Динамические компиляторы в Hotspot

- Клиентский компилятор (`-client`), далее C1
- Серверный компилятор (`-server`), далее C2
- Многоуровневая компиляция (tiered compilation), C1+C2

Общие моменты

- Оба компилятора JITы
- Байткод -> IR(s)-> Машинный код (x86, sparc, ia64, ppc)
- Серьёзно полагаются на интерпретатор для:
  - *Разрешения ссылок на классы*
  - *Инициализации*
  - *Начального исполнения и сбора статистики*
  - *Деоптимизации*
- Переход I2C (возможна асинхронная генерация кода):
  - *Вызовы методов*
  - *OSR (почему?)*
- Переходы C2I:
  - *Добровольная или принудительная деоптимизация (вызов некомпилированного метода, uncommon trap, исключение типа NPE)*

Клиентский компилятор

- Высокая производительность компиляции
- Только простые оптимизации
- Учитывает только количество исполнений метода/цикла (нет частотного анализа веток)
- IR: граф потока исполнения и SSA на линейных участках
- Простой инлайнер – встраивать маленькие методы
- Регистровый аллокатор: linear scan, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.2462&rep=rep1&type=pdf

Клиентский компилятор (2)

- Кодогенератор на основе шаблонов
- Четырёххфазный
  - *Байткод -> HIR (== CFG в SSA форме)*
  - *HIR -> LIR (RISC-подобный с виртуальными регистрами)*
  - *Регистровый аллокатор (linear scan)*
  - *LIR с реальными регистрами -> машинный код*
- Хранит информацию для деоптимизации вместе с кодом метода
- Подробная иифформация: http://www.ssw.uni-linz.ac.at/Research/Papers/Ko08/Ko08.pdf

Традиционные оптимизации

- LVN (идентичная нумерация операций с тождественным эффектом в SSA)
- Constant folding/propagation (вычисление констант при компиляции и использования знания о константах при дальнейшей компиляции)
- Inlining
- Удаление проверок на null
- Удаление мёртвого кода

Серверный компилятор

- Высокая производительность скомпилированного кода при приемлемой скорости компиляции
- Большинство оптимизаций традиционных компиляторов (т.о. многопроходной)
- Оптимизации специфичные для JITa
- Генерация кода для типичных сценариев, иначе откат к интерпретатору
- IR: форма SSA с явными зависимостями по исполнению и данным, без привязки ноды к блоку
- LIR достаточно привязан к целевой архитектуре (из соображений эффективности)
- Регистровый аллокатор: раскраска графа по Чайтину: http://www.cse.ohio-state.edu/~rountev/756/pdf/chaitin81.pdf

Традиционные оптимизации

- Отложенная компиляция (параллельно исполнению)
- GVN (идентичная нумерация операций с тождественным эффектом в SSA)
- Constant folding/propagation (вычисление констант при компиляции и использования знания о константах при дальнейшей компиляции: как ни странно не работает с final)
- Loop unrolling (с учётом знания размеров L1/L2 процессора), unswitching (размыкание)
- Расщепление циклов (peeling)
- Вынос инвариантов цикла (hoisting)
- Учёт алгебраических тождеств и коммутативности
- Соединение смежных доступов

Традиционные оптимизации (2)

- Удаление проверок на null
- Удаление мёртвого кода
- Global Code Motion
- Vectorization (при поддержке целевым процессором)
- Inlining (в частности многоуровневый; сложная часть – блокировки и деоптимизация)
- Range check elimination (доступ к массивам)
- Escape analysis (позволяет многие другие оптимизации, в частности некомпиляторные, такие как выделение объекта на стеке)
- Кодегенератор с матчером (режимы адресации)
- Instruction scheduling
- Финальный peephole

Оптимизации специфичные для JITa

- Оптимистичное удаление проверок на null (использование MMU)
- Устранение полиморфизма, оптимистический inlining виртуальных вызовов
- Различные алгоритмы для мономорфных, полиморфных и мегаморфных вызовов
- Линеаризация fastpath’a
- Замена некоторых методов (например java.lang.System.arraycopy) на оптимизированный вручную ассемблер

Оптимизации специфичные для JITa (2)

- Оптимистичная проверка типов
- Предсказание вероятностей переходов (зачем?)
- Анализ иерархии классов (CHA)
- Уничтожение автобоксинга
- Оптимизация сжатых указателей:
  - *не разжимать для:*
    - проверок на null
    - Load->Store (барьеры!)

Оптимизации блокировок

- Некомпиляторные оптимизации
  - *Быстрые блокировки на атомиках (fastpath)*
  - *Очень быстрые блокировки без атомиков (biased locks)*
  - *Адаптивные блокировки (динамический spinlock/lock)*
- Lock elision (исключение блокировок (EA))
- Lock coarsening/fusion (слияние нескольких последовательных блокировок одного объекта)
- Нетривиальное взаимодействие с исключениями

Многоуровневая компиляция

- Развитие начальной идеи: оптимизировать только то что нужно
- Большая точность: уровень оптимизации пропорционально нарастанию вклада кода в время исполнения программы
- Использовать C1 для генерации профилированного кода
- Требует согласованности интерфейсов между I, C1, C2

Вопросы

- Почему оптимизации зависят от типа IRa ?
- Что должен instruction scheduler знать о процессоре?
- Как loop unrolling может замедлить исполнение кода?
- Почему вызов native методов болезнен для сильнооптимизирующих JITов?
- Почему JITы хороши для больших систем?
- Какие оптимизации можно надеяться что сделает JIT, а какие нельзя?