УДК 681.3.06

С.Ю. Гавриленко, А.Д. Драч

Национальный технический университет «ХПИ», Харьков

ДИНАМИЧЕСКАЯ ГЕНЕРАЦИЯ КОДА С ИСПОЛЬЗОВАНИЕМ ДЕРЕВЬЕВ ТРАСС

Рассмотрены динамические just-in-time (JIT) компиляторы, компилирующие в бинарный код некоторые заранее неизвестные участки кода, предназначенные изначально только для интерпретации. Проанализированы тенденции развития JIT технологий в интерпретируемых языках. Предложена оптимизация скомпилированных участков кода на основе структур деревьев трасс. Обоснован выбор технологии LLVM для генерации машинных кодов. Проанализирована производительность использования оптимизационных методик.

Ключевые слова: *динамические just-in-time (JIT) компиляторы, байт-код, деревья трас, граф управляющей логики.*

Постановка проблемы и анализ литературы

Динамические *just-in-time* (*JIT*) компиляторы – это модули программных виртуальных машин, компилирующие в бинарный код некоторые заранее неизвестные участки кода, предназначенные изначально только для интерпретации виртуальной машиной [1]. Такие выборочные участки кода являются предметом анализа во время интерпретирования программы. Компиляция происходит «на лету» без остановки выполнения программы. Очевидно, что выполнение скомпилированного бинарного кода значительно эффективнее, чем интерпретация кода программы или его промежуточного байт-кода.

Единицей компиляции для традиционных *JIT*-компиляторов является метод. Такой подход применяется, например, в виртуальной машине *Hotspot* для *Java* [2]. Предлагается рассмотреть динамическую компиляцию байт-кодов, основанную на структурах данных – *трассах*. Для них единицей компиляции является программный цикл. В перспективе, трасса может затрагивать несколько методов или даже библиотечный код, вызываемый в пределах обрабатываемого цикла. Используя новое промежуточное представление, обновляющееся во время выполнения программы, компилятор генерирует код, который в значительной степени сравним с кодом традиционных динамических компиляторов, но при этом использует меньшее машинное время для компиляции и меньший объем памяти.

Традиционные динамические компиляторы не сильно отличаются от статических компиляторов [3], за исключением того факта, что они запускаются после начала интерпретации программы. Как следствие, многие *JIT*-компиляторы используют то же промежуточное представление, что и их статические аналоги – графы управляющей логики (*control-flow graphs, CFG*) [3, 4].

Для трансляции байт-кода программы в машинный код, компилятор в первую очередь строит граф управляющей логики на основе специального анализа, и, в случае оптимизационного компилятора, выполняет ряд дополнительных действий, например, выявление информации по созданию, использованию и времени жизни переменных. Известно, что построение промежуточного представления кода в вершинах графа, к которым сходятся множество ребер – сложная задача. Ее решение требует значительное время для графов управляющей логики с большим количеством условных ветвлений. Однако построение для линейных последовательностей кода выполняется крайне быстро [5].

Рассмотрим форму представления программы с *однократным присваиванием переменных* (*Static Single Assignment, SSA*). В такой форме промежуточного представления кода каждая переменная определяется единожды [6]. Таким образом, нельзя написать:

$$z = x + y; \quad z = z + 1 .$$

Та же переменная при каждом присваивании *должна получить новое имя:*

$$z_1 = x + y; \quad z_2 = z_1 + 1 .$$

Для работы с изменяемыми переменными (например, с индексом в цикле), в байт-код в *SSA*-форме добавляется специальная инструкция $\mathfrak{I}$, которая возвращает одну из перечисленных переменных в зависимости от того, какой блок передал управление текущему блоку в управляющем графе:

$$u = \phi = (z_1, z_2) .$$

Трансформация графа управляющей логики программы в *SSA* выполняется во многих оптимизационных компиляторах, так как в такой форме удобно проводить оптимизации [7]. Процесс трансформации является тривиальной задачей, за исключением тех блоков, в которые сходятся несколько ребер графа. В такие блоки должны быть вставлены $\mathfrak{I}$-инструкции. Реализация определение места вставки $\mathfrak{I}$-инструкций достаточно сложная, а значительные временные затраты являются главной причиной

ограниченности использования *SSA*-формы в *JIT*-компиляторах. Если управляющая логика никогда не сходится в пределах всего графа, то трансформация в *SSA*-форму может быть выполнена значительно быстрее и проще [5].

Для статических компиляторов временные затраты на компиляцию не являются критичными. Программист может подождать некоторое разумное время пока программа будет переведена в машинный код. Однако в среде виртуальной машины (*virtual machine, VM*), во время интерпретирования байт-кодов медленный *JIT*-компилятор может создавать заметную задержку, которая негативно сказывается на общей производительности. Эта проблема еще более критична для встраиваемых виртуальных машин (*embedded VM*), которые обладают меньшими ресурсами, доступными для компиляции по сравнению с настольными системами [8]. *JIT*-компиляторы для встраиваемых виртуальных машин, в отличие от традиционных, должны быть небольшими и производительными.

Изложение основного материала

Рассмотрено новое промежуточное представление, эффективно работающее в смешанном режиме интерпретации байт-кода и динамической генерации машинного кода. Такое представление является разновидностью графа управляющей логики и содержит только те его ребра и узлы, которые чаще посещаются во время выполнения программы. Это так называемые значимые элементы. Классический граф содержит все возможные ребра и узлы, при этом не учитывается, как часто они используются. В конечном итоге все значимые элементы графа будут транслированы *JIT*-компилятором в машинный код. И как следствие, будут скомпилированы только лишь критически важные по производительности («горячие») участки программы.

В рассматриваемом представлении «горячие» участки программы представлены множеством связанных трасс, которые строятся на часто повторяющихся программных циклах.

Дерево трасс $\tau = (N, R)$ – ориентированный граф, представляющий множество связанных трасс в программе, где N – множество узлов (инструкций) и R – множество ориентированных ребер $\{(n_i, n_j), (n_k, n_l), ...\}$ между ними. Каждый узел $n \in N$ содержит операцию из множества действительных операций, определенных языком аппаратной (или виртуальной) машины, для которой проводится компиляция. Каждое ориентированное ребро $(n_i, n_j) \in R$ показывает, что инструкция n_j выполняется непосредственно перед инструкцией n_i. Таким образом, о ребре $(n_i, n_j) \in R$ говорят, как о ребре предшественнике. Каждый узел-лист дерева трасс представляет собой последний узел в цикле инструкций, который начинается в опорном узле и заканчивается в нем же. Трассой называется каждый из таких циклов. Опорный узел используется совместно всеми трассами, и они расходятся в деревоподобную структуру, начиная с опорного узла (рис. 1).

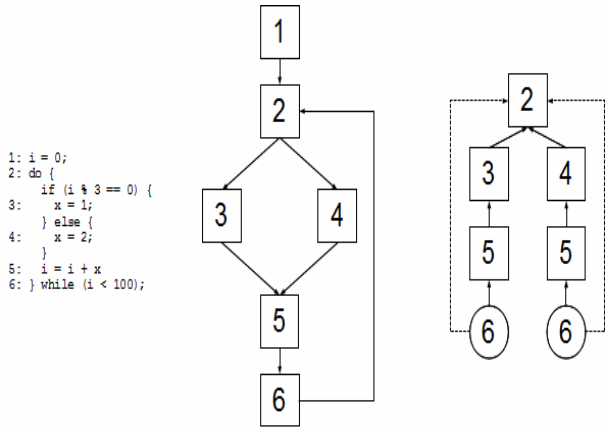

Рис. 1. Построение дерева трасс. Опорный узел №2. Трассы (2, 3, 5, 6) и (2, 4, 5, 6).

Первым шагом к построению трассы является поиск **опорного узла**. Так как рассматриваемые трассы основаны на часто повторяющихся программных циклах, то в нашем случае опорный узел – это начало цикла, его заголовок. Поиск таких потенциальных заголовков выполняется путем подсчета обратных переходов на них. Если число обратных переходов на первую инструкцию предполагаемого цикла превышает заданный порог, то такой цикл можно рассмотреть, как «горячий». В таком случае опорный узел считается найденным, и запись трассы может быть начата.

Не все операции могут быть включены в трассу. Далее приводятся условия, по которым запись трассы может быть прервана:

1. Трасса сталкивается с исключительной или редко повторяющейся семантикой. Очевидно, что такая трасса будет выполняться единожды, поэтому нет нужды ее записывать.

2. Встречаются инструкции, выполняющиеся длительное время, например, выделение памяти в куче. Длительность такой операции может нивелировать эффект от работы скомпилированной трассы. Тем не менее, программистам известен подход, по которому все вызовы по выделению памяти по возможности выносятся за пределы цикла, поэтому ожидается низкая вероятность их встречи.

3. Записанная трасса выходит слишком длинной. Для предотвращения такого условия вводится максимальная длина трассы. Узнать наперед в динамической среде длину трассы достаточно проблематично.

Определенные узлы в дереве трасс помечаются, как **боковые выходы** дерева. Боковой выход – это потенциальное направление из узла дерева, которое им не покрыто. Выполнение скомпилированной трассы может быть прервано только лишь в узле с боковым выходом. В скомпилированный код этого узла

вставляется специальный код, называемый **стражниками**, которые проверяют условие выхода из трассы и перехода к простому интерпретированию с анализом новых трасс. Боковых выходов должно быть как можно меньше, и в идеале, такие узлы должны встречаться только лишь в листьях дерева. Нередко встречаются в программах вложенные циклы. При построении дерева нужно учитывать тот факт, что дерево трасс содержит лишь один опорный узел, который соответствует началу цикла. Таким образом, дерево необходимо строить, начиная с внутреннего цикла (анализатор определяет его первым, как «горячий»). С таким подходом внешние циклы рассматриваются, как продолжение трассы, начатой во внутреннем цикле. Важно при этом отслеживать два ограничения:

1. Трассы в циклах не должны быть слишком длинными.
2. Количество обратных переходов во внешних циклах должно быть ограничено, иначе один и тот же развернутый внешний цикл будет часто повторяться в трассе.

Методы и функции при построении дерева трасс встраиваются (*inlining*), т.е. код вызова функций не генерируется. В случае обработки виртуальных функций, вставляется простое условие, какой именно объект иерархии классов вызывается. Другими словами, виртуальность методов сводится к простому оператору выбора – *switch*. При построении дерева важно следить за тем, чтоб опорные узлы и листья дерева находились в одной области видимости. На операторах типа *return* запись трассы будет прервана.

Код в дереве трасс записывается в измененной *SSA*-форме. Изменение касается расположения $\mathfrak{I}$-инструкции, которая вставляется исключительно в лист трассы. Такой подход предотвращает образование коллизий в опорном узле, в случае, если действовать стандартно – вставлять $\mathfrak{I}$-инструкции в начало цикла (опорный узел) для изменения индексных переменных (рис. 2). Линейный характер трасс дерева позволяет крайне быстро их компилировать и перестраивать все дерево. Это важно, т.к. любое изменение в дереве, например, добавление новой трассы, должно заканчиваться его перекомпилированием.

Удобная *SSA*-форма инструкций позволяет проводить ряд оптимизаций, как то **устранение неиспользуемого кода** *(dead code elimination, DCE)*. Компиляция трассы проводится в обратном порядке – от листьев к опорному узлу. Таким образом, можно определить те переменные, которые впервые объявлены, но нигде ранее по ходу компиляции не использовались.

Также возможно применить ряд других оптимизаций, таких как:

- сворачивание констант, т.е. их подсчет без генерации математических инструкций;
- удаление общих подвыражений, т.е. выявление подвыражений, которые можно вынести и подсчитать отдельно.

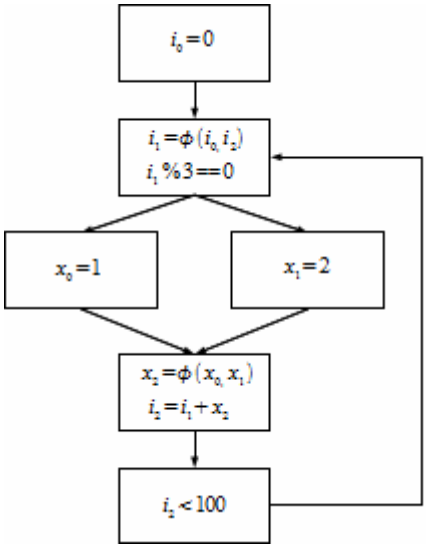

Рис. 2. Пример стандартного подхода вставки $\mathfrak{I}$-инструкций и нового – в листья

Для апробации техники *JIT*-компилирования было выбрано подмножество языка программирования *Lua*. Язык широко используется как скриптовый язык в программах обработки компьютерной графики, встраиваемых устройствах [9]. Подмножество не затрагивает неординарные возможности семантики языка, например, функции не рассматриваются в качестве сопрограмм (*coroutines*) или замыканий (*closures*). Также упрощена многоуровневая поддержка единственной структуры данных — хеш-подобных таблиц.

Выбор Lua основывается на том, что этот язык прост для экспериментов. Семантика языка не усложнена тяжелыми конструкциями. Кодовая база хорошо прокомментирована и документирована, сообщество разработчиков активное.

Был написан оптимизирующий интерпретатор описанного подмножества языка Lua. Реализация выполнена в языковой среде *Python* с использованием библиотеками *PLY* для семантического и синтаксического анализа входных текстов, *llvm-py* для генерации машинного кода в режиме *JIT*-компилятора. Python используется в целях упрощения разработки, т.к. для полноценной низкоуровневой разработки интерпретатора или компилятора на C требуется большое количество человеко-часов. Инфраструктура *LLVM (Low Level Virtual Machine)* используется для генерации машинных x86 кодов на лету во время интерпретации. Промежуточный код *LLVM* поддерживает *SSA*-форму и $\mathfrak{I}$-инструкции. Ручная же генерация машинных кодов и тестирование их корректности – длительный процесс.

Для сравнения производительности интерпретатора в режиме простого выполнения байт-кодов и в режиме задействованного *JIT*-компилятора были написаны элементарные синтетические тесты: расчет фрактала Мальдеброта и поиск простых чисел.

Оба теста используют циклы с ветвлениями, на которых можно показывать оптимизационную работу деревьев трасс.

Было показано, что время выполнения тестов интерпретатора с включенным *JIT* компилятором в среднем в 1,9 и 2,5 раза меньше соответственно. Пример сеанса работы следующий (параметр "–jit" здесь включает оптимизацию):

```
    artem@fish2:~/benchmarks/lua$
./measure_time.py
    *** pylua.py ./mandelbrot_simple.lua
    Time: 701.54671669 seconds
    *** pylua.py -jit
./mandelbrot_simple.lua
    Time: 274.894404411 seconds
    pylua.py        702s  2.55
    pylua.py -jit   275s  1.00

    *** pylua.py
./prime_numbers_simple.lua
    Time: 1365.13750553 seconds
    *** pylua.py -jit
./prime_numbers_simple.lua
    Time: 710.635399818 seconds
    pylua.py        1365s  1.92
    pylua.py -jit    711s  1.00
```

Следует сказать, что включение подсистемы *LLVM* для генерации машинных кодов занимает значительное время. Это оправдано в нашем случае, т.к. демонстрируется реализация экспериментальных идей.

Выводы

В данной статье предложено представление промежуточного кода виртуальных машин в виде деревьев трасс. Динамическая *JIT*-компиляция, проводимая по структурам деревьев трасс, существенно уменьшает время выполнение интерпретационного кода, что было продемонстрировано на примере подмножестве языка *Lua*. Использование модифицированной *SSA*-формы кода, позволяет реализовать множество дополнительных оптимизаций «на лету». Создано и протестировано программное приложение, которое позволило сравнить производительность интерпретатора в режиме простого выполнения байт-кодов и в режиме задействованного JIT компилятора. Данная разработка может быть использована для увеличения производительности работы интерпретаторов.

Целью дальнейших исследований авторов является подбор и реализация таких оптимизационных техник на основе трасс, которые не замедляют выполнение входных программ.

Список литературы

1. A Survey of Adaptive Optimization in Virtual Machines / M. Arnold, S.J. Fink, D. Grove, M. Hind, P.F. Sweeney. – 2005.

2. Sun Microsystems. The Java HotSpot Performance Engine Architecture [Электронный ресурс]. – Режим доступа к ресурсу: http://java.sun.com/products/hotspot/whitepaper.html.

3. Ахо А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман. – М.: Издательский дом "Вильямс", 2003.

4. Aycock J. A Brief History of Just-In-Time / J. Aycock. – 2003.

5. FSF. Control Flow – GNU Compiler Collection (GCC) Internals / [Электронный ресурс]. – Режим доступа к ресурсу: http://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html.

6. Gal A. HotpathVM: an Effective JIT Compiler for Resource-Constrained Devices / A. Gal, C. W. Probst, M. Franz. – 2006.

7. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph / R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. – 1991.

8. Novillo D. Tree SSA: A New Optimization Infrastructure for GCC / D. Novillo. – 2003.

9. Sun Microsystems. CLDC HotSpot Implementation Virtual Machine White Paper [Электронный ресурс]. – Режим доступа к ресурсу: http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf.

10. Ierusalimschy R. The Evolution of Lua / R. Ierusalimschy, L.H. de Figueiredo, W. Celes. – 2007.

Поступила в редколлегию 15.03.2010

Рецензент: д-р техн. наук, проф. И.В. Гребенник, Харьковский национальный университет радиоэлектроники, Харьков.

ОПТИМІЗАЦІЯ МАШИННО-НЕЗАЛЕЖНОГО КОДУ ПРОГРАМ

С.Ю. Гавриленко, А.Д. Драч

Розглянуто динамічні just-in-time (JIT) компілятори, що компілюють в бінарний код деякі заздалегідь невідомі ділянки коду, які призначалися спочатку лише для інтерпретації. Проаналізовані тенденції розвитку JIT технологій в мовах, що інтерпретуються. Запропонована оптимізація скомпільованих ділянок коду на основі структур дерев трас. Обґрунтований вибір технології LLVM для генерації машинних кодів. Проаналізована швидкодія використання оптимізаційних методик.

***Ключові слова**: динамічні just-in-time (JIT) компілятори, байт-код, дерева трас, граф управляючої логіки.*

MACHINE-INDEPENDENT PROGRAM CODE OPTIMIZATION

S.J. Gavrilenko, A.D. Drach

Just-in-time (JIT) compilers are reviewed. They are applied for compiling an interpreted code, that is unknown ahead of execution time, in a binary code. Tendencies of JIT development in a field of interpreted languages are analyzed. Optimization of a compiled code based on trace tree structures is offered. A choice of the LLVM technology for machine codes dumping is grounded. Performance of optimization methods is analyzed.

***Keywords:** dynamic just-in-time (JIT) compiler, byte code, trees tracy, control-flow graphs.*