C++ Russia 2023

elbrus

Компилятор LCC и оптимизация слияния кода

Виктор Шампаров
МЦСТ

viktor.shamparov@yandex.ru

vshamparov

Био

- Разработчик в МЦСТ
  - Универсальные оптимизации в составе компилятора
  - Пара видов профилирования

Структура

- Что такое Эльбрус?
- Компилятор LCC и основные оптимизации
- Слияние кода

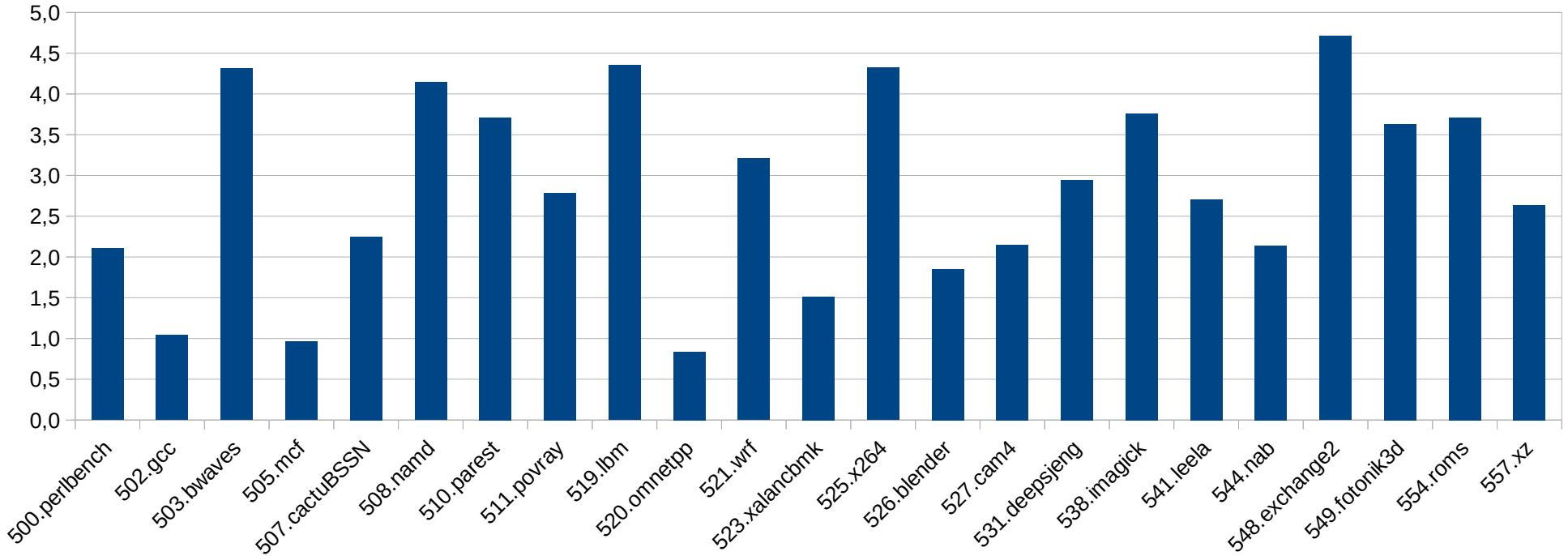
Что это такое?

- Процессор общего назначения
- Архитектура типа VLIW
- Разработан в России

Эльбрус-16С

- 2 ГГц
- 16 ядер

Что это такое? IPC



IPC на бенчмарке SPEC CPU2017r (пиковый режим)

Что такое Эльбрус? Особенности

VLIW = very long instruction word:

- Явный параллелизм инструкций

Предикаты:

- Условное и явно спекулятивное исполнение большинства инструкций

3 стека для безопасности:

- Пользовательский
- Адресов возврата
- Регистровый

Память и регистры с битами тегов

Что такое Эльбрус? VLIW


```
{    ldgdw,0,sm [ %r1 + 0xc ], %r1
     cmplesb,1 %r4, 0x3, %pred0
     adds,2,sm 0x0, _f32s,_lts1 0xf5058, %r9 ? ~%pred1
     merges,3,sm 0x3, 0x2, %r15, %pred2
     ct %ctpr3
     pass %pred0, @p0
     pass %pred2, @p1
     landp ~@p0, @p1, @p5
     pass @p5, %pred3
}
```

- В широкой команде явное указание исполнительных устройств для каждого слога
- Широкая команда выполняется как единая сложная

Что такое Эльбрус? Предикатный режим


```
{    ldgdw,0,sm [ %r1 + 0xc ], %r1
     cmplesb,1 %r4, 0x3, %pred0
     adds,2,sm 0x0, _f32s,_lts1 0xf5058, %r9 ? ~%pred1
     merges,3,sm 0x3, 0x2, %r15, %pred2
     ct %ctpr3
     pass %pred0, @p0
     pass %pred2, @p1
     landp ~@p0, @p1, @p5
     pass @p5, %pred3
}
```

- 32 предикатных (булевых) регистра
- Команда под предикатом выполняется, если в регистре true
  - Есть возможность использовать отрицание предиката
- Есть специальные слоги с предикатными операциями

Что такое Эльбрус? Переходы

Вместо предсказателя — подготовленные переходы

```
{
    nop 4                           # пропускаем 4 такта после ШК
    cmplesb,1 %r0, 0x0, %pred0
    disp %ctpr1, .L5;               # подготовка перехода
}
{
    ct %ctpr1 ? %pred0              # переход
}
```

Компилятор LCC

- C/C++
- Поддерживает архитектуры «Эльбрус» и SPARC
- Поддерживает C++17
  - Экспериментальная поддержка C++20, C++23
- Собственный оптимизатор, фронтенд EDG
  - Экспериментальная поддержка фронтендов LLVM

Компилятор LCC

Ускорение выполнения SPEC CPU2017r в зависимости от режима оптимизации:

- `-O0`
- `-O3`
- `peak` = `-O3/O4` + профиль + ручной подбор опций

	`-O3` vs `-O0`	`peak` vs `-O3`
Среднее геометрическое	x 7,52	x 1,89
Медиана	x 8,46	x 1,77
Min	x 2,04	x 1,07
Max	x 50,70	x 4,35

Оптимизации

Из-за специфики архитектуры оптимизации приобретают особую роль

Особо важны:

- Планирование кода
- Inline-подстановка
- Конвейеризация циклов
- Разрыв зависимостей
- Array Prefetch Buffer
- Слияние кода

Оптимизации
Планирование кода

Задача: собрать операции в широкие команды в оптимальном порядке

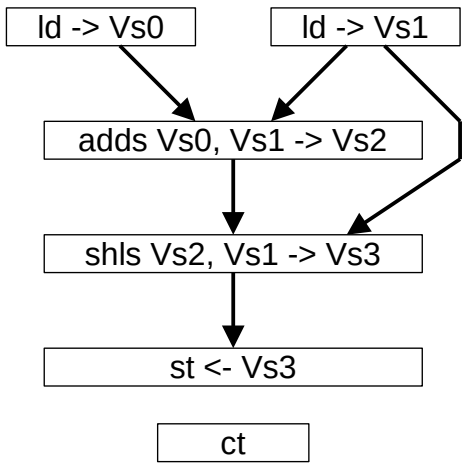
```
14094. CTPD   [6173]              -> C0
13856. CTPD   [14492]             -> C2
12735. MOVs   .s Vs1907           -> Vs1656
    7. MOVs      0x0              -> Vs1
 1435. CMPEs  .s Vs2     0x0      -> P28
 1599. CMPEs  .s Vs2     0x81     -> P42
14160. CLPLAND P28[F]   P42[T]    -> P70
14159. CLPLAND P28[F]   P42[F]    -> P67
12646. ADDs   .s Vs2211 Vs1907    -> Vs2180
12647. SUBs   .s Vs2180 0xa0      -> Vs1736
12644. ADDs   .s Vs2211 Vs1907    -> Vs2158
12645. SUBs   .s Vs2158 0xa0      -> Vs2070
12655. ADDs   .s Vs2197 Vs1907    -> Vs2138
12657. ADDs   .s Vs2189 Vs1907    -> Vs2129
12654. ADDs   .s Vs2480 Vs1907    -> Vs2139
14095. BRANCH   C0                ? P67 [T]
13857. BRANCH   C2                ? P70 [T]
```

```
{  1599. CMPEs .s Rs27 0x81 -> PP0
   1435. CMPEs .s Rs27 0x0  -> PP1
  14094. CTPD  [6173]       -> C0
  15628. ADDs    0x0   0x0  -> Rs0 (7. MOVs)
  12735. MOVs .s Rs28       -> Rs2.m`SP
  12644. ADDs .s Rs29  Rs28 -> Gs16
  12646. ADDs .s Rs29  Rs28 -> Gs17
}
{ 14159. CLPLAND PP1[F] PP0[F] -> PP2
  14160. CLPLAND PP1[F] PP0[T] -> PP0
  13856. CTPD [14492]          -> C2
  12654. ADDs .s Rs10 Rs28     -> Rs9
  12657. ADDs .s Rs11 Rs28     -> Rs12
  12655. ADDs .s Rs26 Rs28     -> Rs15
  12645. SUBs .s Gs16 0xa0     -> Rs16
  12647. SUBs .s Gs17 0xa0     -> Rs4
}
{ 14095. BRANCH  C0            ? PP2 [T]
}
{ 13857. BRANCH  C2            ? PP0 [T]
}
```

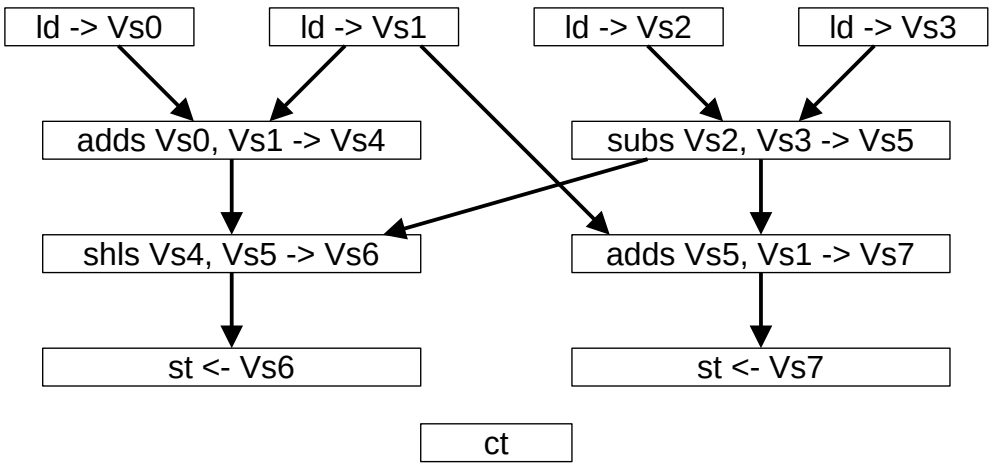
Пример работы планирования и распределения регистров

Оптимизации
Планирование кода

Когда получается хорошо спланировать код?



Пример графа зависимостей с плохим планированием



Пример графа зависимостей с хорошим планированием

Оптимизации
Планирование кода

elbrus

Когда получается хорошо спланировать код?

```
{    ld ... -> Vs0
     ld ... -> Vs1
}
{    adds Vs0, Vs1 -> Vs2
}
{    shls Vs2, Vs1 -> Vs3
}
{    st ... <- Vs3
     ct ...
} 6 ops / 4 ticks
```

```
{    ld ... -> Vs0
     ld ... -> Vs1
     ld ... -> Vs2
     ld ... -> Vs3
}
{    adds Vs0, Vs1 -> Vs4
     subs Vs2, Vs3 -> Vs5
}
{    shls Vs4, Vs5 -> Vs6
     adds Vs5, Vs1 -> Vs7
}
{    st ... <- Vs6
     st ... <- Vs7
     ct ...
} 11 ops / 4 ticks
```

Соответствующие примеры кода после планирования

Оптимизации
Планирование кода

Когда получается хорошо спланировать код?

Плохой код = узкий граф зависимостей

Обычно в маленьких узлах CFG

Хороший код = широкий граф зависимостей

Вероятность такого выше в длинных узлах CFG

Оптимизации
Inline

Подстановка кода вызываемой функции в вызывающую

Достоинства:

- Упрощает анализ
- Улучшает возможность склеить код в длинные узлы
- Убирает операции call/return

Недостатки:

- Уменьшает hitrate кэша инструкций
- Для хорошей работы рекомендуется `-flto/fwhole`

Оптимизации Inline

Ускорение от использования inline

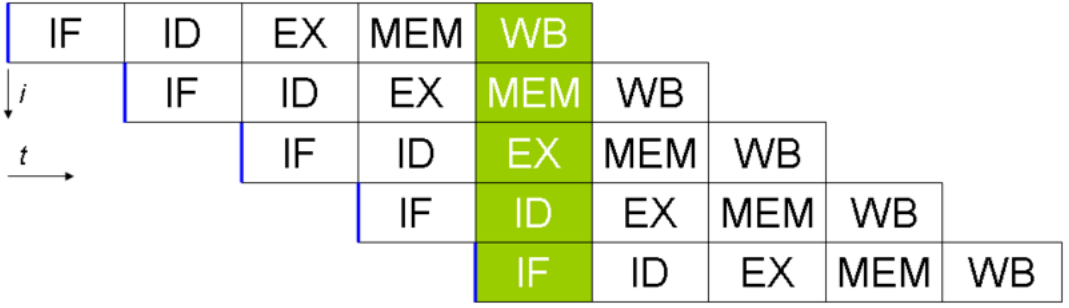
(SPEC CPU 2017r, -O4 + fwhole)

Среднее геометрическое	+49%
Медиана	+14%
Min	без изменения
Max	x 7,0 раз

Оптимизации
Конвейеризация циклов

Разделение работы итерации цикла на стадии и работа подобно конвейеру: следующая итерация запускается, не дожидаясь окончания предыдущей

Таким образом устраняется узость графа зависимостей



Пример конвейера процессора как аналогии

Источник изображения: commons.wikimedia.org

Оптимизации
Конвейеризация циклов

Пример цикла

```
for ( i = 0, j = 0;
      i < n;
      i++, j++ )
    d[j] = a[j] * b[j] + c;
```

В псевдо-IR

```
L_BODY:
 1. add j, 0 -> r6
 2. ld a + j -> r1
 3. ld b + j -> r2
 4. add j, 1 -> j
 5. mul r1, r2 -> r3
 6. add r3, c -> r4
 7. st d + r6 <- r4
 8. add i, 1 -> i
 9. cmpless i, n -> p1
10. ct L_BODY ? p1
```

Расположение (без конвейеризации) по устройствам некоторого VLIW-процессора

Такт	ld/st1	ld/st2	ar/cmp/jmp1	ar/cmp/jmp2
0	2. ld ...	3. ld ...	8. add ...	1. add ...
1			9. cmpless ...	4. add ...
2				
3			5. mul ...	
4				
5				
6			6. add ...	
7	7. st ...		10. ct ...	

IPC: 10 ops / 8 ticks = 1.25

Оптимизации
Конвейеризация циклов

Перенесли часть операций на итерацию раньше

```
L_BODY:
 0. add j, 0 -> jcopy
 1. add r3p, c -> r4
 2. ld a + jp -> r1p
 3. ld b + jp -> r2p
 4. mul r1p, r2p -> r3p
 5. st d + jcopy <- r4
 6. add jp, 1 -> jp
 7. add j, 1 -> j
 8. add i, 1 -> i
 9. cmpless i, n -> p1
10. ct L_BODY ? p1
```

Расположение (после конвейеризации) по устройствам некоторого VLIW-процессора

Такт	ld/st1	ld/st2	ar/cmp/jmp1	ar/cmp/jmp2
0	2. ld ...	3. ld ...	8. add ...	6. add ...
1			0. add ...	9. cmpless ...
2			7. add ...	1. add ...
3	5. st ...		10. ct ...	4. mul ...

IPC: 11 ops / 4 ticks = 2.75

Оптимизации
Конвейеризация циклов

Ограничения:

- Тело цикла - один гиперузел
- Не должно быть вызовов функций
  - Поэтому inline часто помогает

Пример кода, к которому не применяется конвейеризация

```
for ( i = 0; i < n; i++ )
{
    if ( a[i] == 0 )
        b[i] = rand() % 128;
    else
        b[i] = a[i] << 7;
}
```

Оптимизации
Разрыв зависимостей

Пример зависимости по памяти:

```
ld r1 + r2 -> r3
st r4 + r5 <- r0
```

Равны ли `r1 + r2` и `r4 + r5`?

Если да, то нельзя переставить операции

Зачем?

Зависимости мешают оптимальному планированию

Что можно сделать?

- Статически доказать независимость указателей
- Динамически проверять

Оптимизации
Разрыв зависимостей

Статический разрыв

- Анализ указателей
- Можно учесть типы и strict-aliasing
- Пользователь может дать подсказки — признак restrict или опцию -frestrict-all

Динамический разрыв — в коде делаем проверку:

```
*p0 = a;
b = *p1;
```

```
_b0 = *p1;
*p0 = a;
b = (p1 == p0)? a : _b0;
```

Оптимизации Array Prefetch Buffer

Буфер предподкачки массивов:

- В коде — специальные инструкции загрузки
- В специальных секциях — асинхронная программа для буфера

Пример инструкции подкачки

```
movad,0 area = 2, ind = 8, am = 0, be = 0, %db[3]
```

Пример асинхронной программы

```
M_1cc90:
    apb ct=1, si=0, dcd=0, fmt=3, mrng=4, d=1,
        incr=1, ind=0, asz=5, abs=0, disp=0x0
    apb si=0, dcd=0, fmt=3, mrng=4, d=1,
        incr=1, ind=0, asz=5, abs=0, disp=0xa0
```

Оптимизации
Array Prefetch Buffer

Ускорение от использования APB

(SPEC CPU 2017r, -O4 + fwhole)

Среднее геометрическое	+22%
Медиана	без изменения
Min	-2%
Max	x 6,0 раз

Слияние кода
Что происходит без него


```
L0:
    cmpless r0, r1 -> p0
    ct L_NO ? p0
L_YES:
    ld   r2, r0 -> r3
    shls r0, r3 -> r4
    ct L2
L_NO:
    ld   r5, r1 -> r6
    adds r1, r6 -> r4
L2:
    ld   r2, r0 -> r7
    adds r7, r4 -> r6
    st   r2, r0 <- r6
```

Пример кода без слияния

Проблемы этого кода:

- В любом случае есть ct
- Все узлы плохо спланируются

Оценочное время выполнения: 13 тактов

Слияние кода
Предикатный режим

Пример выполнения операций в предикатном режиме

```
cmpless r0, r1 -> p0

ld    r2, r0 -> r3 ? p0
shls  r1, r5 -> r4 ? p0
```

- 32 предикатных (булевых) регистра
- Команда под предикатом выполняется, если в регистре true
- Есть специальные слоги с предикатными операциями

Слияние кода

Пример слитого кода

```
L0:
    cmpless r0, r1 -> p0
    ld   r2, r0 -> r7
    ld   r2, r0 -> r3 ? p0
    shls r0, r3 -> r4 ? p0
    ld   r5, r1 -> r6 ? ~p0
    adds r1, r6 -> r4 ? ~p0
    adds r7, r4 -> r6
    st   r2, r0 <- r6
```

Оценочное время выполнения: 8 тактов

Алгоритм:

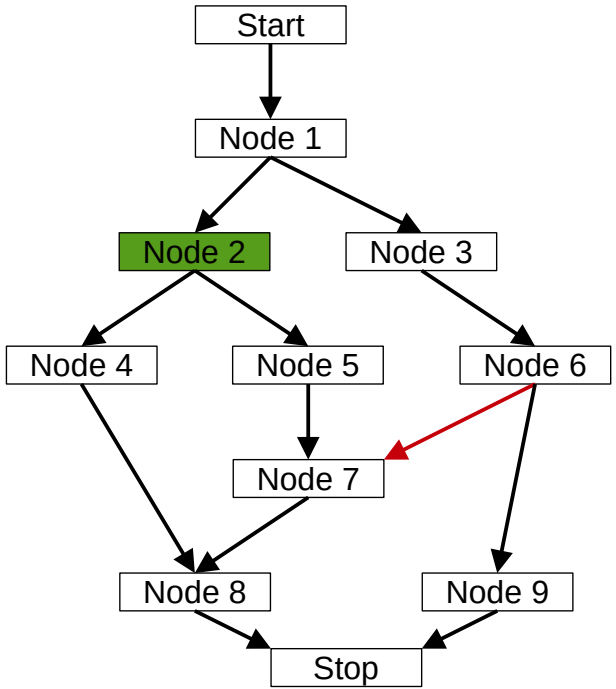
- Собираем регион
- Оптимизируем условия
- Сливаем код
- Выносим вверх некоторые операции

Плюсы:

- Улучшилось планирование

Слияние кода
Выбор регионов

Пример CFG



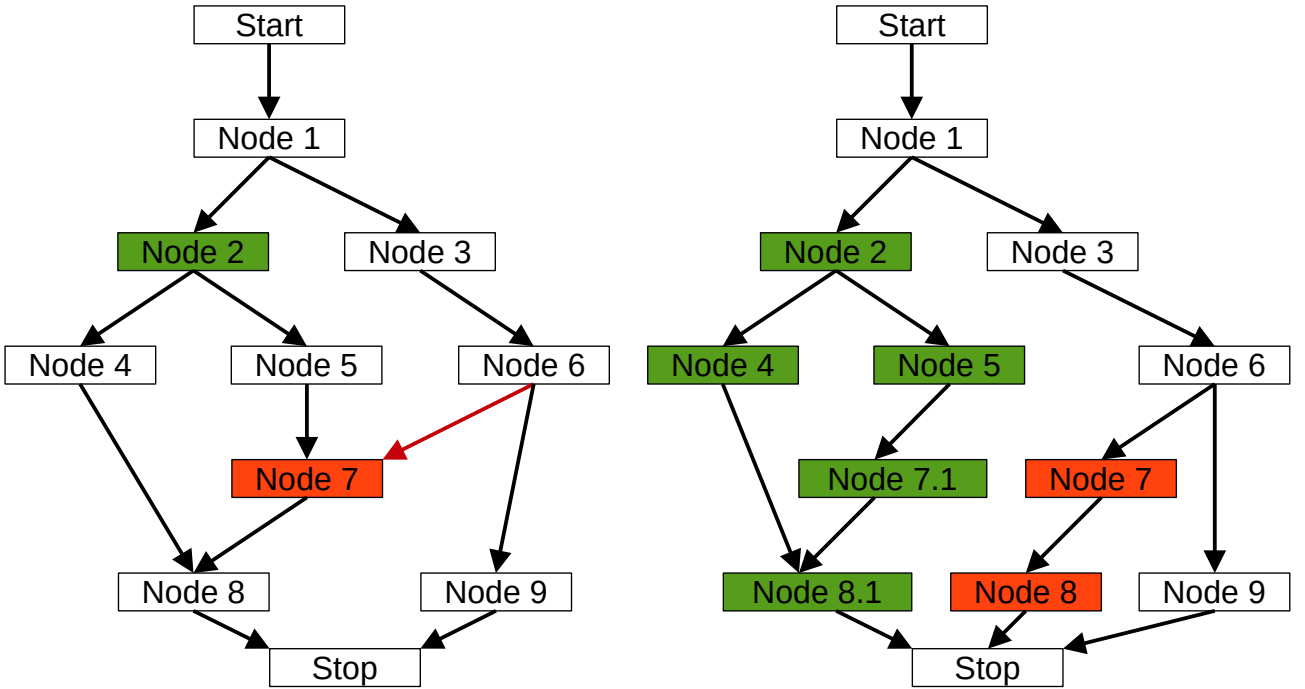
Требования к региону:

- Есть голова региона — все входы в регион только через неё
- Исключаем маловероятные дуги и узлы из региона

Слияние кода
Дублирование узлов

Для узла, в который есть вход извне — можно в регион добавить его копию без внешних входов

Пример дублирования



Слияние кода
Вынос операций вверх

Используется спекулятивный режим:

- Откладывает исключения, вырабатываемые операцией, до момента использования результата
- Но часть операций нельзя исполнять спекулятивно

Пример выноса вверх со спекулятивностью

```
L0:
    cmpless r0, r1 -> p0
    ld      r2, r0 -> r7
    ld .s   r2, r0 -> r3
    ld .s   r5, r1 -> r6
    shls    r0, r3 -> r4 ? p0
    adds    r1, r6 -> r4 ? ~p0
    adds    r7, r4 -> r6
    st      r2, r0 <- r6
```

Оценочное время выполнения: 6 тактов

Слияние кода
Направления выбора регионов

Часть слитых операций под предикатами занимает место, не исполняясь

Следствия:

- Направление выбора регионов — больше по вертикали, чем по горизонтали
- Важен профиль для исключения маловероятных узлов в регионе

Слияние кода
Представление после

На предикатном представлении:

- Обычный граф зависимостей уже слишком консервативен
- Для аккуратного определения зависимых операций нужны специальные техники
  - Это сильно усложняет оптимизации после слияния

Слияние кода
Эффект

Плюсы:

- Лучше планирование
- Включается конвейеризация для циклов со сложным внутренним управлением

Неэффективность:

- Часть операций (условные и спекулятивные) занимают исполнительные устройства, но не обязательно исполняются

Вопросы?