleagă pe cele mai potrivite
- ▶ În general optimizările pentru spațiu se aplică în situații care memoria este limitată și codul nu ar încăpea: microcontrollere (ex: Arduino) sau programe stocate în memoria ROM (ex: BIOS-ul)
- ▶ În celelalte cazuri se preferă optimizările pentru viteză
- ▶ Se poate testa un program în ambele situații, pentru a se decide cea mai potrivită

Ce nu este optimizarea?

- ▶ Optimizarea nu duce la îmbunătățirea unui algoritm, ci doar la eficientizarea execuției sale
- ▶ Exemplu: prin optimizare viteza crește de 5 ori

Tip algoritm	n=10	n=100
Constant $O(1)$	1	1
Logaritmic $O(\log_2(n))$	3.3	6.6
Linear $O(n)$	10	100
Linearitmic $O(n \cdot \log_2(n))$	33.2	664
Pătratic $O(n^2)$	100	10000
Exponențial $O(2^n)$	1024	10^{30}



Includerea optimizărilor în compilator

- ▶ Optimizările pot fi relativ ușor de implementat în anumite faze ale unui compilator, dar foarte greu de implementat în altele
- ▶ Din acest motiv, fiecărei faze de compilare i se atașează optimizări specifice. Pot fi optimizări pentru:
 - Codul sursă (transformări în ASA)
 - Generarea de cod intermediar
 - Codul intermediar (CI)
 - Generarea de cod mașină
 - Codul mașină
- ▶ Unele optimizări se aplică după ce programul a fost executat de mai multe ori, pentru a se colecta date statistice despre cazurile cele mai des întâlnite

Optimizări la nivel de cod sursă

- ▶ Sunt optimizările care țin cont de concepte de nivel înalt ale LP, cunoașterea bibliotecii standard sau operează pe structuri de date mai complexe
- ▶ Din acest motiv, este greu să fie aplicate în fazele următoare, deoarece o fază succesoare renunță la informațiile nerelevante din fazele precedente
- ▶ De obicei se aplică în AST
- ▶ Exemple:
 - Transformarea unor construcții de limbaj sau expresii
 - Optimizări rezultate din inferență de tipuri
 - Concretizarea apelurilor polimorfice

Transformarea unor construcții

- ▶ Dacă se cunosc funcționalitățile pe care LP și biblioteca sa standard le oferă, devine posibil să se înlocuiască unele construcții prin altele, mai eficiente

```
Angajat a=new Angajat("Ion", 3200);           // Java
String text="nume: "+a.nume+", salariu: "+a.salariu;
// poate deveni
String s1=a.nume;
int len1=s1.length();
String s2=a.salariu.toString();
int len2=s2.length();
String text=new StringBuilder(17+len1+len2)
    .append("nume: ").append(s1)
    .append(", salariu: ").append(s2)
    .toString();
```


Optimizări rezultate din inferență de tipuri

- ▶ În special pentru limbajele dinamice, dacă se poate infera tipul unei variabile, atunci se pot folosi în mod direct operațiile aceluși tip, în loc să se testeze de fiecare dată ce tip are valoarea ei

```
require('console') //JavaScript, Node.js
var i,n=7
for(i=0;i<n;i++){
    console.log(i)
}
```

- ▶ Se inferă că *i* și *n* sunt de tip numeric și atunci se cunoaște exact cum se execută operațiile cu ele (ex: <, ++)
- ▶ Altfel, ar fi trebuit înainte de fiecare operație să se testeze tipurile operanzilor, iar apoi să se execute operația corespunzătoare tipului rezultat din testare

Concretizarea apelurilor polimorifice

- ▶ Metodele polimorifice au implementări diferite în clase diferite. Trebuie să existe un mecanism de selecție a metodei corespunzătoare obiectului curent

```
abstract class Figura{    //Cerc, Triunghi, Dreptunghi, ...
    public abstract double arie();
}                                                                    //Java
class Cerc extends Figura{    //implementare concretă
    double r;
    @Override public double arie() {
        return Math.PI*r*r;
    }
}
Figura fig=new Cerc(1);
double a=fig.arie(); //ce implementare trebuie apelată?
```

- ▶ Din ultima expresie atribuită lui **fig**, se inferă faptul că figura este de fapt un **Cerc**, astfel încât se apelează direct **Cerc.arie**, fără să mai fie necesar mecanismul de selecție

Optimizări la generarea CI

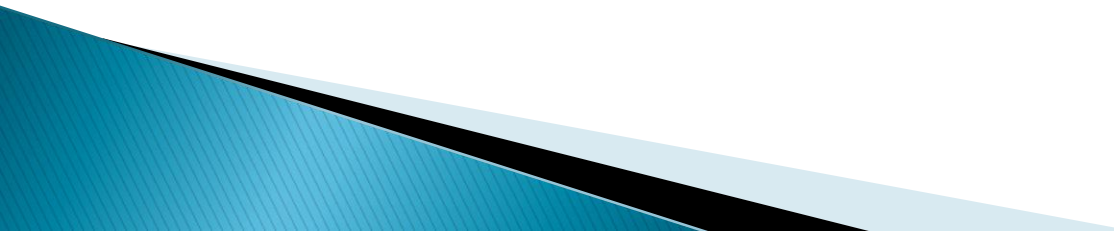
- ▶ Aceeași instrucțiune sursă poate genera structuri diferite, în funcție de felul specific în care e folosită
- ▶ De exemplu, la **switch** se poate ține cont de numărul de **case**-uri, dacă valorile sunt consecutive, etc

```
int x;  
switch(x){  
    case 1:...  
    case 2:...  
    case 3:...  
    case 4:...  
    case 5:...  
    default:  
}
```



```
labels etichete={L1, L2, L3, L4, L5}  
    LESS.i    #1,x,1  
    JT        #1,L_default  
    GRT.i     #1,x,5  
    JT        #1,L_default  
    SUB.i     #1,x,1  
    JMPIDX    etichete,#1  
  
L1:    ...  
L2:    ...  
L3:    ...  
L4:    ...  
L5:    ...  
L_default:...
```

Optimizări la nivel de CI

- ▶ La acest nivel se pot aplica toate optimizările care sunt independente atât de LP cât și de arhitectura destinație:
 - ▶ Eliminarea operațiilor fără efect
 - ▶ Propagarea valorilor constante
 - ▶ Folosirea unor operații mai simple
 - ▶ Eliminarea subexpresiilor comune
 - ▶ Optimizarea salturilor redundante
 - ▶ Înlocuirea apelurilor de funcții cu codul funcțiilor
- 

Eliminarea operațiilor fără efect

- ▶ Se elimină: $x+0$, $x-0$, $x*1$, $x/1$, $x=x$
- ▶ $x==x \rightarrow 1$, $x!=x \rightarrow 0$
- ▶ $\&*p$ (adresa unde se află valoarea pointată)
 $\rightarrow p$
- ▶ $*\&x$ (valoarea de la adresa variabilei) $\rightarrow x$
- ▶ Eliminarea atribuirilor fără efecte colaterale,
a căror destinație nu mai este folosită
ulterior: $x=y/2; x=1; \rightarrow x=1$

Propagarea valorilor constante

```
int DEBUG=0;  
float PI=3.14159;  
float a=2*PI*r;  
if(DEBUG){  
    printf("%g\n",a);  
}
```

```
SET.i    DEBUG,0  
SET.f    PI,3.14159  
MUL.f    #1,2,PI  
MUL.f    a,#1,r  
JF        DEBUG,L1  
CALL     printf "%g\n",a
```

L1:

```
MUL.f    a,6.28318,r  
JF        0,L1  
CALL     printf,"%g\n",a
```

L1:

```
MUL.f    a,6.28318*r
```

Folosirea unor operații mai simple

- ▶ Pe arhitecturi mai simple (ex: microcontrolere), operații precum înmulțire sau împărțire pot să ia sute de tacturi pentru execuție sau pot să nu fie implementate deloc, caz în care iau chiar mai mult timp
- ▶ Din acest motiv, când se poate, se preferă implementarea lor prin operații mai simple

```
int b=a*2;  
int c=a/512;  // 512=29
```



```
int b=a<<1;  // sau a+a  
int c=a>>9;
```

Eliminarea subexpresiilor comune

```
float len(float x1, float y1, float x2, float y2){  
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));  
}
```

```
ENTER 8  
SUB.f  #1,x2,x1  
SUB.f  #2,x2,x1  
MUL.f  #3,#1,#2  
SUB.f  #4,y2,y1  
SUB.f  #5,y2,y1  
MUL.f  #6,#4,#5  
ADD.f  #7,#3,#6  
CALL.f #8,sqrt,#7  
RET    4,#8
```



```
ENTER 8  
SUB.f  #1,x2,x1  
MUL.f  #3,#1,#1  
SUB.f  #4,y2,y1  
MUL.f  #6,#4,#4  
ADD.f  #7,#3,#6  
CALL.f #8,sqrt,#7  
RET    4,#8
```


Optimizarea salturilor redundante

```
while(a>b){  
    if(a<10)a=a-1;  
    else a=a/2;  
}
```

L1: GRT.i #1,a,b
 JF #1,L4
 LESS.i #2,a,10
 JF #2,L2
 SUB.i a,a,1
 JMP L3
L2: DIV.i a,a,2
L3: JMP L1
L4:



L1: GRT.i #1,a,b
 JF #1,L4
 LESS.i #2,a,10
 JF #2,L2
 SUB.i a,a,1
 JMP L1
L2: DIV.i a,a,2
L3: JMP L1
L4:

Înlocuirea apelurilor de funcții cu codul lor

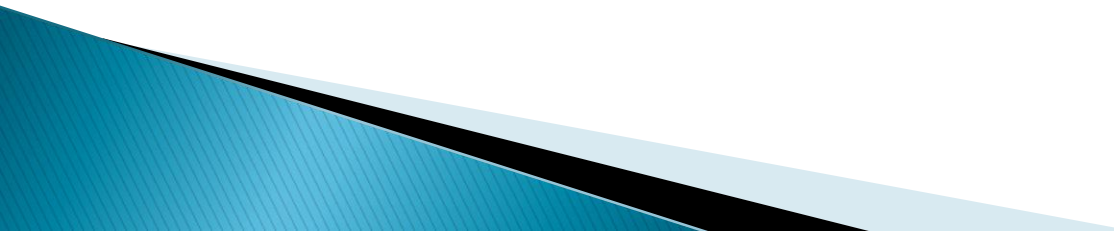
- ▶ Pentru funcțiile care constă doar în câteva instrucțiuni sau cele care se apelează dintr-un singur loc din program, se poate înlocui apelul lor cu codul funcției
- ▶ Nu mai sunt necesare crearea/ștergerea cadrului funcției și apelul, inclusiv transmiterea argumentelor
- ▶ Apar oportunități și pentru alte optimizări

```
int min(int x, int y){  
    if(x<y)return x;  
    else return y;  
}  
printf("%d\n",min(1,3));
```

```
int r;  
if(1<3)r=1;  
else r=3;  
printf("%d\n",r);
```

```
printf("%d\n",1);
```

Optimizări la generarea codului mașină

- ▶ Poate cel mai important aspect la generarea codului mașină este optimizarea folosirii regiștrilor CPU
 - ▶ Folosirea memoriei poate fi de zeci de ori mai lentă decât cea a regiștrilor
 - ▶ Unele instrucțiuni CPU necesită unul sau mai mulți operanzi în regiștri, deci valorile oricum trebuie aduse din memorie în regiștri
- 

Generare de cod mașină

// fără optimizare regiștri

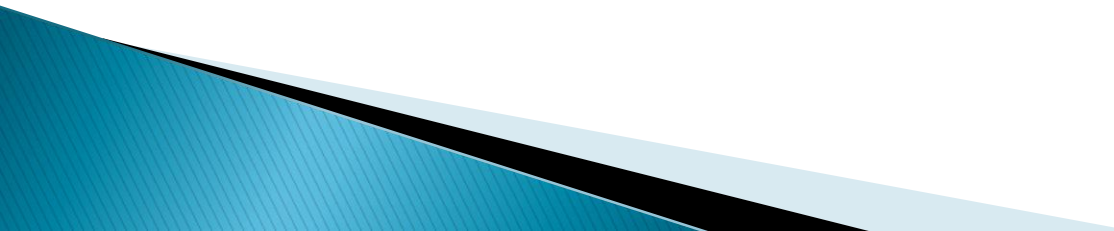
```
L1:  MOV EAX,[ESP-4] // x
      CMP EAX,[ESP-8] // y
      JE L4
      MOV EAX,[ESP-4]
      CMP EAX,[ESP-8]
      JLE L2
      MOV EAX,[ESP-4]
      SUB EAX,[ESP-8]
      MOV [ESP-4],EAX
      JMP L3
L2:  MOV EAX,[ESP-8]
      SUB EAX,[ESP-4]
      MOV [ESP-8],EAX
L3:  JMP L1
L4:  MOV EAX,[ESP-4]
      RET 8
```

```
int cmmdc(int x,int y){
    while(x!=y){
        if(x>y)x=x-y;
        else y=y-x;
    }
    return x;
}
```

// cu optimizare regiștri

```
MOV EAX,[ESP-4] // x
MOV ECX,[ESP-8] // y
L1:  CMP EAX,ECX
      JE L4
      CMP EAX,ECX
      JLE L2
      SUB EAX,ECX
      JMP L3
L2:  SUB ECX,EAX
L3:  JMP L1
L4:  RET 8
```

Optimizarea codului mașină

- ▶ Se optimizează codul mașină generat din CI
 - ▶ Exemple de optimizări:
 - Înlocuirea unei secvențe de instrucțiuni mai simple cu o instrucțiune mai complexă
 - Refolosirea valorilor care rămân constante după o operație sau care apar ca rezultate auxiliare
- 

Înlocuirea secvențelor de instrucțiuni simple cu o instrucțiune mai complexă

```
// int i,v[100];  
// v[i]=-1;
```

```
MOV EAX,[i]           // EAX=i  
SHL EAX,2             // EAX=i*4  
ADD EAX,v             // EAX=v+i*4  
MOV DWORD[EAX],-1     // *(v+i*4)=-1
```



```
MOV EAX,[i]           // EAX=i  
MOV DWORD[v+EAX*4],-1 // *(v+i*4)=-1
```

Refolosirea valorilor constante după operații

```
L1:  MOV EAX,[ESP-4] // x
      MOV ECX,[ESP-8] // y
      CMP EAX,ECX
      JE L4
      CMP EAX,ECX
      JLE L2
      SUB EAX,ECX
      JMP L3
L2:  SUB ECX,EAX
L3:  JMP L1
L4:  RET 8
```

```
      MOV EAX,[ESP-4] // x
      MOV ECX,[ESP-8] // y
      CMP EAX,ECX
L1:  JE L4
      JLE L2
      SUB EAX,ECX
      JMP L3
L2:  SUB ECX,EAX
L3:  JMP L1
L4:  RET 8
```

- ▶ Registrul EFLAGS conține indicatori pentru: egalitate, mai mic, mai mare, negativ, etc.
- ▶ Instrucțiunile aritmetice și de comparație (SUB, CMP) setează registrul EFLAGS conform rezultatului lor
- ▶ EFLAGS rămâne neschimbat până la următoarea instrucțiune care-l modifică
- ▶ Instrucțiunile de salt condiționat (JE, JLE) testează indicatorii din EFLAGS

Optimizări care folosesc date de la execuția programului

- ▶ Aceste optimizări folosesc date statistice colectate de la execuția codului, pentru a favoriza situațiile cele mai des întâlnite

```
switch(x){           // probabilități pentru x, determinate prin execuții multiple
  case 7:...break;    // 40%
  case 100:...break;  // 3%
  case 5000:...break; // 50%
  default:...
```



```
switch(x){
  case 5000:...break;
  case 7:...break;
  case 100:...break;
  default:...
```