



Анализ и оптимизация на софтуерни приложения

Александър Пенев

Васил Василев

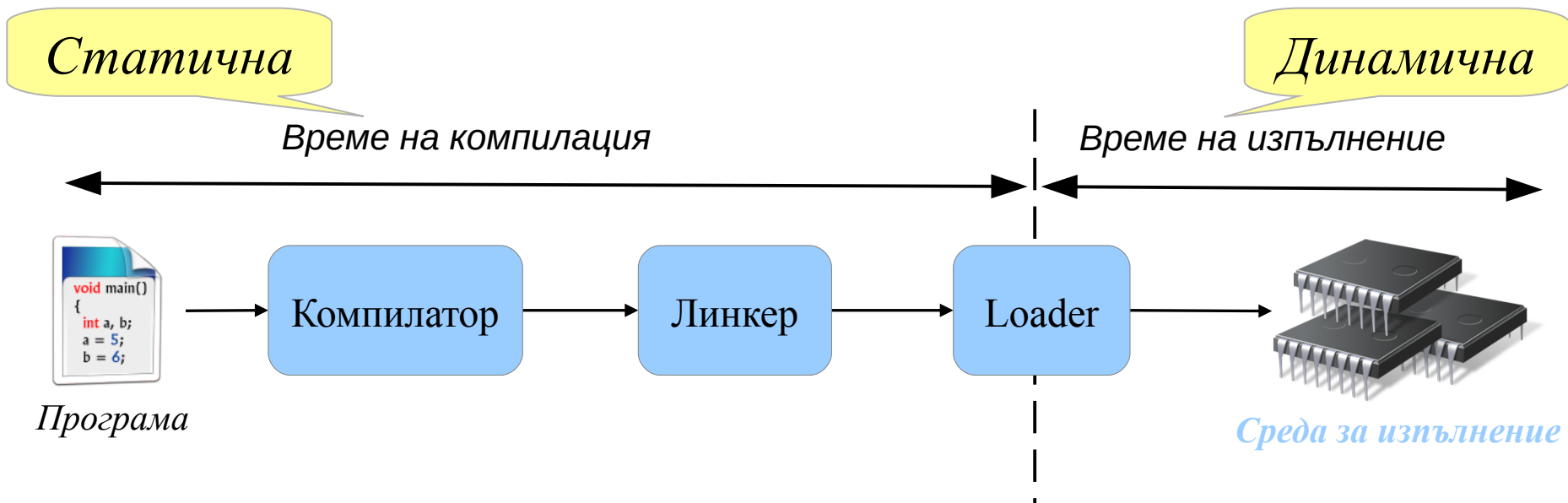
Какво могат и какво не могат
компилаторите

Съдържание

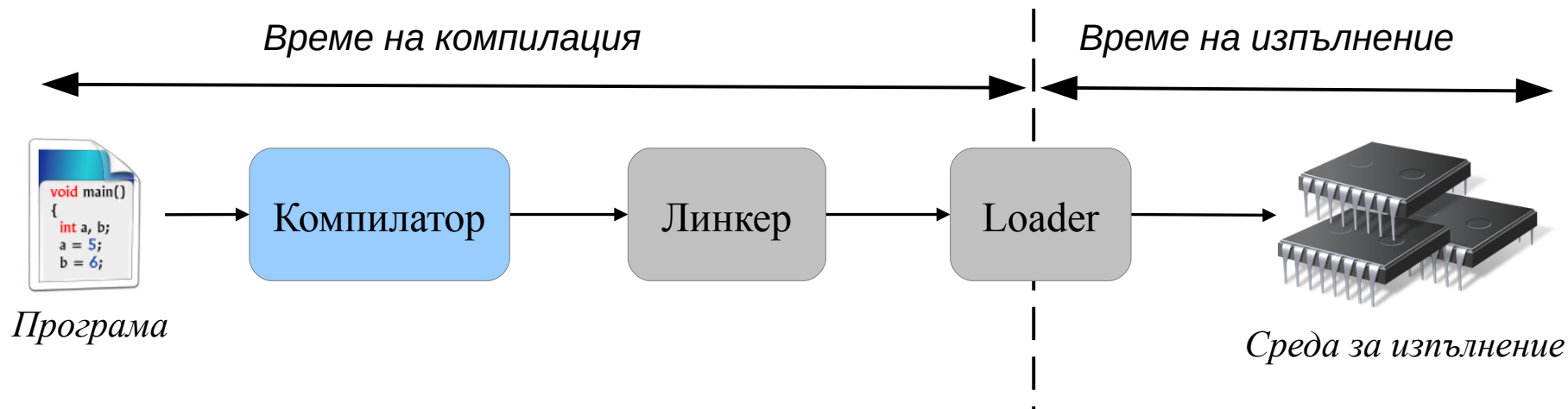
1. Оптимизационен континуум
2. Известни видове оптимизации



Оптимизационен континуум

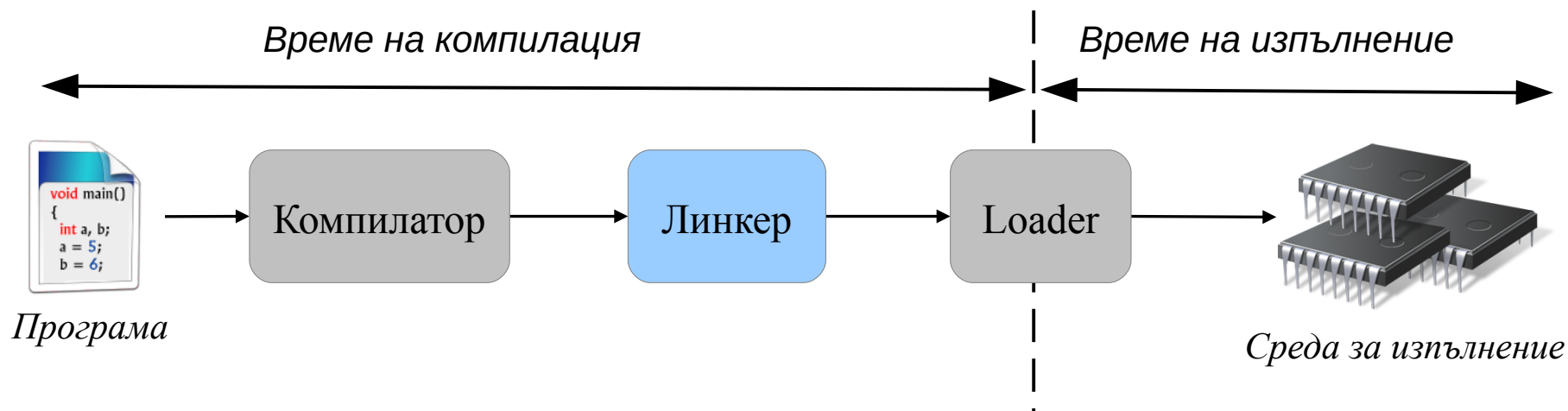


Оптимизационен континуум



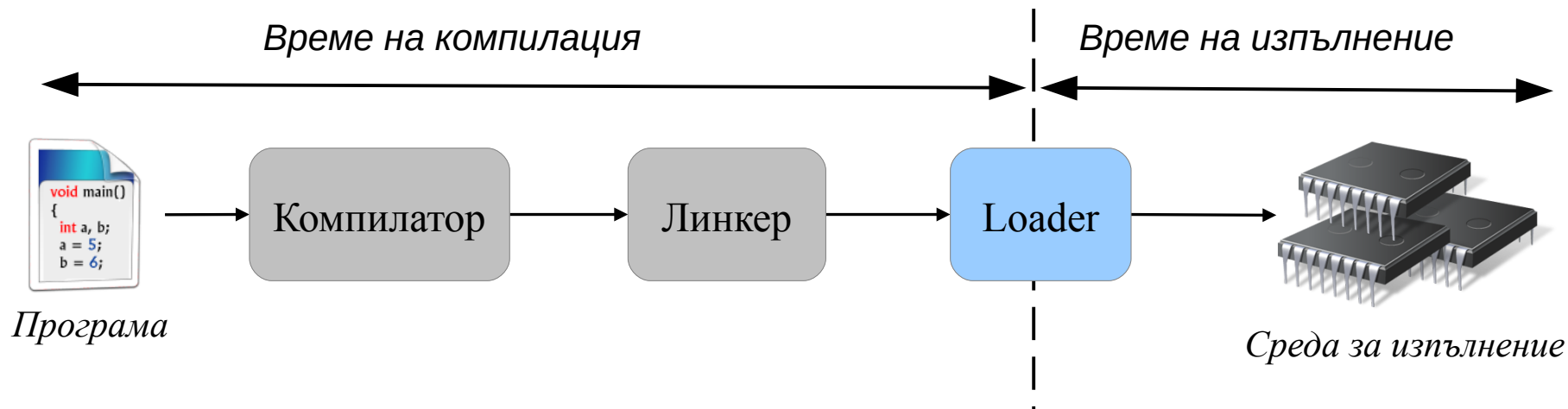
- ❖ Знае цялата входна програма
- ❖ Лесно управляват трансформациите от високо ниво към ниско ниво
- ❖ Времето на компилация не е проблем
- ❖ Не вижда цялата програма
- ❖ Не знае (много) за средата на изпълнение
- ❖ Не знае (много) за архитектурата на изпълнение

Оптимизационен континуум



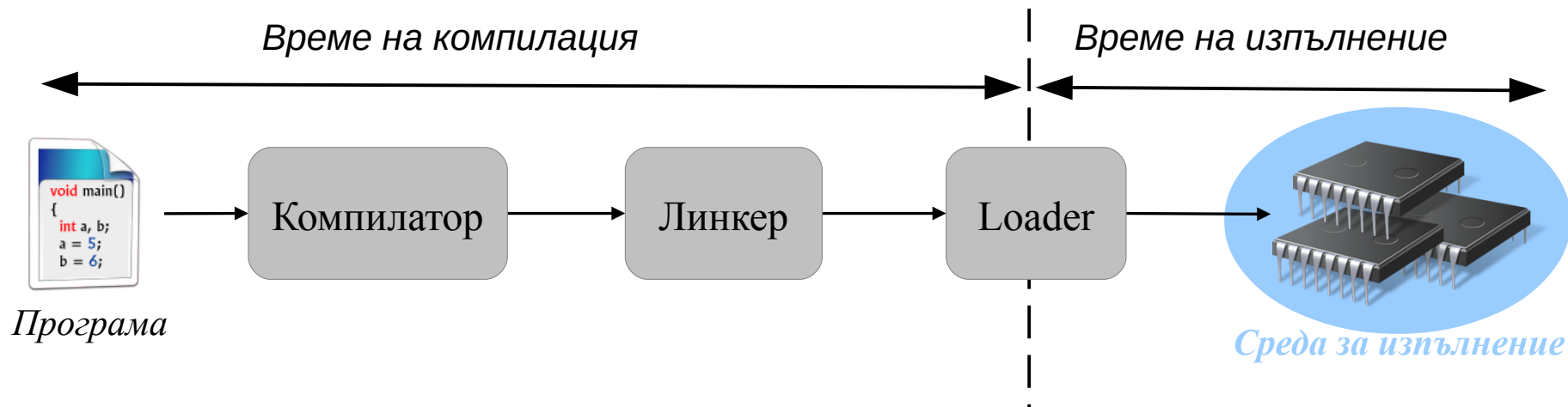
- ❖ Знае цялата входна програма
- ❖ Може да не вижда цялата програма
- ❖ Няма достъп до сорс кода
- ❖ Не знае (много) за средата на изпълнение
- ❖ Не знае (много) за архитектурата на изпълнение

Оптимизационен континуум



- ❖ Знае цялата входна програма
- ❖ Може да не вижда цялата програма
- ❖ Няма достъп до сорс кода
- ❖ Не знае (много) за средата на изпълнение
- ❖ Не знае (много) за архитектурата на изпълнение

Оптимизационен континуум



- ❖ Знае цялата входна програма
- ❖ Знае за средата на изпълнение
- ❖ Няма достъп до сорс кода
- ❖ Времето за оптимизации отнема от времето за изпълнение

Анализ на потока данни

Data Flow Analysis

Техниката се използва за събиране на информация за възможното множество от стойности, изчислени в различни точки на програмата

По време на компилация получаваме информация за стойностите по време на изпълнение за променливи или изрази. Намира:

- ❖ Кои оператори за присвояване са били използвани за получаване на текущата стойност на променливата в дадената точка
- ❖ Кои променливи имат стойности, които не се използват от дадена точка нататък в програмата
- ❖ Какъв е множеството възможни стойности на променливата в дадената точка на програмата



Пример

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```



Пример

```
**** int calc(int N, int t1, int t2){
9     .loc 1 2 0
10    .cfi_startproc
11    pushq %rbp
12    .LCFI0:
13    .cfi_def_cfa_offset 16
14    .cfi_offset 6, -16
15    movq %rsp, %rbp
16    .LCFI1:
17    .cfi_def_cfa_register 6
18    movl %edi, -20(%rbp)
19    movl %esi, -24(%rbp)
20    movl %edx, -28(%rbp)
****   x = 0;
21    .loc 1 6 0
22    movl $0, -8(%rbp)
****   y = 0;
23    .loc 1 7 0
24    movl $0, -4(%rbp)
****   for (i = 0; i <= N; i++) {
25    .loc 1 9 0
26    movl $0, -12(%rbp)
27    jmp  .L2
28    .L3:
29    .loc 1 10 0 discriminator 2
****   x = x + (8*t2/t1)*i + (i+N)*(i+N);
30    movl -28(%rbp), %eax
31    sall $3, %eax
32    movl %eax, %edx
33    sarl $31, %edx
```

```
34    idivl -24(%rbp)
35    movl %eax, %edx
36    imull -12(%rbp), %edx
37    movl -20(%rbp), %eax
38    movl -12(%rbp), %ecx
39    leal (%rcx,%rax), %esi
40    movl -20(%rbp), %eax
41    movl -12(%rbp), %ecx
42    addl %ecx, %eax
43    imull %esi, %eax
44    addl %edx, %eax
45    addl %eax, -8(%rbp)
46    .loc 1 11 0 discriminator 2
****   x = x + y*(t1+t2);
47    movl -28(%rbp), %eax
48    movl -24(%rbp), %edx
49    addl %edx, %eax
50    imull -4(%rbp), %eax
51    addl %eax, -8(%rbp)
****   for (i = 0; i <= N; i++) {
52    .loc 1 9 0 discriminator 2
53    addl $1, -12(%rbp)
54    .L2:
****   for (i = 0; i <= N; i++) {
55    .loc 1 9 0 is_stmt 0 discriminator 1
56    movl -12(%rbp), %eax
57    cmpl -20(%rbp), %eax
58    jle  .L3 ****   }
****   return x;
59    .loc 1 14 0 is_stmt 1
60    movl -8(%rbp), %eax ****   }
```



Разпространение на константи

Constant Propagation

Анализ

- ❖ Ако при всички възможни пътища на изпълнение, стойността на променливата е константна за дадена точка на използване

Трансформация

- ❖ Замества се променливата с тази константа

Последствия:

- ❖ Няма нужда да се съхранява тази стойност в променлива

Освобождава се памет или регистър

- ❖ Може да доведе до нужда от допълнителна оптимизация



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```

*Не е константа при всички
пътища на изпълнение*



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```

*Не е константа при всички
пътища на изпълнение*



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + y*(t1+t2);
    }

    return x;
    x++;
}
```



Разпространение на константи

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + 0*(t1+t2);
    }

    return x;
    x++;
}
```



Опростяване на изрази

Algebraic Simplification

Анализ

- ❖ Ако даден израз е изчислим по време на компилация
- ❖ Ако могат да се използват по-бързи инструкции

Трансформация

- ❖ $X * 0 = 0$; $X * 1 = X$; $X * 32 = X \ll 4$
- ❖ $X + 0 = X$; $X + X = X \ll 2$

Последствия:

- ❖ По-малко работа по време на изпълнение
- ❖ Може да доведе до допълнителна оптимизация
- ❖ Машината, която изпълнява кода може да се държи различно от тази, на която е компилиран
Например: underflow, overflow
- ❖ Проблеми с комутативността и транзитивността



Опростиаване на изрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + 0*(t1+t2);
    }

    return x;
    x++;
}
```



Опростиаване на изрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + 0*(t1+t2);
    }

    return x;
    x++;
}
```



Опростиране на изрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + 0;
    }

    return x;
    x++;
}
```



Опростиаване на изрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x + 0;
    }

    return x;
    x++;
}
```



Опростиаване на изрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x;
    }

    return x;
    x++;
}
```



Разпространение на копията

Copy Propagation

Анализ

- ❖ Ако се правят излишни копия на променливи

Трансформация

- ❖ $Y = X$ $Z = 3 + X$
 $Z = 3 + Y$

Последствия:

- ❖ По-малко генерирани инструкции
- ❖ По-малко използвана памет/регистри
- ❖ Може да доведе до неправилно заделяне на регистри (register spills)



Разпространение на копията

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
        x = x;
    }

    return x;
    x++;
}
```



Разпространение на копията

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
    }

    return x;
    x++;
}
```



Елиминиране на общите подизрази

Common Expression Elimination

Анализ

- ❖ Ако един и същ подизраз се изчислява повече от един път

Трансформация

- ❖ Подизразът се присвоява на променлива и се използва променливата на останалите места

Последствия:

- ❖ По-малко генерирани инструкции
- ❖ По-малко изчисления
- ❖ Има нужда от допълнителна памет/регистри и може да доведе до неправилно заделяне на регистри (register spills)
- ❖ Може да възпрепятства паралелизацията като добави излишни зависимости



Елиминирање на обичите подизрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N)*(i+N);
    }

    return x;
    x++;
}
```



Елиминиране на общите подизрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        x = x + (8*t2/t1)*i + (i+N) * (i+N);
    }

    return x;
    x++;
}
```



Елиминиране на общите подизрази

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y, t;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
    x++;
}
```

*Какво може да се
направи още?*



Елиминиране на мъртъв код

Dead Code Elimination

Анализ

- ❖ Ако резултатът от изчислението не се използва
Dead store
- ❖ Ако няма път на изпълнение, по който да се стигне до оператора
Unreachable code

Трансформация

- ❖ Премахва се

Последствия:

- ❖ По-малко генерирани инструкции
- ❖ Може да освободи ресурса по-рано
- ❖ По-малко памет/регистри (няма нужда от съхраняване на резултата)



Елиминиране на мъртъв код

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y, t;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;

    }

    return x
    x++;
}
```



Елиминиране на мъртъв код

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, y, t;

    x = 0;
    y = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
    x++;
}
```



Елиминиране на мъртъв код

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t;

    x = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
    x++;
}
```



Елиминиране на мъртъв код

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t;

    x = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
    x++;
}
```



Елиминиране на мъртъв код

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t;

    x = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
}
```



Изваждане на инварианти от цикли

Loop Invariant Removal

Анализ

- ❖ Ако изчислението не зависи от итерациите на цикъла

Трансформация

- ❖ Изважда се извън цикъла

Последствия:

- ❖ Много по-малко работа в цикъла
- ❖ Трябва да съхрани в променлива извън цикъла, което увеличава областта на видимост и жизнения цикъл на променливата и може да доведе до неоптимално алокиране на регистрите (register spills)



Изваждане на инварианти от цикли

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t;

    x = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
}
```



Изваждане на инварианти от цикли

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t;

    x = 0;

    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + (8*t2/t1)*i + t*t;
    }

    return x
}
```



Изваждане на инварианти от цикли

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t, u;

    x = 0;

    u = (8*t2/t1);
    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + u*i + t*t;
    }

    return x
}
```



Замяна на операции

Strength Reduction

Анализ

- ❖ Ако може да замени операция с друга изискваща по-малко изчисления

Трансформация

- ❖ $\text{for}(i=0) t=a*i;$ $t=0; \text{for}(i=0) t=t+a;$

Последствия:

- ❖ По-малко изчисления
- ❖ Трябва да съхрани в променлива извън цикъла, което увеличава областта на видимост и жизнения цикъл на променливата и може да доведе до неоптимално алокиране на регистрите (register spills)
- ❖ Въвежда зависимост и паралелния цикъл става последователен.

Оправя се с инверсната (обратната) оптимизация



Замяна на операции

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t, u;

    x = 0;

    u = (8*t2/t1);
    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + u*i + t*t;
    }

    return x
}
```



Замяна на операции

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t, u;

    x = 0;

    u = (8*t2/t1);
    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + u*i + t*t;
    }

    return x
}
```

$u \cdot 0 \quad v = 0$
 $u \cdot 1 \quad v = v + u$
 $u \cdot 2 \quad v = v + u$
 $u \cdot 3 \quad v = v + u$
...



Замяна на операции

```
int calc(int N, int t1, int t2)
{
    int i;
    int x, t, u, v;

    x = 0;
    u = (8*t2/t1);
    v = 0;
    for (i = 0; i <= N; i++) {
        t = i+N;
        x = x + v + t*t;
        v = v + u;
    }

    return x
}
```



Какво спира оптимизациите

Оптимизаторът трябва да гарантира еквивалентността на програмата:

- ❖ За всички възможни входни данни
- ❖ За всички възможни начини на изпълнение
- ❖ За всички възможни архитектури

За да оптимизира програмата, компилаторът трябва да знае за и да разбира:

- ❖ Потока инструкции (Control Flow)
- ❖ Начинът на достъп до данните

През повечето време компилаторът не разполага с цялата информация за програмата. Тогава той:

- ❖ Редуцира региона, където трансформацията е приложима
- ❖ Редуцира агресивността на трансформациите
- ❖ Не оптимизира



Анализ на потока инструкции

Control Flow Analysis

Техниката се използва за събиране на информация изпълнението на отделни оператори, инструкции или извиквания на подпрограми

Представяния в компилатора:

- ❖ Граф на извикванията (Call graph)
- ❖ Граф на изпълнението (Control flow graph)

Какво възпрепятства анализа:

- ❖ Указатели към функции
- ❖ Индиректни преходи
- ❖ Изчислими goto оператори
- ❖ Големи switch оператори
- ❖ Цикли с exit и break оператори
- ❖ Цикли с неизвестни граници
- ❖ Условия с непредвидими преходи



Анализ на достъпа до данни

Data Accessors Analysis

Кой още може да чете и пише по данните

Представяния в компилатора:

- ❖ Дефинирай-използвай верига (Def-Use chain)
- ❖ Вектор на зависимостите

Какво възпрепятства анализа:

- ❖ Променливи по адрес
- ❖ Глобални променливи
- ❖ Параметри
- ❖ Масиви
- ❖ Указатели
- ❖ Volatile променливи



Instruction Scheduling

Правилната последователност от инструкции, които биха използвали максимално възможностите на процесора

Непоследователно изпълнение:

- ❖ Хардуера се опитва да разпредели инструкциите, които да бъдат изпълнени
- ❖ Компиляторът също може да помогне
Хардуерът няма достатъчно време да оптимизира инструкциите, а и няма цялата информация

Последователно изпълнение:

- ❖ Разпределението на инструкции е КРИТИЧНО



Ограничения при разпределяне на инстр.

От данните

Ако две инструкции използват една променлива те могат да зависят една от друга

❖ Видове зависимости:

❖ RAW/True: запис – четене

Когато инструкция 1 записва стойност и тя се използва от инструкция 2

❖ WAR/Anti: четене – запис

Когато инструкция 1 чете стойност, която по-късно е презаписана от инструкция 2

❖ WAW/Output: запис – запис

Когато инструкция 1 записва стойност и инструкция две записва стойност в една и съща променлива

От последователността на изпълнение

От ресурсите

