УДК 004.051
DOI: 10.25559/SITITO.17.202104.988-997

Научная статья

Автоматическая генерация интерпретатора для многоязыковой виртуальной машины

М. Г. Гонопольский
ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Российская Федерация
Адрес: 119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1
gmarkmgw@yandex.ru

Аннотация

В данной статье проведено исследование существующих реализаций генераторов интерпретаторов и предложен подход к генерации интерпретаторов на ассемблере для нескольких архитектур по описанию множества инструкций (BISA – Bytecode Instruction Set Architecture) многоязыковой регистровой виртуальной машины (VM – Virtual Machine) с интерпретатором, компилятором и сборщиком мусора. Каждой инструкции из BISA однозначно соответствует обработчик BISA. Для каждого обработчика BISA задан номер, определяющий положение обработчика, частота использования соответствующей инструкции в множестве целевых приложений (МЦП), и набор инструкций псевдоязыка, характеризующий логику обработки соответствующей инструкции BISA виртуальной машиной. МЦП состоит из приложений, предназначенных для запуска на рассматриваемой виртуальной машине. Интерпретатор представляет собой последовательный набор обработчиков BISA. Согласно выбранному автором подходу, для получения интерпретаторов на ассемблере по BISA необходимо выполнить три последовательных шага. На первом шаге осуществляется преобразование обработчиков BISA в промежуточное представление интерпретатора. На втором шаге проводятся машинно-независимые оптимизации промежуточного кода интерпретатора. На третьем шаге выполняются машинно-зависимые оптимизации и преобразование промежуточного представления интерпретатора в представления выбранных архитектур. Было разработано инструментальное средство на языке C++, реализующее упрощенный, не включающий оптимизации, вариант предложенного подхода для двух инструкций. Проведено экспериментальное исследование, в ходе которого предложенный генератор сравнивался с компилятором clang++ по количеству генерируемых инструкций целевой машины для каждого обработчика BISA. Экспериментальное исследование показало, что выбранные подходы эквивалентны, даже при использовании упрощенного алгоритма генерации интерпретаторов. В перспективе планируется оптимизировать предложенный подход, что позволит превзойти по производительности известные подходы.

Ключевые слова: генерация интерпретаторов, регистровая многоязыковая виртуальная машина, интерпретатор, язык ассемблера, компилятор

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Гонопольский М. Г. Автоматическая генерация интерпретатора для многоязыковой виртуальной машины // Современные информационные технологии и ИТ-образование. 2021. Т. 17, № 4. С. 988-997. doi: https://doi.org/10.25559/SITITO.17.202104.988-997

Original article

Automatic Generation of Interpreter for Multilingual Virtual Machine

M. G. Gonopolskiy
Lomonosov Moscow State University, Moscow, Russian Federation
Address: 1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation
gmarkmgw@yandex.ru

Abstract

The article says about existing implementations of interpreter generators and proposes an approach for generating interpreters in assembly for several architectures by describing a set of instructions (BISA – Bytecode Instruction Set Architecture) of a multilingual stack virtual machine (VM – Virtual Machine) with an interpreter, compiler and garbage collector. Each instruction from BISA has a unique BISA handler. For each BISA handler, there are a specified number that defines its position, a frequency of usage of the corresponding instruction in the set of target applications (STA), and a set of pseudo-language instructions that characterizes the logic of handling the corresponding BISA instruction by the virtual machine. STA consists of applications designed to be run on the given virtual machine. An interpreter is a sequential set of BISA handlers. According to the approach chosen by the author, to obtain interpreters in assembly using BISA, it is necessary to do three consecutive steps. During the first step, the BISA handlers are converted into an intermediate representation of the interpreter. In the second step, machine-independent optimizations of the intermediate code of the interpreter are performed. At the third step, the intermediate representation of the interpreter is converted into representations of the selected architectures and machine-dependent optimizations are performed. A C ++ tool was developed that implements a simplified, non-optimized version of the given approach for two instructions. An experimental study was carried out, during which the proposed generator was compared with the clang ++ compiler in terms of the number of generated instructions of the target machine for each BISA processor. Experimental research has shown that the selected approaches are equivalent, even when using a simplified algorithm for generating interpreters. In the longer run, it is planned to optimize the proposed approach, which will allow surpassing the known approaches in performance.

Keywords: interpreter generation, register multilingual virtual machine, interpreter, assembly language, compiler

The author declares no conflict of interest.

For citation: Gonopolskiy M.G. Automatic Generation of Interpreter for Multilingual Virtual Machine. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2021; 17(4):988-997. doi: https://doi.org/10.25559/SITITO.17.202104.988-997

1. Введение

В данной статье рассматривается многоязыковая многопоточная регистровая [1-2] виртуальная машина [2-5] (ВМ), поддерживающая параллелизм и статически [6-8] и динамически [8-9] типизированные языки, включающая в себя компилятор [10], позволяющий компилировать исходный код программы под соответствующую архитектуру; сборщик мусора [10-11], обеспечивающий очистку памяти; и интерпретатор [10, 12], исполняющий код программы в реальном времени.

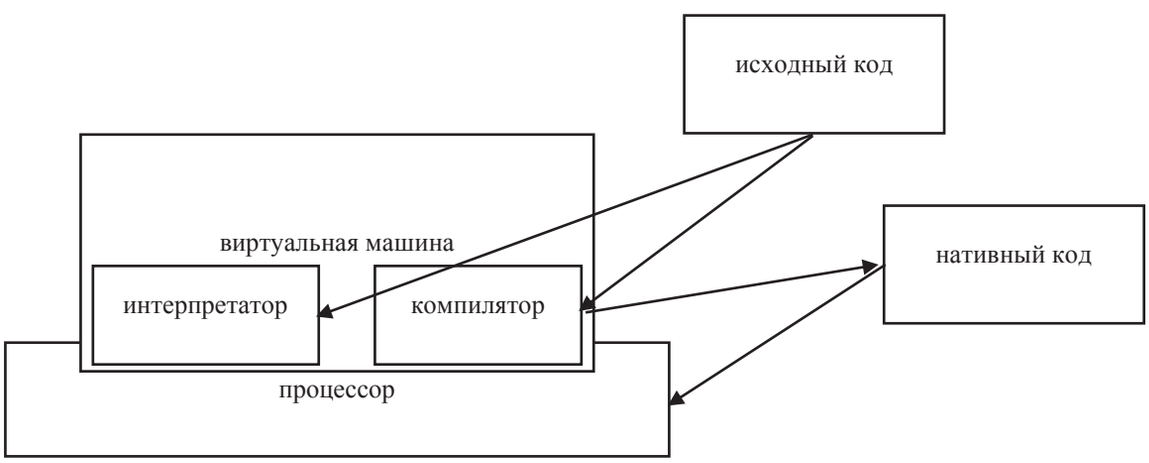
Для ВМ задан набор инструкций (BISA – Bytecode Instruction Set Architecture), которые она поддерживает. Каждой инструкции из BISA однозначно соответствует обработчик BISA и набор характеристик, определяющих природу инструкции, принимаемые параметры и т.д. Для каждого обработчика BISA задан номер, определяющий положение обработчика, частота использования соответствующей инструкции в множестве целевых приложений (МЦП), и набор инструкций псевдоязыка (IRH – Intermediate Representation of Handler), характеризующий логику обработки соответствующей инструкции BISA виртуальной машиной. МЦП состоит из приложений, предназначенных для запуска на рассматриваемой виртуальной машине.

Под конфигурацией интерпретатора понимается совокупность характеристик и взаимосвязей обработчиков инструкций и инструкций из BISA: порядок расположения обработчиков, частоты использования инструкций, IRH обработчиков, характеристики инструкций BISA.

Под архитектурой Arch системы будем понимать архитектуру процессора, на котором исполняется ВМ. Для каждой Arch задан набор инструкций (ISA – Instruction Set Architecture), которые способен обрабатывать процессор [13-15].

Виртуальная машина обычно позволяет исполнять одну и ту же программу на системах с разной архитектурой. Это позволяет пользователю писать программы, не задумываясь о реализации тех или иных системных функций и устройстве процессора. Однако даже системы с одинаковой архитектурой могут отличаться друг от друга и одно из самых значительных отличий – это количество оперативной памяти. В системах с большим количеством памяти виртуальная машина может не использовать интерпретатор, поскольку в нем нет нужды, ведь есть возможность скомпилировать код всей программы в нативный код ISA Arch и исполнять его напрямую на процессоре. В иных системах, с меньшим количеством памяти, это невозможно, в связи с чем принимается решение использовать интерпретатор, поскольку он занимает значительно меньше места.

Многие виртуальные машины изначально не содержали интерпретатора, но эта проблема вынудила разработчиков добавить его в продукт [15]. Однако, за экономию памяти приходится расплачиваться скоростью: интерпретировать программу значительно медленнее, чем исполнять напрямую на процессоре. В связи с этим возникает новая проблема, заключающаяся в ускорении процесса интерпретации.



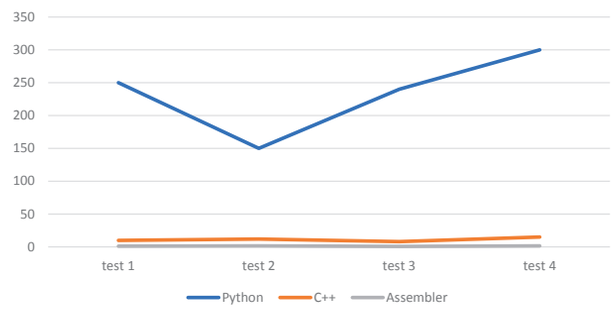
Р и с. 1. Виды исполнения программы

F i g. 1. Types of program execution

Существуют разные подходы к ускорению интерпретатора. К одним из наиболее важных можно отнести способ, заключающийся в написании интерпретатора на ISA Arch. Согласно исследованию проведенному в разделе «Обзор работ», интерпретатор, написанный на ассемблере вручную обычно значительно быстрее аналогов, получающихся путем компиляции высокоуровневых программ, даже если они написаны на таких языках как C или C++. Кроме того, имея такой интерпретатор, можно согласовать некоторые договоренности с другими частями платформы (например, со сборщиком мусора), что позволит ускорить выполнение программ.

Именно такой способ, в совокупности с некоторыми видами оптимизаций [16-18], обеспечивает максимальную скорость исполнения программ путем интерпретации.

Ниже приведен график сравнения разных реализаций интерпретаторов на простой программе с циклами.



Р и с. 2. Сравнение реализаций интерпретаторов (switch-case)
F i g. 2. Comparison of interpreter implementations (switch-case)

Приведенный пример демонстрирует, что интерпретатор, написанный на ассемблере, действительно превосходит свои аналоги по скорости исполнения, хотя все еще довольно сильно отстаёт от варианта с прямым исполнением программы на процессоре. Кроме того, у этого подхода есть свои проблемы:

1. Написание программы на ассемблере – довольно трудоемкий процесс, на который требуется значительное количество человеко-часов.
2. Для каждой архитектуры необходим собственный интерпретатор.

Для решения этих задач применяют следующий подход – автоматическую генерацию интерпретаторов. Заключается он в том, чтобы один раз заданным образом описать интерпретатор, после чего специальное программное средство сгенерирует на основе этого описания несколько интерпретаторов под заданные архитектуры. Однако реализовать генератор, который способен получать корректный и эффективный интерпретатор – это большая проблема, которая, вообще говоря, не решена.

Согласно выполненному автором настоящей статьи обзору, результаты которого приведены в разделе «Обзор существующих методов», не существует методов, в полной мере реализующих генерацию ассемблерного интерпретатора для рассматриваемой в данной работе ВМ. В связи с чем был разработан собственный принцип, позволяющий получить интерпретатор на языке ассемблера по описанию BISA.

Предлагаемый метод состоит в последовательном выполнении следующих шагов:

1. Задание описания BISA.
2. Трансляция заданного описания BISA в интерпретатор на промежуточном платформо-независимом языке.
3. Выполнение машинно-зависимых оптимизаций.
4. Трансляция промежуточного интерпретатора в интерпретатор на ISA Arch для всех заданных архитектур и выполнение машинно-зависимых оптимизаций для каждого получившегося интерпретатора.

2. Постановка задачи генерации интерпретатора

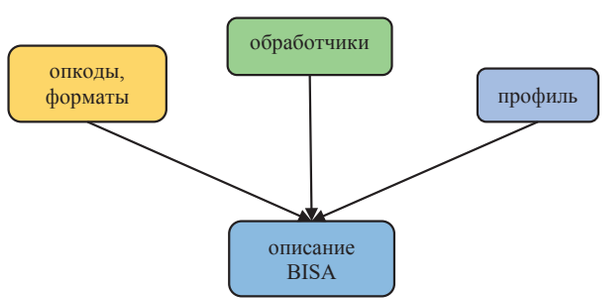
Каждая виртуальная машина, как, собственно, и каждый процессор, реализует свою собственную схему вычислений, выражаемую в наборе операций, называемых опкодами [19]. Они задают BISA.

Каждый опкод регламентирует преобразование данных. Например, опкод add позволяет складывать числа. Так результатом выполнения следующей команды: add a, b – будет сумма a + b.

Каждый опкод имеет собственное описание на архитектуро-независимом языке. Это описание мы будем называть обработчиком, оно регулирует работу виртуальной машины при обработке соответствующей инструкции. Так же опкод может иметь специальные флаги, оповещающие о формате соответствующих инструкций. Например, флаг call может говорить о том, что это инструкция вызова метода, а флаг mem_write – о том, что данная инструкция записывает какую-то информацию в память машины.

Помимо этого, опкод может иметь профильную информацию, такую как частота использования на заданном наборе тестов, или другие статистические данные.

Таким образом, описание BISA – это набор опкодов, которые могут быть обработаны виртуальной машиной, их обработчики, форматы и профильная информация.



Р и с. 3. Описание BISA ВМ
F i g. 3. Description of BISA VM

Целевым интерпретатором будем называть интерпретатор, написанный на языке ассемблера, ориентированный на определенную архитектуру. Часто виртуальная машина предназначена для работы на многих архитектурах, что требует нескольких целевых интерпретаторов.

Обобщенный целевой интерпретатор – это набор целевых интерпретаторов для всех целевых архитектур для данной виртуальной машины.

Минимальный по времени обобщенный целевой интерпретатор – это обобщенный целевой интерпретатор, который показывает лучшую производительность по времени исполнения на наборе целевых тестов, выбираемых вручную.

Таким образом задача заключается в следующем: требуется разработать метод, который позволит для виртуальной машины рассматриваемого типа сгенерировать минимальный обобщенный целевой интерпретатор по целевым тестам и описанию BISA.

3. Обзор существующих методов

Целью обзора являлся поиск метода построения эффективного интерпретатора, для виртуальной машины, удовлетворяющей следующим критериям обзора:

1. ВМ должна быть многоязычной.

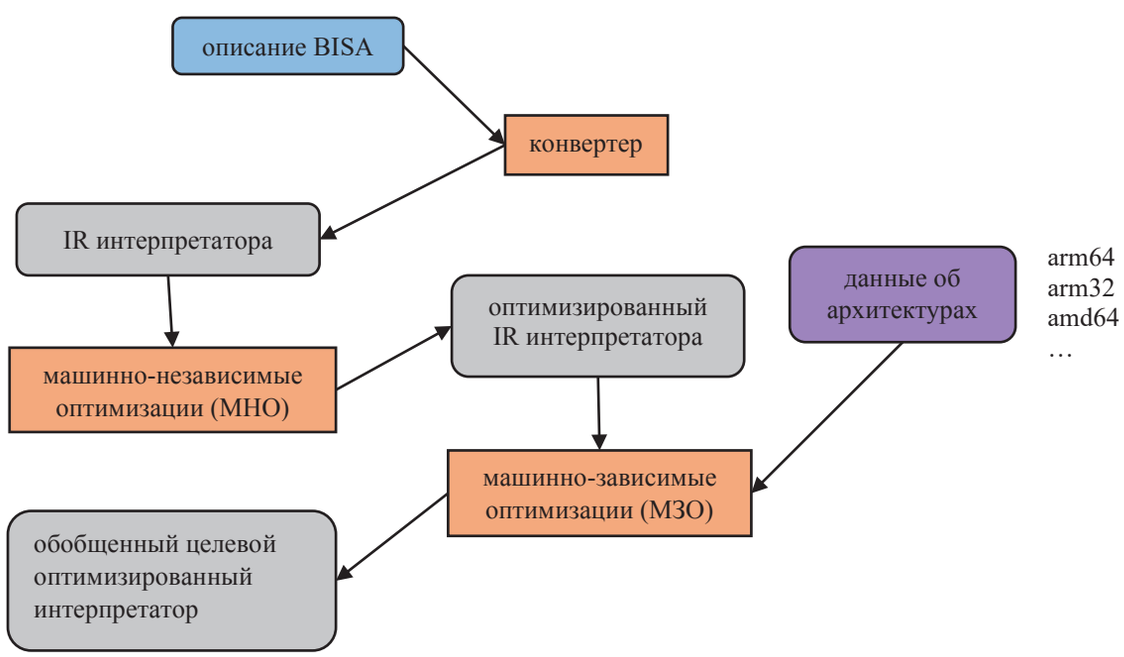
2. Виртуальная машина должна поддерживать языки со статической и динамической типизацией.
3. ВМ должна быть регистровой.
4. ВМ должна включать компилятор, интерпретатор и сборщик мусора.
5. Интерпретатор виртуальной машины должен быть сгенерирован из промежуточного независимого от платформы представления в архитектурно-зависимый ассемблерный код.
6. ВМ должна быть многопоточной.
7. ВМ должна поддерживать многопоточные языки программирования.
8. ВМ должна поддерживать сопрограммы.
9. ВМ должна иметь открытый исходный код и / или подробную документацию.

В следующей таблице представлены результаты проверки удовлетворения существующих известных реализаций виртуальных машин выбранным критериям.

Т а б л и ц а 1. **Обзор реализаций ВМ**

T a b l e 1. **Overview of VM Implementations**

Реализации	1	2	3	4	5	6	7	8	9
V8	+	+	+	+	+	+	-	+	+
ART	+	+	+	+	-	+	+	+	+
Core CLR	+	+	-	+	-	+	+	+	+
Hermes	+	+	+	-	-	+	-	+	+
Lua JIT	-	-	+	+	-	-	-	+	+
OpenJDK	+	+	-	+	-	+	+	+	+
GraalVM	+	+	-	+	-	+	+	+	+
JavaScript Core	+	+	+	+	+	+	-	+	+



Р и с. 4. Описание подхода

F i g. 4. Description of the approach

Ни одна из рассмотренных реализаций не удовлетворяет в полной мере критериям описанным в начале раздела, однако многие из них имеют части кода, которые генерируются из некоторого промежуточного представления, в том числе интерпретаторы.

Помимо реализаций ВМ от крупных компаний, существуют подходы, разработанные и описанные в научных работах. Не все из них находят применение на практике, однако многие из них выражают интересные идеи, которые могут быть использованы для разработки нового подхода или доработки иных подходов.

Важным замечанием является то, что реализация генерирования интерпретатора обычно рассматривается в контексте генерирования всей виртуальной машины или ее частей [20], таких как компилятор и часть инструментария. Так же, существует довольно мало научных работ, по которым можно судить о структуре генератора и применяемых оптимизациях. Подробное описание процесса генерации обычно встречаются в работах, подкрепленных реальными реализациями [20].

Чаще всего встречаются работы, в которых либо приводятся результаты работы сгенерированного интерпретатора, либо краткое описание, не говорящее обо всем множестве применяемых оптимизаций и принципах и тонкостях их реализации. Примером подобных работ могут служить следующие статьи [21-23], в них рассматривается задача генерирования части платформы, некоторые из них не включают в себя интерпретатор, например [22].

Однако есть статьи, подкрепленные примерами реализаций, например [24-28]. Авторы данных статей генерируют виртуальную машину на основе некоторого описания, однако их конфигурация интерпретатора генерируется в программу на языке C, что не удовлетворяет критериям текущего обзора. В дальнейшем оптимизации из данных работ могут найти применение для достижения цели поставленной задачи.

На основании обзора можно сделать вывод, что среди существующих методов нет подходящего для решения поставленной в данной статье задачи. В связи с этим было принято решение разработать собственную схему генерации интерпретатора.

4. Описание подхода

Предлагаемый метод состоит в последовательном выполнении шагов, показанных на рисунке 4.

4.1. Получение IR интерпретатора

На вход методу подается описание BISA ВМ, включающее в себя список опкодов, форматы инструкций, реализации обработчиков и профильную информацию.

Обработчики опкодов, как было сказано ранее, определяют поведение интерпретатора в процессе обработки инструкций. Иными словами, они задают логику исполнения, используя специальный язык (специально разработанный язык, существующий язык программирования, макроассемблер – любой) или программный интерфейс. Например, рассмотрим инструкцию ADD_64 vd1, vs1, которая суммирует содержимое регистров vd1 и vs1 и кладет сумму в регистр vd1.

Ее обработчик на макроассемблере может выглядеть следующим образом:

```
v0: frame_ptr # регистр v0 содержит указатель на фрейм
v1: inst_ptr  # регистр v1 содержит указатель на инструкцию в памяти
Load64 v2, v1, offset_vd1_inst # загружаем в регистр v2 номер регистра vd1
Load64 v3, v1, offset_vs1_inst # загружаем в регистр v3 номер регистра vs1
Load64 v4, v0, v2, reg_size # загружаем в регистр v4 значение соотв. размера из фрейма
Load64 v5, v0, v3, reg_size
Add64 v6, v4, v5 # суммируем содержимое регистров v4 и v5 и кладем в регистр v6
Store64 v0, v2, v6 # кладем значение регистра v6 во фрейм по смещению регистра vd1
```

Регистры, используемые в примере – виртуальные. Они не имеют прямого отношения к регистрам машины, их количество задается разработчиками ВМ.

На основе этих данных строится промежуточное представление (IR – intermediate representation) интерпретатора, представляющее собой полный код интерпретатора на некотором архитектуро-независимом языке, как правило, более низкоуровневом и обладающем более подробным синтаксисом, чем язык описания обработчиков. Например, для этих целей можно переиспользовать IR компилятора. В таком случае код рассмотренного выше опкода в IR интерпретатора будет выглядеть так:

```
1. LoadByOffset inst_ptr, offset_vd1_inst[int8]
2. Mul (1), reg_size
3. LoadByOffset inst_ptr, offset_vs1_inst[int8]
4. Mul (3), reg_size
5. LoadByOffset frame_ptr, (2)[int64]
6. LoadByOffset frame_ptr, (4)[int64]
7. Add64 (5), (6)
8. StoreByOffset frame_ptr, (2)[int64], (7)
... # далее выбор следующей инструкции и переход на ее выполнение
```

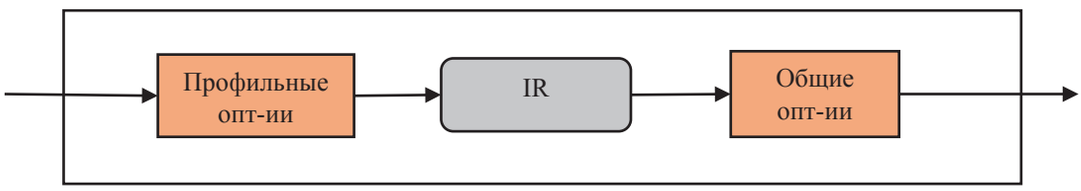
Так же, сохраняется профиль и другая информация об инструкциях, которая будет использоваться в оптимизациях.

4.2. Машинно-независимая оптимизация (МНО)

Данный процесс преобразует IR интерпретатора, проводя оптимизации, которые не учитывают особенности архитектуры, на которой будет выполняться интерпретатор.

Такие оптимизации делятся на две группы:

1. Оптимизации, основанные на профильной информации. Поскольку профильная информация собирается на некотором множестве целевых приложений, такого рода оптимизации нацелены на повышение эффективности интерпретатора на этом множестве и могут не учитывать особенностей приложений, не входящих в него. Во множество таких оптимизаций может входить, например, нахождение быстрых путей для исполнения (fastpath) и переписывание обработчиков опкодов.
2. Общие оптимизации. Нацелены на общее повышение производительности на любом приложении. Например, peephole-оптимизации.



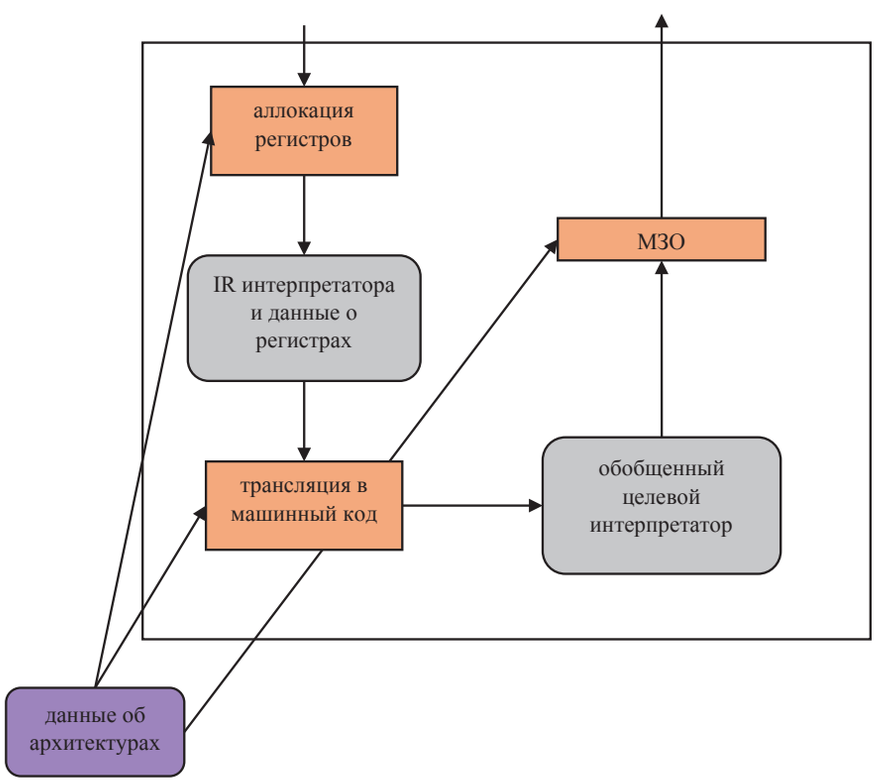
Р и с. 5. Схема МНО
F i g. 5. MIO (Machine-Independent Optimization) scheme

4.3. Машинно-зависимая оптимизация (МЗО)

После получения IR интерпретатора и проведения МНО, из него можно получить несколько интерпретаторов на языке ассемблера, по одному для каждой из целевых архитектур. Идеологически, этот шаг состоит из нескольких этапов, однако, как все из них, так и некоторые могут быть объединены в один:

1. Распределение регистров. Для того, чтобы получить интерпретатор, написанный под соответствующую архитектуру, необходимо распределить машинные регистры так, чтобы их использование было наиболее эффективным. Этот процесс называют аллокацией регистров. Данные о регистрах для каждой из архитектур сохраняются в памяти.
2. Трансляция в машинный код. После получения информации о том, как лучше распределить регистры, появляется возможность полностью оттранслировать IR интерпретатора в код на языке ассемблера для каждой соответствующей архитектуры – получить обобщенный целевой интерпретатор.
3. Проведение машинно-зависимых оптимизаций. Такие оптимизации, как и МНО, делятся на два типа:

1. Оптимизации на основе профильной информации. Например, переставление местами обработчиков опкодов для наиболее эффективного использования кэша. Однако такая оптимизация не может использоваться в реальном времени, поскольку она теряет свою релевантность после того, как каждому обработчику назначается собственный порядковый номер, иначе говоря, после «заморозки» ISA. Иначе для каждой версии интерпретатора придется заново собирать приложение.
2. Общие оптимизации. Например, перестановка инструкций, позволяющая более эффективно исполнять код на процессоре.



Р и с. 6. Схема трансляции IR
F i g. 6. Translation scheme of IR interpreter and MDO (Machine-Dependent Optimization)

4.4. Выводы

В результате проделанных шагов можно получить обобщенный целевой интерпретатор для выбранного множества архитектур, эффективность которого зависит от примененных оптимизаций, способа распределения регистров и метода трансляции. Однако вопрос о его минимальности на данный момент остается открытым и может быть разрешен путем создания аналитического или основанного на случайном переборе метода оценки времени выполнения, с учетом вызова скомпилированных функций. Так же, он может быть частично разрешен, обеспечив получение интерпретатора близкого по свойствам к минимальному.

Однако, для доказательства практической ценности генератора достаточно того, что получаемый им интерпретатор будет превосходить текущее решение, например, интерпретатор, написанный на C++.

5. Программная реализация

Данная работа выполняется для компании Huawei Technologies Co. Ltd. и будет являться частью виртуальной машины, удовлетворяющей критериям обзора настоящей статьи[1].

Для тестирования начальной стадии разработки программного средства и настройки параметров был разработан прототип генератора интерпретаторов, генерирующий интерпретатор для двух опкодов используемой ISA. Данная реализация не включает в себя оптимизации и использует простой алгоритм аллокации регистров – Linear Scan, заимствованный из библиотеки генерации кода действующего компилятора.

6. Тестирование генератора

Вопрос о релевантности использования генератора интерпретаторов на ассемблере очень важен и сводится к тому, во сколько раз сгенерированный интерпретатор будет превосходить свой хорошо оптимизированный аналог, написанный на C++.

Идеологически, интерпретатор на ассемблере имеет больше преимуществ, так как компиляторы C++ не всегда генерируют качественный код. Кроме того, используя сторонний компилятор нельзя рассчитывать на некоторые виды договоренностей, которые могут дать большой прирост в производительности. Особенно, если речь идет о множестве архитектур, а не об одной.

С другой стороны, генератор интерпретаторов фактически представляет собой компилятор, которой компилирует высокоуровневое представление интерпретатора в ассемблерный код под соответствующую архитектуру. И для того, чтобы обеспечить качественный ассемблерный интерпретатор нужно реализовать множество оптимизаций, в том числе архитектуро-зависимых и основанных на статистических данных.

Без таких оптимизаций далеко не всегда может получиться результат сравнимый с компилятором C++, который, в каком-то смысле, тоже может считаться генератором интерпретаторов, если в качестве высокоуровневого описания интерпретатора использовать программу на C++.

Данный прототип не содержит оптимизаций и использует примитивный алгоритм аллокации регистров, однако его результаты для выбранных двух опкодов равны с результатами компилятора C++: clang++ (с флагом -O2) для архитектуры amd64.

Результаты тестирования вы можете видеть в следующей таблице:

Таблица 2. **Результаты тестирования генераторов**

Table 2. **Generator test results**

Название	clang++	ITA
LDA.8	10	10
MOV.4.4	14	13

LDA.8 – Инструкция загрузки содержимого 8-ибитного регистра в аккумулятор;

MOV.4.4 – Инструкция перемещающая данные из одного 4-хбитного регистра в другой 4-хбитный регистр.

В данном случае под «битностью» регистра понимается количество битов, отведенное на номер регистра. То есть 4-хбитный регистр – это регистр, имеющий номер с 0 по 15, 8-ибитный: с 16 по 255 и т.д.

Сравнение проводилось по количеству генерируемых ассемблерных команд на обработчик опкода. Выяснилось, что при текущей реализации интерпретатора генерируется одна лишняя ассемблерная команда, в остальном результаты идентичны. Из этого можно сделать вывод, что скорость исполнения двух рассмотренных опкодов будет одинаковой при использовании обоих решений.

7. Заключение

В данной статье предложен собственный подход к задаче генерации интерпретатора для многоязыковой многопоточной регистровой виртуальной машины с компилятором и сборщиком мусора, поддерживающей параллелизм и статические и динамические языки. Разработан прототип, частично реализующий этот подход и проведено тестирование, показавшее, что даже без оптимизаций для некоторых опкодов текущий подход дает сравнимые результаты с компилятором clang++.

В дальнейшем планируется доработать предложенный подход: реализовать полное программное средство, позволяющее генерировать интерпретаторы на ассемблере для трех архитектур: amd64, arm64, arm32; включить в реализацию оптимизации на основе статистики, зависимые и независимые от архитектуры оптимизации и авторский алгоритм распределения регистров. Кроме того, планируется провести экспериментальное исследование на основе математического аппарата статистических гипотез.

[1] Gitee: Ark Runtime Core [Электронный ресурс] // OSCHINA, 2021. URL: https://gitee.com/openharmony/ark_runtime_core (дата обращения: 14.10.2021).

References

[1] Schoeberl M. Design and implementation of an efficient stack machine. *19th IEEE International Parallel and Distributed Processing Symposium.* IEEE Computer Society; 2005. p. 1-8. (In Eng.) doi: https://doi.org/10.1109/IPDPS.2005.161

[2] Marques I.L., Ronan J., Rosa N.S. TinyReef: a register-based virtual machine for Wireless Sensor Networks. *SENSORS.* IEEE Computer Society; 2009. p. 1423-1426. (In Eng.) doi: https://doi.org/10.1109/ICSENS.2009.5398437

[3] Goldberg R.P. Architecture of virtual machines. *Proceedings of the workshop on virtual computer systems.* Association for Computing Machinery, New York, NY, USA; 1973. p. 74-112. (In Eng.) doi: https://doi.org/10.1145/800122.803950

[4] Craig I.D. Virtual Machines. Springer, London; 2006. 269 p. (In Eng.) doi: https://doi.org/10.1007/978-1-84628-246-1

[5] Gregg D., et al. The case for virtual register machines. *Science of Computer Programming.* 2005; 57(3): 319-338. (In Eng.) doi: https://doi.org/10.1016/j.scico.2004.08.005

[6] Mugridge W.B., Hamer J., Hosking J.G. Multi-methods in a statically-typed programming language. In: America P. (eds.) *ECOOP'91 European Conference on Object-Oriented Programming. ECOOP 1991. Lecture Notes in Computer Science.* Vol. 512. Springer, Berlin, Heidelberg; 1991. p. 307-324. (In Eng.) doi: https://doi.org/10.1007/BFb0057029

[7] Abadi M., Cardelli L., Pierce B., Plotkin G. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems.* 1991; 13(2):237-268. (In Eng.) doi: https://doi.org/10.1145/103135.103138

[8] Tratt L. Chapter 5 Dynamically Typed Languages. *Advances in Computers*. 2009; 77:149-184. (In Eng.) doi: https://doi.org/10.1016/S0065-2458(09)01205-4

[9] Nierstrasz O., Bergel A., Denker M., Ducasse S., Gälli M., Wuyts R. On the Revival of Dynamic Languages. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds) *Software Composition. SC 2005. Lecture Notes in Computer Science.* Vol. 3628. Springer, Berlin, Heidelberg; 2005. p. 1-13. (In Eng.) doi: https://doi.org/10.1007/11550679_1

[10] Hickman G.D. An overview of virtual machine (VM) technology and its implementation in IT student labs at Utah Valley State College. *Journal of Computing Sciences in Colleges*. 2008; 23(6):203-212. (In Eng.) doi: https://doi.org/10.5555/1352383.1352419

[11] Cheng P., Blelloch G.E. A parallel, real-time garbage collector. *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI'01).* Association for Computing Machinery, New York, NY, USA; 2001. p. 125-136. (In Eng.) doi: https://doi.org/10.1145/378795.37882

[12] Sasada K. YARV: yet another RubyVM: innovating the ruby interpreter. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05).* Association for Computing Machinery, New York, NY, USA; 2005. p. 158-159. (In Eng.) doi: https://doi.org/10.1145/1094855.1094912

[13] Hofstee H.P. Power efficient processor architecture and the Cell processor. *11th International Symposium on High-Performance Computer Architecture.* IEEE Computer Society; 2005. p. 258-262. (In Eng.) doi: https://doi.org/10.1109/HPCA.2005.26

[14] Burd T.D., Brodersen R.W. Processor design for portable systems. *Journal of VLSI signal processing systems for signal, image and video technology*. 1996; 13(2):203-221. (In Eng.) doi: https://doi.org/10.1007/BF01130406

[15] Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., Grimmer M. Practical partial evaluation for high-performance dynamic language runtimes. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2017).* Association for Computing Machinery, New York, NY, USA; 2017. p. 662-676. (In Eng.) doi: https://doi.org/10.1145/3062341.3062381

[16] Kuck D.J., Kuhn R.H., Padua D.A., Leasure B., Wolfe M. Dependence graphs and compiler optimizations. *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'81).* Association for Computing Machinery, New York, NY, USA; 1981. p. 207-218. (In Eng.) doi: https://doi.org/10.1145/567532.567555

[17] McKeeman W.M. Peephole optimization. *Communications of the ACM*. 1965; 8(7):443-444. (In Eng.) doi: https://doi.org/10.1145/364995.365000

[18] Dubach C., Cavazos J., Franke B., Fursin G., O'Boyle M.F.P., Temam O. Fast compiler optimisation evaluation using code-feature based performance prediction. *Proceedings of the 4th international conference on Computing frontiers (CF '07).* Association for Computing Machinery, New York, NY, USA; 2007. p. 131-142. (In Eng.) doi: https://doi.org/10.1145/1242531.1242553

[19] McCandless J., Gregg D. Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: or: how I learned to stop worrying and love hill climbing. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11).* Association for Computing Machinery, New York, NY, USA; 2011. p. 161-170. (In Eng.) doi: https://doi.org/10.1145/2093157.2093183

[20] Henriques P.R., et al. Automatic Generation of Language-based Tools. *Electronic Notes in Theoretical Computer Science*. 2002; 65(3):77-96. (In Eng.) doi: https://doi.org/10.1016/S1571-0661(04)80428-6

[21] Vergu V., Visser E. Specializing a meta-interpreter: JIT compilation of Dynsem specifications on the Graal VM. *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18).* Association for Computing Machinery, New York, NY, USA; 2018. Article number: 16. p. 1-14. (In Eng.) doi: https://doi.org/10.1145/3237009.3237018

[22] Steensgaard-Madsen J. A Generator for Composition Interpreters. In: Bosch J., Mitchell S. (eds.) *Object-Oriented Technologys. ECOOP 1997. Lecture Notes in Computer Science.* Vol. 1357. Springer, Berlin, Heidelberg; 1998. p. 369-373. (In Eng.) doi: https://doi.org/10.1007/3-540-69687-3_75

[23] Rubio M.S., Civera G.L., Herraiz J.J.M. Automatic Generation Of Virtual Machines For Security Training. *IEEE Latin America Transactions.* 2016; 14(6):2795-2800. (In Eng.) doi: https://doi.org/10.1109/TLA.2016.7555257

[24] Aguzzi G., Cesarini F., Pinzani R., Soda G., Sprugnoli R. Towards an Automatic Generation of Interpreters. In: Brauer, W. (eds) *GI Gesellschaft für Informatik e. V. Lecture Notes in Computer Science*. Vol. 1. Springer, Berlin, Heidelberg; 1973. p. 94-103. (In Eng.) doi: https://doi.org/10.1007/978-3-662-41148-3_9

[25] Leduc M., et al. Automatic generation of Truffle-based interpreters for Domain-Specific Languages. *The Journal of Object Technology.* 2020; 19(2):1-21. (In Eng.) doi: https://doi.org/10.5381/jot.2020.19.2.a1

[26] Ertl M.A., Gregg D.,Krall A., Paysan B. Vmgen – a generator of efficient virtual machine interpreters. *Software: Practice and Experience.* 2002; 32(3):265-294. (In Eng.) doi: https://doi.org/10.1002/spe.434

[27] Gregg D., Ertl M.A. A Language and Tool for Generating Efficient Virtual Machine Interpreters. In: Lengauer C., Batory D., Consel C., Odersky M. (eds.) *Domain-Specific Program Generation. Lecture Notes in Computer Science.* Vol. 3016. Springer, Berlin, Heidelberg; 2004. p. 196-215. (In Eng.) doi: https://doi.org/10.1007/978-3-540-25935-0_12

[28] Lim J., Reps T. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *ACM Transactions on Programming Languages and Systems*. 2013; 35(1):4. (In Eng.) doi: https://doi.org/10.1145/2450136.2450139

Поступила 14.10.2021; одобрена после рецензирования 30.11.2021; принята к публикации 05.12.2021.
Submitted 14.10.2021; approved after reviewing 30.11.2021; accepted for publication 05.12.2021.

Об авторе:

Гонопольский Марк Геннадьевич, магистрант кафедры информационной безопасности факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1), **ORCID: https://orcid.org/0000-0002-0670-8603,** gmarkmgw@yandex.ru

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Mark G. Gonopolskiy, Master degree student of the Chair of Information Security, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation), **ORCID: https://orcid.org/0000-0002-0670-8603,** gmarkmgw@yandex.ru

The author has read and approved the final manuscript.