Structure of the client compiler

Alexander Bozhnyuk 371

Изменения по сравнению с предыдущими версиями

- Более агрессивная оптимизация машинного кода
- Раньше: клиентский компилятор - несколько высокоэффективных оптимизаций, серверный - более глобальные оптимизации по границам блоков (Global Value Numbering, Loop Unrolling etc.)
- Раньше HIR не был в Static Single Assignment форме => не подходил для глобальных оптимизаций. Теперь в SSA форме.
- Реализация Linear scan register allocator.

Структура клиентского компилятора

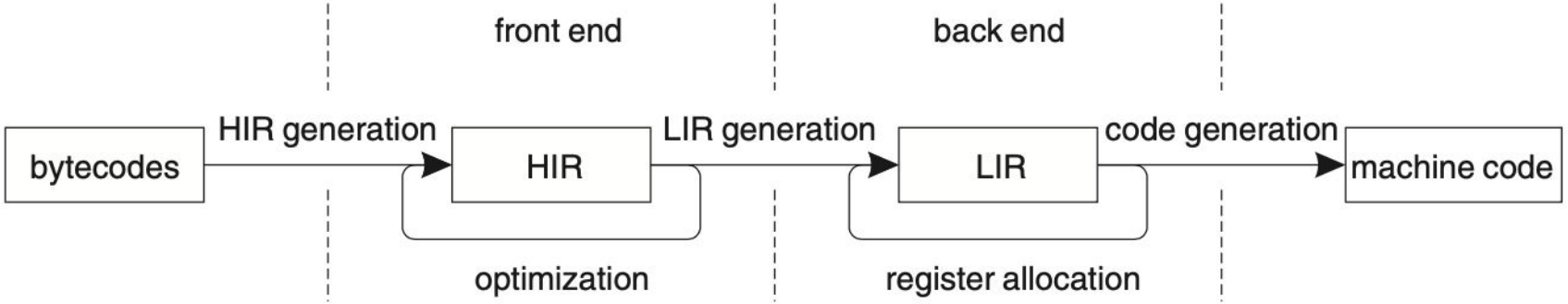

Fig. 2. Structure of the Java HotSpot™ client compiler.

- 3 фазы компиляции, позволяет делать больше оптимизаций
- HIR generation
- LIR generation
- Code generation

High-level Intermediate Representation

- Графическое представление метода в SSA форме.
- Платформо-независим, высокоуровневый, позволяет применять различные оптимизации.
- SSA создается во время синтаксического анализа байт-кодов.
- Можем моделировать управление через граф потока управления (CFG). Узлы - **базовые блоки** (максимально длинные последовательности инструкций без переходов в середине).
- **Узлы инструкций** тоже образуют поток управления: они ссылаются на свои аргументы через указатели на другие инструкции.
- **Инструкция** представляет собой как вычисление результата, так и сам результат
- Блок инструкции может содержать только инструкцию, аргумент может быть даже в другом блоке, но обязательно предшествующем.

High-level Intermediate Representation

Java code fragment:

```
int i = 1;
do {
  i++;
} while (i < f())
```

Bytecodes:

```
10: iconst_1
11: istore_0
12: iinc 0, 1
15: iload_0
16: invokestatic f()
19: if_icmplt 12
```

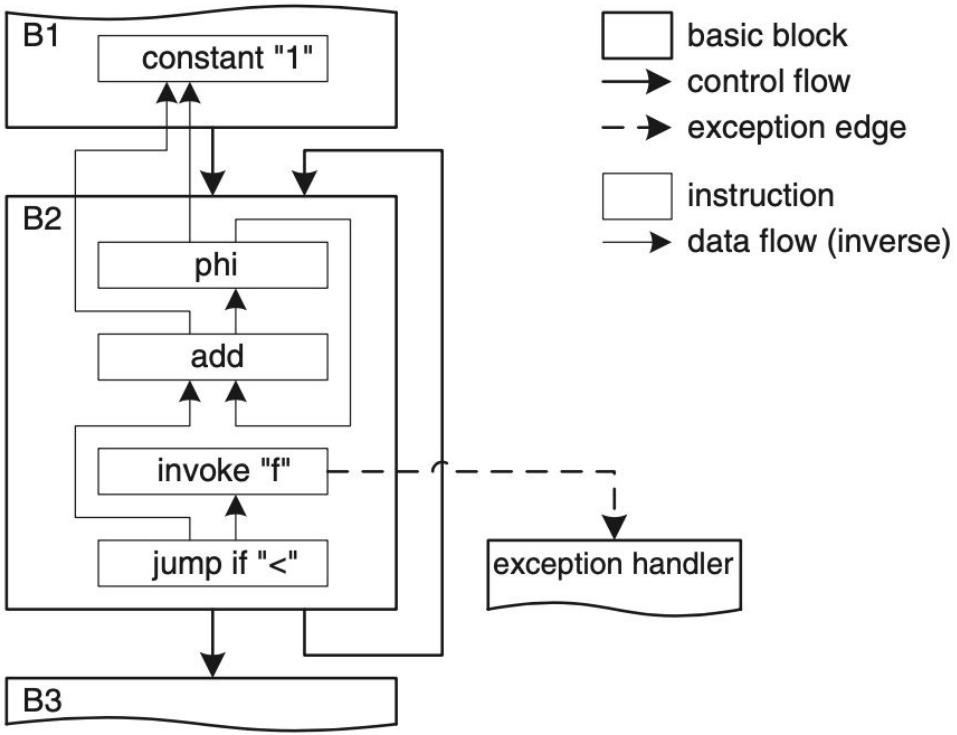

Fig. 3. HIR example with control and data flow.

High-level Intermediate Representation

Построение в 2 прогона по байткоду:

- Первый прогон. Определить границы всех базовых блоков. Произвести анализ циклов.
- Второй прогон. Создать инструкции через абстрактную интерпретацию байт-кодов, определить в нужные блоки, соединить блоки для построения CFG.

Inline методов. Если байт-код содержит вызов короткого метода, который может быть привязан статически, построение HIR вызывается рекурсивно для вызываемого, и необходимые базовые блоки добавляются в граф.

High-level Intermediate Representation: SSA

- SSA форма: одно присваивание для каждой переменной
- Когда у нас объединяются 2 потока управления, используются фи-функции.
- Если у блока 2 и более родителя, то в начале блока может быть нужна фи-функция.
- Фи-функции создаются перед тем, как внести блоки инструкций.
- Если в блоке нет заголовка цикла, то у родителей инструкции заполнены. Смотрим, если результаты переменных разные, создаем фи-функции.
- Если заголовок цикла, то то состояние переменных неизвестно. Для каждой переменной создаем фи-функцию.
- Первый проход делает анализ цикла и позволяет создавать фи-функции более оптимизировано. Также часть лишних фи-функций после построения HIR удаляется.

Оптимизации

- Constant folding
- Local value numbering
- Method inlining (в том числе оптимистичные оптимизации для virtual calls)
- Null-check elimination
- Conditional expression
- Global value numbering

Low-Level Intermediate Representation

- Близок к машинному коду с тремя операндами. Но с добавлением инструкций более высокого уровня, например, для выделения и блокировки объектов.
- Содержит код, зависящий от платформы.
- Использует явные операнды: виртуальные регистры, физические регистры, адреса памяти, слоты стека, константы. Более подходит для низкоуровневых оптимизаций (аллокация регистров)
- Не использует форму SSA, функции фи разрешаются перемещениями регистров.
- После замены виртуальных регистров на физические, для каждой LIR инструкции идет сопоставление с шаблонами из одной или нескольких машинных команд.