

Обзор оптимизирующих преобразований

Синявин А.В.

Ранние оптимизации

Вычисление константных выражений (constant folding/CF)

```
for(i=1;i<=100;i++)  
{  
    fun1(5+6, 20*3+7);  
}  
→  
for(i=1;i<=100;i++)  
{  
    fun1(11, 67);  
}
```


Scalar replacement/SC

```
for(int i=1;i<=N; i++)  
{  
    for(int j=1;j<=N;j++)  
    {  
        A[i] = A[i] + B[j];  
    }  
}  
→  
for(int i=1;i<=N; i++)  
{  
    T=A[i];  
    for(int j=1;j<=N;j++)  
    {  
        T = T + B[j];  
    }  
    A[i] = T;  
}
```

T — некоторый регистр

Алгебраические упрощения (algebraic simplification/AS)

$$i + 0 = 0 + i = i - 0 = i$$

$$0 - i = -i$$

$$i * 1 = 1 * i = i / 1$$

$$i * 0 = 0 * i = 0$$

$$-(-i) = i$$

$$i + (-j) = i - j$$

$$b \text{ or } \text{TRUE} = \text{TRUE or } b = \text{TRUE}$$

$$b \text{ or } \text{FALSE} = \text{FALSE or } b = b$$

$$i \ll 0 = i \gg 0 = 0$$

$$i \ll w = i \ll w = 0, \text{ где } w \geq \text{кол-во битов в } i$$

и т.д.

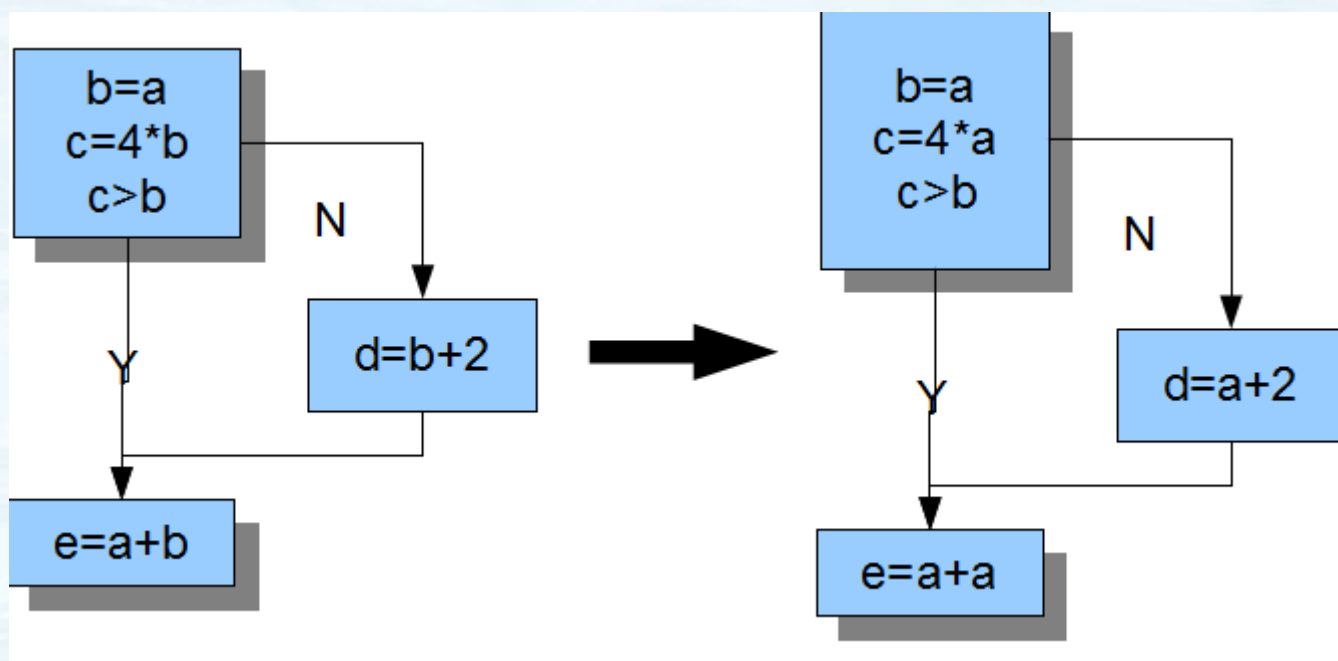
Алгебраические упрощения (algebraic simplification / AS)

Важно

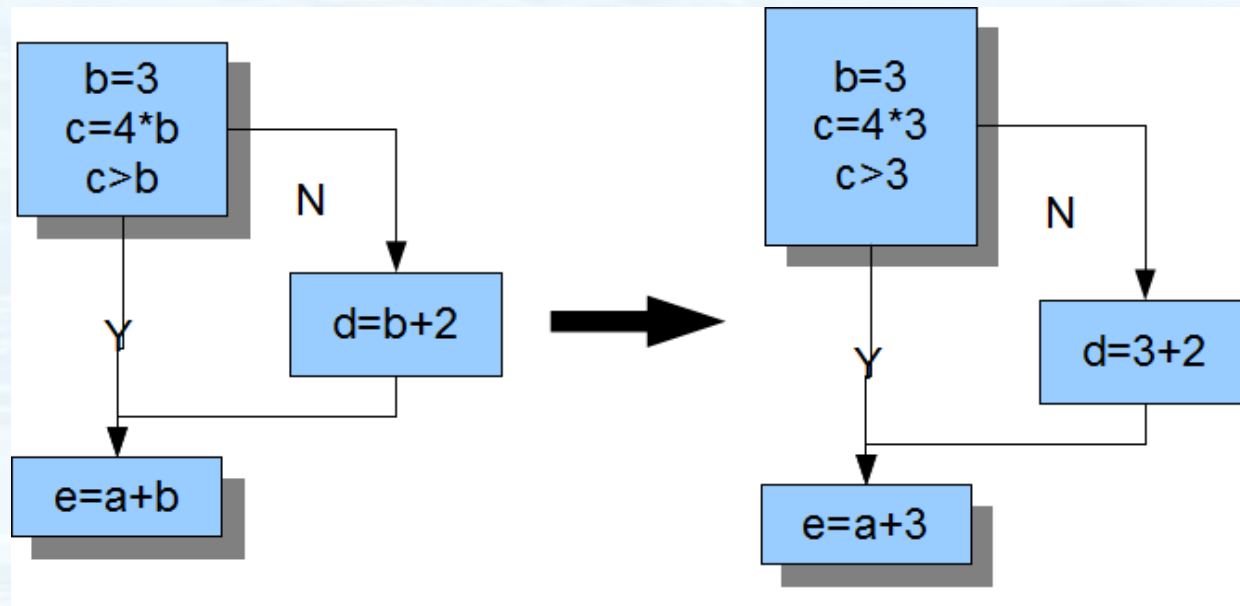
```
float f1, f2, f3;  
f1 = ...;  
f2 = ...;  
f3 = ...;  
bool res = ( (f1 + f2) + f3 == f1 + (f2 + f3) );  
res – false
```

Реассоциация может изменить результат
вычислений

Распространение копий (copy propagation / CopyP)



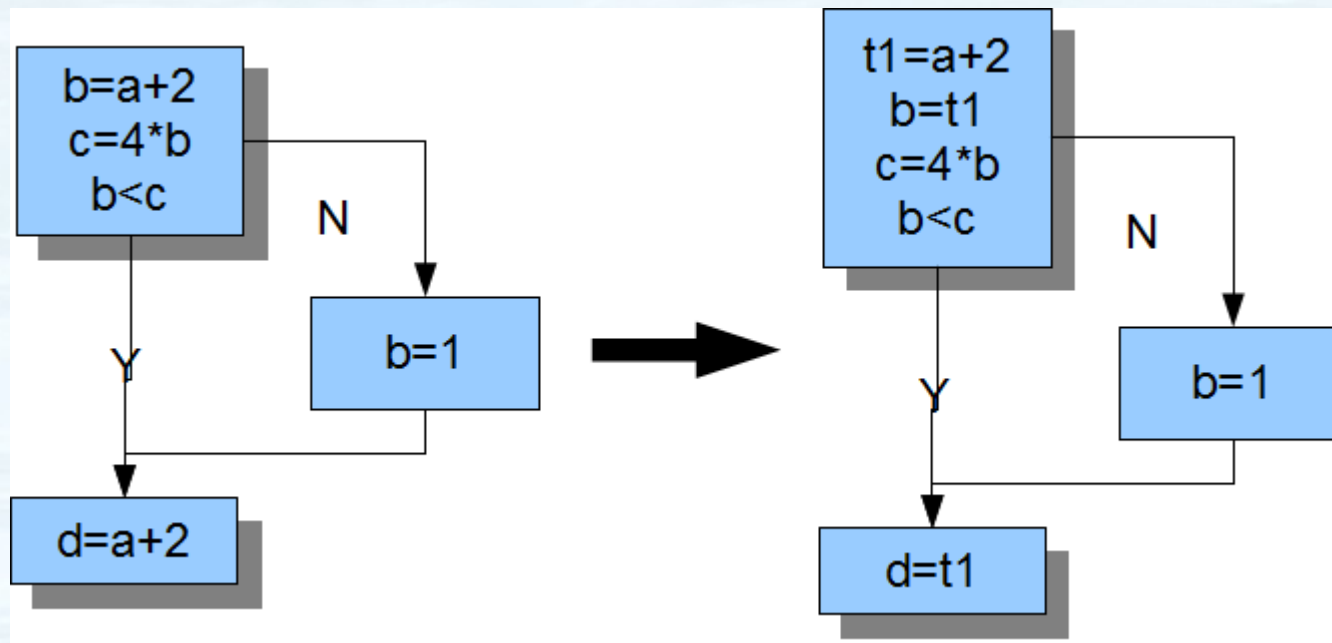
Распространение констант (Constant propagation / ConstP)



В отдельных случаях после преобразования логические выражения могут вычислены на этапе компиляции, т.е. CFG может измениться

Устранение избыточности

Устранение общих подвыражений (common subexpression elimination/CSE)



Вынесение инвариантного кода (loop-invariant code motion/LICM)

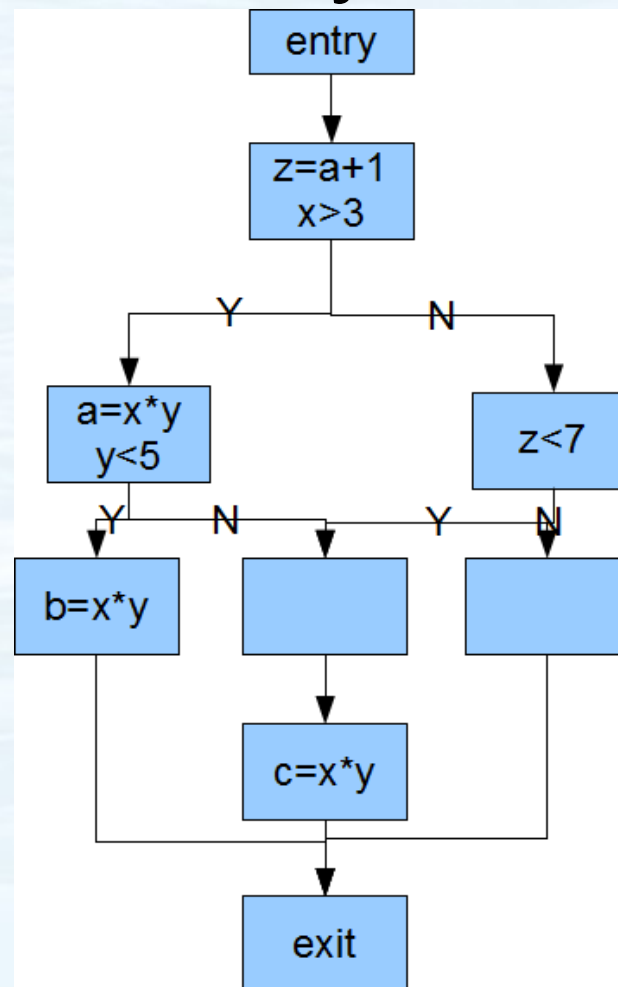
```
for(i=1;i<=100;i++)  
{  
    l=i*(n+2);  
    for(j=1;j<=100;j++)  
    {  
        A[i][j]=100*n + 10*l  
        + j;  
    }  
}
```



```
t1=10*(n+2);  
t2=100*n;  
for(i=1;i<=100;i++)  
{  
    t3=t2+i*t1;  
    for(j=1;j<=100;j++)  
    {  
        A[i][j]=t3+j;  
    }  
}
```

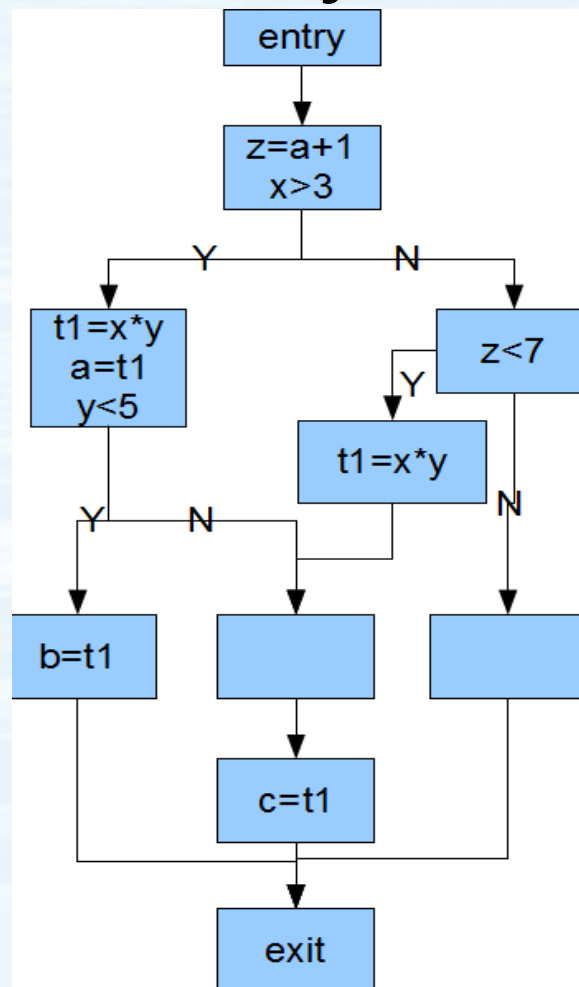

Устранение частичной избыточности (partial-redundancy elimination/PRE)

До
преобразования

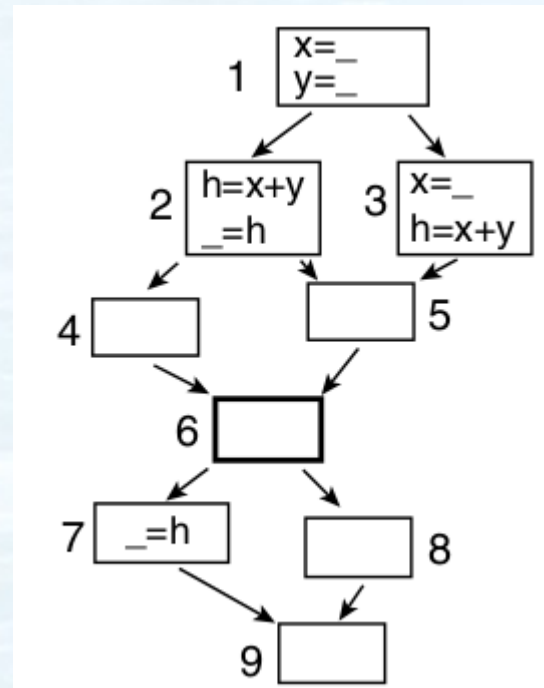
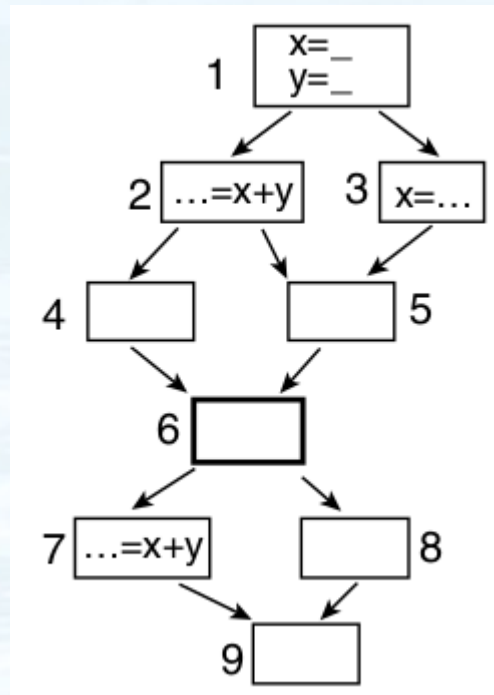


Устранение частичной избыточности (partial-redundancy elimination/PRE)

После
преобразования



Устранение частичной избыточности (учёт профильной информации)

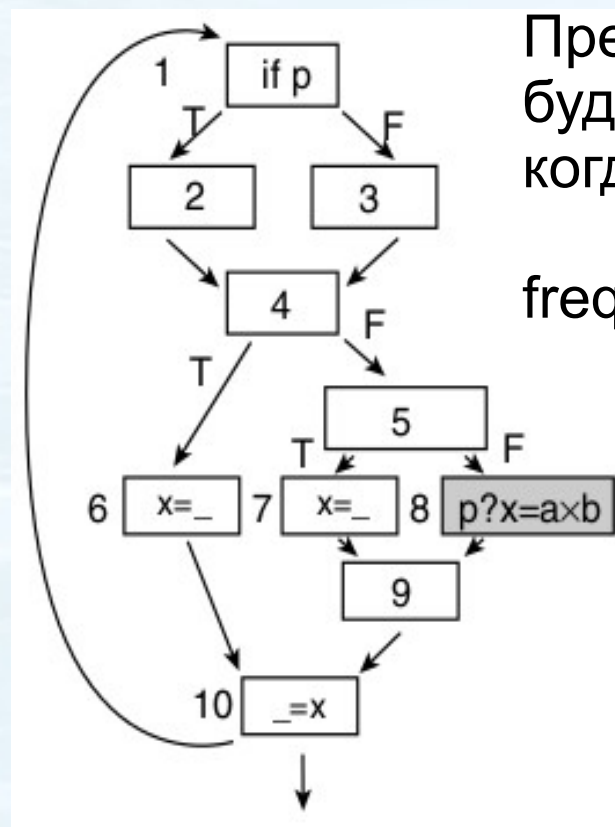
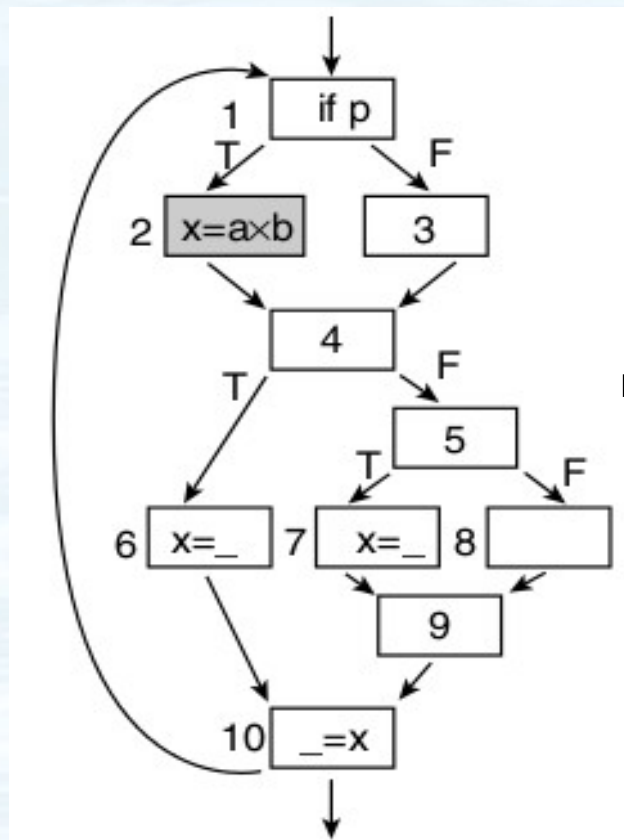


Преимущество
будет тогда,
когда

$$\text{freq}(3) < \text{freq}(7)$$

путь	x + y (до)	x + y (после)
1,2,4,6,7,9	выч. дважды	выч. один раз
1,2,5,6,8,9	выч. один раз	выч. один раз
1,3,5,6,8,9	не вычисл.	выч. один раз

Устранение частично мёртвого кода (учёт профильной информации)

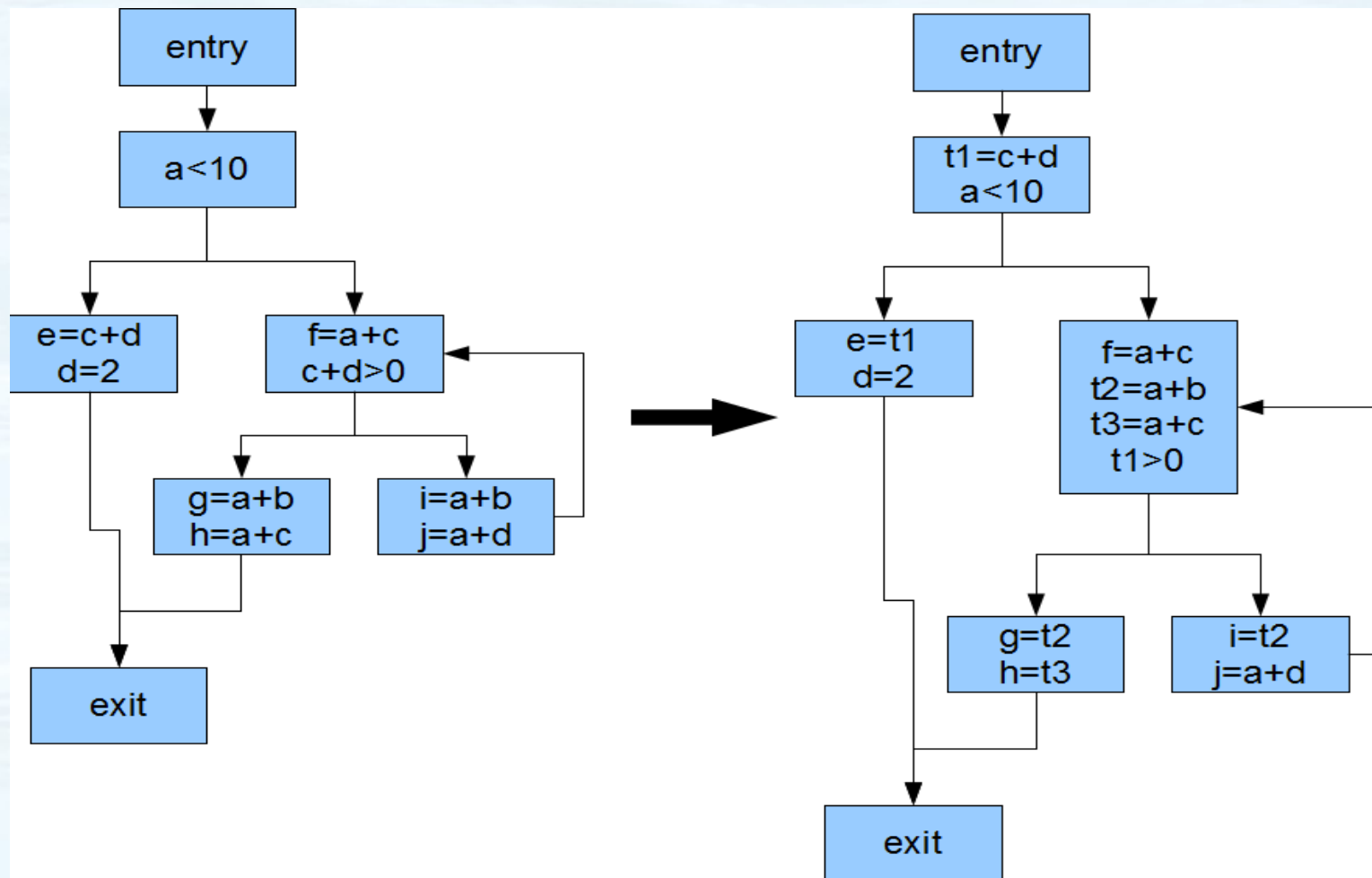


Преимущество
будет тогда,
когда

$$\text{freq}(8) < \text{freq}(2)$$

путь	$a * b$ (до)	$a * b$ (после)
1,2,4,6,10	мёртвое в 2	нет мёртв. выч.
1,2,4,5,7,9,10	мёртвое в 2	нет мёртв. выч.
1,3,4,5,8,9,10	нет мёртв. выч.	добавлено выч.

Подъём кода (code hoisting/CH)



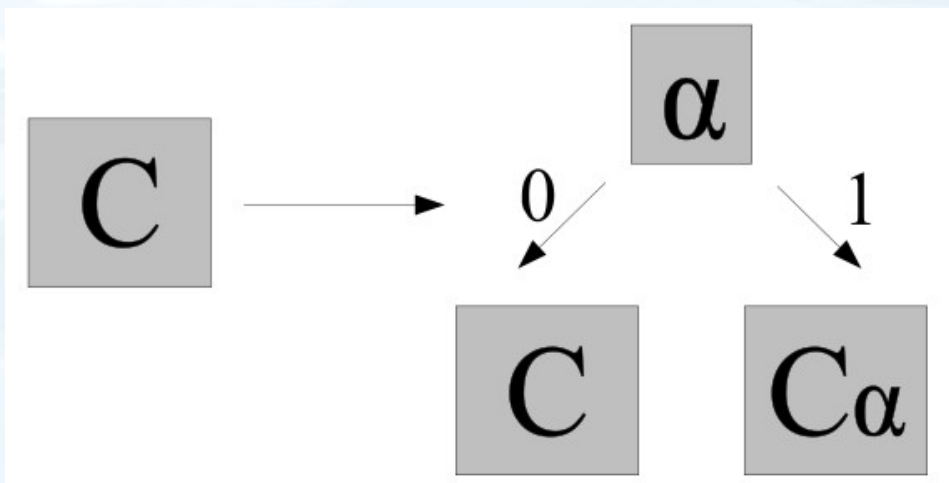
Специализация кода (общий случай)

C код для специализации

α условие, которое часто истинное

$C \longrightarrow \text{if}(\alpha) \text{ then } C_\alpha \text{ else } C$

или



Специализация кода (частный случай)

```
for(i=0; i<N; i++ )
{
    a[i] = c[i] * A;
    b[i] = d[i] * B;
    c[i] *= C;
}
```



```
if ( A == 0 )
{
    for(i=0; i<N; i++ )
    {
        a[i] = c[i] * 0;
        b[i] = d[i] * B;
        c[i] *= C;
    }
}
else
{
    for(i=0; i<N; i++ )
    {
        a[i] = c[i] * A;
        b[i] = d[i] * B;
        c[i] *= C;
    }
}
```

Очень часто $A = 0$
(например 99%)

**$c[i] * 0$ может быть
заменено на 0**

Давление на регистры (register pressure)

- Давление на регистры – требуемое количество физических регистров на этапе кодогенерации
- Если требуемое количество превышает действительное значение, то мы вынуждены сохранять значение регистров в стеке (т.е. появляются обращения к памяти)
- CSE/LCIM/PRE/CH – увеличивают давление на регистры

Оптимизация циклов

Раскрутка цикла (loop unroll)

```
for(i=1;i<=4*N;i++)  
{  
    body;  
}
```



```
for(i=1;i<=4*N;i+=4)  
{  
    body;  
    body;  
    body;  
    body;  
}
```

Уменьшается кол-во выполнений команд ветвлений и проверок условий.

Разрастается код => код может не влезть в instruction cache

Понижение силы операций (Strength reduction / SR)

```
for( i=0; i<N; i++ )  
{  
    f(i*25);  
}
```



```
t1=0;  
for(i=0;i<N;i++)  
{  
    f(t1);  
    t1 += 25;  
}
```

```
for( i=0; i<N; i++ )  
{  
    A[i] = B[i]*4;  
}
```



```
for(i=0;i<N;i++)  
{  
    A[i] = B[i] << 2;  
}
```

Замена дорогих операций на дешёвые внутри тела цикла

Удаление бесполезных индуктивных переменных

```
T = 0;  
for( i=0; i<N; i++ )  
{  
    ...  
    T += 34;  
    ...  
    // T нигде  
    // не используется  
}  
// T нигде  
// не используется
```



```
for(i=0;i<N;i++)  
{  
    ...  
    ...  
}
```


Удаление бесполезных bound-check

```
int A[100][10];  
for(i=0; i<50; i++)  
    for(j=0; j<3; j++)  
        if (i<100 && j<10)  
            A[i][j]=0;
```



```
int A[100][10];  
for(i=0; i<50; i++)  
    for(j=0; j<3; j++)  
        A[i][j]=0;
```

Переупорядочение операций (statement reordering / SR)

```
A[10] = 0;
for(int i=0; i<N; i++)
{
    B[i] = i;
}
for(int j=0; j<N; j++)
{
    A[j] = j + 2;
}
```



```
for(int i=0; i<N; i++)
{
    B[i] = i;
}
A[10] = 0;
for(int j=0; j<N; j++)
{
    A[j] = j + 2;
}
```

Цель:

- улучшить использование кэша данных
- планирование инструкций

Unswitching

```
for (i=0;i<N;i++)
{
    for(j=2;j<N;j++)
    {
        if (T[i]>0)
            A[i][j]=A[i][j-1]*T[i]+B[j];
        else
            A[i][j]=0;
    }
}
```



```
for (i=0;i<N;i++)
{
    if (T[i]>0)
    {
        for(j=2;j<N;j++)
            A[i][j]=A[i][j-1]*T[i]+B[j];
    }
    else
    {
        for(j=2;j<N;j++)
            A[i][j]=0;
    }
}
```

Уменьшается количество
операций ветвления

Отпиливание итераций (loop peeling)

```
for (i=1;i<=N;i++)  
{  
    A[i]=X*B[i];  
}
```



```
if (N>=1)  
{  
    A[1]=X*B[1];  
    for (i=2;i<=N;i++)  
    {  
        A[i]=X*B[i];  
    }  
}
```

Используется как
вспомогательное
преобразование, для
изменения количества
итераций цикла

Loop splitting

```
for(i=1; i<100; i++)  
{  
    A[i] = B[i] + C[i];  
    if (i > 10)  
        D[i] = A[i] + A[i-10];  
}
```




```
for(i=1; i<100; i++)  
{  
    A[i] = B[i] + C[i];  
}  
  
for(i=11; i<100; i++)  
{  
    A[i] = B[i] + C[i];  
    D[i] = A[i] + A[i-10];  
}
```

- позволяет регулировать количество итераций в цикле
- позволяет убрать проверки условия в цикле

Слияние циклов (loop fusion)

```
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];  
}
```



```
for (i=0;i<N;i++)  
{  
    A[i]=B[i]+1;  
    C[i]=A[i]/2;  
    D[i]=1/C[i+1];  
}
```

- если количество итераций неодинаково, то может использоваться loop peeling перед loop fusion
- улучшение локальности и уменьшение кол-ва ветвления
- увеличение размера тела цикла => могут появиться новые возможности по оптимизации, например, CSE
- увеличение размера тела цикла => может ухудшить instruction locality

Расширение скалярных переменных (scalar expansion)

```
for (i=0;i<N;i++)  
{  
    T=A[i] + B[i];  
    C[i] = T + 1/T;  
}  
→  
if (N>=1)  
{  
    Tx = new int[N];  
    for (i=0;i<N;i++)  
    {  
        Tx[i]=A[i] + B[i];  
        C[i] = Tx[i] + 1/Tx[i];  
    }  
}
```

Вспомогательное
преобразование — может дать
возможность для векторизации

Деление циклов (loop fission / loop distribution)

```
for(i=0;i<N;i++)  
{  
    A[i] = A[i+1]*3;  
    B[i] = sin(B[i]);  
}
```



```
for(i=0;i<N;i++)  
{  
    A[i] = A[i+1]*3;  
}
```

```
for(i=0;i<N;i++)  
{  
    B[i] = sin(B[i]);  
}
```

- позволяет раздробить большое тело цикла на несколько малых, чтобы поместить в instruction cache
- если тело цикла работает с несколькими большими циклами, то есть возможность улучшить data locality
- даёт возможность для других оптимизаций, например, векторизация

Реверс циклов (loop reversal)

Изменение порядка итераций в цикле
вспомогательное преобразование

```
for (i=1;i<=N;i++)  
{  
    A[i] = B[i] + 1;  
    C[i] = A[i] / 2;  
}
```



```
for(i=1;i<=N;i++)  
{  
    D[i] = 1 / C[i+1];  
}
```

```
for(i=N; i>=1; i--)  
{  
    A[i] = B[i] + 1;  
    C[i] = A[i] / 2;  
    D[i] = 1 / C[i+1];  
}
```


Перестановка циклов (loop interchanging)

```
for(i=0;i<N;i++)  
{  
    for(j=0;j<M;j++)  
    {  
        body  
    }  
}
```



```
for(j=0;j<M;j++)  
{  
    for(i=0;i<N;i++)  
    {  
        body  
    }  
}
```

Loop collapsing

```
int a[100][300];
```

```
for (i = 0; i < 300; i++)  
  for (j = 0; j < 100; j++)  
    a[j][i] = 0;
```



```
int a[100][300];  
int *p = &a[0][0];
```

```
for (i = 0; i < 30000; i++)  
  *p++ = 0;
```

Может облегчить другие
оптимизации, например loop unroll

Межпроцедурные оптимизации

Inlining

Достоинства:

- избавление от издержек, которые связаны с инструкцией вызова
- могут появиться дополнительные возможности по оптимизации

Недостатки:

- возрастание объёма кода и как следствие ухудшение instruction locality

Interprocedural constant propagation

```
void my_print(int p)
{
    if ( p > 0 )
        printf( "PAPA" );
    else
        printf( "MAMA" );
}
```

```
int main()
{
    int param = 3;
    my_print(param);
    return 0;
}
```



```
void my_print()
{
    if ( 3 > 0 )
        printf( "PAPA" );
    else
        printf( "MAMA" );
}
```

```
int main()
{
    int param = 3;
    my_print();
    return 0;
}
```

Devirtualization

```
class A
{
public:
    virtual void f();
};
```

До преобразования:

- вызов метода через таблицу виртуальных функций

```
class B : public A
{
public :
    virtual void f();
};
```

После преобразования:

- обычный вызов функции

```
A * obj = new B();
obj->f();
```



```
A * obj = new B();
obj->B::f();
```


Indirect call conversion

Похожа на devirtualization

Вызовы через указатель
на процедуру заменяются
на прямые вызовы

Function cloning

```
void f(int param)
{
    if (param > 0)
    {
        stmts1;
    }
    else
    {
        stmts2;
    }
}
```



```
void f_1()
{
    stmts1;
}

void f_2()
{
    stmts2;
}
```

- может улучшить instruction locality
- может дать дополнительные возможности по оптимизации каждой функции в отдельности

Function splitting

```
void func (...)  
{  
    if (test)  
        something_small  
    else  
        something_big  
}
```



```
void func_part (...)  
{  
    something_big  
}  
  
void func (...)  
{  
    if (test)  
        something_small  
    else  
        func_part (...);  
}
```

test — очень часто true

=>

function splitting даёт возможность для
function inlining

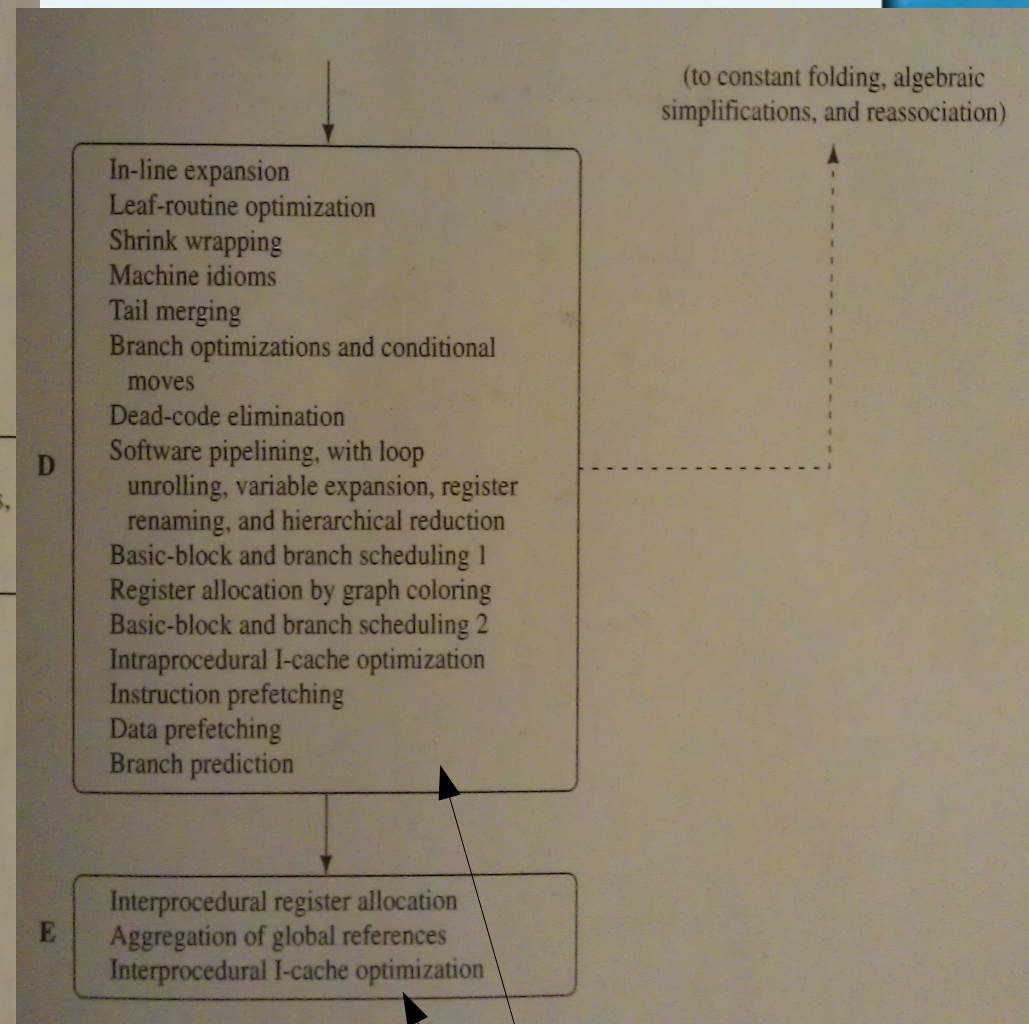
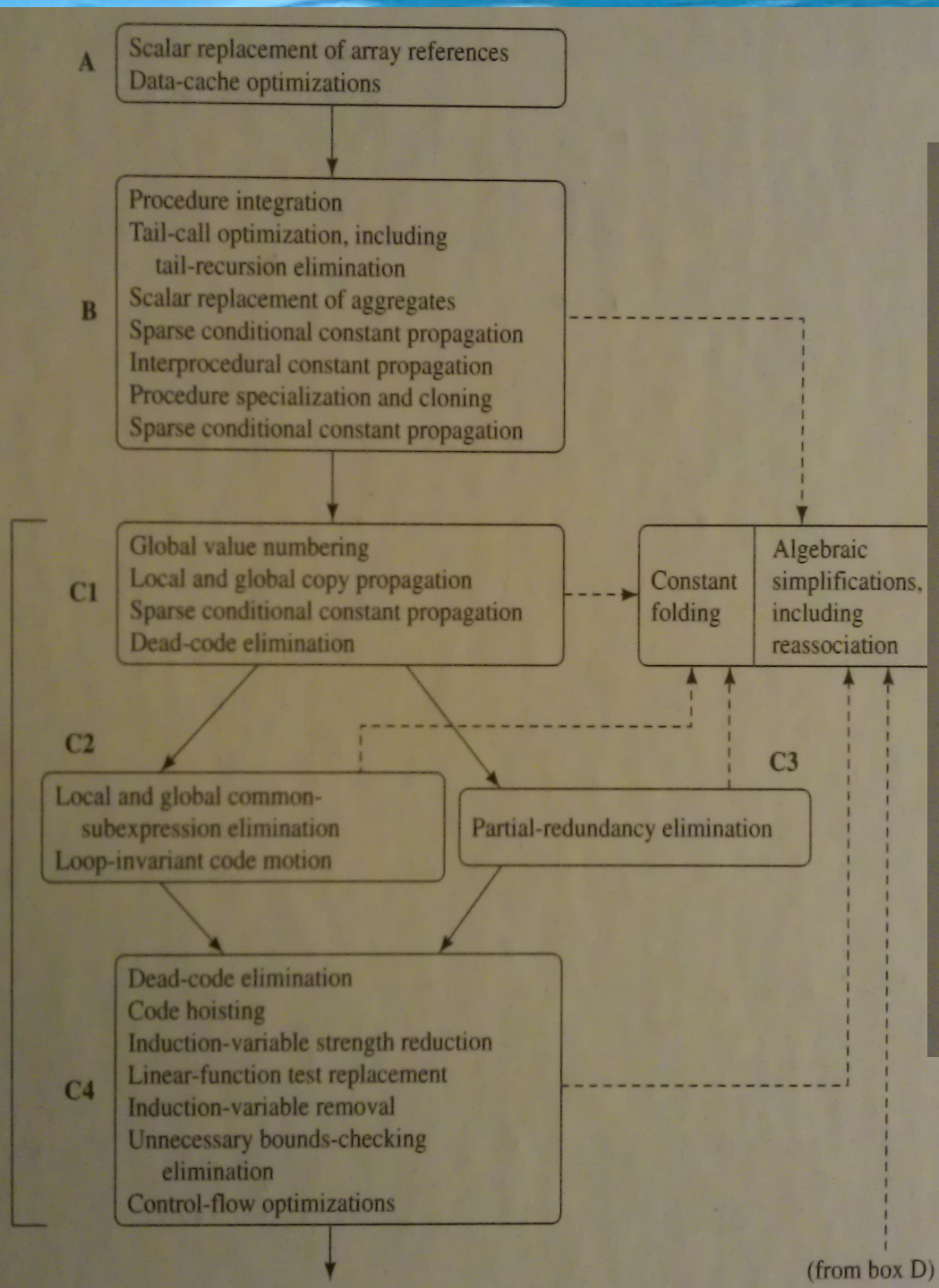
Другие оптимизации

Другие оптимизации

- Data transformation
- Векторизация циклов
- Распараллеливание циклов
- Конвейеризация циклов
- Оптимальное распределение регистров
- Basic block reordering

Возможная последовательность оптимизаций

(согласно Steven Muchnick. Advanced
compiler design & implementation)



Машинно-зависимые
оптимизации

Остальные как правило
машинно-независимые