инженерия программного обеспечения

Пилипенко Артур Витальевич

УДК 004.453

ОБЗОР ИНТЕРПРЕТАЦИИ И КОМПИЛЯЦИИ В ВИРТУАЛЬНЫХ МАШИНАХ

Аннотация

Исполнение байткода под управлением виртуальной машины имеет ряд преимуществ перед традиционным исполнением машинного кода. Это переносимость, безопасность, удобство компиляции и отладки. В то же время, такое исполнение влечет за собой дополнительные накладные расходы. В данной статье описываются два способа исполнения байткода: интерпретация и компиляция. Для каждого из этих методов обсуждаются накладные расходы и узкие места с точки зрения производительности. Дается обзор основных оптимизаций этих методов, которые позволили достичь скорости исполнения, сравнимой со скоростью исполнения машинного кода.

Ключевые слова: виртуальные машины, байткод, интерпретация, AOT компиляция, JIT компиляция, динамическая компиляция, адаптивные оптимизации.

Виртуальная машина представляет собой абстрактную вычислительную машину, не зависящую от реальной аппаратной или программной платформы. Аналогично реальной вычислительной машине она определяет архитектуру команд, которая включает в себя соглашения о передаче аргументов, стеки операндов или набор регистров, модель памяти, непосредственно набор команд. Также виртуальная машина определяет исполняемое представление программы. Примером такого представления может служить закодированная последовательность инструкций, аналогичная машинному коду. Такое представление называется байткодом и является одним из самых распространенных. По сравнению с машинным кодом, байткод зачастую компактнее, так как набор инструкций виртуальной машины более высокоуровневый. В отличие от других представлений, например текста или синтаксических деревьев, байткод прост в декодировании. Это упрощает его эффективное исполнение. Байткод используется в качестве исполняемого представления в таких языках и платформах как Java (Java байт-код), .NET (.NET CIL ассемблер), Android (Dalvik VM байткод), Pascal (P-code) и других.

Исполнение промежуточного представления, несомненно, влечет за собой дополнительные накладные расходы по сравнению с исполнением машинного кода. Но, благодаря исполнению байткода путем его компиляции, применению адаптивных и спекулятивных (или оптимистических) оптимизаций, современные реализации виртуальных машин по скорости исполнения вплотную приблизились к скорости исполнения машинного кода. Данная статья описывает способы исполнения байткода и дает обзор основным методам их оптимизации, позволившим добиться такого результата.

© Пилипенко А.В., 2012

ИНТЕРПРЕТАЦИЯ

Самый простой и очевидный способ исполнения байткода состоит в его последовательной интерпретации. Интерпретаторы просты в реализации и легко переносимы. Интерпретация не требует никакой предварительной работы перед исполнением программы. Благодаря этим преимуществам интерпретация широко применяется для исполнения байткода. Главный недостаток этого способа – невысокая скорость, которая по разным оценкам в 2–50 раз меньше скорости исполнения машинного кода [11, 32, 33].

Для того чтобы определить причины низкой производительности, рассмотрим работу интерпретатора подробнее. Интерпретация инструкции виртуальной машины состоит из декодирования, непосредственного исполнения и перехода к следующей инструкции. Непосредственная работа инструкции выполняется только на втором шаге, остальные действия – это вынужденные накладные расходы на интерпретацию. Производительность интерпретатора определяется соотношением полезной работы и этих накладных расходов. Соотношение это варьируется в зависимости от реализации интерпретатора и архитектуры набора инструкций. Очевидно, что это соотношение можно улучшить, увеличив количество работы выполняемой одной инструкцией, то есть, используя более высокоуровневый набор инструкций. Кроме того, увеличить долю полезной работы можно, уменьшив накладные расходы на обработку каждой инструкции. Как показано в [35], на декодирование и переход между инструкциями тратится большая часть времени работы интерпретатора. Это объясняется тем, что большинство реализаций использует непрямые переходы (переходы по значению, определяемому на этапе исполнения) при обработке инструкции. Конвейерные архитектуры современных процессоров максимально эффективно исполняют последовательные участки кода, изменение же хода программы приводит к сбрасыванию конвейера и существенной потере скорости исполнения. Для того чтобы избежать подобных ситуаций, процессоры используют системы предсказания переходов. Правильно предсказанный переход позволяет продолжить исполнение без каких-либо последствий. К сожалению, системы предсказания переходов почти не работают для непрямых переходов. Таким образом, обработка каждой инструкции почти всегда приводит к сбросу конвейера и потере производительности. Проблеме неправильного предсказания непрямых переходов при интерпретации посвящен ряд работ, в которых предлагаются различные способы решения проблемы как на программном, так и на аппаратном уровне [25, 30, 35].

Рассмотрим, как накладные расходы на декодирование и диспетчеризацию инструкций зависят от реализации интерпретатора. Для представления, кодирующего инструкции с помощью целочисленных констант, реализация интерпретатора на языке высокого уровня может выглядеть следующим образом (см. листинг 1).

Подобная реализация проста и легко переносима (требуется лишь перекомпиляция интерпретатора под новую платформу). В то же время, она не отличается высокой эффективностью. Это связано с особенностями реализации конструкции switch-case большинством компиляторов. Обычно эта конструкция реализуется с помощью таблицы переходов [4].Такая реализация требует проверки входного значения на принадлежность диапазону, определения смещения в таблице переходов по этому значению и непрямого перехода по адресу из таблицы. Помимо этого, переход к следующей итерации цикла для обработки следующей инструкции требует еще одного перехода.

ШИТЫЙ КОД

Эти накладные расходы можно уменьшить, используя альтернативное кодирование инструкций, например используя прямой шитый код, который кодирует инструкцию адресом, по которому расположен код реализации этой инструкции [36]. При использовании такого кодирования переход к следующей инструкции вместе с её декодированием сводится к переходу по следующему адресу из тела программы. С точки

Листинг 1

```
typedef enum {
     push1, add, …
} Instruction;
…
Instruction code[] = {push1, push1, add, …};
Instruction *ip = code;
while (true) {
     switch (*ip++) {
          case push1:
               /* Реализация инструкции push1 */
               break;
          case add:
               /* Реализация инструкции add */
               break;
               …
     }
}
```

зрения интерпретации, прямой шитый код является наиболее эффективным представлением программы, но это достигается за счет менее компактного кодирования (для кодирования одной инструкции используется машинное слово). Реализация интерпретатора прямого шитого кода выглядит следующим образом (используется синтаксис расширения GCC «Labels and Values» [19]) (см. листинг 2).

Другой разновидностью шитого кода является косвенный шитый код. От прямого он отличается тем, что для кодирования инструкции вместо адреса её реализации используется адрес элемента в таблице инструкций. Таблица инструкций, в свою очередь, содержит адреса их реализаций [22]. Дополнительная косвенность увеличивает расходы на декодирование инструкции, но при этом дает большую гибкость. Например, переписыванием таблицы инструкций можно переключаться между отладкой и обычным исполнением (листинг 3).

Подход, аналогичный интерпретации косвенного шитого кода, можно применить для интерпретации байткода. В этом случае значения, кодирующие инструкции, используются как индексы в таблице инструкций, хранящей адреса их реализаций. Декодирование инструкции при этом состоит из чтения адреса из таблицы, а переход к её исполнению осуществляется переходом по этому адресу (листинг 4).

Подобным образом реализован интерпретатор виртуальной машины OpenJDK HotSpot [18] . В HotSpot машинный код интерпретатора и таблицы инструкций генерируются динамически при старте виртуальной машины. Такой подход имеет сразу несколько преимуществ:

– дает полный контроль над генерируемым машинным кодом;

– позволяет генерировать разный машинный код без перекомпиляции виртуальной машины, что может быть использовано, например, для отладки;

Листинг 2

```
void* program[] = {&&l_push1, &&l_push1, &&l_add, …}
void** ip = program;
goto **ip++;
l_push1: { /* Реализация инструкции push1 */ } goto **ip++;
l_add: { /* Реализация инструкции add */ } goto **ip++;
…
```

Листинг 3

```
static void *lut[] = {&&l_push1, &&l_add, …};
…
void* program[] = {lut, lut, lut + 1, …}
void** ip = program;
goto ***(ip++);
l_push1: { /* Реализация инструкции push1 */ } goto ***(ip++);
l_add: { /* Реализация инструкции add */ } goto ***(ip++);
…
```

– упрощает процесс построения виртуальной машины за счет использования только высокоуровневого языка;

– упрощает описание зависимостей между высокоуровневым и машинным кодом, позволяя использовать при генерации кода такие параметры, как размеры объектов, смещения внутри объектов и другие;

– позволяет генерировать наиболее эффективную реализацию интерпретатора под архитектуру конкретного процессора;

– кроме того, генератор кода интерпретатора может использовать ту же инфраструктуру, что и динамический компилятор.

Код порожденного интерпретатора можно получить, запустив виртуальную машину с ключом –
XX:+UnlockDiagnosticVMOptions –
XX:+PrintInterpreter в командной строке.

Существует много способов, с помощью которых можно ускорить исполнение инструкций. Например, для стековых виртуальных машин можно ускорить доступ к операндам инструкций с помощью кэширования вершины стека в регистрах процессора. Аналогичного эффекта можно добиться для регистровых виртуальных машин с помощью отображения виртуальных регистров в физические. Другой пример – это устранение избыточных проверок времени исполнения там, где это возможно, путем замены инструкции на её упрощенную версию. Но все эти способы не исправляют главных причин низкой производительности интерпретаторов. Даже самая эффективная реализация интерпретатора остается в несколько раз медленнее исполнения скомпилированного кода. Поэтому следующим шагом для увеличения скорости исполнения является компиляция промежуточного представления в код целевой платформы. Стоит отметить, что работы по ускорению интерпретации велись и в другом направлении. Ряд исследований был посвящен специализированному аппаратному обеспечению, призванному ускорить интерпретацию, например за счет аппаратной поддержки подмножества инструкций [14, 26, 20]. Однако, если сравнивать с компиляцией в машинный код архитектур общего назначения, эти аппаратные реализации не принесли значительного ускорения и не обладали достаточной гибкостью.

Листинг 4

```
typedef enum {
    push1, add, …
} Instruction;
static void *lut[] = {&&l_push1, &&l_add, …};
…
Instruction program[] = {push1, push1, add, …}
Instruction *ip = program;
goto *lut[*ip++];
l_push1: { … /* Code to handle "const_1" */ … } goto *lut[*ip++];
l_add: { … /* Code to handle "add" */ … } goto *lut[*ip++];
…
```

КОМПИЛЯЦИЯ

В отличие от интерпретатора скомпилированный код не выполняет лишней работы по декодированию и переходам между инструкциями байткода. Кроме того, скомпилированный код может быть дополнительно оптимизирован. Возможности компилятора в этом плане значительно шире, чем у интерпретатора, который исполняет инструкции в отрыве от контекста. Например, компилятор может более эффективно распределять регистры в пределах методов, обладая информацией об используемых локальных переменных и временных значениях. Если компилятор виртуальной машины работает одновременно с исполнением программы, то он может использовать информацию о динамическом состоянии программы и может реагировать на изменение этого состояния. На этой возможности основан целый ряд адаптивных и спекулятивных оптимизаций.

С другой стороны, компиляция имеет свои недостатки. Компилятор значительно усложняет архитектуру виртуальной машины и ухудшает её переносимость, так как использует информацию о целевом процессоре. Компилятор требует больше памяти по сравнению с интерпретатором для хранения внутренних структур и скомпилированного кода, который обычно в несколько раз больше размера соответствующего байткода. В отличие от интерпретатора, компилятор не может сразу приступить к исполнению программы: всегда существует задержка, связанная с компиляцией байткода в машинный код. Если компиляция осуществляется на лету, то компилятор конкурирует с исполняемой программой за процессорные ресурсы. Все это в совокупности усложняет применение компиляции в условиях ограниченности ресурсов.

AOT КОМПИЛЯЦИЯ

Можно выделить две различные стратегии компиляции байткода в код целевой платформы: это компиляция перед исполнением (ahead-of-time, AOT) и компиляция на лету (just-in-time, JIT). В первом подходе AOT компилятор выступает в роли backend статического компилятора, который транслирует машинно-независимое промежуточное представление в бинарный образ целевой платформы. Для этого применяются хорошо известные и изученные алгоритмы статической компиляции. Как и статический компилятор, AOT компилятор не ограничен по времени и может осуществлять сложные оптимизации, требующие глубокого анализа программы. Но вместе с этим AOT компилятор так же ограничен в применении данных времени исполнения. Использование AOT компиляции позволяет существенно упростить среду исполнения и тем самым уменьшить потребление памяти и время запуска программы.

JIT КОМПИЛЯЦИЯ

Компиляция программы одновременно с её исполнением называется JIT компиляцией, или динамической компиляцией. Так как динамический компилятор работает одновременно с выполняемой программой, то он конкурирует с выполняемой программой за вычислительные ресурсы, и время его работы становится критичным для скорости работы программы. Это обстоятельство сильно влияет на алгоритмы и оптимизации, применяемые при динамической компиляции. Если статический компилятор имеет возможность осуществлять более сложные оптимизации ценой увеличения времени компиляции, то динамический компилятор должен балансировать между качеством порождаемого кода и временем работы для достижения оптимальной производительности. В то же время, динамический компилятор обладает дополнительными сведениями о статистике исполнения программы, её динамическом состоянии и текущей аппаратной платформе. Это открывает дополнительные возможности по оптимизации, не доступные статическому и AOT компилятору.

Эти возможности особенно актуальны для объектно-ориентированных и динамических языков, оптимизация которых на этапе статической компиляции усложняется рядом факторов. Рассмотрим один из них в качестве примера. Подстановка методов

(method inlining) – одно из самых эффективных оптимизирующих преобразований. Кроме того, что эта оптимизация позволяет избавиться от дорогостоящей операции вызова, она также открывает возможности для сквозных оптимизаций подставленных методов. В свою очередь, полиморфизм, являющийся одним из основных механизмов объектно-ориентированного подхода, влечет за собой использование динамической диспетчеризации вызовов вместо статической. Это значит, что конкретная реализация метода, которая будет вызвана, определяется только на этапе исполнения. При этом не только увеличивается стоимость вызовов, но компилятор лишается возможности подстановки методов и дальнейших оптимизаций. Возможность динамической загрузки классов лишает компилятор возможности использовать статическую диспетчеризацию для виртуальных вызовов, так как набор классов, который будет доступен на этапе исполнения, не известен. Ситуация усугубляется тем, что написание удобочитаемого и повторно используемого кода поощряет использование значительного количества небольших методов. В то же время, во время исполнения программы известен точный набор классов на текущий момент. Обладая этой информацией, динамический компилятор может осуществлять подстановку методов там, где это возможно. Подобные возможности являются одной из причин широкой распространенности виртуальных машин в качестве сред исполнения для современных объектно-ориентированных и динамических языков.

АДАПТИВНАЯ ОПТИМИЗАЦИЯ

Как было сказано ранее, время работы динамического компилятора вносит свой вклад в общее время работы программы. Но не всегда временные затраты на компиляцию окупаются уменьшением времени выполнения кода. Эта особенно актуально для различных инициализаторов, которые выполняются всего один раз. Их компиляция зачастую приводит к замедлению старта программы вместо ускорения. Решением этой проблемы является выборочная оптимизация – подход, который состоит в применении различного уровня оптимизации к разным частям программы. Тот факт, что большинство программ тратят основную часть времени на исполнение небольшой части кода [9], позволяет виртуальной машине сфокусироваться на оптимизации часто исполняемого кода и минимальными временными затратами достичь существенного прироста производительности.

Для нахождения участков программы, требующих оптимизации, используется адаптивный подход: оптимизируется лишь тот код, который будет активен в ближайшем будущем. Для нахождения такого кода применяется простое эвристическое предсказание: если программная единица была активна в ближайшем прошлом, то она будет активна и в ближайшем будущем. Активность программы отслеживается профилировщиком (profiler), и те части программы, которые были отмечены как активные (так же называются «горячими»), впоследствии оптимизируются.

Не оптимизированным исполнением может быть как интерпретация [23, 29], так и простейшая не оптимизирующая компиляция [8, 31]. Кроме того, виртуальная машина может иметь несколько уровней компиляции, каждый последующий из которых отличается более сложными оптимизациями и большим временем компиляции. Выбор уровня компиляции в таком случае также осуществляется адаптивно под управлением профилировщика. Такая компиляция называется многоуровневой. На практике количество уровней компиляции ограничивается двумя или тремя, так как большее количество уровней компиляции не приносит значительного выигрыша производительности [2, 13]. Многоуровневая компиляция применяется в таких виртуальных машинах, как OpenJDK HotSpot [23], Jikes RVM [8], Azul [29].

Адаптивный подход ориентирован на достижение максимальной пиковой производительности и имеет побочные эффекты. Так, для достижения максимальной производительности требуется время на «прогрев» виртуальной машины. Это время необходимо для выявления «горячего» кода и его ком-

пиляцию. «Прогрев» виртуальной машины не является проблемой для серверных приложений с большим временем жизни, но может быть недостатком для пользовательских интерактивных приложений. Виртуальная машина OpenJDK HotSpot использует различные компиляторы в клиентской и серверной конфигурации для разрешения этой проблемы [15, 34]. Клиентский компилятор ориентирован на максимально быструю генерацию кода, для того чтобы минимизировать время отклика интерактивных приложений. Серверный компилятор, в свою очередь, осуществляет гораздо более сложные оптимизации ценой большей времени компиляции. Также в OpenJDK HotSpot поддерживается многоуровневая компиляция, которая позволяет объединить преимущества обоих подходов.

ПРОФИЛИРОВАНИЕ

Профилировщик является обязательной частью механизма адаптивной компиляции. Профилирование не должно сильно замедлять исполнение программы и не должно искажать картины её активностей. Для выявления активностей программы применяют два различных подхода. Первый из них состоит в точном подсчете некоторых событий. В контексте адаптивной оптимизации это такие события, как количество вызовов каждого метода и количество совершенных итераций каждого цикла. Для этого каждый метод и переход назад внутри нее снабжается счетчиком. Увеличение счетчиков осуществляется или интерпретатором, или инструментированными инструкциями в скомпилированном коде. При переполнении счетчики могут масштабироваться. Этот способ профилирования является точным, но накладные расходы на хранение и обновление счетчиков могут быть довольно велики. Такой подход применяется в виртуальной машине Open JDK HotSpot [34]. В этой виртуальной машине для обоих типов счетчиков существуют пороговые значения, превышение которых инициирует компиляцию соответствующего метода. Аналогично осуществляется профилирование в виртуальной машине Azul [29].

Второй механизм профилирования называется сэмплированием (sampling). Он заключается в сборе статистической информации и не является точным. Для этого через равные интервалы времени анализируется состояние программы. Для выявления активных методов через равные промежутки времени анализируется состояние стеков вызовов программы. Этот механизм эффективно распознает «горячие» методы, содержащие циклы, но может пропускать вызовы коротких методов. Накладные расходы сэмплирования не велики, но применение неточного механизма вносит недетерминизм в поведение виртуальной машины, что может усложнить её отладку. Сэмплирование используется для профилирования в следующих виртуальных машинах: Oracle JRockit [21], Jikes RVM [8].

Системы с многоуровневой компиляцией могут применять комбинированный подход, как это делается в IBM Development Kit for Java [16]. Профилирование неоптимизированного кода осуществляется подсчетом событий. По сравнению со скоростью неоптимизированного исполнения, накладные расходы на обновление счетчиков не слишком велики. Для профилирования оптимизированного кода применяется сэмплирование как менее затратный механизм.

ON-STACK REPLACEMENT

Рассмотрим, как осуществляется переключение между оптимизированным и неоптимизированным исполнением кода. После компиляции или перекомпиляции метода виртуальная машина заменяет его последующие вызовы на вызов оптимизированной версии. То есть переход к оптимизированной версии осуществляется только при следующем вызове. Что если метод содержит бесконечный цикл или просто долго исполняющийся код? В первом случае переход к исполнению оптимизированной версии не произойдет никогда, во втором случае произойдет не скоро. Эффективная оптимизация подобных случаев требует возможности переключения исполнения на новую версию кода непосредственно во время выполнения метода. Механизм, позволяющий осу-

Листинг 5

```
class C {
   static int sum(int c) {
      int y = 0;
      for (int i=0; i<c; i++) {
          y += i;
      }
      return y;
   }
}
```

ществлять такое переключение, называется On-Stack Replacement (OSR). Он состоит в извлечении состояния исполняемого метода, не зависящего от конкретного способа исполнения, и воссоздании этого состояния для новой версии кода. Это состояние включает в себя указатель на текущую инструкцию исполняемого байткода, содержимое локальных переменных, записи активации на стеке вызовов. Обычно это состояние хранится в записи активации на стеке вызовов, и его воссоздание требует замены этой записи, что и дало название способу. Стоит отметить, что из-за подстановки методов одна секция активации оптимизированного метода может соответствовать нескольким секциям активации неоптимизированных методов. В случае интерпретации такое состояние можно восстановить в любой момент. Для скомпилированного кода подобный переход возможен лишь в заранее подготовленных точках, в которых можно восстановить такое состояние. Обычно это точки вызовов методов и итерации циклов.

Листинг 6

```
0 iconst_0
1 istore_1
2 iconst_0
3 istore_2
4 goto 14
7 iload_1
8 iload_2
9 iadd
10 istore_1
11 iinc 2 1
14 iload_2
15 iload_0
16 if_icmplt 7
19 iload_1
20 ireturn
```

Приведем пример сохраняемого состояния исполняемого метода при замещении на стеке [13]. Код метода на языке Java (см. листинг 5). Соответствующий ему байткод приведен в листинге 6. Состояние метода после 50 итераций цикла перед исполнением байткода с индексом 16 (листинг 7). Специальная версия метода sum для OSR перехода (см. листинг 8).

Подобный механизм также может использоваться для обратного перехода к неоптимизированному исполнению. Такой переход называется деоптимизацией. Изначально он был описан для языка Self как способ перехода от оптимизированного исполнения к отладке [12]. Сейчас он применяется, в том числе, для реализации целого ряда спекулятивных (оптимистических) оптимизаций, которые описаны ниже.

Гранулярность выборочной оптимизации на уровне методов зачастую является не самым эффективным решением. Некоторые ветвления внутри методов могут исполняться гораздо реже остальных. Отложенная компиляция (deferred compilation), реализуемая при помощи OSR, позволяет применять выборочную оптимизацию к более мелким программным единицам [3, 13, 24]. Другое название этого подхода – частичная компиляция (partial compilation). Применяя этот подход, компилятор может не генерировать код для редко исполняемых условий. Вместо этого, компилятор вставляет вместо них заглушки, которые осуществляют переход к исполнению неоптимизированной версии. Переход к исполнению неоптимизированной версии осуществляется при помощи механизма OSR. Так, например, серверный компилятор виртуальной машины OpenJDK HotSpot не генерирует код, связанный с загрузкой и инициализацией классов, так как в большинстве случаев подобные инициализации к моменту компиляции оказываются выполненными [34].

ОПТИМИЗАЦИИ ДИНАМИЧЕСКОГО КОМПИЛЯТОРА

Одной из первых и самых эффективных оптимизаций, применяемых при компиля-

Листинг 7

```
running thread : MainThread
frame pointer : 0xSomeAddress
program counter : 16
local variables : L0(c) = 100; L1(y) = 1225; L2(i) = 50;
stack expressions : S0 = 50; S1 = 100;
```

ции, является подстановка методов, которая состоит в подстановке тела вызываемого метода вместо его вызова. Такая оптимизация позволяет избавиться от выполнения операций вызова и возврата, прологов и эпилогов методов. Подстановка методов положительно влияет на локальность кода, что приводит к более эффективному использованию кэш памяти. И, наконец, подстановка методов расширяет возможности компилятора по применению дальнейших оптимизаций для подставленного кода. Главным недостатком этой оптимизации является увеличение размеров результирующего кода. Увеличение размера кода может привести к увеличению количества промахов кэша инструкций. Кроме того, при подстановке методов увеличивается давление на регистры. При заданном ограничении на размер кода определение оптимального набора подстановок сводится к задаче о рюкзаке [6]. На практике для принятия решений о подстановке применяются различные эвристики. Это могут быть как статические эвристики, например размер метода, меньший некоего порога [15], так и эвристики, основанные на статистической информации об исполняемой программе [1, 7].

Для описания дальнейших оптимизаций стоит рассмотреть процесс компиляции подробнее. В самом простом случае компиляция состоит в шаблонной генерации фрагментов машинного кода для каждой инструкции байткода. Такой подход называется шаблонной компиляцией. Шаблонная компиляция характеризуется высокой скоростью компиляции, простотой реализации и невысоким качеством порождаемого кода. Реализация шаблонного компилятора может быть переносимой, если для генерации шаблонов кода используется язык высокого уровня [28]. Невысокое качество кода связано с тем, что при такой компиляции глубокий анализ программы затруднен, и возможности по оптимизации существенно ограничены. При такой компиляции обычно применяют локальные низкоуровневые оптимизации (peephole optimization), локальную свертку и распространение констант, ограниченное исключение общих подвыражений [11, 17]. Высокая скорость компиляции позволяет использовать шаблонную компиляцию вместо интерпретации для неоптимизированного исполнения. Другая область применения шаблонной компиляции – это оптимизированное исполнение в условиях ограниченности ресурсов.

Многие оптимизации, применяемые при статической компиляции, используют анализ потока данных и потока управления программы. Статический компилятор получает необходимую информацию путем преобразования текста программы в промежуточные представления более низкого уровня. Для того чтобы иметь возможность осуществлять подобные оптимизации, динамический компилятор должен выполнять обратную работу – генерировать более высокоуровневое представление из байткода. Обычно это

Листинг 8

```
ldc 100
istore_0
ldc 1225
istore_1
ldc 50
istore_2
ldc 50
ldc 100
goto 16
0 iconst_0
...
16 if_icmplt 7
...
20 ireturn
```

представление в Single Static Assignment (SSA) форме. На уровне этого представления осуществляются следующие преобразования:

– распространение констант и копий,
– свертка констант и упрощение выражений,
– исключение общих подвыражений,
– исключение «мертвого» кода,
– различные оптимизации циклов.

Эти оптимизации являются адаптацией хорошо известных алгоритмов классической компиляции. Помимо этих оптимизаций, на уровне промежуточного представления осуществляется:

– устранение лишних проверок на null,
– устранение лишних проверок границ массивов.

Использование промежуточного представления открывает возможности для применения многочисленных оптимизаций. В то же время, построение этого представления увеличивает время компиляции и требует дополнительной памяти для его хранения.

Рассмотрим высокоуровневое промежуточное представление (high-level intermediate representation, HIR), применяемое в клиентском компиляторе виртуальной машины OpenJDK HotSpot. Поток управления в этом представлении моделируется с помощью графа потока управления, узлы которого являются базовыми блоками, то есть последовательностями инструкций наибольшей длины без переходов или с переходами в эту же последовательность.

Базовые блоки, в свою очередь, состоят из инструкций высокоуровневого представления, которые формируют граф потока данных в SSA форме. Инструкция представляет собой как вычисление, так и результат вычисления. Инструкции ссылаются на свои аргументы при помощи указателей на другие инструкции.

Ниже приведен пример HIR для небольшого фрагмента кода с циклом [15]. Код на языке Java (см. листинг 9).

Соответствующий ему байткод приведен в листинге 10.

На рис. 1 показана визуализация промежуточного представления для этого примера.

Как было сказано ранее, динамический компилятор, в отличие от обычного компилятора, может использовать статистические данные об исполняемой программе. Профилировщик является обязательной частью среды исполнения с выборочной оптимизацией. Он также может использоваться для сбора статистической информации, которая может помочь в оптимизации наиболее часто исполняемых случаев [5]. Как было сказано ранее, эта информация может быть использована для принятия решений о подстановке методов. Другой пример подобной оптимизации: руководствуясь статистической информацией об условных переходах, компилятор может размещать часто исполняемые ветви программы непрерывно, улучшая тем самым локальность кода и упрощая предсказание переходов в нем.

Листинг 9

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

Листинг 10

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

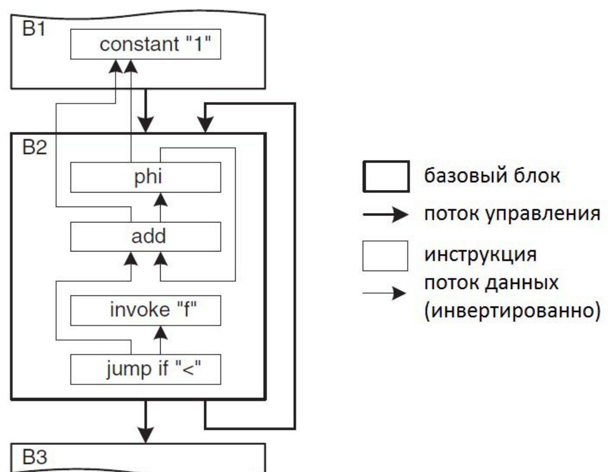

Рис. 1

Известно, что подавляющее большинство виртуальных вызовов типичного приложения при исполнении связываются с одним и тем же методом, то есть являются мономорфными. Такие вызовы не нуждаются в динамической диспетчеризации и могут быть связаны статически. Замена виртуального вызова на статический, называемая девиртуализацией, упрощает и ускоряет вызов метода, позволяет осуществлять подстановку и дальнейшую оптимизацию вызываемого кода. Сложность в определении мономорфности вызова составляет возможность динамической загрузки классов. Вызов может быть мономорфен для конкретного набора классов, но загрузка нового класса может нарушить это условие.

Рассмотрим следующий пример на языке Java (листинг 11). Если класс A не имеет загруженных подклассов, то виртуальный вызов bar в методе foo является мономорфным, а значит, может быть заменен на статический. В то же время, загрузка виртуальной машиной следующего класса нарушает мономорфность вызова (листинг 12).

Виртуальная машина имеет возможность не только использовать информацию о динамическом состоянии программы при компиляции, но и реагировать на изменение этого состояния. На этой возможности основан целый ряд спекулятивных оптимизаций. Они состоят в генерации оптимизированного кода, корректного при определенных условиях. При нарушении этих условий оптимизированная версия отбрасывается, осуществляется откат к исполнению неоптимизированной версии – деоптимизация. Впервые этот подход был описан именно для девиртуализации вызовов в языке Self.

Спекулятивная девиртуализация выглядит следующим образом:

– мономорфность вызова доказывается для набора классов доступного на текущий момент,

– компилятор осуществляет девиртуализацию вызова и применяет дальнейшие оптимизации,

– при последующей загрузке класса осуществляется проверка условия мономорфности оптимизированного вызова,

– если это условие было нарушено, осуществляется деоптимизация.

Примеры других спекулятивных оптимизаций: оптимистичное устранение проверок времени исполнения [10, 27], спекулятивный escape-анализ, спекулятивная подстановка объектов [15]. В последних двух примерах, как и в случае девиртуализации, анализ программы осуществляется, исходя из текущего набора загруженных классов. Загрузка классов в дальнейшем может повлиять на корректность осуществленных преобразований.

ЗАКЛЮЧЕНИЕ

В данной статье были рассмотрены два способа исполнения байткода: интерпретация и компиляция. Интерпретация, несмотря на низкую производительность, остается популярным способом исполнения на платформах с ограниченными ресурсами или в системах с выборочной оптимизацией. Простота реализации и переносимость являются главными преимуществами интерпретации. Динамическая компиляция на сегодняшний день является стандартом де-факто для высокопроизводительных виртуальных машин. Применение динамической компиляции позволяет добиться скорости исполнения переносимых программ сравнимой со скоростью исполнения машинного кода. Немаловажно и то, что динамическая компиляция открыла дополнительные возмож-

Листинг 11

```
void foo() {
    A p = create();
    p.bar();
}
class A {
    void bar() { ... }
}
```

Листинг 12

```
class B extends A {
    void bar() { ... }
}
```

ности по оптимизации объектно-ориентированных и динамических языков. Это особенно актуально в свете растущей популярности динамических языков, оптимизация которых на этапе статической компиляции затруднительна.

Условия работы динамического компилятора значительно отличаются от условий, в которых работает статический компилятор. Динамический компилятор конкурирует за ресурсы с исполняемой программой. Но, в то же время, он обладает большей информацией об исполняемой программе и текущей аппаратной платформе. Это привело к появлению новых алгоритмов и оптимизаций, специфичных для динамической компиляции. Основные идеи динамической компиляции, адаптивных и спекулятивных оптимизаций были заложены в рамках проектов Smalltalk и Self в 80-х – 90-х годах. Сегодня они применяются на практике повсеместно. В статье эти концепции описывались без углубления в детали конкретных реализаций. Эти детали можно найти в публикациях, описывающих архитектуры конкретных виртуальных машин. Кроме того, исходный код большинства из упомянутых виртуальных машин открыт.

Литература

1. *Matthew Arnold, Stephen Fink, Vivek Sarkar, Peter F. Sweeney.* A Comparative Study of Static and Dynamic Heuristics for Inlining / DYNAMO'00 Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, 2000. P. 52–64.

2. *Toshiaki Yasue, Motohiro Kawahito, Hideaki Kornatsu, Toshio Nakatani.* A dynamic optimization framework for a Java just-in-time compiler / OOPSLA'01 Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2001. P. 180–195.

3. *Toshio Suganuma, Toshiaki Yasue, Toshio Nakatani.* A region-based compilation technique for a Java just-in-time compiler / PLDI'03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, 2003. P. 312–323.

4. *Roger Anthony Sayle*. A Superoptimizer Analysis of Multiway Branch Code Generation / Proceedings of the GCC Developers' Summit, 2008. P. 103–116.

5. *Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, Peter F. Sweeney.* A Survey of Adaptive Optimization in Virtual Machines // Proceedings of the IEEE, 2004. Vol. 93. Is. 2. P. 449–466.

6. *Schiefler R.* An analysis of inline substitution for a structured programming language // Communications of the ACM, 1977. Vol. 20. Is. 9. P. 647–654.

7. *Kim Hazelwood, David Grove.* Adaptive Online Context-Sensitive Inlining / CGO'03 Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2003. P. 253–264.

8. *Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney.* Adaptive Optimization in the Jalapeno JVM / ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2000.

9. *Donald E. Knuth*. An Empirical Study of Fortran Programs // Software – Practice and Experience, 1970. Vol. 1. P. 105–133.

10. *Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck, Array Bounds*. Check Elimination in the Context of Deoptimization // Science of Computer Programming archive, 2009. Vol. 74. Is. 5–6, P. 279–295.

11. *Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, Mario Wolczko*. Compiling Java Just In Time // IEEE Micro archive, 1997. Vol. 17. Is. 3. P. 36–43.

12. *Urs Hölzle, Craig Chambers, David Ungar.* Debugging Optimized Code with Dynamic Deoptimization / PLDI'92 Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, 1992. P. 32–43.

13. *Stephen J. Fink , Feng Qian.* Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement / CGO'03 Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2003. P. 241–252.

14. *Guy Lewis Steele Jr., Gerald Jay Sussman*. Design of a LISP-based microprocessor // Communications of the ACM, 1980. Vol. 23. Is. 11. P. 628–645.

15. *Thomas Kotzmann, Christian Wimmer, Hanspeter Mossenbock, Thomas Rodriguez, Kenneth Russell, David Cox*. Design of the Java HotSpot™ client compiler for Java 6 // ACM Transactions on Architecture and Code Optimization (TACO) TACO Homepage archive, 2008. Vol. 5. Is. 1, № 7.

16. *T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, T. Nakatani.* Evolution of a Java just-in-time compiler for IA-32 platforms // IBM Journal of Research and Development archive, 2004. Vol. 48. Is. 5/6. P. 767–795.

17. *Ali-Reza Adi-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M.* Stichnoth, Fast, effective code generation in a just-in-time Java compiler / PLDI'98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998. P. 280–290.

18. *Robert Griesemer.* Generation of Virtual Machine Code at Startup. OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, 1999.

19. «Extensions to the C Language Family «Labels As Values» / http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html (дата обращения 30.06.12).

20. *H.J. Burkle, A. Frick, C. Schlier.* High level language oriented hardware and the post-von Neumann era / ISCA '78 Proceedings of the 5th annual symposium on Computer architecture, 1978. P. 60–65.

21. *K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, O. Lindholm.* Impact of JIT/JVM Optimizations on Java Application Performance / INTERACT '03 Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures, 2003. P. 5.

22. *Robert B. K. Dewar*. Indirect Threaded Code // Communications of the ACM, 1975. Vol. 18. Is. 6. P. 330–331.

23. «Java HotSpot™ Virtual Machine Performance Enhancements» / http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#tieredcompilation (дата обращения 30.06.12).

24. *Craig Chambers, David Ungar*. Making Pure Object-Oriented Languages Practical / OOPSLA '91 Conference proceedings on Object-oriented programming systems, languages, and applications, 1991. P. 1–15.

25. *Kevin Casey, M. Anton Ertl, David Gregg.* Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters / ACM Transactions on Programming Languages and Systems (TOPLAS), 2007. Vol. 29. Is. 6, № 37.

26. *Andrew Berlin, Henry Wu*. Scheme86: a system for interpreting scheme / Technical Report, Massachusetts Institute of Technology Cambridge, MA, USA, 1988.

27. *Lixin Su, Mikko H. Lipasti.* Speculative optimization using hardware-monitored guarded regions for java virtual machines / VEE'07 Proceedings of the 3rd international conference on Virtual execution environments, 2007. P. 22–32.

28. *Alex Iliasov*. Templates-based portable Just-In-Time compiler // ACM SIGPLAN Notices Homepage archive, Vol. 38. Is. 8. P. 37–43.

29. *Cliff Click*. Tiered Compilation, 2010.

30. *M. Anton Ertl, David Gregg*. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures / Euro-Par'01 Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, 2001. P. 403–412.

31. *David Detlefs, Ole Agesen*. The Case for Multiple Compilers / OOPSLA '99 Virtual Machine Workshop, 1999.

32. *Blair McGlashan, Andy Bower*. The Interpreter is Dead (slow). Isn't it? / OOPSLA 99 Workshop: «Simplicity, Performance and Portability in Virtual Machine Design», 1999.

33. *Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V.C. Sreedhar, Harini Srinivasan, John Whaley*. The Jalapeco Dynamic Optimizing Compiler for Java / JAVA '99 Proceedings of the ACM 1999 conference on Java Grande, 1999. P. 129–141.

34. *Michael Paleczny, Christopher Vick, Cliff Click*. The java hotspotTM server compiler // JVM'01 Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, 2001. Vol. 1. P. 1.

35. *M. Anton Ertl, David Gregg*. The Structure and Performance of Efficient Interpreters // Journal of Instruction-Level Parallelism 5? 2003. P. 1–25 .

36. *James R. Bell*. Threaded Code // Communications of the ACM, 1973. Vol. 16. Is. 6. P. 370–372.

Abstract

Bytecode execution by virtual machine has several advantages over traditional native code execution. These are portability, safety, ease of compiling and debugging. However, using virtual machine gives an additional overhead. Two ways of bytecode execution are described in this paper. These are interpretation and compilation. Bottlenecks and overheads of each of these methods are discussed. A survey of major optimizations of these methods which allows achieving execution speed comparable with native code execution speed is given.

Keywords: virtual machines, bytcode, interpretation, AOT compilation, JIT compilation, adaptive optimizations.

Пилипенко Артур Витальевич,
аспирант кафедры информатики математико-механического факультета СПбГУ,
artur.pilipenko@gmail.com

Наши авторы, 2012.
Our authors, 2012.