Виртуальные машины

Лекция 2: Интерпретация и компиляция

Олег Плисс
Oleg.Pliss@oracle.com

август 2013

Sun microsystems
We make the net work.

Интерпретация

- Ядро VM — интерпретатор кода виртуального процессора
- Интерпретация исходного текста программы
  - Не эффективна из-за многократного повторения лексического и синтаксического анализа
  - Текстовое представление выбирается для облегчения взаимодействия с человеком
- Интерпретация промежуточного кода
  - Промежуточное представление выбирается для облегчения анализа и интерпретации
  - Исходный текст компилируется в бинарный промежуточный код
  - Бинарный промежуточный код интерпретируется
  - Компиляция исходного текста в промежуточный код может быть неявной динамической

Виды интерпретаторов

- Рекурсивный интерпретатор
  - Вызывает себя с параметрами при обработке всех или некоторых инструкций (вызов подпрограммы)
  - Не требует отдельных стеков
  - Обычно применяется для ПЯ, представленных текстом, деревьями или графами
- Итеративный интерпретатор
  - Последовательно в цикле перебирает инструкции программы
  - Параметры и результаты передаются в виртуальных регистрах или на стеке операндов
  - При вызове подпрограммы формирует новую секцию в стеке вызовов, меняет указатель текущей инструкции, продолжает выполнение цикла
  - Применяется для ПЯ, представленных потоками, последовательностями или массивами инструкций

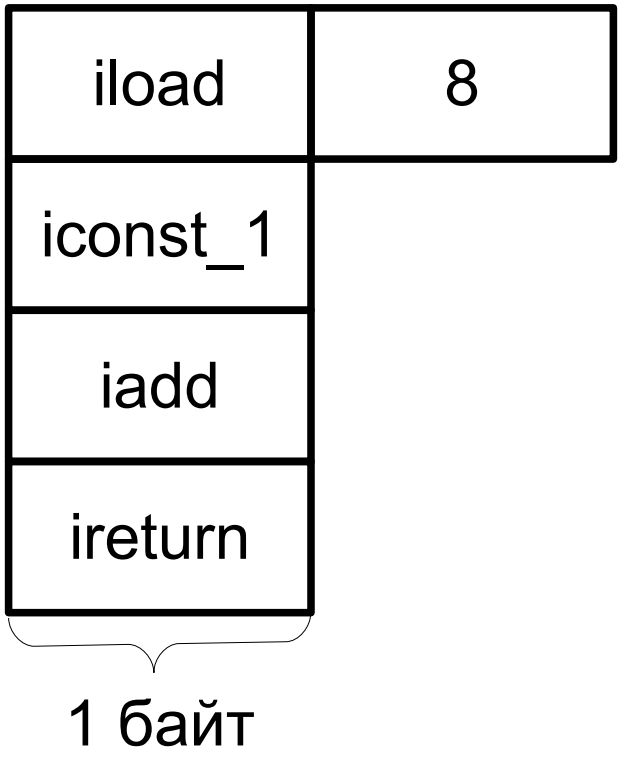
Компоненты итеративного интерпретатора

- Набор инструкций
  - add, load, branch... (может быть расширяемым)
- Виртуальные регистры
  - *ip* — адрес текущей инструкции
  - *sp* — указатель стека
  - *fp* — адрес текущей рамки стека вызовов
- Стеки
  - *Стек вызовов* для организации вызовов подпрограмм, обычно состоит из рамок/секций/записей активации
  - *Стек операндов* для передачи параметров и выдачи результатов виртуальных инструкций (может быть частью стека вызовов или вообще отсутствовать)
- Цикл интерпретатора
  - Включает декодер инструкций и их реализацию

Представление инструкции

- Токен
  - Целое число, однозначно идентифицирующее инструкцию и, возможно, все или некоторые из ее операндов
  - Остальные операнды могут быть закодированы в потоке инструкций вслед за токеном
  - Токены не зависимы от аппаратуры и размещения кода в памяти
  - Пример: байтовые коды Smalltalk VM или JVM
- Адрес подпрограммы
  - Единственная инструкция — вызов подпрограммы, подразумеваемая по умолчанию
  - С другой стороны, расширяемый набор инструкций, представляемых адресами их реализаций
  - Пример: шитый код Forth-машины
  - Bell, J.K., Threaded code, Communications of the ACM vol 16, nr 6 (Jun) 1973, pp.370-372

Пример: Байтовый код JVM



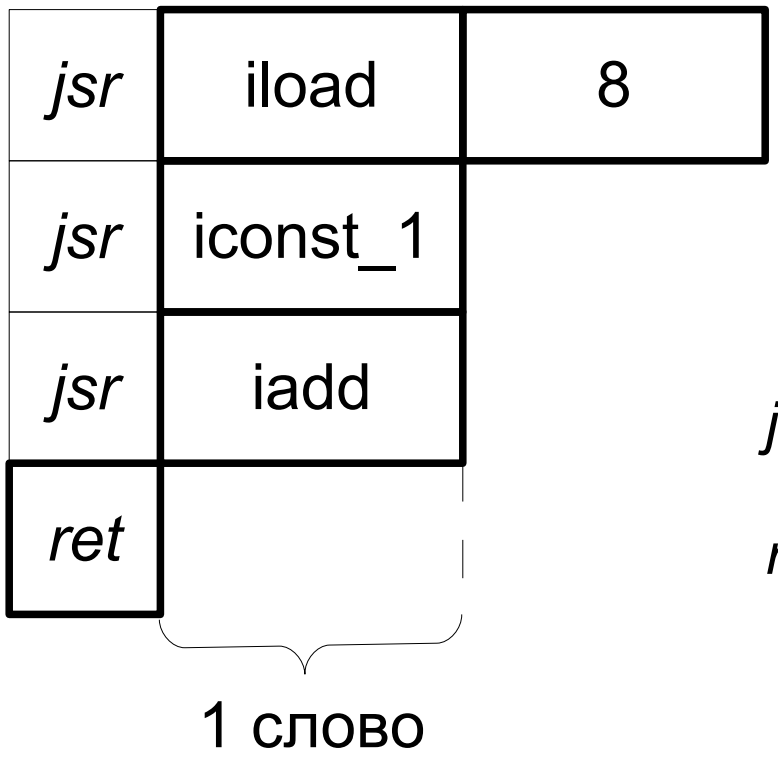
Интерпретатор байтового кода на C

```
void Interpreter (void) {
        const byte* ip;
        …
        for (;;) {
              switch (*ip++) {
                    ...
                    case iadd: {
                          const int a = pop();
                          const int b = pop();
                          push(a+b);
                          break;
                    }
                    ...
              }
        }
}
```

Доступ к дополнительным параметрам

- Простая выборка из потока инструкций при помощи виртуального регистра ip
  - const byte arg = *ip++;
- Доступ к 16- и более битовым операндам сложнее
  - Порядок байтов виртуального и аппаратного процессоров может быть различным
  - Аппаратура может требовать выравнивания на соответствующую границу

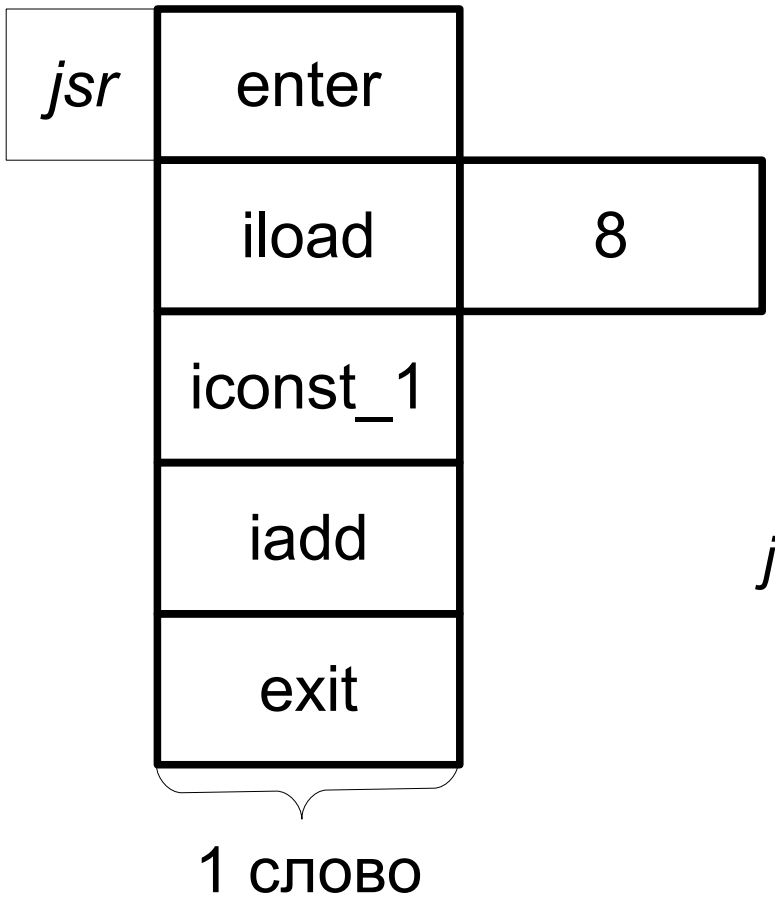
Подпрограммный шитый код



jsr — машинная инструкция вызова подпрограммы
ret — машинная инструкция возврата из подпрограммы

Цикла интерпретатора не требуется.

Прямой шитый код



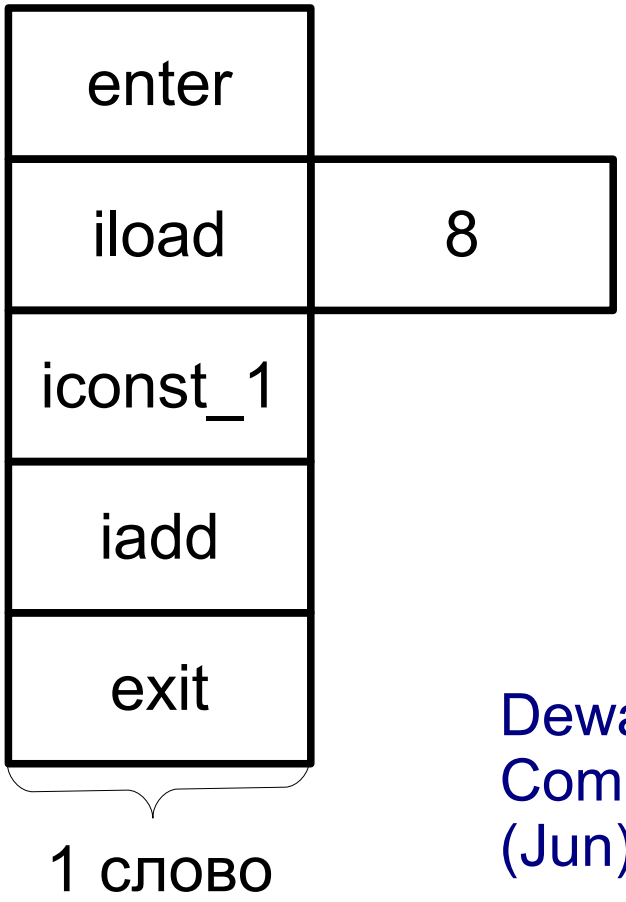

```
enter:  rpush(ip);
        pop(ip);
next:   goto(*ip++);

exit:   rpop(ip);
        goto(*ip++);
```

jsr — машинная инструкция вызова подпрограммы

enter и exit — вход и выход в интерпретатор данного фрагмента кода. Интерпретаторов может быть несколько.

Косвенный шитый код




```
enter:  rpush(ip);
        ip = tmp++;
next:   tmp=*ip++;
        goto(*tmp);

exit:   rpop(ip);
        tmp=*ip++;
        goto(*tmp);
```

Dewar, R.B.K., Indirect threaded code, Communications of the ACM vol 18, nr 6 (Jun) 1975, pp.330-331

Единственная разновидность шитого кода, не содержащая машинных инструкций и потому портируемая.

Передача параметров инструкциям

- Явная через виртуальные регистры
  - Обычно большое число, чтобы свести распределение регистров к их назначению
  - Немного ускоряется интерпретация
  - Удлиняется код
  - Усложняется декодирование инструкций
  - Усложняется компилятор языка в код VM — он должен распределять регистры
  - За счет этого может упроститься распределение регистров в динамическом компиляторе
- Неявная через стек операндов
  - Потенциально неограниченной глубины
  - Упрощает набор виртуальных инструкций, реализацию интерпретатора и компилятора языка в виртуальный код
  - Используется в большинстве современных VM

Пример: Байтовый код JVM

- Стек-ориентированный набор из ≈200 инструкций
- Аналогичен коду Smalltalk VM
  - В Java примитивные типы данных не являются объектами
- Четыре виртуальных регистра
  - ip, sp, fp, lp
- По одному стеку вызовов для каждого потока (thread)
- Стек операндов — часть секции активации
  - Компилятор в виртуальный код должен вычислить требуемую глубину стека операндов
  - Верификатор проверяет достаточность глубины стека операндов

Пример: Байтовый код JVM (2)

- Большинство инструкций однобайтовые
  - Операнды на вершине стека операндов
- Некоторые инструкции извлекают дополнительные 8- и 16-битовые операнды из потока инструкций
- Три инструкции переменной длины
  - Длина инструкции определяется операндами
  - lookupswitch, tableswitch, wide
- Несколько кодов зарезервировано для внутреннего использования VM

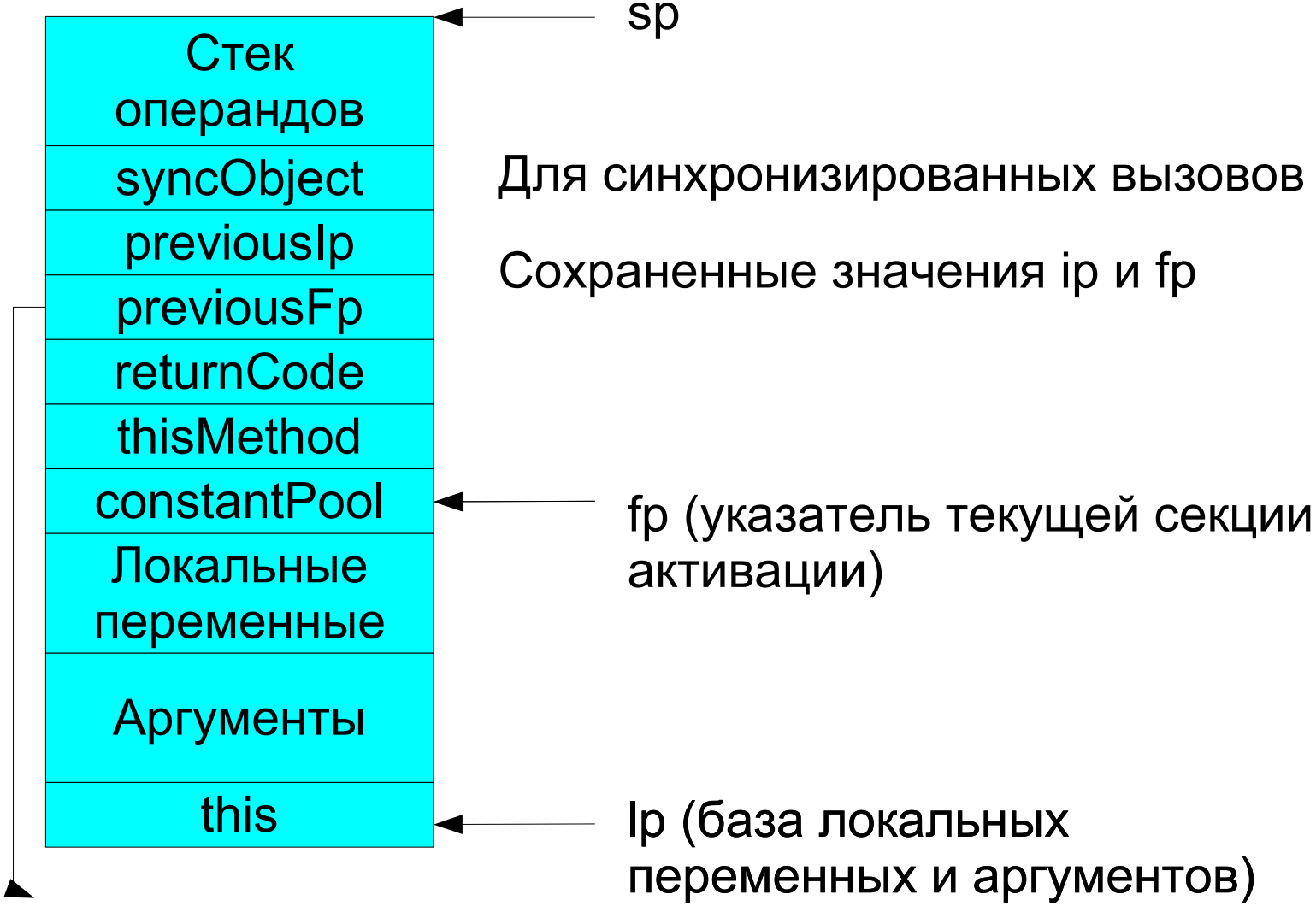
Пример: Байтовый код Self

- Всего 8 инструкций:
  - SELF
  - LITERAL
  - NON-LOCAL RETURN
  - DIRECTEE
  - SEND
  - IMPLICIT SELF SEND
  - RESEND
  - INDEX-EXTENSION
- 3 бита — код операции, остальные 5 — параметр
- Большинство примитивов вызывается с помощью обычной посылки сообщений

Стек вызовов и записи активации

- Smalltalk и Self: стека вызовов нет
  - Дисциплина вызовов может отличаться от LIFO
  - Записи активации — обычные объекты, отводятся в «куче»
- Forth: стек вызовов есть, записей активации нет
  - Запись активации редуцирована до адреса возрата
- Java: свой стек вызовов у каждой нити
  - Запись активации создается при каждом вызове и сбрасывается при каждом возврате
  - Запись активации содержит:
    - аргументы вызова
    - место для локальных переменных
    - место для стека операндов
    - служебные данные

Пример: Запись активации KVM



Организация стека вызовов

- Стек конечной глубины может переполниться
- Ограничить глубину стека достаточно большим числом и зарезервировать по максимуму
  - Неэффективное использование памяти
  - При наличии страничной адресации можно зарезервировать адресное пространство, а память отводить по мере необходимости
- Стек переменной длины
  - При переполнении увеличивать размер и копировать
  - При нехватке памяти сокращать размер
- Кусочный стек
  - При переполнении отводить новый фрагмент
  - Возможно патологическое поведение вблизи границы фрагмента

Обработка исключений в Java

```
try {
    ... Сделать что-нибудь ...
} catch (IOException e1) {
    ... Обработка ошибок ввода-вывода ...
} catch (MyException e2) {
    ... Обработка определенных пользователем
        исключений ...
}
```

- При компиляции метода создается таблица диапазонов адресов содержащихся в нем *try*-блоков и соответствующих обработчиков

Обработка исключений в Java (2)

- При возникновении исключения секции активации последовательно сканируются сверху донизу
- Если метод текущей секции содержит try-блок вокруг заданного адреса с обработчиком подходящего типа исключений, все верхние секции сбрасываются и вызывается этот обработчик
- Иначе процесс продолжается для предыдущей секции и соответствующего адреса вызова
- При сбрасывании синхронизированных секций снимаются замки синхронизации

Производительность интерпретатора

- По разным исследованиям интерпретатор в 2.5-50 раз медленнее соответствующего нативного кода, порожденного оптимизирующим компилятором
- Замедление определяется в основном реализацией VM, а не приложением
- Накладные расходы интерпретатора
  - Декодирование виртуальных инструкций
  - Доступ к виртуальным регистрам
  - Вызов примитивов
  - Лишние проверки при выполнении инструкций

Ускорение интерпретации

- Шитый код часто быстрее байтового
  - На 20-30%
- Переписать цикл и критические инструкции на ассемблере
  - Ускорение в 2-4 раза
  - Усложнение
  - Ухудшение портируемости
- Отобразить виртуальные регистры в физические
  - На ассемблере или при помощи специальных директив компилятора
  - Ухудшение портируемости

Ускорение интерпретации (2)

- Кэширование 1-2 верхних элементов стека операндов в физических регистрах
  - Без кэширования

```
pop(tmp1); pop(tmp2);
tmp1 += tmp2;
push(tmp1);
```

  - Кэширование верхнего элемента в регистре top экономит 2 обращения к памяти

```
pop(tmp1);
top += tmp1;
```

  - Кэширование двух верхних элементов

```
top1 += top2;
pop(top2);
```

  - Число регистровых пересылок увеличивается
  - Утяжеление вызовов за счет сохранения и восстановления дополнительных регистров

Кэширование вершины стека

- Отдельные циклы интерпретатора для 0-2 кэшированных элементов с взаимными переходами
  - add0:

```
pop(top1); pop(tmp);
top1 += tmp;
goto Loop1;
```

  - add1:

```
pop(tmp);
top1 += tmp;
goto Loop1;
```

  - add2:

```
top1 += top2;
goto Loop1;
```

  - Увеличение размера кода интерпретатора

Ускорение интерпретации (3)

- Введение упрощенных инструкций для исключения избыточных проверок (quickening)
  - `fast_putfield` не проверяет объект на null
  - Если из контекста можно заключить, что проверка не нужна, заменяем в коде исходную инструкцию упрощенной
- Супер-инструкции
  - Объединение и оптимизированная реализация часто используемых последовательностей инструкций
- Вынесение редко используемых сложных инструкций в отдельный цикл
  - При реализации на языке высокого уровня улучшает распределение регистров

Компиляция

- Дальнейшее ускорение интерпретации требует компиляции
  - Ускорение в 10-20 раз по сравнению с интерпретатором на ANSI C
  - Порождаемый код обычно в 4-8 раз длиннее виртуального
  - Увеличение сложности
  - Дальнейшее затруднение портирования — в компиляторе существенно используется знание о целевом процессоре
  - Способы облегчения портирования:
    - Порождение ассемблера и дизассемблера по описанию набора инструкций
    - Порождение кодогенератора по описанию проекций (в стиле BURG)

Когда компилировать?

- До выполнения (статическая компиляция)
  - a.k.a. Ahead-Of-Time, AOT
- Во время выполнения
  - Варианты:
    - При старте приложения (с кэшированием порожденного кода для облегчения последующих стартов)
    - Лениво при первом выполнении программной единицы
    - Адаптивно под управлением профилятора

Статическая компиляция

- Классическая компиляция
  - Или кросс-компиляция (host != target)
  - Хорошо известные алгоритмы
  - Во время исполнения не нужен интерпретатор
- Глубокий анализ кода
  - Требует памяти и времени
  - Слабо применим к динамическим языкам
  - Слабо применим к открытой модели Вселенной
  - Ограничен в использовании статистики исполнения
  - Не может использовать особенности динамического состояния приложения

Динамическая компиляция

- При старте приложения
  - Откладывание классических компиляции и линковки до старта приложения
  - Выполняется на целевом устройстве
  - \- Target обычно более ограничен в ресурсах, чем Host — нет ни времени, ни места компилировать весь код
  - \- Время первого старта существенно для клиентских приложений
- При первом выполнении программной единицы
  - Способ распределения компиляций во времени и исключения не используемых программных единиц
  - \+ Можно использовать динамическое состояние
  - \- Компиляции распределяются неравномерно
  - \- Многие инициализаторы (например, классов в Java) выполняются лишь однажды

Динамическая адаптивная компиляция

- Компилировать те единицы, которые будут использоваться в ближайшем будущем
  - Динамическое профилирование
  - Предсказание активности в ближайшем будущем по активности в ближайшем прошлом
- Для перехода от интерпретации к исполнению кода переписать секцию активации
  - On-Stack Replacement, OSR
- Для перехода от исполнения к интерпретации выполнить обратную замену
  - Переход возможен только в подготовленных во время компиляции точках

Динамическая адаптивная компиляция (2)

- Частичная компиляция
  - Исходный интерпретируемый код сохраняется
  - Поддерживается переход от исполнения скомпилированного кода к интерпретации исходного
  - Значит, можно компилировать частично, заменяя отсутствующие фрагменты на переходы к интерпретации
  - Можно исключать сложные для компиляции конструкции и редко исполняемые фрагменты
  - Можно по мере необходимости наращивать набор компилируемых конструкций
- Спекулятивная специализация кода
  - Специализация — порождение более оптимального кода для заданного частного случая (заданных условий)
  - Спекулятивная специализация — заданные при компиляции условия могут в будущем оказаться неверными

Динамическое профилирование

- Не должно слишком замедлять интерпретатор
- Counter-based
  - Каждая программная единица снабжается счетчиком вызовов, а каждый переход внутри нее счетчиком выполнений
  - В интерпретатор добавляется код для инкремента счетчиков
  - При угрозе переполнения счетчики масштабируются
  - Получаем динамический профиль с точностью до относительных частот выполнения базовых блоков
  - Много счетчиков, накладные расходы сравнительно велики
  - Пример: профилирование в HotSpot SE

Time-based profiling

- Выявление активности кода в интервалах выбранной временной шкалы
- Сэмплинг
  - Сканирование верхних секций стека вызовов для определения текущего адреса исполнения
  - Эффективен для циклов
  - Пропускает вызовы коротких программных единиц
  - Результаты зависят от смещения момента старта приложения внутри временного интервала
- Инструментирование
  - Дополнительный код в интерпретаторе
  - Не эффективен для циклов
  - Достаточно эффективен для вызовов

Пример: профилятор Monty VM

- Временная шкала с управляемым шагом
  - Базовая частота 1 ms
- Комбинация сэмплинга и инструментирования
  - Сэмплинг верхней секции стека для обнаружения цикла в текущем методе
  - Интерпретатор записывает вызовы в короткий циклический буфер
  - Буфер сканируется и сбрасывается в момент тика таймера временной шкалы

Эвристическое предсказание активности

- Если программная единица была активна в ближайшем прошлом, то она будет активна и в ближайшем будущем
  - Большинство программ проводит большую часть времени в небольшом числе циклов
  - Действует в широком диапазоне временных шкал
  - Временная шкала чувствительна к изменению приоритета процесса и ожиданию событий
  - Не действует вблизи завершения циклов с большим, но фиксированным числом итераций и при фазовых переходах
  - Парадокс 1: поздняя компиляция пустого цикла с большим числом повторений замедляет его выполнение
  - Парадокс 2: пустой цикл с фиксированным числом итераций может исполняться медленнее аналогичного цикла с непустым телом

Что компилировать?

- Программные единицы
  - С учетом инлайновой подстановки некоторых вызываемых программных единиц
- Трассы исполнения
  - Линейная последовательность инструкций, заканчивающаяся переходом внутрь этой последовательности
  - В точках вызова последовательность может пересекать границы программных единиц
  - Обычно не исполняемые выходы из трассы защищаются переходами в режим интерпретации
  - Трассы с общим префиксом объединяются в деревья
  - A.Gal, C.Probst, and M.Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pp. 144-153, 2006

Как компилировать?

- Остановиться, скомпилировать, продолжить
  - Выполнение текущей нити приостанавливается до завершения компиляции
  - Переход от интерпретации к исполнению кода
  - Возобновление выполнения текущей нити
  - Компиляция больших единиц запрещается, чтобы не вызывать длительных пауз в выполнении приложения
- Concurrent compilation
  - Компилятор как сопрограмма — выполнение нити и компиляция перемежаются с начала компиляции и до ее завершения
  - Планирование компиляций — управление длительностью шагов выполнения компилятора
- Параллельная компиляция
  - Компилятор выполняется отдельной нативной нитью вне зависимости от виртуальных нитей

Пример: планирование компиляций в Monty VM

- Concurrent compilation
  - Из-за ограниченной памяти не более одного компилятора активно в каждый момент времени
- Динамическое ограничение длительности шага компиляции
- Предотвращение кластеризации пауз
  - Каждая отдельная пауза возникает в результате системной активности: сборки мусора, компиляции, загрузки классов, ...
  - Выделение приложению не менее заданной доли в каждом временном интервале
- Подавление компиляций в фазе старта
  - Ускорение старта, фазовый переход
- Подавление компиляций при нехватке памяти

Оптимизации

- Выполняемые динамическим компилятором оптимизации не должны быть слишком ресурсоемкими
  - Длительность компиляций добавляется к длительности выполнения приложения
    - Исключение: параллельная компиляция при избытке аппаратного параллелизма
  - Отводимая компилятором рабочая память временно не доступна приложению
  - Код компилятора добавляется к коду VM
- Многоуровневая (tiered) оптимизация
  - Наборы оптимизаций разбиваются на уровни по длительности
  - Сначала программная единица компилируется с самыми быстрыми оптимизациями
  - Впоследствии уровень оптимизации может быть увеличен

Наиболее эффективные оптимизации

- Constant folding & algebraic simplifications
  - Свертка константных выражений и упрощение выражений
- Constant & copy propagation
  - Протяжка констант и присваиваний, часто только локальная
- Common subexpression elimination
  - Изведение общих подвыражений, часто ограниченное
- Multi-level inlining of method calls
  - Многоуровневая инлайновая подстановка
- Loop & branch optimizations
  - Расцикливание и выворачивание циклов
  - Переходы на переходы, выпрямление кода

Наиболее эффективные оптимизации (2)

- Null check elimination
  - Устранение избыточных проверок на NULL
  - С помощью Reaching definitions DFA
- Dynamic cast elimination
  - Устранение повторных проверок принадлежности к подклассу указанного класса
  - С помощью Reaching definitions DFA
- Range check elimination
  - Изведение лишних проверок выхода индекса за пределы границ массива
  - Сложная оптимизация, обычно реализуется для некоторых часто встречающихся форм циклов
  - Что делать, если не удается доказать, что в теле цикла никакие проверки не нужны? Оставить единственный цикл с проверками? Расщепить цикл?

Девиртуализация вызовов

- Виртуальный вызов называется *мономорфным*, если при исполнении связывается с одним и тем же методом.
- *Во время исполнения* типичного приложения более 90% произведенных виртуальных вызовов являются мономорфными бесконтекстно и 98% в динамическом контексте вызовов глубиной не более 1.
- Нужно *во время компиляции* обнаружить большинство таких виртуальных вызовов и заменить их на прямые (статические) вызовы.

Девиртуализация вызовов (2)

- Вычислить набор классов для аргументов диспетчеризации виртуального вызова
  - Конструкторы и литералы выдают конкретный класс
  - Если язык типизированный, то для всех прочих случаев проще всего отфильтровать классы по типу
  - Если тип — это класс, то набор всех его подклассов
  - Если тип — это интерфейс, то набор всех подклассов классов, реализующих этот интерфейс и его подинтерфейсы
  - Альтернатива — не полагаясь на языковые типы, вывести набор классов при помощи DFA
- Если в полученном наборе классов имеется единственный метод с подходящей для виртуального вызова сигнатурой, заменить этот виртуальный вызов статическим вызовом этого метода

Девиртуализация вызовов (3)

- После девиртуализации часто применяется инлайновая подстановка с последующей оптимизацией тела вызываемого метода
- Редко работает в больших библиотеках классов
  - Даже если приложение использует лишь несколько классов, все доступные классы принимаются во внимание
- Редко работает при открытой системе классов
  - Классы и методы редко объявляются финальными
  - Новый загруженный класс может оказаться подклассом интересующего нас класса или реализовать интересующий нас интерфейс

Сужение типов локальных переменных

- Выведение типа локальной переменной в точке вызова при помощи Reaching Definitions DFA
  - Декларированный тип переменной бывает шире множества классов ее возможных значений в указанной точке
  - NULL выдает пустое множество классов
  - Литерал и конструктор выдают соответствующий класс
  - Аргумент, поле или вызов типа «класс» выдают все подклассы этого класса
  - Аргумент, поле или вызов типа «интерфейс» выдают набор подклассов классов, реализующих этот интерфейс и его подинтерфейсы
  - Переменные преинициализированы пустым набором классов
- Вычисленный тип переменной не может быть шире ее декларированного типа

Спекулятивная девиртуализация

- Применить алгоритм девиртуализации не ко всем классам, а лишь к их инициализированному в момент компиляции подмножеству
- Инициализация нового класса может нарушить условия произведенных девиртуализаций
- Варианты алгоритма отличаются способом обработки такой ситуации
  - Guarded speculative devirtualization
  - Unguarded speculative devirtualization

Защищенная спекулятивная девиртуализация

- Девиртуализованный вызов предваряется проверкой, что виртуальный вызов действительно связывается с использованным методом
- Если нет, перейти в скомпилированный код с виртуальным вызовом или режим интерпретации
  - Скомпилированный код всегда корректен, но, возможно, не оптимален
- Изведение избыточных проверок и их вынесение из циклов
  - Некоторое количество проверок остается

Незащищенная спекулятивная девиртуализация

- В скомпилированном коде отсутствуют какие-либо проверки
- Вместо них скомпилированный метод ассоциируется с предикатом корректности всех произведенных в нем спекулятивных девиртуализаций
- При инициализации класса необходимо проверить предикаты всех скомпилированных методов и при необходимости их деоптимизировать
  - Обычно путем перевода в режим интерпретации

Незащищенная спекулятивная девиртуализация (2)

- Как получить предикат корректности?
  - При анализе не редуцировать подклассы и интерфейсы в наборы классов, а оперировать ими символически
  - Для получателя каждого девиртуализуемого вызова получаем фильтр класса - логическое объединение из конкретных классов, подклассов класса и подклассов классов, реализующих подинтерфейсы интерфейса. Это выражение можно упрощать.
  - Предикат корректности данного вызова — уникальность метода с заданной сигнатурой в наборе классов, вычисляемом данным фильтром классов.
  - Предикат корректности скомпилированного метода — логическое произведение предикатов всех вызовов внутри этого метода.

Когда производить деоптимизацию

- Eager deoptimization
  - Как только скомпилированный метод перестал быть корректным, деоптимизировать все его активации
  - Возможно, совершаем лишнюю работу и зря замедляем выполнение существующих активаций
  - Можно сразу избавиться от устаревшего кода
- Lazy deoptimization
  - Деоптимизировать секцию активации устаревшего кода, если она верхняя в стеке
  - Остальные активации деоптимизировать при возврате в них — подменяем адрес возврата
  - Кроме обычных возвратов, бросаются исключения!
  - Необходимо хранить устаревший код до тех пор, пока существуют его активации
  - В каждый момент для каждого метода может существовать не более одной текущей версии скомпилированного кода, но множество устаревших

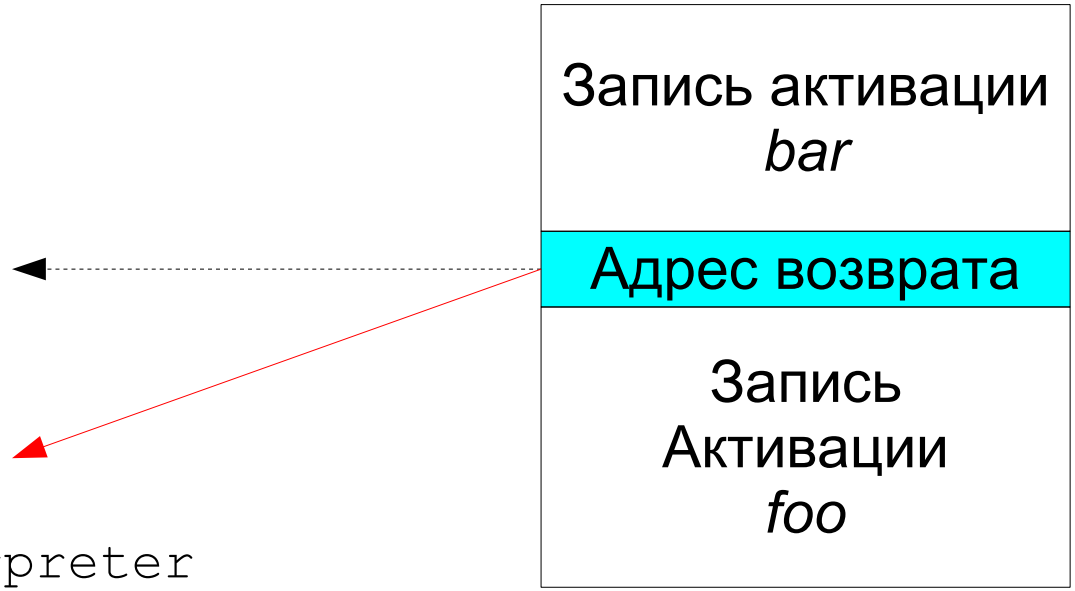
Ленивая деоптимизация

Скомпилированный код метода foo

```
foo_compiled_entry:
   …
   direct_call bar
   …
   return

deoptimize_call_bar:
   …
   jmp Switch_to_Interpreter
```

Стек



Сокращение числа деоптимизаций (1)

- Все ли активации требуют деоптимизации?
  - Во время компиляции метода его предикат корректности заведомо верен
  - Инициализация класса может нарушить предикат, после этого устаревший код нельзя вызывать
  - Но нельзя ли продолжить выполнение уже существующих активаций?
- Инициализация класса не меняет ранее созданных объектов
  - Для некоторых объектов это известно во время компиляции
    - Параметры вызова и их константные поля
    - Локальные вычисления, предшествовавшие инициализации
  - Предикат корректности продолжения выполнения слабее предиката корректности нового вызова!
  - Учитываем это во время оптимизации представления предикатов

Сокращение числа деоптимизаций (2)

- Возможность продолжать выполнение зависит от текущей позиции
  - Предикат корректности скомпилированного метода — это логическое произведение условий всех спекулятивно девиртуализованных вызовов в нем
  - Не все такие вызовы достижимы из текущей позиции
  - Во время компиляции для каждой позиции вычисляем свой предикат продолжения как произведение условий всех достижимых из нее вызовов
  - Позиции — точки возврата из вызовов
  - Пишем предикаты в таблицу диапазонов адресов, оптимизируем представление
  - Вычисление предикатов требует времени, их запись занимает память

Замена полей локальных объектов локальными переменными

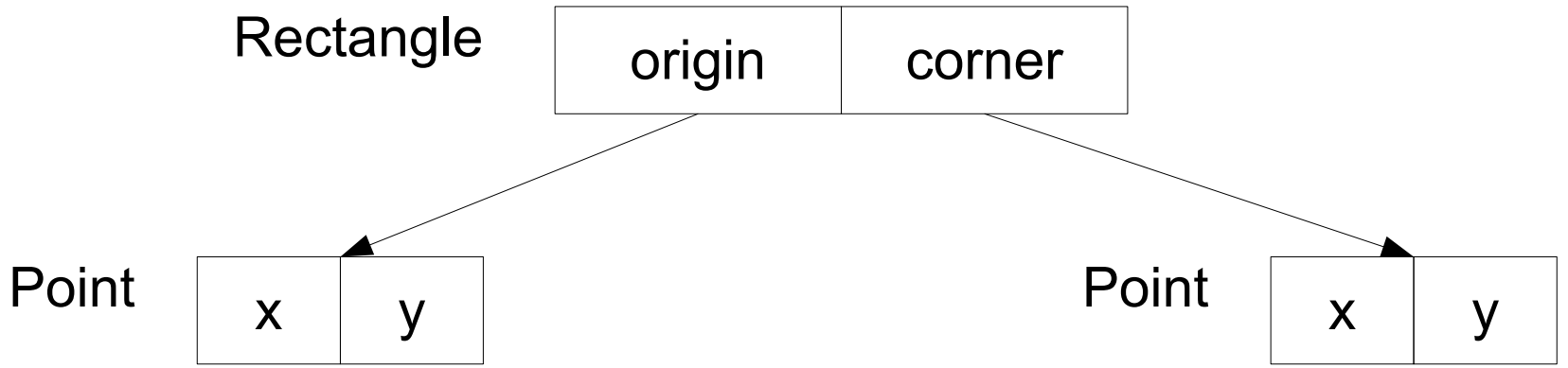
- Если созданный объект не покидает локальной области, можно его разложить в набор переменных, локальных в этой области
  - Не нужно отводить объект в «куче»
  - Обращения к локальным переменным хорошо оптимизируются
  - Класс объекта и поля должны быть инициализированы
- Escape analysis
  - Нельзя присваивать ссылку на объект нелокальным переменным, в т.ч. полям объектов в «куче»
  - Нельзя передавать этот объект параметром неинлайновых методов
  - Нельзя выдавать этот объект результатом

Замена полей локальных объектов локальными переменными (2)

- Трудность деоптимизации
  - Если по какой-либо причине нужно заменить скомпилированный метод с локализованными объектами интерпретируемым, необходимо воссоздать эти объекты из локальных переменных
  - Нужно заранее позаботиться, чтобы объектам хватило памяти — кидать OOME в этот момент нельзя
  - Подсистема памяти должна уметь резервировать память, разрезервировать и отводить ранее зарезервированную память
  - При входе в локальную область зарезервировать количество памяти, необходимое для локальных объектов, при выходе из локальной области, в т.ч. и по исключению, разрезервировать.
  - Если зарезервировать не удалось, перейти в режим интерпретации

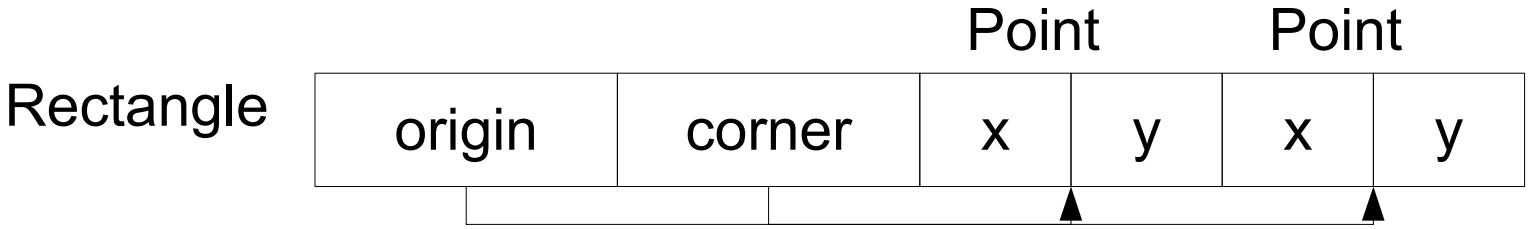
Инлайновая подстановка объектов

- Объекты образуют устойчивые конфигурации



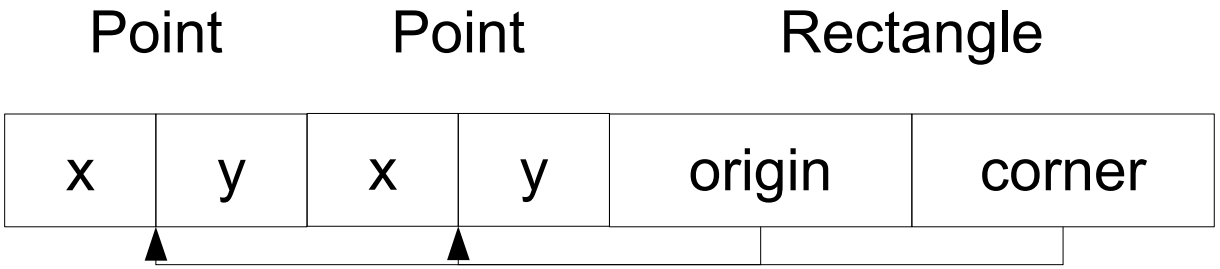
- Конфигурация часто фиксируется в пределах конструктора объекта
  - В сильно динамических языках все сложнее

Слабая инлайновая подстановка объектов



- Если у класса нет подклассов с полями && удается доказать, что в объекте на протяжении его жизни ссылки на некоторые подобъекты не меняются, эти подобъекты можно отвести в памяти вслед за объектом и адресовать относительно его базы
  - Сборщик мусора должен знать, что такие конгломераты нельзя разрушать — например, использовать таблицы коллокации
  - Структура объектов не меняется — оптимизируется только доступ к ним

Слабая инлайновая подстановка объектов и наследование

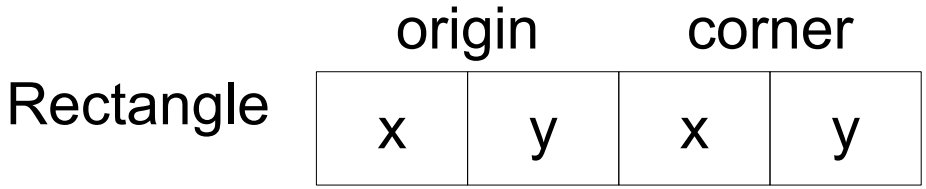


- Подкласс может добавлять поля
- При добавлении полей меняются размер объекта и смещения подобъектов
  - Для сохранения корректности скомпилированного кода размещаем подобъекты перед головным объектом

Слабая спекулятивная инлайновая подстановка объектов

- При открытой иерархии классов неизменность ссылок на подобъекты и отсутствие полей в подклассах редко удается доказать.
  - Классы редко объявляют финальными
  - Доказываем для инициализированной части иерархии
  - Если у класса заводится и инициализируется подкласс, проверяем отсутствие в нем полей и присваиваний наследованным ссылкам на подобъекты
  - Если свойство нарушено, проще всего перейти к интерпретации ранее оптимизированного кода
  - Сложнее для одного и того же исходного метода компилировать разные версии кода в классе и подклассе

Сильная инлайновая подстановка объектов



- Если дополнительно удается доказать, что ссылки на подобъекты не распространяются вне объекта и не передаются в неинлайновые методы && поля ссылок не наследованы, поля подобъектов можно перенести в объект, а поля ссылок на подобъекты устранить.
  - Доказать такое не удается почти никогда
  - Меняется структура объектов
  - Сильную спекулятивную инлайновую подстановку объектов трудно реализовать

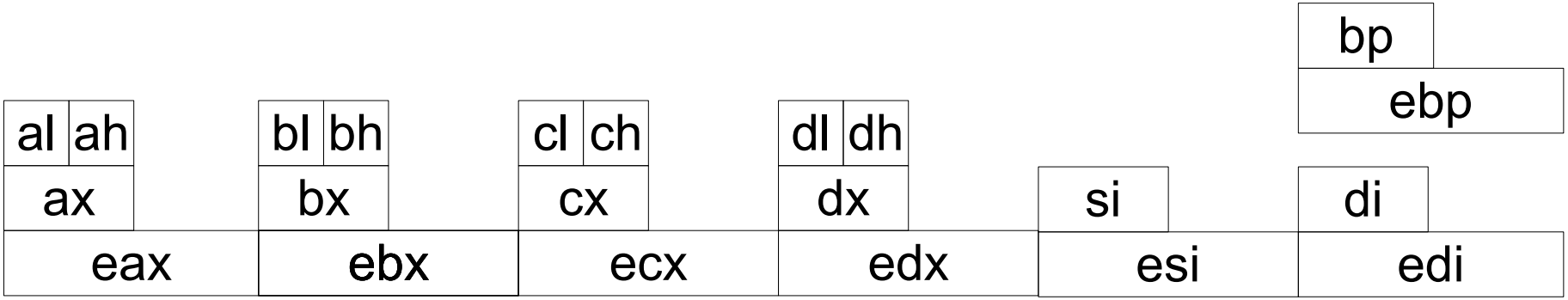
Распределение регистров

- FIFO
  - Выбор регистра, который отведен раньше всех
  - Связываем все регистры в очередь, при отведении регистра перемещаем его в конец, при освобождении — в начало
- Round Robin
  - Циклическое сканирование от текущей позиции
  - Близок к FIFO
- LRU
  - Выбор регистра, который дольше всего не использовался
  - Связываем все регистры в очередь, при использовании регистра перемещаем его в конец, при освобождении — в начало
- Раскраска графа
  - Классический медленный алгоритм
- Линейное сканирование
  - С разнообразными улучшениями приближается к раскраске графов
- Puzzle Solving

Puzzle Solving Register Allocation

- Генерируемый код — параметризованные регистрами паттерны машинных инструкций
- Параметры паттерна могут быть входными, выходными и входными-выходными
- Некоторые инструкции требуют операндов в фиксированных регистрах
  - Целочисленное деление, строчные операции, обращения к портам ввода-вывода
- Регистры могут быть разной ширины и частями других регистров

Система регистров x386

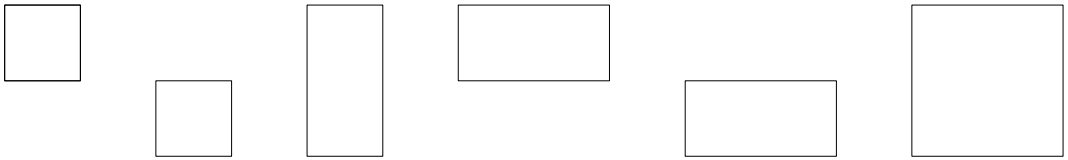


Puzzle Solving Register Allocation (2)

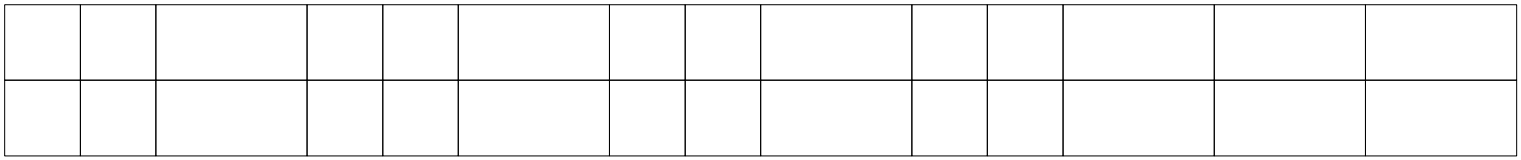
- Параметры паттернов — кусочки паззла

in

out



- Наборы входных и выходных регистров формируют поле для размещения кусочков паззла



Puzzle Solving Register Allocation (3)

- При выделении регистров для паттерна сначала пытаемся выставить фиксированные кусочки
- Если не получается — задача неразрешима
  - Алгоритм не занимается спиллингом — нужно где-то вытеснить что-то из регистров и повторить размещение регистров
- Затем пытаемся выставить прочие кусочки, последовательно применяя правила
- Правила определяются статически набором регистров и набором всевозможных кусочков

Puzzle Solving Register Allocation (4)

- Для всех современных процессоров эти правила образуют строгую последовательность
  - Применение следующего правила не может привести к лучшему результату, чем применение предыдущего
  - Это свойство доказывается как теорема в теории паззлов
  - Более сложные взаимоотношения между регистрами могли бы это свойство нарушить
- Влияние размера паттерна
  - Для слишком крупных паттернов задача может оказаться неразрешимой
  - Между слишком мелкими паттернами может оказаться слишком много регистровых пересылок
- Алгоритм можно начинать применять с любого паттерна — например, с внутренних циклов

Кодогенерация

- Вручную
- С использованием грамматик
  - Так удобнее записывать правила применения семантик
- BURG и вариации

Пример: динамический компилятор Monty VM

- Однопроходный
  - Нет памяти для промежуточного представления
  - Память слишком медленная для построения ПЯ
  - На самом деле, почти однопроходный — производится быстрый предварительный просмотр для определения границ базовых блоков, ставятся метки в байтовом массиве
  - Иногда заглядывает вперед для распознавания коротких циклов
- Компиляция путем абстрактной интерпретации байткода
- Вручную написанный кодогенератор
- Скомпилированный код размещается в непрерывном расширяемом буфере

Компиляция путем абстрактной интерпретации

- Контекст выполнения моделирует состояние рамки стека
- Вместо значений времени выполнения используются значения времени компиляции
  - У каждого значения есть тип и размещение в памяти
  - Значение может быть константой
  - Значению может быть назначен регистр или регистровая пара
- При интерпретации байтового кода меняется состояние рамки и может генерироваться код
- Контексты выполнения помещаются в очередь абстрактного интерпретатора

Ветвление потока управления

- Клонировать текущий контекст, поместить его в очередь
- Продолжить интерпретацию той ветви, которая выполняется чаще
- Если метка перехода еще не посещалась, ассоциировать с ней копию текущего состояния
- Иначе при объединении потоков управления согласовать состояния
  - Состояние в начале базового блока определяется при первом входе в этот блок
  - Все последующие входы должны подчиняться этому состоянию
  - При согласовании состояний может потребоваться перемещение значений между регистрами и памятью

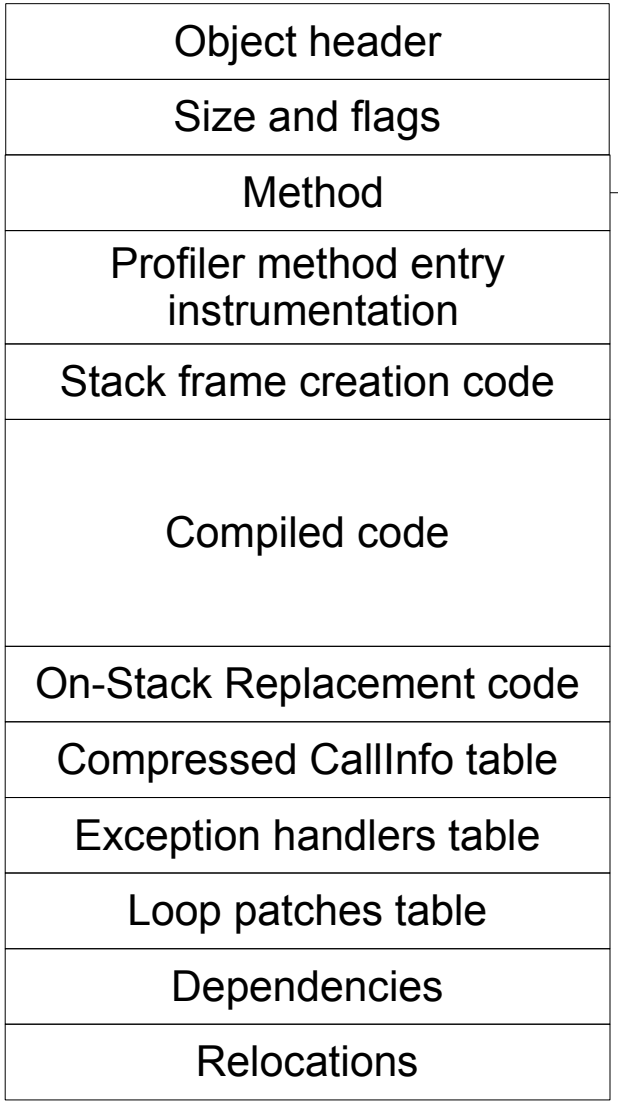
Отведение регистров в Monty VM

- Модифицированный Round Robin
  - Сначала пустые
  - Потом вычисляемые константы и общие подвыражения
  - Потом обычный Round Robin

Оптимизации в Monty VM

- Свертка и протяжка констант
- Изведение общих подвыражений (ограниченное)
- Инлайновая подстановка мелких методов
- Оптимизации циклов и переходов
- Незащищенная спекулятивная девиртуализация
- Специализированные и инлайновые вызовы копирования массивов
  - Оптимизированное копирование блоков памяти
  - Исключение избыточных проверок и вычислений
  - Массовый барьер записи (для GC)

Структура скомпилированного метода в Monty VM

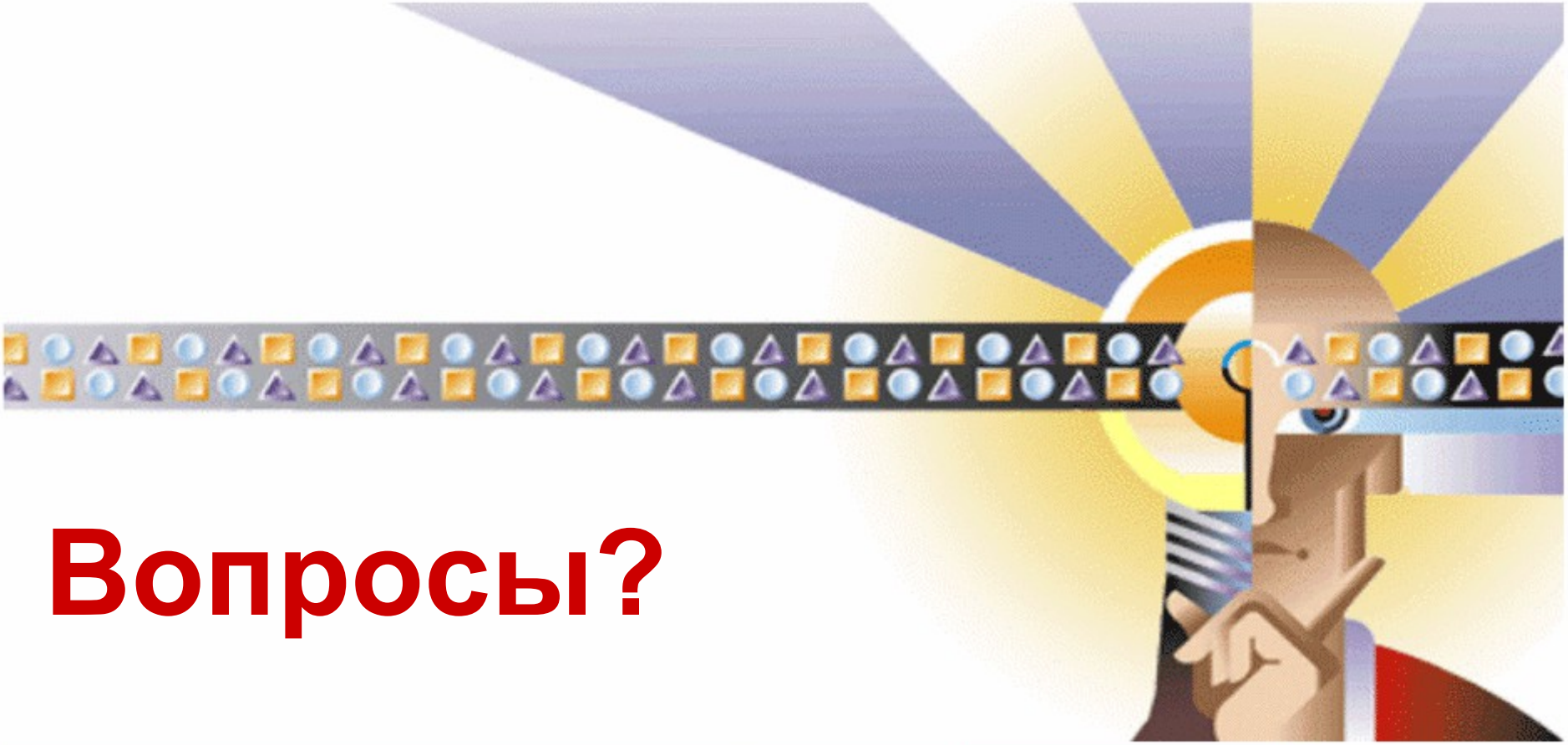


Интерпретируемый метод

x86: move byte [execution_sensor+method_index], 0

Для коротких циклов

Условия спекулятивных оптимизаций



Вопросы?

Sun microsystems
We make the net work.