УДК 004.4'422
DOI: 10.25559/SITITO.17.202103.613-624

Научная статья

Задача автоматической генерации peephole-оптимизаций: обзор подходов, решение проблемы оптимального расширения архитектуры набора managed инструкций

А. В. Антипина
ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Российская Федерация
119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1
anya.antipina@gmail.com

Аннотация

В данной статье рассматривается проблема автоматической генерации peephole-оптимизаций статического оптимизатора байт-кода виртуальной машины (ВМ) применительно к задаче усовершенствования архитектуры набора инструкций (ISA) ВМ. Peephole-оптимизация является одним из этапов работы оптимизирующего компилятора, применяющаяся к коду исходной входной программы в промежуточном представлении (IR или LLIR) и заключающаяся в поиске последовательностей инструкций, совпадающих с известными шаблономи, и замене их на более короткие или более быстрые семантически эквивалентные наборы. Существующие подходы peephole-оптимизаций предполагают наличие некоторого ограниченного набора шаблонов или функций, которые закодированы вручную, полагаясь при этом на глубокие знания инженеров соответствующего IR и структуры выходных программ данного компилятора. Также для различных IR необходимо заново кодировать аналогичные peephole-оптимизации, что влечет дополнительные денежные и временные расходы. Альтернативным подходом является *супероптимизация*, заключающаяся в автоматической генерации шаблонов и правил их сопоставления. Среди основных частей супероптимизатора присутствуют алгоритм поиска, функция затрат и верификатор эквивалентности. В статье реализован и протестирован метод решения более специализированной подзадачи, заключающейся в поиске и подсчете наиболее частых шаблонов инструкций, которые в дальнейшем могут быть заменены на одну новую инструкцию для оптимизации времени выполнения и/или размера байт-кода и быть проанализированы программистом для получения информации о специфике исходных данных. Данный алгоритм может быть использован как инструмент проектирования managed ISA.

Ключевые слова: автоматическая генераций peephole-оптимизаций, компиляторы, peephole-оптимизация, супероптимизаторы, архитектура набора команд, байткод, виртуальные машины

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Антипина, А. В. Задача автоматической генерации peephole-оптимизаций: обзор подходов, решение проблемы оптимального расширения архитектуры набора managed инструкций / А. В. Антипина. – DOI 10.25559/SITITO.17.202103.613-624 // Современные информационные технологии и ИТ-образование. – 2021. – Т. 17, № 3. – С. 613-624.

Original article

The Task of Automatic Generation of Peephole-Optimizations: Approaches Overview, Solution of the Optimal Expansion of Managed Instructions Set Architecture

A. V. Antipina
Lomonosov Moscow State University, Moscow, Russian Federation
1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation
anya.antipina@gmail.com

Abstract

This article discusses the problem of automatic generation of peephole-optimization of the static bytecode optimizer of a virtual machine (VM) concerning the task of improving the instruction set architecture (ISA) of the VM. Peephole optimization is one of the stages of the optimizing compiler. It applies to the code of the original input program in an intermediate representation (IR or LLIR) and resides in finding sequences of instructions that match known templates and replacing them with shorter or faster semantically equivalent sequences. The current peephole optimization approaches assume some limited set of templates or functions that are manually encoded while relying on the engineers' in-depth knowledge of the corresponding IR and the structure of the compiler output programs. Also, for various IRs you must re-encode similar peephole optimizations, which entails additional monetary and time costs. An alternative approach is *superoptimization*, which consists of the automatic generation of templates and their matching rules. Among the main parts of the superoptimizer there are the search algorithm, the cost function, and the equivalence verifier. In the article algorithm for solving a more specialized subtask was implemented and tested. It consists in finding and counting the most frequent instruction templates, which can later be replaced with one new instruction to optimize execution time and/or bytecode size or be analyzed by a programmer to investigate the weaknesses of the existing ISA and obtain information about the specifics of the source data.

Keywords: automatic generation of peephole optimizations, compilers, peephole optimizations, superoptimizers, instruction set architecture, bytecode, virtual machines

The author declares no conflict of interest.

For citation: Antipina A.V. The Task of Automatic Generation of Peephole-Optimizations: Approaches Overview, Solution of the Optimal Expansion of Managed Instructions Set Architecture. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2021; 17(3):613-624. DOI: https://doi.org/10.25559/SITITO.17.202103.613-624

1. Введение

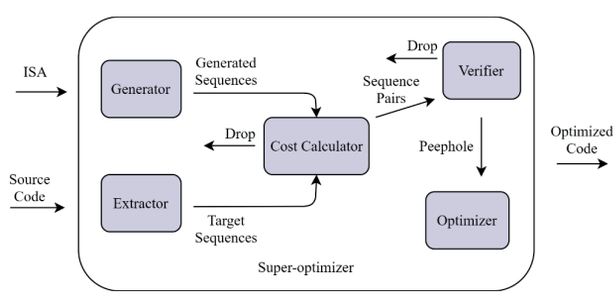
Компилятор является большой и сложной программной системой, которая принимает в качестве входных данных исходную программу и транслирует ее функциональность в программу для целевой машины[1]. Различная природа этих двух задач разделяет компиляцию на две основные фазы: внешний интерфейс или front-end и back-end. Front-end переводит исходную программу в промежуточное представление (IR) и предоставляет в back-end для последующего его сопоставления с набором команд и конечными ресурсами целевой машины [1].

Создание максимально производительного кода всегда имело большое значение во многих областях применения. На практике невозможно создать оптимальную программу для любой целевой машины, применяя оптимизации только в IR, так как каждая архитектура имеет свои особенности и возможности для улучшения исполняемого кода. Существуют три концептуальные возможности: front-end, IR, back-end. Выбирая первый вариант, для каждого интерфейса придется реализовать множество общих оптимизаций, что увеличит усилия в разработке [2]. Аналогично, для третьего варианта необходимо дублирование оптимизаций для каждой целевой архитектуры. Поэтому современные компиляторы выполняют оптимизации на IR. В 1965 году Уильям МакКиман изобрел первый метод peephole-оптимизации [3]. Peephole-оптимизация – это тип оптимизации, выполняемый на очень небольшом наборе инструкций IR, называемым "peephole" или "window" [4]. Суть оптимизации заключается в распознавании такого набора инструкций (или входного шаблона), которые могут быть заменены более коротким или более быстрым семантически эквивалентным набором инструкций (или выходным шаблоном) для достижения скорости или производительности при выполнении оптимизируемой программы[2]. Могут применяться следующие методы для улучшения [5]: сворачивание констант, снижение сложности вычислений, удаление нулевых последовательностей, комбинирование операций, применение алгебраических законов, специальные инструкции, операции в режиме адресации.

Peephole-оптимизаторы обычно используют правила сопоставления шаблонов (впервые использовадись в peephole-оптимизаторе Лэмба [6]) или функции преобразования, написанные человеком. В этом случае от инженера требуется глубокое знание соответствующей архитектуры процессора и выполнение кодирования соответствующих частей программы на машинном языке вручную. Это безусловно является тяжелой задачей и требует больших временных затрат и человеческих усилий. Альтернативным подходом является супероптимизация, заключающаяся в автоматической генерации шаблонов и правил их сопоставления. Это не молодая техника – термин «супероптимизация» ввел Генри Массалин еще в 1987 году в своей статье, где он представил новаторский подход к построению peephole-оптимизатора [7]. Первый супероптимизатор выполнял полный перебор возможных кодовых последовательностей в порядке увеличения длины. Каждая последовательность исполнялась на наборе тестовых данных и последовательность, прошедшая все тесты, печаталась в качестве кандидата. Однако даже с использованием метода усечения пространства поиска супероптимизатор работал очень медленно и мог генрировать последовательности с максимальной длиной в пять инструкций. К тому же тестирование эквивалентности найденных последовательностей не покрывало всех возможных поведений программы, соответственно, могли генерироваться некорректные программы. Поэтому данный подход был далек от практического применения.

Принципы построения системы автоматической генерации, предложенные Массалиным, легли в основу будущих решений в данной области: на вход супероптимизатору подается ISA целевой архитектуры и исходная программа на языке, представленным данной ISA; выходными данными является оптимизированная семантически эквивалентная программа на том же языке; фундаментальными частями супероптимизатора являются:

- алгоритм поиска – способ генерации новой последовательности инструкций (выполняется в элементе “Generator” на рис. 1);
- функция затрат – способ сравнения степени оптимальности последовательностей (рассчитывается в элементе “Cost Calculator” на рис. 1);
- верификатор эквивалентности – способ подтверждения эквивалентности найденных последовательностей (“Verifier” на рис. 1).



Р и с. 1. Структура супероптимизатора

F i g. 1. Super optimizer structure

Последние тенденции технологического развития увеличивают потребность в автоматической генерации и дают новые возможности для устранения ее недостатков, что может помочь данной области стать более популярной и практически применимой. Во-первых, возросло давление на разработчиков компиляторов в связи с внедрением языков программирования более высокого уровня и распространением разнообразных аппаратных платформ. Во-вторых, SAT- и SMT-решатели, которые могут использоваться в верификаторе эквивалентности, продолжают совершенствоваться. Они уже способны осуществлять доказательства эквивалентности, необходимые для проверки оптимизаций компилятора, включающих десятки и сотни инструкций [8]. В-третьих, компиляторы, включающие верификацию, гораздо сложнее масштабировать, чем традиционные компиляторы. Таким образом, задача построения

[1] Cooper K.D., Torczon L. Engineering a Compiler. 2nd ed. Morgan Kaufmann Publ., 2011. 824 p. DOI: https://doi.org/10.1016/C2009-0-27982-7

[2] Aho A.V., Lam M.S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools. 2nd ed. Addison Wesley, 2006. 1040 p.

системы автоматической генерации peephole-оптимизаций и внедрение ее в цикл компиляции является востребованной во многих областях применения благодаря:

- уменьшению затрат при переносе существующих компиляторов на новые архитектурные платформы;
- редуцированию ресурсов, затрачиваемых на разработку систем peepholeоптимизаторов;
- увеличению систематичности peephole-оптимизаторов;
- появлению потенциально новых возможностей для улучшения производительности целевого кода.

2. Цель работы

Целью работы является исследование метода автоматической генерации peephole-оптимизаций и разработка метода усовершенствования ISA регистровой виртуальной машины с помощью добавления новых инструкций для оптимизации получаемого байт-кода.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Проанализировать существующие методы создания систем peephole-оптимизаций;
2. Разработать метод автоматической генерации входных шаблонов peephole-оптимизаций, выполнимых с помощью добавления новый инструкций в ISA;
3. Реализовать функциональность метода;
4. Разработать методику экспериментального исследования;
5. Провести экспериментальное исследование и проанализировать полученные результаты.

3. Постановка задачи

Зададим множество $ISA = \{isa\}$ архитектур набор команд для регистровых виртуальных машин. Каждая $isa = \{opcode_{isa}\}_{i=0}^{n}$ представляет собой множество из n опкодов – кортежей, имеющих структуру

$opcode_{isa} = \{OP, OPERANDS, DESC, PROPS, VALID, EXEPS\}$.

Определим множество программ (или байт-кодов) P_{isa} над $isa \in ISA$, каждый элемент которого является множеством специфицированных опкодов $p_{isa} = \{\widetilde{opcode}_{isa}\}, p_{isa} \in P_{isa}$.

Для каждого элемента множества программ существует множество *I* входных данных и множество *O* выходных данных такие, что $\forall i \in I\ \exists o \in O: P_{isa}(i) = o$.

Пусть $S \subseteq P_{isa}$ подмножество участков программы. Назовем две последовательности опкодов $s_1, s_2 \in S$ эквивалентными $(s_1 \sim s_2)$, если $\forall i \in I\ P^1_{isa}(i) = P^2{}_{isa}(i)$, где $P^2{}_{isa}$ получена из P^1_{isa} подстановкой s_2 вместо s_1. Две последовательности опкодов $s_1, s_2 \in S$ являются частично эквивалентными на подмножестве входов I' $(s_1 \sim^{I'} s_2)$, если существует подмножество входов программы $I' \subset I$, для которого s_2 и s_1 являются эквивалентными.

Определим низкоуровневую оптимизацию peephole как отображение $peephole_{isa}: S \times S$, преобразующее исходную последовательность опкодов $s_1 \epsilon P^1_{isa}$ в эквивалентную или частично эквивалентную результирующую (может быть пустую) последовательность $s_2 \epsilon P^2_{isa}$, при этом замена в исходной программе s_1 на s_2 приводит к увеличению производительности и/или уменьшению размера программы. Первое условие обозначим как критерий корректности. Последнее условие обозначим, как то, что стоимость программы P^2_{isa} должна быть меньше, чем P^1_{isa}.

Таким образом, в задаче супероптимизации необходимо найти множество peephole-оптимизаций для $isa \in ISA$ на основе входного множества программ $P^T_{isa} \subset P_{isa}$.

В статье рассматривается более специализированная задача нахождения множества исходных последовательностей $s_1 \in P_{isa}$ таких peephole-оптимизаций, правые части $s_2 \in P_{isa}$, которых состоят из одной уже существующей или новой инструкции. То есть необходимо найти S_1, $S_1 \subseteq P_{isa}, \forall s_1 \in S_1\ \exists\ s_2 = peephole_{isa'}(s_1), |s_2| = 1, s_2 \in S_2 \subseteq P_{isa}$,
где $isa' = isa \cup \{opcode'\}, opcode' \epsilon\ S_2$.

4. Анализ существующих методов генерации peephole-оптимизаций

PO и HOP

Первый машинно-независимый peephole-оптимизатор был разработан Дэвидсоном и Фрейзером в 1979-80 годах [9; 10] и получил название PO. PO примечателен еще и тем, что использовал автоматическую генерацию шаблонов, хотя структурно не являлся «супероптимизатором» согласно определению, введенному в главе Введение. PO принимает на вход ассемблерным код программы и описание целевой машины (ISA), на которой предполагается выполнение программы.

Работа peephole-оптимизатора разделена на три фазы. На первой он выясняет сайд-эффекты каждой ассемблерной инструкции, представляющие собой шаблоны относительно регистров. Собранная информация формирует двунаправленную грамматику перевода между инструкцией ассемблерного кода и шаблоном передачи регистров. На второй фазе каждая пара смежных инструкций в ассемблерном коде анализируется и преобразуется в эквивалентные экземпляры шаблонов передачи регистров с помощью двунаправленной грамматики из предыдущего анализа. На последней фазе PO находит наилучшую единственную инструкцию ассемблерного кода, соответствующую шаблону, с помощью которого затем заменяется исходная пара инструкций во входной программе. Данный алгоритм является полным в том смысле, что он находит все возможные пары объектного кода, которые могут быть сведены к одной инструкции. Существенным недостатком было времня работы и необходимость выполнения генерации при каждой компиляции. Авторы указывают скорость оптимизации от 1 до 10 инструкций в секунду на PDP-11/70, в то время как машинно-независимый peephole-оптимизатор, разработанный Таненбаумом в 1980 году, достигает скорости более 1000 инструкций в секунду на (значительно более медленном) PDP-11/45. Однако сам подход с использованием паттернов «передачи регистров» вдохновил авторов GCC компилятора и нашел отражение в промежуточном представлении кода RTL, для

которого также применяются peephole-оптимизации[3] [11-13].
В 1984-ом году Дэвидсон и Фрейзер сделали еще один шаг к улучшению автоматической генерации и представили быстрый peephole-оптимизатор HOP, который работает во время компиляции и использует PO [14]. PO запускался в офлайн-режими и получал «обучающий набор» шаблонов для автоматического поиска шаблонов, которые могут быть использованы в правилах оптимизаций.

GNU супероптимизатор

В период с 1991 по 1995 года программистами проекта GNU разрабатывался собственный супероптимизатор (GSO), который использует исчерпывающий поиск подход генерации и тестирования (generate-and-test) для поиска кратчайшей последовательности команд для данной функции[4]. Пользователь должен сообщить супероптимизатору, для какой функции и для какого процессора он хочет сгенерировать код, и задать максимальную ширину окна поиска (в количестве инструкций) [15]. Супероптимизатор не может генерировать очень длинные последовательности, если у пользователя нет очень быстрого компьютера или очень много свободного времени, так как временная сложность используемого алгоритма составляет приблизительно $O((m * n)^{2n})$, где m – количество доступных инструкций в архитектуре, а n – самая короткая последовательность для целевой функции. Практический предел длины последовательности зависит от целевой архитектуры и количества аргументов целевой функции, обычно это значение равно 5. Главным недостатком GSO является то, что функция, которую пользователь хочет оптимизировать, должна быть записана в конфигурационный файл на языке C. GSO также оптимизирует только операции регистр-регистр, где предполагается, что выходные и входные данные целевых функций находятся в определенных регистрах. Для тестирования используется тест выполнения сначала на тестовых векторах, предоставленных пользователем, а затем на рандомно сгенерированных.

LLVM супероптимизатор Souper

Souper – это синтезирующий супероптимизатор, который автоматически выводит новые оптимизации для промежуточного представления кода [8]. Первоначально он был разработан для LLVM[5] [16; 17], но также использовался авторами для поиска новых оптимизаций для компилятора Microsoft Visual C++ (ранее в LLVM уже использовалась автоматическая генерация peephole-оптимизаций, но только частично[6] [18; 19]). Базовой абстракцией Souper является направленный ациклический граф потока данных (DAG). Souper IR состоит из 51 инструкции, которые получены из эквивалентов в целочисленном скалярном подмножестве набора инструкций LLVM IR. Единственными типами данных являются бит-вектор, кортеж битвекторов и специальный тип блока.
Правила peephole-оптимизации, которые синтезирует Souper, состоят из правой и левой частей (RHS, LHS). В качестве функции затрат используется подсчет размера инструкции в байтах для конкретной архитектуры или количество инструкций. Проверка корректности преобразования выполняется по стандартной методике: Souper спрашивает SMT-решателя, существует ли какая-либо оценка входных данных, которая приводит к неравенству левой и правой сторон оптимизации. Если этот запрос неудовлетворителен, эквивалентность доказанна и оптимизация обоснована. Если запрос выполним, обнаружен контрпример, и он представлен пользователю. Для сокращения пространства поиска Souper использует улучшенную версию алгоритма CEGIS [20]. CEGIS избегает исчерпывающего поиска, а также избегает создания запросов для SMTрешателя, содержащих сложные кванторы. Souper может быть запущен несколькими способами: существуют инструменты командной строки для обработки Souper IR и LLVM IR, и также Souper может быть связан с общей библиотекой для динамически загрузки в качестве прохода оптимизации LLVM. Souper использует Klee [21] в качестве библиотеки для генерации запроса в SMT-LIB формате и не требует модели памяти, поэтому достаточно выдавать запросы в теории квантифицированных битвекторов.

Стохастический супероптимизатор STOKE

STOKE[7] – это стохастический оптимизатор и синтезатор программ для набора инструкций x86-64. Авторы формулируют задачу безцикловой бинарной супероптимизации как задачу стохастического поиска [22; 23]. Хотя их метод не является полным, объем программ, которые STOKE может обрабатывать, и качество программ, которые он синтезирует, намного превосходят программы рассмотренных ранее супероптимизаторов. Причиной тому ограничения, которые накладывает полнота явных методов перечисления на длину обрабатываемых программ, и в настоящее время эта граница значительно ниже порога, при котором происходит много интересных оптимизаций. Неявные методы могут преодолеть это ограничение за счет сокращения пространства поиска.
Авторы определяют конкурирующие требования корректности и скорости, как термины в функции затрат в сложном пространстве поиска всех безцикловых последовательностей инструкций и формулируют задачу оптимизации программы как задачу минимизации затрат. Хотя результирующее про-

[3] Stallman R.M., et al. Using The GNU Compiler Collection: A GNU Manual for GCC Version 4.3.3. CreateSpace Independent Publishing Platform, 2009. 638 p.; Liška M., Jelínek J., et al. GenMatch [Электронный ресурс] // GCC GitHub repository, 2021. URL: https://github.com/gcc-mirror/gcc/blob/master/gcc/genmatch.cc (дата обращения: 03.08.2021); Machine-Specific Peephole Optimizers [Электронный ресурс] // GNU Compiler Collection, 2021. URL: https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gccint/Peephole-Definitions.html#Peephole-Definitions (дата обращения: 03.08.2021).

[4] GNU Superoptimizer [Электронный ресурс] // GNU Operating System, 2021. URL: https://www.gnu.org/software/superopt (дата обращения: 03.08.2021).

[5] Lattner C., Adve V. The LLVM Instruction Set and Compilation Strategy. Technical Report. No. UIUCDCS-R-2002-2292. Computer Science Dept., Univ. of Illinois, 2002. URL: https://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.html (дата обращения: 03.08.2021); LLVM [Электронный ресурс] // GitHub repository, 2021. URL: https://github.com/llvm-mirror/llvm/tree/master (дата обращения: 03.08.2021).

[6] van Oirschot J. Extending tree pattern matching for application to peephole optimizations: Master Thesis. EE Dept., Eindhoven University of Technology, Netherlands, 2019. URL: https://research.tue.nl/en/studentTheses/extending-tree-pattern-matching-for-application-to-peephole-optim (дата обращения: 03.08.2021).

[7] STOKE [Электронный ресурс] // GitHub repository, 2021. URL: https://github.com/StanfordPL/stoke (дата обращения: 03.08.2021).

странство поиска сильно нерегулярно и не поддается точным методам оптимизации, общий подход использования сэмплера марковской цепи Монте-Карло (Markov Chain Monte Carlo, MCMC) для изучения функции и получения недорогих выборок достаточен для получения высококачественных кодовых последовательностей.

Еще одним новаторским решением является разделение генерации на два этапа: этап синтеза, ориентированный исключительно на корректность, который пытается определить регионы эквивалентных программ; этап оптимизации, ориентированный на минимизацию метрики производительности для поиска самой быстрой программы в каждом из этих регионов. Эти две фазы используют одну и ту же реализацию поиска. Различаются только начальная точка поиска и функция стоимости. Синтез начинается со случайной начальной точки (последовательности случайно выбранных инструкций), в то время как оптимизация начинается с последовательности, эквивалентной целевой. В качестве целевой архитектуры авторы STOKE выбрали X86-64, так как ее ISA является одной из самых сложных в производстве, что должно упроситить обобщение предложенной техники для других архитектур. Для доказательства корректности преобразования STOKE использует SMT-решатель в роли валидатора. Кодовые последовательности преобразуются в формулы SMT в свободной от кванторов теории битовой векторной арифметики.

Супероптимизатор Бансала

Бансал [24] описывает систему, полностью автоматического построения peepholeоптимизаторов с использованием супероптимизации грубой силы. Для генерации оптимизаций используется алгоритм полного перечисления, но пространсво поиска существенно сокращается благодаря введенным ограничениям. Вычисление правил для peephole-оптимизаций производится в автономном режиме на основе набор тестовых программ. Затем оптимизации организуются в таблицу поиска, сопоставляя исходные последовательности с их оптимизированными аналогами. Эта архитектура, в которой оптимизации вычисляются в автономном режиме, а затем представляются в виде индексированной структуры для эффективного поиска, гораздо ближе к базе данных поисковой системы, чем к традиционному оптимизатору. Тем не менее, эти правила могут быть использованы так же эффективно, как и правила в стандартном peepholeоптимизаторе.

Таким образом, супероптимизатор Бансала достигает достаточной скорости и систематичности, чтобы его можно было использовать в каждой компиляции. Основные достижения Бансала состоят в следующем:

- целевые последовательности извлекаются или собираются из обучающего набора программ. Идея заключается в том, что важными последовательностями для оптимизации являются те, которые уже генерируются компиляторами. Поэтому авторы просто берут все последовательности команд до заданной длины из репрезентативной коллекции существующих двоичных файлов в качестве обучающего набора;
- супероптимизатор стремится быть применим к общим двоичным файлам. Существующий прототип обрабатывает почти 300 опкодов 32-битной архитектуры x86. В частности, оптимизации включают инструкции, связанные с доступом к памяти и ветвями, и учитывают контекст (например, набор живых переменных);
- для сокращения времени поиска применяются две техники:
  1. Канонизация, основанная на наблюдении, что, однажды рассмотрев один набор инструкций, никогда не нужно рассматривать другой, равный последовательному переименованию регистров и символических констант;
  2. Устранение инструкций, которые функционально эквивалентны другим более дешевым инструкциям. Из этого следует, что все подпоследовательности последовательности команд должны быть оптимальными.
- для увеличения скорости работы авторы сохраняют найденные правила замены в хэш-таблицу. Хэш вычисляется следующим образом: сгенерированная последовательность I выполняется для двух тестовых состояний машины, а затем вычисляем хэш результата. Наиболее важные свойства такого подхода заключаются в том, что хэш вычисляется очень быстро и приводит к небольшому набору целевых последовательностей, которые могут быть эквивалентны I;

Авторы используют различные функции затрат для различных целей; например, время выполнения для оптимизации скорости, количество байтов команд для оптимизации размера двоичного файла. Тест эквивалентности выполняется в два этапа: быстрый, но неполный тест выполнения и более медленный, но точный логический тест. Тест выполнения аналогичен тому, как вычисляются хэши последовательностей. Сравниваемые последовательности запускаются на наборе тестовых векторов, и сравниваются результаты выполнения. Логический тест представляет последовательность команд в виде булевой формулы и выражает отношение эквивалентности как ограничение выполнимости. Ограничение выполнимости проверяется с помощью решателя SAT. Состояние машины представлено конечным набором регистров и моделью полной памяти и стека: регистры представлены в виде битовых векторов, а память моделируется сопоставление выражений адресов на биты данных, модель стека идентична модели памяти, с дополнительными битами, представляющими указатели стека и кадра. Инструкции кодируются в виде логических схем, преобразующих исходное состояние машины в конечное. Инструкции ветвления обрабатываются путем прогнозирования выполнения инструкций по истинному и ложному путям с условием ветвления или его отрицанием. Две последовательности команд эквивалентны, если регистры, память и выражения стека, полученные в конечном состоянии, эквивалентны. Отношение эквивалентности состояний выражается в виде ограничения выполнимости, прежде чем передать его решателю SAT.

Результаты обзора

В приведенном обзоре были рассмотрены существующие методы автоматической генерации peephole-оптимизаций, применяющиеся в ранних и современных супероптимизаторах. Краткие результаты приведены в таблицах 1.1 и 1.2.

Таблица 1.1. **Результаты обзора супероптимизаторов**
Table 1.1. **Super Optimizer Review Results**

	Search Algorithm				
	Sequence Constraints			Completeness	Execution Phase
	Representation	Length	Instructions		
J. W. Davidson, C. W. Fraser. PO and HOP [9, 10, 14]	Register Transfer Patterns	2	Register-based	Complete brute force search	Online-Compile(PO); Offline-Compile(HOP)
GNU Superoptimizer GSO[8]	Correspond to the input	5	Register-based	Complete brute force search	Online-Compile
LLVM Superoptimizer Souper [8]	Souper IR (DAG)	5	All (in loop-free code)	Complete solver-based synthesis	Online-Compile
STOKE[9]	Correspond to the input	Not specified	All (in loop-free code)	Incomplete randomized search using MCMC sampler	Online-Compile
S. Bansal, A. Aike. Superoptimizer [24]	Correspond to the input	6	All (in loop-free code)	Complete brute force search with restrictions for reducing the search space	Offline-Compile

Таблица 1.2. **Результаты обзора супероптимизаторов**
Table 1.2. **Super Optimizer Review Results**

	Cost Function	Equivalence test	
		Execution Test	Boolean Test
J. W. Davidson, C. W. Fraser. PO and HOP [9, 10, 14]	Number of instructions	-	-
GNU Superoptimizer GSO[10]	Number of instructions	User's tests and random tests	-
LLVM Superoptimizer Souper [8]	Number of instructions or bytes	-	Register states -> queries in the theory of quantified bitvectors for SMT-solver
STOKE[11]	Castomizing	User's tests or generated tests. User should provide an annotated driver and appropriate contex for test execution	Registers, stack, memory -> SMT formulae in the quantifier free theory of bit-vector arithmetic
S. Bansal, A. Aike. Superoptimizer [24]	Running time or instruction byte count	User's test-vectors or random test-vectors	Registers, stack, memory -> SAT formulae in the bit-vector arithmetic

5. Разработка метода автоматической генерации peephole-оптимизаций

Разрабатываемый метод направлен на решение задачи более специализированной, чем супероптимизация в целом. Как было сказано в постановке задачи, совершается поиск левых частей peephole-оптимизаций, которые делают замену этих правых частей на одну уже существующую или новую инструкцию. Данный метод использует те же принципы, что и супероптимизатор:

- алгоритм поиска последовательностей, но пространство поиска сокращается, принимая во внимание, что найденная последовательность должна замениться одной инструкцией;
- функция стоимости – вычисляет, что предполагаемая правая часть должна быть меньше по стоимости, чем сгенерированная левая;
- верификатор эквивалентности. В роли верификатора выступает разработчик, который принимает решение нужно ли ему заменять предложенный паттерн и добавлять новую инструкцию в ISA, имеющую тот же семантический смысл, что и данный паттерн, но выполняющуюся быстрее или занимающую меньше места.
- Решение поставленной задачи базировалось на алгоритмах, используемых в методе Бансала [24]. Описанные алгоритмы наиболее подходят для применения к байт-коду регистровой ВМ по следующим причинам:
- правила peephole-оптимизаций генерируются из тестового набора программ на ассемблерном коде в автономной фазе, значит, наиболее времязатратный этап можно выполнять независимо от компиляции целевых программ;
- входные и выходные данные являются программами на ассемблерном коде. Это является преимуществом, так как ас-

[8] GNU Superoptimizer [Электронный ресурс] // GNU Operating System, 2021. URL: https://www.gnu.org/software/superopt (дата обращения: 03.08.2021).
[9] STOKE [Электронный ресурс] // GitHub repository, 2021. URL: https://github.com/StanfordPL/stoke (дата обращения: 03.08.2021).
[10] GNU Superoptimizer [Электронный ресурс] // GNU Operating System, 2021. URL: https://www.gnu.org/software/superopt (дата обращения: 03.08.2021).
[11] STOKE [Электронный ресурс] // GitHub repository, 2021. URL: https://github.com/StanfordPL/stoke (дата обращения: 03.08.2021).

семблер ВМ является достаточно низкоуровневым для нахождения интересных оптимизаций и, учитывая, что ассемблерный код исполняется интерпретатором ВМ, не требуется генерация паттернов для каждой целевой архитектуры;
- данный метод сочетает в себе полноту поиска с приемлемым временем генерации, что достигается благодаря введенным ограничениям;
- разрабатываемый супероптимизатор не нуждается в большой длине обрабатываемых последовательностей по следующим наблюдениям:
  1. Во-первых, рассматриваемая ВМ ориентирована главным образом на выполнение программ на языке Java, который используется в основном для написания разного рода приложений, следовательно, в генерируемом ассемблерном коде будут преобладать обращения к полям классов, вызовы функций и циклы. Поэтому длинные линейные участки кода без вызовов или ветвления являются редкими.
  2. Во-вторых, peephole-оптимизация изначально предполагает локальность своего действия, что несовместимо с обработкой последовательностей длиной более шести инструкций. Замена длинных наборов команд потенциально ведет к ошибкам, связанным с отсутствием глобального анализа [25].

Описание работы метода

Необходимо найти наиболее часто встречающиеся последовательностей инструкций, которые потенциально могут быть заменены на одну инструкцию. Назовем такие последовательности паттернами. Условия, согласно которым можно считать последовательность паттерном, C_P:

1. Частота последовательности в коде программ из тестового набора высока;
2. Длина последовательности $l: 2 \leq l \leq 3$;
3. Последовательность удовлетворяет ограничениям C_S:
   3.1. Последовательность S должна иметь единственную точку входа.
   3.2. В последовательность S не должно быть опкодов ветвления, кроме последней инструкции.
4. Между инструкциями последовательности существует связь либо по общим регистрам, либо по значению в аккумуляторе (одна инструкция записывает в аккумулятор, другая читает из него).

Алгоритм нахождения паттернов:

1. Извлечение последовательности, удовлетворяющего условиям C_P;
2. Канонизация паттерна;
3. Сохранение паттерна в базу данных;
4. Если паттерн уже присутствует в базе данных, то увеличивается счетчик его частоты;
5. Сортировка паттернов по частоте;
6. Создание удобного представления для анализа результатов программистами.

Канонизация

Чтобы сократить время поиска без ущерба полноте алгоритма применяется техника канонизации.

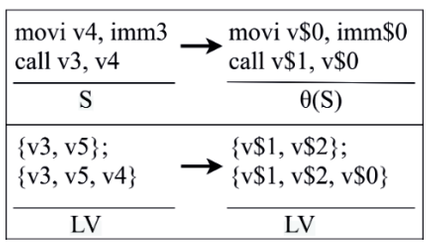
Многие последовательности, встречающиеся в ассемблерном коде являются просто преобразованиями друг друга при переименовании регистров или замене конктант. Например, на машине с восемью регистрами инструкция «mov r1, r0» имеет 8 * 7 = 56 эквивалентных версий с разными именами регистров. Процесс канонизации направлен на уменьшение количества подобных последовательностей до одной канонической. Последовательность команд является канонической, если ее регистры и константы названы в порядке их появления в последовательности команд, т. е. первый используемый регистр всегда r_0 , второй используемый отдельный регистр всегда r_1 и так далее. Аналогично, первая константа – это (символическая константа) c_0, вторая константа c_1 и так далее. Последовательность команд канонизируется путем переименования регистров и констант. Контекст живых переменных, ассоциированный с рассматриваемой последовательностью, также канонизируется в соответствии с уже примененной канонизацией к последовательности (см. рис. 2).

Пусть O_S – множество операндов последовательности S. Таким образом, канонизация последовательности (S, LV) представляет собой совокупность:

$$\theta(S, LV) = \begin{cases} \theta_1 = \theta(O_S) \\ \theta_2 = \theta(\theta_1(LV)) \end{cases}$$

Для сопоставления канонических последовательностей неканоническим введем операцию, обратную канонизации:

$$\theta(S, LV) = \begin{cases} \theta_1 = \theta(O_S) \\ \theta_2 = \theta(\theta_1(LV)) \end{cases}$$



Р и с. 2. Пример канонизации
F i g. 2. Canonization example

Функция затрат

Стоимость сгенерированных последовательностей рассчитывается по следующей функции затрат (в терминах постановки задачи):

$$cost(S) = \sum_{j=0}^{l-1} \left| opcode_{isa}{}^{(j)} \right|$$

где отношение $|\cdot|$ означает количество байт, которое занимает данный опкод. В стоимость не включено время выполнения,

так как для рассматриваемой ISA и ВМ оно коррелирует с длиной опкода.

7. Реализация разработанного метода

Программная реализация метода генерации новых инструкций для ISA регистровой виртуальной машины (ВМ) выполнена в виде приложения Pattern Extractor.

Язык программирования: Python 3.8

Библиотеки: Redis[12], Openpyxl[13]

Файлы: pattern_extractor.py, isa_reader.py, canonicalizer.py, database_accessor.py, pattern_reader.py

Входные данные:

- Тренировочный набор программ на языке ассемблера ВМ.
- ISA данной ВМ в формате YAML.

Выходные данные:

- Файл в формате XLSX, в котором перечислены канонические паттерны, которые могут быть заменены на новые инструкции.

Структура приложения состоит из нескольких скриптов, разделяющих функциональность приложения (см. рис. 3):

- Extractor считывает программу и извлекает паттерны из ассемблерного кода;
- ISA Reader предоставляет интерфейс для получения свойств опкодов;
- Canonicalizer канонизирует найденный паттерн;
- Database Accessor предоставляет интерфейс взаимодействия с базой данных;
- Pattern Reader читает паттерны из базы данных и сохраняет их в XLSX файле.

Тренировочные программы поступают на вход экстрактору, который извлекает из ассемблерного кода последовательности, удовлетворяющие ограничениям . Экстрактор может обращаться к ISA для получения информации о свойствах опкода, например, тип операндов (регистр, идентификатор, константа и тд.), какие операнды подаются на чтение и какие на запись, использует ли опкод аккумулятор, если да, то читает или пишет в него. Эти знания используются для принятия решения, удовлетворяет ли последовательность ограничениям паттерна . Если да, то последовательность передается канонизатору, который выполняет процесс канонизации, после чего обращается к базе данных с помощью интерфейса, предоставленного Database Accessor. Если паттерн существует в базе данных, то увеличивается счетчик его частоты, иначе новый паттерн добавляется в таблицу найденных паттернов в базе данных. После обработки всех тренировочных программ Pattern Reader обращается к базе данных, считывает найденные паттерны, сортирует их по частоте и записывает в удобно читаемое представление в XLSX файл.

8. Экспериментальное исследование

Тестирование метода проводится для ISA регистровой Java ВМ с аккумулятором с использованием набора библиотек class-файлов java-приложений и специальных математических бенчмарков.

Исследование включает в себя получение паттернов новых инструкций для ассемблерного кода разнородных приложений и тестов, а также ручной анализ дизассемблированных файлов на предмет корректности извлеченных паттернов и возможности изменения ISA на основе полученных данных.

Для проведения экспериментального исследования используются:

- набор библиотек class-файлов, предоставляющий базовые API конечным приложениям на мобильных устройствах (219 Мб, 104268 class-файлов);
- набор математически-специфичных java-тестов;
- машина для запуска приложения:
  - процессор Intel(R) Core(TM) i7-8700 CPU 3.20GHz;
  - операционная система Ubuntu 20.04 (Linux 5.4.0-72-generic x86_64);
  - оперативная память 31.2 Гб.
- компилятор и дизассемблер рассматриваемой виртуальной машины.

Тестирование состоит из следующих этапов:

0. Компиляция тестовых наборов для получения ассемблерных файлов;
1. Запуск приложения:
   - 1.1. Для всех библиотек из тестового набора – характеризует «managed» код;
   - 1.2. Для математических тестов – характеризует код для математических операций;
   - 1.3. Для специфичной библиотеки, содержащей много константных массивов – характеризует работу с массивами;
2. Измерение времени работы приложения;
3. Анализ полученных паттернов;
4. Сравнение наиболее частых паттернов для трех тестовых наборов.

Результаты тестирования

Managed-код

Время работы приложения для всех библиотек составило 5 минут 30 секунд. Наиболее частые паттерны представлены на рисунке 3. Можно видеть сдедующие характерные структуры паттернов:

1. Вызов функции (сохраняющей объект в аккумулятор) и сохранение объекта из аккумулятора;
2. Загрузка и сохранение объектов из аккумулятора;
3. Загрузка в аккумулятор и выход из функции.

Первая структура паттернов говорят о том, что необходима инструкция вызова с сохранением результата не в аккумулятор, а в регистр. Была проведена работа по внесению данных опкодов в ISA, что дало улучшение размера секции кода бинарного файла на 2%. Вторая структура паттернов показывает несовершенство работы с аккумулятором, что может быть исправлено улучшением алгоритма распределения аккумулятора. Также полезной инструкцией могла бы быть загрузка строкового объекта, не в аккумулятор, а в регистр. Третья

[12] Redis [Электронный ресурс] // Redis Ltd., 2021. URL: https://redis.io (дата обращения: 03.08.2021).

[13] Gazoni E., Clark Ch. OpenPyXL. MIT/Expat, 2021 [Электронный ресурс]. URL: https://openpyxl.readthedocs.io/en/stable (дата обращения: 03.08.2021).

структура подсказывает о целесообразности введения такой инструкции, как «return» с аргументом.

Если посмотреть на дизассемблированный код рассматриваемых библиотек, то можно убедиться, что по своей природе «managed-код» в основном содержит вызовы функций или обращение к членам класса, так как это частые операции при написании приложений, для чего обычно и используется язык Java.

Математически специфичный код

Время работы приложения для небольшого количества математических тестов составило 0,18 секунды. Видно, что математический код отличается от managed-кода (см. рис. 3). В нем преобладают следующие структуры:

1. Операция сложения с аккумулятором и сохранение из аккумулятора;
2. Загрузка констант в регистр и вызов функции с этой константой;
3. Загрузка констант в регистр и сохранение значения из аккумулятора в элемент массива по индексу, равному этой константе;

Во второй структуре появляется специфичный случай вызова, когда аргументом является константа, предварительно загруженная в регистр. Однако введение такого опкода вызова с константным аргументом непрактично, так как дополнительно необходимо кодировать, какую позицию он занимает среди всех аргументов вызова. Наиболее частым паттерном является сложение с аккумулятором и сохранение результата в регистр, а не в аккумулятор, поэтому было бы практично введение подобной новой инструкции. Также добавляется частая работа с массивами, что видно из третьей структуры. Возможно, целесообразно ввести инструкцию, которая сохраняла бы элемент в массив по индексу, заданному константным значением. Однако в этом случае размер этого значения будет ограничен битовой шириной поля для непосредственных операндов инструкции.

Код с константными массивами

Библиотеку, имеющую размер 652 Кб и содержащую 48 class-файлов, приложение обработало за 6 секунд. Можно заметить следующие особенные паттерны (см. рис. 3):

1. Загрузка в аккумулятор значения и сохранение из аккумулятора в элемент массива;
2. Загрузка константы в регистр и загрузка этого регистра в аккумулятор;
3. Создание нового объекта и сохранение его в регистр;
4. Загрузка элемента массива в аккумулятор и сохранение аккумулятора в регистр.

Заметно преобладает структура первого и четвертого типов – данный паттерн составляет практически 20% от исходного кода. Действительно, так как данная библиотека содержит много константных массивов, при инициализации они должны быть заполнены, поэтому ожидаемо, что в таком коде будут преобладать операции заполнения элементов массивов. Поэтому возникает необходимость в специальной инструкции, которая сначала создавала новый массив (тогда уйдет паттерн третьего типа) и сразу заполняла бы весь массив из подготовленной константной полезной нагрузки. Реализация данной инструкции уже находится в работе и ее использование дает уменьшение конечного бинарного файла более, чем в два раза. Вторая структура говорит о нехватке инструкции загрузки константы в аккумулятор.

	Pattern Name	Amount	Amount Ratio	Canonical Pattern	
Managed code	call.virt.short sta.obj	240016	3.38%	call.virt.short id$0, v$0 sta.obj v$0	call.virt.short v$0, id$0, v$0
	lda.str sta.obj	206616	2.91%	lda.str id$0 sta.obj v$0	lda.str v$0, id$0
	newobj sta.obj	136627	1.92%	newobj id$0 sta.obj v$0	
	lda return	79831	1.12%	lda v$0 return	return v$0 / return.obj v$0
	lda.obj return.obj	60232	0.85%	lda.obj v$0 return.obj	
Math	fadd2.64 sta.64	22	4.31%	fadd2.64 v$0 sta.64 v$0	fadd2.64 v$0, v$0
	fmovi.64 call.short	14	2.75%	fmovi.64 v$0, imm$0 call.short id$0, v$0	call.short id$0, imm$0
Constant arrays	lda.64 fstarr.32	24448	18.82%	lda.64 v$0 fstarr.32 v$0, v$1	lda.const id$0
	fmovi.64 lda.64	3923	3.02%	fmovi.64 v$0, imm$0 lda.64 v$0	lda.64 imm$0
	newarr sta.obj	832	0.64%	newarr v$0, id$0 sta.obj v$0	
	lda.obj starr.obj	829	0.64%	lda.obj v$0 starr.obj v$0, v$1	

Р и с. 3. Наиболее частые паттерны для тестовых наборов и предложенные новые инструкции. Amount – количество появления паттерна в коде. Amount Ratio – доля, которую составляет паттерн от всего кода

F i g. 3. Most common test case patterns and suggested new instructions. Amount - is the amount of pattern appearing in the code. AmountRatio - the proportion of the pattern in the entire code

9. Выводы

Тестирование показало, что данный метод генерации новых инструкций может иметь практическое применение не только для внедрения недостающих опкодов в ISA ВМ, но и для обнаружения характерных особенностей кода, например, преобладание вызовов с константами в математически специфичном коде или работа с константными массивами. Также полученные данные могут сказать о несовершенстве работы какого-либо алгоритма при компиляции кода. Таким образом, разработанное приложение может быть полезно как для наборов инструкций ВМ, так и для ISA с аппаратной поддержкой.

References

[1] Watson D. Compilers and Interpreters. In: Ed. by D. Watson. *A Practical Approach to Compiler Construction. Undergraduate Topics in Computer Science*. Springer, Cham; 2017. p. 13-36. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-52789-5_2

[2] Tanenbaum A.S., Van Staveren H., Stevenson J.W. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems*. 1982; 4(1):21-36. (In Eng.) DOI: https://doi.org/10.1145/357153.357155

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

[4] Lopes N.P., Regehr J. Future Directions for Optimizing Compilers. arXiv:1809.02161. 2018. (In Eng.) DOI: https://doi.org/10.48550/arXiv.1809.02161

[5] Bhatt C.H., Bhadka H.B. Peephole Optimization Technique for analysis and review of Compile Design and Construction. *IOSR Journal of Computer Engineering (IOSR-JCE)*. 2013; 9(4):80-86. (In Eng.) DOI: https://doi.org/10.9790/0661-0948086

[6] Lamb D.A. Construction of a peephole optimizer. *Software: Practice and Experience*. 1981; 11(6):639-647. (In Eng.) DOI: https://doi.org/10.1002/spe.4380110608

[7] Massalin H. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*. 1987; 15(5):122-126. (In Eng.) DOI: https://doi.org/10.1145/36206.36194

[8] Sasnauskas R., et al. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422. 2018. (In Eng.) DOI: https://doi.org/10.48550/arXiv.1711.04422

[9] Davidson J.W., Fraser C.W. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*. 1980; 2(2):191-202. (In Eng.) DOI: https://doi.org/10.1145/357094.357098

[10] Fraser C.W. A compact, machine-independent peephole optimizer. *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (POPL '79). Association for Computing Machinery, New York, NY, USA; 1979. p. 1-6. (In Eng.) DOI: https://doi.org/10.1145/567752.567753

[11] Chakraborty P. Fifty years of peephole optimization. *Current Science*. 2015; 108(12):2186-2190. Available at: https://www.jstor.org/stable/24905654 (accessed 03.08.2021). (In Eng.)

[12] Hickey J., Nogin A. Formal compiler construction in a logical framework. *Higher-Order and Symbolic Computation*. 2006; 19(2-3):197-230. (In Eng.) DOI: https://doi.org/10.1007/s10990-006-8746-6

[13] Heberle A., Gaul T., Goerigk W., Goos G., Zimmermann W. Construction of Verified Compiler Front-Ends with Program-Checking. In: Ed. by D. Bjøner, M. Broy, A.V. Zamulin. *Perspectives of System Informatics. PSI 1999. Lecture Notes in Computer Science.* 2000; 1755:481-492. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/3-540-46562-6_43

[14] Davidson J.W., Fraser C.W. Automatic generation of peephole optimizations. *Proceedings of the 1984 SIGPLAN symposium on Compiler construction (SIGPLAN '84).* Association for Computing Machinery, New York, NY, USA; 1984. p. 111-116. (In Eng.) DOI: https://doi.org/10.1145/502874.502885

[15] Granlund T., Kenner R. Eliminating branches using a superoptimizer and the GNU C compiler. *ACM SIGPLAN Notices*. 1992; 27(7):341-352. (In Eng.) DOI: https://doi.org/10.1145/143103.143146

[16] Leidel J., Kabrick R., Donofrio D. Toward an Automated Hardware Pipelining LLVM Pass Infrastructure. *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* IEEE Press, St. Louis, MO, USA; 2021. p. 39-49. (In Eng.) DOI: https://doi.org/10.1109/LLVMHPC54804.2021.00010

[17] Davidson J.W., Fraser C.W. Automatic generation of peephole optimizations. *ACM SIGPLAN Notices*. 2004; 39(4):104-111. (In Eng.) DOI: https://doi.org/10.1145/989393.989407

[18] Menendez D., Nagarakatte S. Alive-Infer: data-driven precondition inference for peephole optimizations in LLVM. *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. 49-63. (In Eng.) DOI: https://doi.org/10.1145/3062341.3062372

[19] Lopes N.P., Menendez D., Nagarakatte S., Regehr J. Practical verification of peephole optimizations with Alive. *Communications of the ACM*. 2018; 61(2):84-91. (In Eng.) DOI: https://doi.org/10.1145/3166064

[20] Gulwani S., Jha S., Tiwari A., Venkatesan R. Synthesis of loop-free programs. *ACM SIGPLAN Notices*. 2011; 46(6):62-73. (In Eng.) DOI: https://doi.org/10.1145/1993498.1993506

[21] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08).* USENIX Association, USA; 2008. p. 209-224. Available at: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf (accessed 03.08.2021). (In Eng.)

[22] Luk C.-K., et al. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*. 2005; 40(6):190-200. (In Eng.) DOI: https://doi.org/10.1145/1064978.1065034

[23] Schkufza E., Sharma R., Aiken A. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*. 2013; 41(1):305-316. (In Eng.) DOI: https://doi.org/10.1145/2451116.2451150

[24] Bansal S., Aiken A. Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News*. 2006; 34(5):394-403. (In Eng.) DOI: https://doi.org/10.1145/1168857.1168906

[25] Buchwald S. OPTGEN: A Generator for Local Optimizations. In: Ed. by B. Franke. *Compiler Construction. CC 2015. Lecture Notes in Computer Science*. 2015; 9031:171-189. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/978-3-662-46663-6_9

Поступила 03.08.2021; одобрена после рецензирования 10.09.2021; принята к публикации 13.09.2021.
Submitted 03.08.2021; approved after reviewing 10.09.2021; accepted for publication 13.09.2021.

Об авторе:

Антипина Анна Вячеславовна, магистрант 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/0000-0001-9122-4680,** anya.antipina@gmail.com

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Anna V. Antipina, Master student of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation), **ORCID: https://orcid.org/0000-0001-9122-4680,** anya.antipina@gmail.com

The author has read and approved the final manuscript.