Методи на Транслация

Оптимизация

доц. д-р Александър Пенев

Оптимизация

Под **оптимизация** на програма ще разбираме нейната преработка с цел получаване на по-оптимална (от някаква гледна точка) програма.

Понеже не е известен алгоритъм за получаване на абсолютно оптимална програма, обикновено се цели минимизиране или максимизиране на някакво качество на програмата, например бързодействие, размер заемана памет, енерго-ефективност и др.

Много често оптимизацията е една от задачите на компилатора, но това не е ограничение т.е. може например програмиста да прави преработката.

Видове Оптимизация

Видове Оптимизация

- В зависимост от това дали се базира на специфични особеностите на целевата машина или не:
  - Машинно зависима – извършва се над вътрешното представяне на изходната програма (SSA и др.). Използват се знания за архитектурата на целевата машина (инструкции, регистри, кеш и др.);
  - Машинно независима – извършва се над вътрешното представяне на входната програма (AST и др.). Използват се факти от математиката, логиката и др. за подобряване на различни части програмата;

Видове Оптимизация

- В зависимост от това дали е специфична за конкретния входен език за програмиране или е общо приложима:
  - Езиково специфична/зависима – някои езици за програмиране имат специфични възможности или конструкции, които затрудняват някои видове оптимизация. Например в C и C++ наличието на указатели затруднява оптимизацията на достъпа до масиви и др. Функционалните езици предполагат някои специфични видове оптимизация;
  - Езиково независима – много от съвременните езици за програмиране имат сходни конструкции като *if*, *while*, *обекти* и др. Това позволява някои методи за оптимизация да бъдат лесно пренасяни и прилагани в различните езици;

Видове Оптимизация

- В зависимост от това дали се извършва по време на компилация или по-време на изпълнение:
  - Статична – прилагането на оптимизация по време на компилация. Предимствата са че обикновено има достатъчно време за сложни и време отнемащи анализи и трансформации на кода;
  - Динамична – оптимизацията по време на изпълнение на програмата може да се извършва на различни нива и от различни елементи на системата (например от JIT). ;

Видове Оптимизация

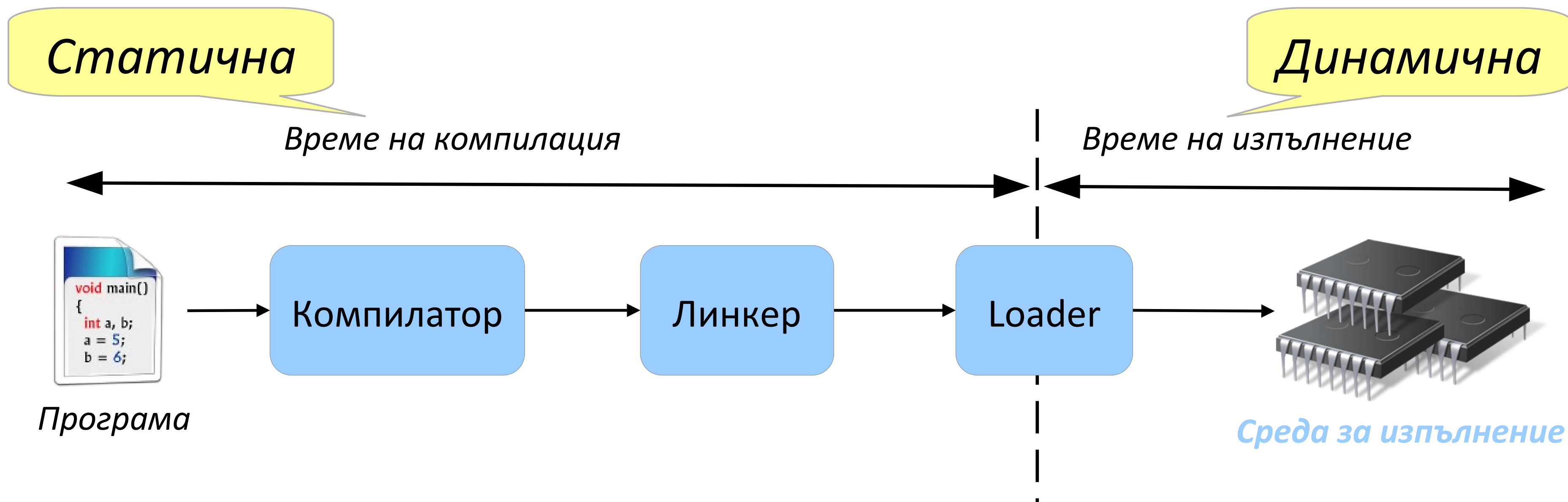
- В зависимост от това дали се извършва малки обособени части на програмата (изрази, функции/методи, модули), обхваща повече от една част или програмата като цяло:
  - Локална – най-често това са оптимизации съсредоточени до анализ и трансформация на малки части на програмата. Много често това са функциите/методите на програмата, един по един без анализ на взаимодействието им;
  - Между процедурна (IPO) – когато се анализира поведението на програмата не само в отделните функции/методи, но и тяхното взаимодействие;
  - Глобална (WPO) – обхваща по-големи единици от програмата като цяло (модули, класове, дори цялата програма);

Видове Оптимизация

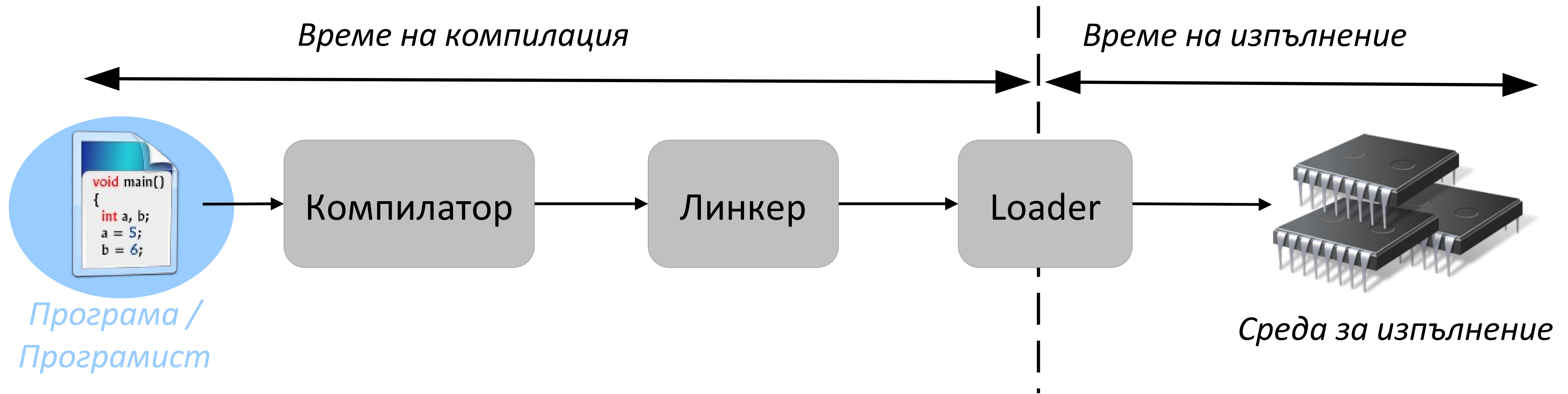
- В зависимост от това дали семантиката на входната програма се запазва се запазва напълно или само в допустими граници:
  - Семантично еквивалентно оптимизации – повечето класически оптимизации (тези извършвани от компилаторите) запазват напълно семантиката на оригиналната, спрямо оптимизираната програма;
  - Семантично нееквивалентни оптимизации – понякога с цел подобряване на бързодействието (и други) е допустимо семантиката да не бъде запазена напълно. Такива оптимизации се правят или от програмиста „на ръка“ върху входната програма или от специализирани системи за мета програмиране и оптимизации;

Оптимизационен Контунуум

Оптимизационен континуум

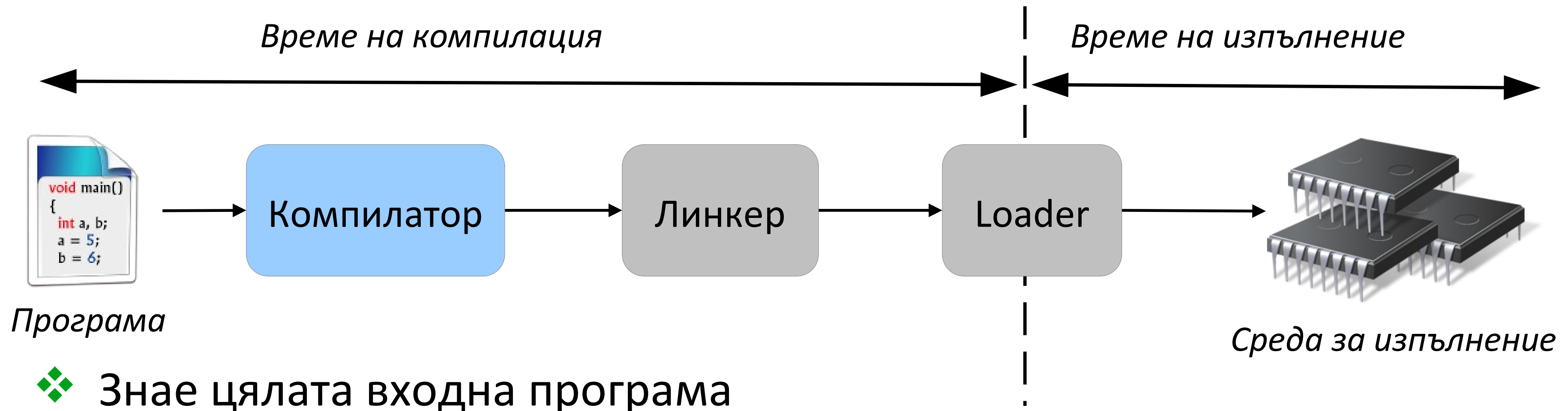


Оптимизационен континуум



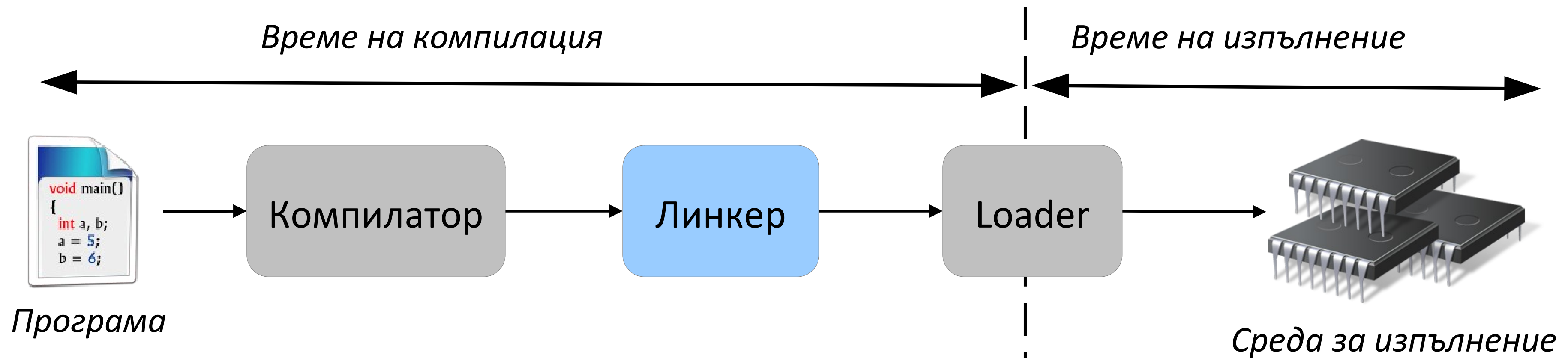
- Програмиста (потенциално) знае всичко за цялата входна програма
- Може да бъдат правени произволни промени на алгоритъма
- Промяната е сравнително бавна и остава за постоянно в изходния код
- Не знае (много) за средата на изпълнение
- Не знае (много) за архитектурата на изпълнение

Оптимизационен континуум



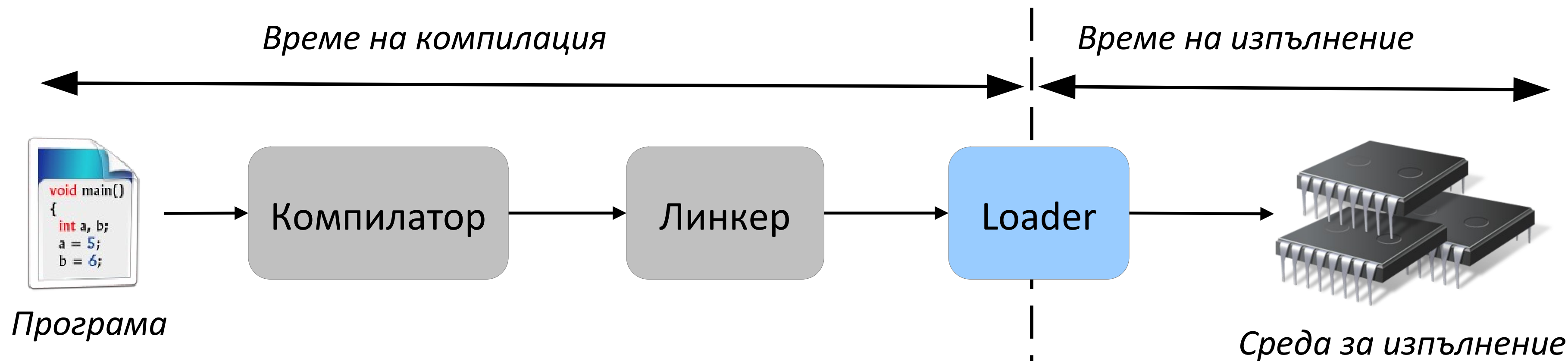
- Знае цялата входна програма
- Лесно управляват трансформациите от високо ниво към ниско ниво
- Времето на компилация не е проблем
- Не вижда цялата програма
- Не знае (много) за средата на изпълнение
- Не знае (много) за архитектурата на изпълнение

Оптимизационен континуум



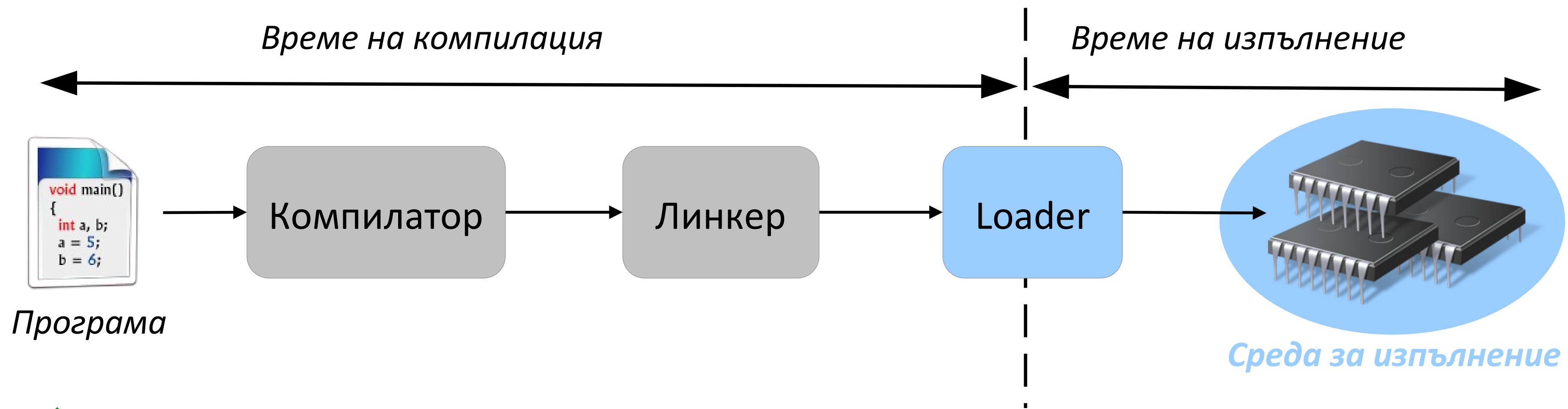
- Знае цялата входна програма
- Може да не вижда цялата програма
- Няма достъп до сорс кода
- Не знае (много) за средата на изпълнение
- Не знае (много) за архитектурата на изпълнение

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 оптимизация отнема от времето за изпълнение

Известни Методи за Оптимизация

Как действат оптимизационните методи

Оптимизацията в съвременните компилатори е разделена на множество методи, които последователно (а понякога и циклично) се прилагат над вътрешните представяния на програма (AST, CFG, CG, SSA, TAC, и др.).

Всеки оптимизационен метод се състои от две части:

- Анализ;
- Трансформация;

Някои видове анализ могат да се използват от повече от един оптимизационни методи. Това помага за по-добро бързодействие.

Някои класове оптимизационни методи

- Анализ и оптимизация на потоците данни (Data-flow optimizations);
- SSA базирани оптимизации (SSA-based optimizations);
- Оптимизация на цикли (Loop optimizations);
- Функционални оптимизации (Functional language optimizations);
- Между процедурни оптимизации (Interprocedural optimizations);
- Оптимизация в генератора на код (Code generator optimizations);
- Peephole оптимизация (Peephole optimization);
- Оптимално разпределяне на регистри;
- Рематериализация (Rematerialization);
- Оптимизация по време на свързване (Link time optimizations);
- Други;

Примери за Оптимизационни Методи

(машинно независими)

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;

Анализ на потока инструкции

Control Flow Analysis

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

Представяния в компилатора:

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

Какво може да 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:

- Верига на Използване-Дефиниране (Use-Define chain);
- Вектор на зависимостите;

Какво възпрепятства анализа:

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

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 ****   }
```

Изчисляване на константни изрази

Constant Expressions (constexpr)

Анализ:

Ако при всички възможни пътища на изпълнение, стойността на даден израз е константна (или е обявен за константен от програмиста – constexpr), то той може да се изчисли още по време на компилация.

Трансформация:

Изчислява се израза и се замества със стойността (като константа).

Последствия:

- Няма нужда израза да се изчислява по време на изпълнение
  Освобождава се памет или регистри, отпадат инструкции;
- Може да доведе до нужда/възможност от допълнителна оптимизация;

Изчисляване на константни 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 + (2*4*t2/t1)*i + (i+N)*(i+N);
    x = x + y*(t1+t2);
  }

  return x;
  x++;
}
```

Изчисляване на константни 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 + y*(t1+t2);
  }

  return x;
  x++;
}
```

Разпространение на константи

Constant Propagation

<u>Анализ:</u>

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

<u>Трансформация:</u>

Замества се променливата с тази констант.

<u>Последствия:</u>

- Няма нужда да се съхранява тази стойност в променлива
  Освобождава се памет или регистър;
- Може да доведе до нужда/възможност 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++;
}
```

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 + 0*(t1+t2);
  }

  return x;
  x++;
}
```

Опростяване на изрази

Algebraic Simplification

<u>Анализ:</u>

Ако даден под израз е изчислим по време на компилация.

Ако могат да се използват по-бързи инструкции.

<u>Трансформация:</u>

X * 0 = 0;   X * 1 = X; $X * 2^4 = X << 4$

X + 0 = X;   X + X = X << 2

…

Опростяване на изрази

Algebraic Simplification

<u>Последствия:</u>

- 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;

Опростяване 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++;
}
```

Опростяване 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++;
}
```

Опростяване 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++;
}
```

Опростяване 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

<u>Анализ:</u>

Ако се правят излишни копия на променливи.

<u>Трансформация:</u>

Y = X Z = 3 + Y	→	Z = 3 + X
X = X	→	ε (вид „премахване на ненужни действия“)

<u>Последствия:</u>

- По-малко генерирани инструкции
- По-малко използвана памет/регистри
- Може да доведе до неправилно заделяне на регистри (reg. 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

Елиминиране на общите подизрази

Common Expression Elimination

<u>Последствия:</u>

- 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;

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;

  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 резулт.)

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;

}
```

Инверсия на цикъл

Loop inversion

Анализ:

Търсим цикли от вида „while/do“.

Трансформация:

Заменя се с еквивалентен цикъл „do/while“, като преди него се поставя проверка дали условието е изпълнено. Ако по време на компилация първото изпълнение на условието е известно че е истина, то „защитния“ if може да се пропусне;

Последствия:

- Намаля наполовина броя на преходите, когато цикъла се изпълнява. Премахва безусловния преход в края (намаля stalls);
- Увеличава кода (изисква изпълнение на условието на две места);

Развиване (разделяне, сливане и др.) на цикли

Loop Unrolling

Анализ:

Ако границите на цикъла са ясни по време на компилация и в случаите, когато броя на итерациите е сравнително малък.

Трансформация:

Тялото на цикъла се повтаря, като в него управляващата променлива се заменя с константа (различна за всяка итерация) и се опимизирас.

Последствия:

- Няма излишни условни и безусловни преходи;
- Няма операции за обновяване на управляващата променлива, както и такива за проверка за край на цикъла;
- Програмата става по-голяма;

Развиване на цикли (при фикцирано N=3)

```
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;

}
```

Развиване на цикли (при фиксирано N=3)

```
int calc(int N, int t1, int t2)
{

  int x, t;

  x = 0;


  t = 0+3; x = x + (8*t2/t1)*0 + t*t; // i=0
  t = 1+3; x = x + (8*t2/t1)*1 + t*t; // i=1
  t = 2+3; x = x + (8*t2/t1)*2 + t*t; // i=2
  t = 3+3; x = x + (8*t2/t1)*3 + t*t; // i=3


  return x;

}
```

Развиване на цикли (при фиксирано N=3)

```
int calc(int N, int t1, int t2)
{

  int x, t;

  x = 0;


  t = 3; x = x + 0 + t*t;              // i=0
  t = 4; x = x + (8*t2/t1) + t*t;      // i=1
  t = 5; x = x + (8*t2/t1)*2 + t*t;    // i=2
  t = 6; x = x + (8*t2/t1)*3 + t*t;    // i=3


  return x;

}
```

Развиване на цикли (при фикцирано N=3)

```
int calc(int N, int t1, int t2)
{

  int x, temp;





  temp = (8*t2/t1);
  x = 9 + temp + 16;    // i=1
  x = x + temp*2 + 25; // i=2
  x = x + temp*3 + 36; // i=3


  return x;

}
```

Развиване на цикли (при фикцирано N=3)

```
int calc(int N, int t1, int t2)
{

  int x, temp;





  temp = (8*t2/t1);
  x = 9 + temp + 16 + temp*2 + 25 + temp*3 + 36;




  return x;

}
```

Развиване на цикли (при фиксирано N=3)

```
int calc(int N, int t1, int t2)
{

  int x, temp;




  temp = (8*t2/t1);
  x = temp*6 + 86;




  return x;

}
```

Замяна на операции

Strength Reduction

<u>Анализ:</u>

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

<u>Трансформация:</u>

for(i=0) t=a*i; → t=0; for(i=0) t=t+a;

Замяна на операции

Strength Reduction

<u>Последствия:</u>

- 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 инверсна (обратна) 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*0  →  v = 0
u*1  →  v = v+u
u*2  →  v = v+u
u*3  →  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;

}
```

Вграждане на метод/функция

Inline

<u>Анализ:</u>

Ако дадена функция/метод са отбелязани (например с ключовата дума *inline*) от програмиста за вграждане в мястото си на извикване и/или функцията/метода се извиква само на едно място в програмата и/или кода за предаване на параметри е по-голям от кода на функцията и/или тя е сравнително проста (някаква евристична метрика).

<u>Трансформация:</u>

Използва се правилото за копиране на тялото на функцията/метода в местата на извикване, като формалните параметри се заменят с изразите на актуалните параметри (може да се направят и оптимизации), локалните променливи на функцията/метода се преместват в мястото на извикване като нови локални променливи.

Вграждане на метод/функция

Inline

<u>Последствия:</u>

- Избягват се инструкции за предаване на параметри и извикване на функцията, както пролог и епилог кода на самата функция;
- Бързодействието на изходната програма обикновено се подобрява;
- Увеличава се кода на изходната програма;
- Може да доведе до неоптимално алокиране на регистрите (register spills);

Вграждане на метод/функция

```
inline int max(int a, int b)
{
  if (a > b) return a; else return b;
}

int calc(int t1, int t2, int t3)
{
  int i, j;

  i = max(t1, t2);
  j = max(i, t3);

  return j;
}
```

Вграждане на метод/функция

```
inline int max(int a, int b)
{
  if (a > b) return a; else return b;
}

int calc(int t1, int t2, int t3)
{
  int i, j;

  if (t1 > t2) i = t1; else i = t2;   // i = max(t1, t2);
  if (i > t3) j = i; else j = t3;     // j = max(i, t3);

  return j;
}
```

Клониране на метод/функция

Method/Function clone

Анализ:

Ако дадена функция/метод се извиква много често с едни и същи параметри (често константи, default параметри и др.) или или има много параметри, един или няколко от които са фикцирани. Това се проверява с някаква евристична метрика специфична за ЕП и компилатора на съответния език.

Трансформация:

Създават се едно или повече копия на функцията/метода (наречени клонинги), които отговарят на различните често срещани комбинации на извикващите параметри. На местата на извикване се извиква или оригиналната функция/метод или най-подходящия клонинг.

Клониране на метод/функция

Method/Function clone

Последствия:

- Бързодействието на изходната програма обикновено се подобрява;
- Намаля се броя на параметрите за предаване;
- Увеличава се кода на изходната програма;

Клониране на метод/функция

```
int f(int a, int b, int c = 0) {
  return (a * b) + c;
}




int calc(int t1, int t2, int t3) {
  return f(t1, t2) + f(t2, t3) + f(t1, t3, 2);
}

int calc1(int x1, int x2) {
  return f(x1, 2) + f(x2, 2);
}
```

Клониране на метод/функция

```
int f(int a, int b, int c = 0) {
  return (a * b) + c;
}

int f_clone1(int a, int b) { return a * b; }
int f_clone2(int a) { return a * 2; }

int calc(int t1, int t2, int t3) {
  return f_clone1(t1, t2) + f_clone1(t2, t3) + f(t1, t3, 2);
}

int calc1(int x1, int x2) {
  return f_clone2(x1) + f_clone2(x2);
}
```

Сливане на функции/методи

Merge Functions/Memxoдc

<u>Анализ:</u>

Търси функции с еквивалентен код (не изисква доказване на еквивалентност на две функции в общия случай).

<u>Трансформация:</u>

Оставя кода само на една от тях, като извикванията на другите се заменят с извиквания към нея.

<u>Последствия:</u>

- Намаля размера на програмата;

Сливане на функции/методи

```
int rgb_sum(int r, int g, int b) {
  return r + g + b;
}

int xyz_sum(int x, int y, int z) {
  return x + y + z;
}

int calc(int t1, int t2, int t3) {
  return rgb_sum(t1, t2, t3) + xyz_sum(t1, t2, t3);
}
```

Сливане на функции/методи

```
int merged(int a, int b, int c) {
  return a + b + c;
}





int calc(int t1, int t2, int t3) {
  return merged(t1, t2, t3) + merged(t1, t2, t3);
}
```

Други

Съществуват много други методи, например:

- Линеаризация на многомерни масиви;
- Девиртуализация на методи;
- Премахваме/Опростяване на шаблони за дизайн, като Итератор и др.;
- Минимизиране на броя на използваните променливи;
- Автоматична векторизация и паралелизация;
- Профилно-управляема оптимизация (Profile-guided optimization);
- Анализ на указатели;

Други

- Премахване на рекурсията (най-често чрез Tail-call optimization);
- Deforestation (Data structure fusion);
- Частично изчисление (Partial Evaluation);
- „Мързеливо“ изчисление (Lazy Evaluation);
- Премахване на проверките за излизане извън границите /на масива/ (Bounds-checking elimination);
- Премахване на ненужни обработки на exceptions;
- Пренареждане на проверките (Test reordering);
- Polyhedral model optimization;
- и много други;

Примери за Оптимизационни Методи

(машинно зависими)

Машинно зависима оптимизация

- Извършва се над вътрешното представяне на изходната програма;
- Използват се знания за архитектурата на целевата машина (инструкции, регистри, кеш и др.);
- Най-често работи над малки (обикновено линейни) последователности от машинни инструкции, наречени базови блокове, в които се търсят инструкции и се заменят с по-оптимален техен еквивалент. Това може да включва: премахване на ненужни инструкции; замяна на една или повече инструкции с други; дори понякога добавяне на инструкции;

Премахване на излишни инструкции

Много често при генерация на изходния код се генерират инструкции, които лесно може да се прецени че не са наобходими.

Когато се генерират последователности от изрази и присвоявания, обикновено генератора на код ги обработва един по един и помежду различните изрази се генерират инструкции, които може да не са необходими.

```
...
t1 = a + b;
d = t1 + c;
...
```

→

```
...
MOV    AX, [a]
ADD    AX, [b]
MOV    [t1], AX
MOV    AX, [t1]    ; зачеркната
ADD    AX, [c]
MOV    [d], AX
...
```

→

```
...
MOV    AX, [a]
ADD    AX, [b]
MOV    [t1], AX
ADD    AX, [c]
MOV    [d], AX
...
```

← Това също може да е излишна инструкция, ако „***t1***“ е временна променлива.

Замяна на по-бавни инструкции с по-бързи

```
...                 ...
aload 1             aload 1
aload 1      →      dup
mul                 mul
...                 ...
```

```
...                 ...                     ...
MOV   AX, 0         XOR   AX, AX      или   XOR   AX, AX
MOV   BX, AX  →     MOV   BX, AX      дори  XOR   BX, BX
...                 ...                     ...
```

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;
- Компилаторът също може да помогне

  Хардуерът няма достатъчно време да оптимизира инструкциите, а и няма цялата информация;

Последователно изпълнение:

- Разпределението на инструкции е КРИТИЧНО;

Оптимално разпределяне на регистрите

Optimal Register Allocation

Оптималното разпределяне на регистрите е една (от многото) NP-пълни задачи, които трябва да се решават с евристични алгоритми в реално време, за да се постигне оптимална генерация на изходния код на компилираната програма.

Задачата зависи от много параметри:

- Броя и вида на наличните регистри в архитектурата на процесора;
- Конвенцията за предаване на параметри;
- Може да става локално за всеки базов блок или глобално за цялата функция или между процедурно;
- Други;

Оптимално използване на йерархията на пам.

Използват се свойствата на йерархията на паметта. Най-използваните данни се записват първо в регистрите, след това в кеша, в паметта и т.н.

```
...
x = (a + b) * c;
...
```

→

```
...
MOV     AX, [a]
ADD     AX, [b]
MOV     [t1], AX
MOV     AX, [t1]
MUL     AX, [c]
MOV     [x], AX
...
```

→

```
...
MOV     AX, [a]
ADD     AX, [b]
MOV     BX, AX
MOV     AX, BX
MUL     AX, [c]
MOV     [x], AX
...
```

Оптимално използване на йерархията на пам.

Използват се свойствата на йерархията на паметта. Най-използваните данни се записват първо в регистрите, след това в кеша, в паметта и т.н.

```
...
x = (a + b) + c / (a + b);
...
```

→

```
...
MOV     AX, [a]
ADD     AX, [b]
MOV     [t1], AX
ADD     AX, [c]
DIV     AX, [t1]
...
```

→

```
...
MOV     AX, [a]
ADD     AX, [b]
MOV     BX, AX
ADD     AX, [c]
DIV     AX, BX
...
```

По-малко преходи. Премахване на усл. Преходи

Branch-free code

Преходите (условни или безусловни) пречат на предварителното извличане на инструкциите, като по този начин забавят кода. Използването на вграждане на методи или развиване на циклите може да намали броя на преходите с цената на увеличаване на размера на крайната програма. Съвременните процесори (вкл GPGPU) имат инструкции, които помагат за премахване на преходи.

if (x) a=5; **else** a=a+1; →

```
      MOV   BX, 5
      TEST  [x]
      JE    E1
      MOV   [a], BX
      JMP   E2
E1:   INC   [a]
E2:   ...
```

→

```
MOV     AX, [a]
INC     AX
MOV     BX, 5
TEST    [x]
CMOVNZ  AX, BX
MOV     [a], AX
...
```

Рематериализация/Преизчисляване

Rematerialization

В съвременните компютърни системи съотношението скорост на CPU към скорост достъпа до паметта става все по-голямо. Това води до това, че понякога е много по-добре даден временен резултат, съхранен във временна променлива в паметта да бъде преизчислен наново в CPU, вместо да бъде прочетен от паметта.

Разбира се наличието на кеш на няколко нива може да е причина за неприложимост на тази оптимизация. Друг възпрепятстваща прилагането на метода причина е ако в регистрите (и кеша) не се намират всички необходими изходни данни за преизчисление на подизраза.

Намаляне на дълбочината на стека

Stack height reduction

При сложни изрази, редът на обхождане на AST има съществено значение за броя на временните резултати съхранявани в стека. В зависимост от изразите, следвайки правилата комутативност в математиката често изразът може да се запише в еквивалентна форма, но използващ по-малко временни резултати и съответно дълбочината на стека става по-малка. Използването на Tail-call също подобрява използването на стека при рекурсии и др.

```
x = a + (b  * c);
//x = (b  * c) + a;
```

→

```
MOV     AX, [a]
MOV     [TMP1], AX
MOV     AX, [b]
MUL     AX, [c]
ADD     AX, [TMP1]
MOV     [x], AX
```

→

```
MOV     AX, [b]
MUL     AX, [c]
ADD     AX, [a]
MOV     [x], AX
```

Възможни Проблеми

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;

Какво спира оптимизациите

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;

Ограничения при разпределяне на инстр.

1. 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

2. От последователността на изпълнение

3. От ресурсите

Въпроси?

apenev@uni-plovdiv.bg