Реализация JVM: разное

Николай Иготти

Дизайн интерпретатора

- Язык реализации, типичная структура
- Быстрое исполнение, быстрая диспетчеризация
- Перезапись байткодов (прямые ссылки)
- Быстрые и медленные байткоды – расщеплённый интерпретатор
- Динамическая генерация интерпретатора
- Представление стека (кэширование TOS, тэгированный стек)
- Динамическое переключение интерпретаторов

Блокировки

- Статус блокировки хранится в объекте
- Быстрые, лёгкие блокировки (оригинальная работа) на основе CAS с поддержкой рекурсии
- Резервирование блокировок (часто можно и без CASa)
- Адаптивные мьютексы
- Удаление блокировок JITом
- Транзакционная память и блокировки

Нативный интерфейс

- Иногда не обойтись без вызова кода на C (когда?)
- Стандартизовано в JNI (есть другие решения KNI, CNI, RNI)
- Байткод *invokenative*, стандартный протокол именования методов
- Проблема для разработчиков JVM: производительность, стабильность, портируемость
- Другие подходы

Взаимодействие с ОС

- Выделение памяти (страничная гранулярность, большие страницы)
- Создание потоков (системные, пользовательские), приоритеты
- ЦП с раздельными I и D кэшами
- Интерфейс к системным вызовам (JIT может порождать вызовы в ОС)
- Примитивы для доступа к системному времени (syscalls, кэширование, TSC процессора, HPET, ещё?)

Реализация профайлера

- Точный
  - *Что учитывать:*
    - Количество вызовов метода
    - Количество ветвлений назад
    - Частотные характеристики ветвлений
- Сэмплирующий
  - *С какой частотой вызывать*
  - *Что делать с потоком на время анализа (синхронно, асинхронно)*
  - *Как ловить маленькие методы*
- Как выяснять что нагрузка изменилась?

Отладка ВМ

- Внутренние проверки (верификация инвариантов)
- Автоматический поиск deadlock’ов системой исполнения (как?)
- Подключение к ВМ (jstack, jhat, SA)
- Post-mortem анализ падений
  - *Стеки вызовов*
  - *Получение хипа (core, gcore, jhat)*
  - *Анализ с учётом версии ВМ (симуляция структур ВМ поверх данных в дампе)*
  - *Serviceability agent*

Взаимодействие с аппаратными ускорителями

- Проблемы, решения:
  - *Сложность реализации абстрактной машины в кремнии*
  - *Взаимодействие с ВМ (трапы, спец. режим для Java)*
  - *Структуры данных, особенно формат стека*
- picoJava – непосредственное исполнение байткода
- Jazelle – аппаратное ускорение частых байткодов на фазе декодирования для ARM ядер, требует JVM

Заключение по JVM

- Байткод - удачное промежуточное представление
- ВМ – ключевой момент успеха платформы
- Ключевой момент в стабильности и корректности платформы
- Ключевой компонент в безопасности платформы
- Не только Java (см. JSR-292, он же invokedynamic)
- ВМ должна хорошо знать целевую архитектуру для оптимального отображения абстрактной машины в конкретную
- Реализация нетривиальной оптимизирующей логики
- Новые оптимизации ускоряют существующие программы на существующем железе

Вопросы

- Как ускорить диспетчеризацию если мы знаем что байткоды ходят типичными парами, тройками и т.п.?
- Почему в Lisp’е нет блокировок?
- Можно ли использовать performance counters процессора в профиляторе?
- Какие ещё отладочные функции способна выполнять ВМ?
- Какие инструкции стоит добавить в x86 чтобы JVM было писать проще?