Оптимизация и генерация кода

четверг, 6 января 2022 г. 14:54

Структура компилятора

1. Лексический анализ
2. Синтаксический анализ (парсинг)
3. Сематический анализ
4. Оптимизация
5. Генерация кода

Оптимизация

- Оптимизация - модификация программ с целью улучшения их характеристик без изменения функциональности (логики работы)
  - Производительность
  - Компактность
- Оптимизации
  - Машинно-зависимые
  - Машинно-независимые
  - Локальные
    - Оператор
    - Последовательность операторов
  - Внутрипроцедурные
  - Межпроцедурные
  - Внутримодульные
  - Глобальные

Сворачивание констант (constant folding)

```
Int main (int argc, char **argv) {
    Struct point {
        Int x;
        Int y;
    } p;

    Int a = 32*32;
    Int b = 32 * 32 * 4;
    Long inc c;

    C = (a+b)*(4*4*sizeof(p) - 2 + 32);
    Return 0;
}

Int main (int argc, char ** argv) {
    Strukr point {
        Int x;
        Int y;
    } p;

    Int a = 1024;
    Int b = 4096;

    Long int c;

    C = (a + b) * (16 * sizeof(p) + 30);
    //c = (a + b) * (16 * 8 + 30);
    //c = (a + b) * 158;

    Return 0;
}
```

Распространение констант (constant propagation)

```
Int main(int argc, char **argv) {
    Struct point {
        Int x;
        Int y;
    } p;
    Int a = 1024;
    Int b = 4096

    Long int c;
     c = (a+b)*158;
    //c = 481280;
```

```
    Return 0;
}
```

Распространение копий (cope propagation)

```
Int calc(int x, int y) {
    Int a = x;
    Int b = y;
    Return a * a + b * b;
}

Int calc(int x, int y) {
    //
    //
    Return x * x + y * y;
}
```

Удаление недоступного/недостижимого кода (unreachable code elimination)

```
Void main() {
    Int y = 0;
    Int x;
    Scanf("%d", &x);
    If (x >= 10) {
        Printf("x >= 10\n);
        Return 0;
    }
    Else {
        Printf("x < 10\n);
        Return 0;
    }
    //Printf("x = %d\n", &x);
}

Int sum(int x, int y) {
    Return x + y;
}

    //Int sub(int x, int y) {
        //Return x - y;
    //}

Int main(int argc, char **argv) {
    Return sum(2, 2);
}
```

Удаление мертвого кода (dead code elimination)

```
Void main() {
    Int y = 0;
    Int x;
    Scanf("%d", &x);
    If (x >= 10) {
        //Int y = x * x + 42;
        Printf("x >= 10\n);
        Return 0;
    }
    Else {
        Printf("x < 10\n);
        Return 0;
    }
    //Printf("x = %d\n", &x);
}
```

Устранение общих подвыражений (common sub-expression elimination)

```
Int calc (int x, int y) {
    Int a = (x + y) * (x - y) - x * y;
    Int b = x * (x + y) - y * (x - y);
    Return (a * b + x - y) * (a * b + x + y);
}

Int calc(int x, int y) {
    Int tmp1 = x + y;
    Int tmp2 = x - y;
```

```
    Int a = tmp1 * tmp2 - x * y;
    Int b = x * tmp1 - y * tmp2;
    Int tmp3 = a * b;
    Return (tmp3 + tmp2) * (tmp3 + tmp1);
}
```

Низкоуровневые оптимизации

- Разворачивание циклов
- Векторизация кода
- Подстановка процедур (inline)
- Изменение порядка следования переменных в памяти

Разворачивание циклов

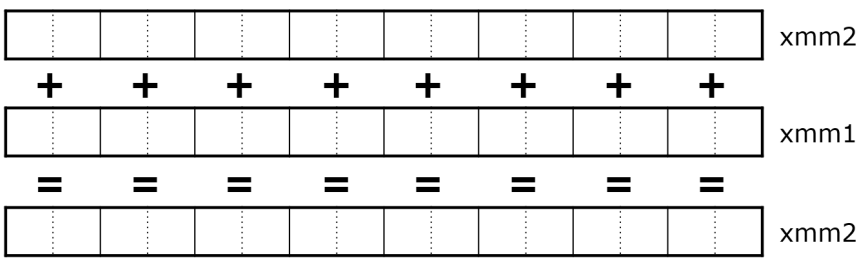
```
    Mov cx, $1000
    Xor dx, dx
    Mov si, array
.CountLoop:
    Lodsw
    Add dx, ax
    Loop .CountLoop

//////////

    Mov cx, $0400
    Xor dx, dx
    Mov si, array
.CountLoop:
    Lodsw
    Add dx, ax
    Lodsw
    Add dx, ax
    Lodsw
    Add dx, ax
    Loop .CountLoop
```

Векторизация кода

- Современные процессоры x86/x64 поддерживают различные наборы инструкций для векторизации:
  - MMX, 3DNow!
  - SSE, SSE2, SSE3, SSSE3, SSE4
  - AVX, AVX2, AVX-512
  - …
- Все 64-битные процессоры поддерживают SSE2
  - 8 регистров по 128 бит каждый (xmm=-xmm7)
  - 5 типов данных для регистров
    - 2 * double
    - 16 * byte
    - 8 * word
    - 4 * Dword
    - 2 * Qword
- Пример инструкции PADDW
  - Paddw xmm2, xmm1



- Intel AVX
  - 8 регистров по 256 бит (ymm0-ymm7)
    - Xmm0-xmm7 - младшие 128 бит
  - Intel AVX-512
    - 32 регистра по 512 бит (zmm0-zmm31)
      - Ymm0-ymm31 - младшие 256 бит
      - Xmm0-xmm31 - младшие 128 бит

- Проблемы

- Отсутствие некоторых инструкций
- Не все задачи поддаются векторизации (распараллеливанию)
- Компилятору необходимо распознать параллельное вычисление
- Нужно время, чтобы поддержка новых инструкций была добавлена в компиляторы

Генерация кода

- Первоначально выполняется только операционная система
- При запуске программы
  - ОС выделяет место в памяти для программы
  - В отведенное место загружается код программы
  - ОС передает управление (jmp/call) на точку входа (entry point) программы

- Компилятор
  - Генерирует код
  - Определяет способ использования памяти данных
- На этапе генерации
  - Выбрать используемые инструкции
  - Спланировать порядок из размещения
  - Распределить данные по регистрам
- Модуль генерации кода сильно зависит от целевой платформы
- Целевая платформа может быть
  - Со стековой, регистровой, аккумуляторной и т.п. архитектурой
  - CISC/RISC/VLIW и т.п
  - …
  - MISP, SuperH и др. имеют так называемые branch delay slots
    - Могут выполняться даже инструкции, расположенные ПОСЛЕ выполненного уловного перехода
  - Intel Itanium отдает управление работой конвейера на откуп компилятору
    - Инструкции объединяются в bundle'ы по 16 байт каждый (по 3 слота)
    - Каждый слот содержит одну инструкцию определенного вида
    - Виды инструкций внутри bundle'а ограничены одним из 32 шаблонов

- Способы организации компиляторов
  - Генерация кода сразу на целевом языке
  - Генерация кода в 2 этапа с использованием промежуточного языка
  - JIT-компиляция

JIT-компиляция

- Генерация кода может происходить по-разному
  - Ahead-of-time (AOT) - преобразование в машинный код происходит до запуска программы
  - Just-in-time (JIT) - преобразование в машинный код происходит во время работы программы
  - Just-in-time compilation
  - Применяется в платформах, использующих байт-код
  - Компиляция программы в них разделяется на два этапа
    - На компьютере разработчика
      - Исходный код --> байт-код
    - На компьютерах пользователей
      - Байт-код --> машинный код
  - Современные JIT-компиляторы
    - Генерируют код на лету
    - Пытаются оптимизировать выполнение программы основываясь на данных о ходе ее выполнения (в том числе перекомпилировать фрагменты уже выполняющейся программы с учетом характера их выполнения)
  - Почему на современных компьютерах это работает очень медленно?