Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 20.Б09-мм

Разработка JIT компилятора на основе LLVM для OpenJDK

Бочкарев Арсений Петрович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
старший преподаватель кафедры системного программирования А. П. Козлов

Санкт-Петербург
2023

Оглавление

1. Введение

По данным университета Berkeley [5] на 2022 год Java находится в пятёрке самых популярных языков программирования. Отличительной чертой языка является широкая переносимость. Java программы компилируются в платформонезависимый байт-код, который потом исполняется виртуальной машиной. Для ускорения исполнения Java программ широко применяется Just-in-Time компиляция байт-кода [12]: превращение байт-кода в оптимизированный набор инструкций процессора прямо во время исполнения.

Особенности JIT-компилятора:

- оптимизации с учётом семантики языка Java. Например, это динамическая загрузка классов [21];
- оптимистичные оптимизации [7];
- глубокая интеграция с конкретной JVM. Пример: контракты времени исполнения;

От качества JIT-компилятора зависит скорость исполнения Java программ.

Компанией Azul был разработан проприетарный компилятор «Falcon», поддерживаемый на данный момент только виртуальной машиной «Zulu Prime», также разработанной Azul [4]. Falcon показывает большую эффективность, нежели его ближайший аналог в OpenJDK[1], а потому, в целях увеличения производительности OpenJDK, в данный момент ведётся работа по интеграции данного JIT-компилятора в HotSpot Java Virtual Machine, являющейся на 2020 год самой популярной [6] виртуальной машиной Java. Для этого используется интерфейс JVMCI, позволяющий использовать внешний компилятор вместе с этой JVM.

В компиляторе Falcon используется большое количество фактов о внутренних структурах виртуальной машины, которые отличаются у

[1]Подробнее в разделе «Обзор»

Zulu Prime и HotSpot JVM. Например, это структура Java кучи, представление классов, контракты доступа к куче и взаимодействия с библиотекой функций времени исполнения. Сам Falcon основан на технологии LLVM, для которой характерно наличие большого количества оптимизаций, что позволяет порождать более эффективные машинные коды [13].

Таким образом, компилятор Falcon нуждается в наборе изменений, чтобы код, порождённый им, можно было использовать в HotSpot JVM. Целью данной работы является внесение подобных изменений в Falcon.

2. Постановка задачи

Целью работы является адаптация JIT-компилятора Falcon к виртуальной машине HotSpot OpenJDK.
В ходе работы была поставлена следующая задача:
Сделать обзор элементов реализации HotSpot JVM и произвести модификацию JIT-компилятора Falcon для согласования с реализацией HotSpot следующих концепций:

1. сборки мусора;
2. виртуальных вызовов через «Inline Cache»;
3. обращений к полям классов;
4. релокаций в Data Section;
5. деоптимизаций.

3. Обзор

3.1. Обзор существующих решений

3.1.1. Интерпретатор Java байткода

Выполнение Java программ начинается с интерпретатора байткода виртуальной машины, поочередно выполняющего полученные команды. Методы, вызываемые достаточно часто во время исполнения, или методы, про которые заранее известно, что они будут работать достаточно долго, будут скомпилированы JIT-компилятором вместо интерпретации [12]. Существует также возможность заменять интерпретированный код на скомпилированный прямо во время исполнения [11].

3.1.2. Промежуточные представления

Промежуточное представление («IR») в компиляторах используется для упрощения и оптимизации процесса компиляции. IR представляет собой промежуточный код, который создается из исходного кода программы и используется для генерации машинного кода.

В Falcon используется промежуточное представление LLVM, которое представляет собой последовательность базовых блоков, код в которых написан в SSA форме[2].

3.1.3. JIT-компиляторы внутри HotSpot JVM

Делятся на C1 («клиентский») и C2 («серверный»)[20]:

- C1 компилятор нацелен в первую очередь на быструю компиляцию и уменьшенное использование памяти (по сравнению с C2). При работе он использует несколько промежуточных промежуточных представлений. Через абстрактную интерпретацию Java

[2] https://llvm.org/docs/LangRef.html

байткодов строится высокоуровневое промежуточное представление («HIR») компилируемого метода. Он состоит из графа потока управления («CFG»), основные блоки которого представляют собой односвязные списки инструкций. HIR находится в SSA форме, что означает, что для каждой переменной существует только одна точка в программе, где ей присваивается значение. Инструкция, которая загружает или вычисляет значение, представляет как операцию, так и ее результат, так что операнды могут быть представлены, как указатели на предыдущие инструкции. Как во время, так и после создания HIR выполняется несколько оптимизаций, таких как «constant folding», «value numbering», «method inlining» и «null-check elimination». После этого другая часть компилятора переводит оптимизированный HIR в низкоуровневое промежуточное представление («LIR»). LIR концептуально похож на машинный код, но по большей части не зависит от платформы. В отличие от инструкций HIR, операции LIR работают с виртуальными регистрами, а не со ссылками на предыдущие инструкции. LIR облегчает различные низкоуровневые оптимизации и используется для составления соответствия между физическими и виртуальными регистрами методом «linear scan». После выделения регистров может быть сгенерирован машинный код [12];

- C2 компилятор нацелен в первую очередь на скорость работы производимого кода. Для его замены изначально разрабатывался Falcon, и, согласно официальному сайту Azul, на 2023 год обходит его по эффективности[3]. Работа серверного компилятора Java состоит из традиционных нескольких фаз:

  1. синтаксический анализ,
  2. машинно-независимые оптимизации,
  3. выбор инструкций,

[3] https://www.azul.com/products/components/falcon-jit-compiler/

4. global code motion,
5. распределение регистров,
6. планирование инструкций,
7. peephole optimizations,
8. кодогенерация.

В своей работе C2 также использует собственное промежуточное представление в SSA форме, применяющееся как во время фазы машинно-независимых оптимизаций, так и во время фазы выбора регистров. Распределение регистров, в отличие от компилятора C1, выполнено в виде раскраски графа [19].

3.1.4. Graal

«Graal» — JIT-компилятор с открытым исходным кодом, основанный на идеях и структурах C1/C2 компиляторов, в частности, Sea of Nodes [8]. По умолчанию интегрирован с виртуальной машиной «GraalVM», интеграция же с HotSpot JVM происходит при помощи интерфейса JVMCI. Также, Graal имеет своё внутреннее промежуточное представление, отличное от таковых в Falcon и OpenJDK [9].

3.1.5. Сборщики мусора

Сборка мусора — это процесс освобождения из памяти объектов, помеченных, как «мусор». Объект считается таковым, и его память может быть повторно использована виртуальной машиной, когда на него больше не осталось ссылок из какого-либо другого объекта, не являющегося мусором. На данный момент в HotSpot JVM реализовано пять сборщиков[17], предназначенных для разных задач:

- «Serial GC», нацеленный на работу с небольшим количеством данных и/или на одном процессоре (до 100 Мб);

- «Parallel GC» — для приложений, в которых важна пиковая производительность без требований на время задержки при сборке мусора;
- «G1 GC» и «Concurrent Mark-and-Sweep GC», предназначенные для задач, в которых необходимо как можно более быстрое время отклика приложения (есть требования к времени задержки при сборке мусора);
- «ZGC», рассчитанный для задач с требованиями к времени задержки при сборке мусора и работе с кучей большого размера.

Без сборки мусора программист вынужден вручную управлять выделением и освобождением памяти, что может привести к ошибкам и неэффективному использованию ресурсов [15]. Поскольку в HotSpot JVM для этого используются продвинутые алгоритмы [17], для достижения цели по увеличению производительности OpenJDK сборке мусора должно быть уделено большое внимание. Поскольку Falcon на данный момент поддержан только собственным сборщиком мусора «C4»[4] (как следствие его тесной интеграции с виртуальной машиной Zulu Prime), использование с ним промышленных, поддерживаемых в HotSpot JVM сборщиков мусора, невозможно. Для введения сборки мусора, в качестве первого из таких сборщиков был выбран Serial GC. Основное понятие, на котором строится его работа — это так называемые «поколения» объектов. Данный сборщик мусора работает в предположении о том, что большинство объектов (например, итераторы), будут существовать относительно короткий промежуток времени, и только небольшое число объектов просуществует достаточно долго.

Подавляющее большинство объектов размещается в области памяти, предназначенной для «молодого» поколения объектов. Когда память из области для молодого поколения наполняется, происходит *миноритарная* сборка мусора, в ходе которой собирается только молодое

[4] https://www.azul.com/products/components/pgc/

поколение; мусор в других поколениях не утилизируется. Уцелевшие в ходе нескольких из таких сборок объекты из молодого поколения перемещается в другую область памяти, предназначенную для «старого» поколения. Когда память старого поколения заполняется, происходит *мажоритарная* сборка мусора, в ходе которой собирается вся куча. Обычно, такая сборка занимает гораздо больше времени, так как в ней задействовано значительно большее количество объектов [17].

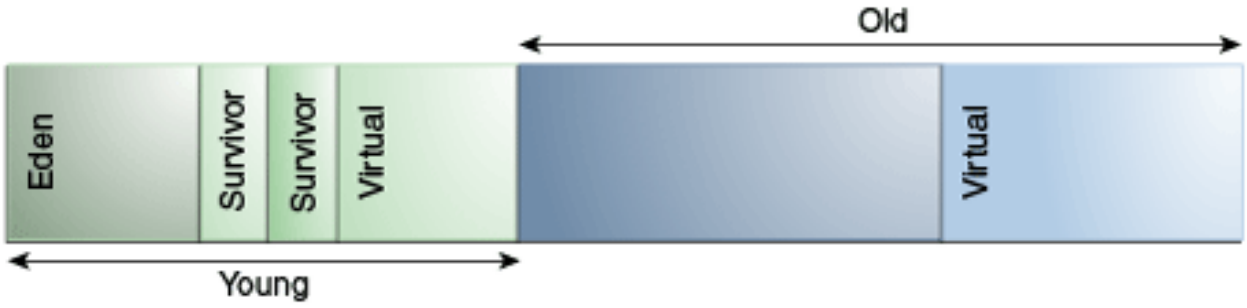

Рис. 1: Поколения при сборке мусора. Источник: статья «Java Garbage Collection Basics»

В некоторых старых объектах могут быть ссылки на молодые объекты, и в ходе миноритарной сборки мусора необходимо следить, чтобы такие объекты не были освобождены. Это достигается путём использования так называемой «Card Table» — специальной битовой структуры, в которую заносится запись при появлении каждой подобной ссылки. Такие записи называются «барьерами на запись», или «stored-value barrier» («SVB»). Сборщик мусора, в свою очередь, видя подобный барьер, сканирует объект, на который была обнаружена ссылка, на предмет того, необходимо ли его освободить, или нет [1].

Каждая из миноритарных (а также большинство мажоритарных) итераций сборки является «Stop the World» событием, то есть таким, при котором все потоки приложения останавливаются до завершения операции [18]. Помимо сборки мусора, концепция «StW» может использоваться, например, для деоптимизации вызовов при неверном предсказании переходов. Кроме простой остановки Java-потоков, виртуальной машине может также потребоваться состояние регистров и стека. Для

реализации подобного используется схема, согласно которой потоки в известных точках программы (когда существует гарантия того, что их состояние известно) запрашивают, должны ли они передать управление виртуальной машине. Моменты работы программы, когда потоки приостанавливаются, называются «Safepoints», а точки, в которых происходят опросы, называются «Safepoint Polls» [2].

3.1.6. Виртуальные вызовы

Большинство вызовов функций в Java являются виртуальными, то есть такими, для которых конкретная реализация будет определяться во время исполнения. Поэтому их эффективной компиляции уделяется большое внимание. Одной из возможных техник их оптимизаций является «Inline Caching». Inline caching — это техника оптимизации по ускорению динамической диспетчеризации путём запоминания результатов предыдущих поисков соответствующих методов прямо в структуре, хранящей расположение функции в коде, из которого она была вызвана («Call site») [14].

3.1.7. Деоптимизации

В случае, если система выполнения начнёт заранее исполнять код, который при ветвлении никогда не должен быть доступен (такая ситуация возможна при неверном предсказании перехода), выставляется специальная «ловушка» («Uncommon trap»), благодаря которой выполнение перейдёт в следующие блоки кода [3].

3.2. Инструменты

- Язык C++ версии 14;
- Технология LLVM версии 13;
- Реализованные в Falcon методы по построению LLVM IR — load, getelementptr и другие.

4. Реализация

4.1. Поддержка сборки мусора

4.1.1. Барьеры на запись

Реализация в OpenJDK В OpenJDK SVB реализованы «по области», то есть, если на момент цикла сборки мусора была обнаружена запись в card table, то на предмет наличия молодого объекта внутри старого будет просканирован не только тот объект, на который непосредственно есть ссылка внутри таблицы, а те объекты, которые внутри card table находятся на некотором расстоянии от него [1]. Также важно отметить, что пометка, например, в интерпретаторе внутри OpenJDK, делается без дополнительных условий, проверяющих запись нового поколения внутри старого во время исполнения[5].

Результаты Поскольку сборщик мусора C4, чтобы делать как можно меньше записей в память [22], во многом опирается на метаданные от профилировщика, получаемые прямо во момент работы программы, необходимо было изменить генерацию соответствующего кода. В то же время, на данном этапе требуется лишь наладить работу сборщика мусора, поэтому было принято решение о генерации SVB для каждой записи. Помимо этого, также была изменена и генерация функции проставления барьеров в скалярном случае следующим образом:

- В порождение функции добавлен отдельный базовый блок[6] для безусловной записи в Card Table, в котором загружаются необходимые константы и происходит подсчёт адресов внутри таблицы, соответствующих объектам, в которые делается запись. Код этого базового блока в форме промежуточного представления LLVM приведён в листинге 1;

- Изменена область ответственности каждой записи в Card Table —

[5] https://github.com/openjdk/jdk17/blob/master/src/hotspot/cpu/x86/gc/shared/cardTableBarrierSetAssembler_x86.cpp#L88

[6] https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html

барьеры отвечают не за запись в конкретном объекте, а за область, в которой все объекты будут просканированы [1];

- Исправлено само значение, записываемое в таблицу.

Код функции по генерации SVB приведён в листинге 1.

Листинг 1: Безусловная запись в card table

```
SerialGCMarkBB:
    ; Fetch required globals
    %bytesPerGPGCPage = load i64, i64* @LogBytesPerGPGCPage
    %bytesPerGPGCPageInfo = load i64, i64*
        @LogBytesPerGPGCPageInfo
    %pageInfoBase = load i8*, i8** @GPGC_PageInfo.page_info_base
    %cardsBaseAddrOffsetInBytes = load i64, i64*
        @GPGC_PageInfo.cards_base_addr_offset_in_bytes
    %bytesPerWord = load i64, i64* @LogBytesPerHeapOop

    ; Get the bit index and card table address
    %addr.asInt = ptrtoint i8 addrspace(1)* addrspace(1)* %addr
        to i64, !verifier_exception !1
    ; read the base address of cards from page info
    %pageNum = lshr i64 %addr.asInt, %bytesPerGPGCPage
    %pageInfoOffset = shl i64 %pageNum, %bytesPerGPGCPageInfo
    %pageInfoAddr = getelementptr inbounds i8, i8* %pageInfoBase,
         i64 %pageInfoOffset
    %cardsAddrAddr = getelementptr inbounds i8, i8* %pageInfoAddr
        , i64 %cardsBaseAddrOffsetInBytes
    %cardsAddrAddr.i64 = bitcast i8* %cardsAddrAddr to i64**
    %cardsAddr = load i64*, i64** %cardsAddrAddr.i64
    %cardsAddr.asInt = ptrtoint i64* %cardsAddr to i64, !
        verifier_exception !1
    %addr.offset = sub i64 %addr.asInt, %cardsAddr.asInt
    %bit.index = lshr i64 %addr.offset, %bytesPerWord
```

```
%bit.index.shift = load i64, i64*
    @LogCardBitsPerGPGCCardMarkSummaryWord
%word.index = lshr i64 %bit.index, %bit.index.shift
%word.addr = getelementptr i64, i64* %cardsAddr, i64 %
    word.index
; In OpenJDK case we don't need to be precise
; Yet we're still need to be atomic

; The summary card mark can race with other threads since they're
; all writing the same value

; Note that we store 0 here instead of 1
store atomic i64 0, i64* %word.addr unordered, align 4, !
    noalias !{!12}
br label %poison
```

4.1.2. Генерация Safepoint Poll

Реализация в OpenJDK Основной недостаток подхода с safepoint poll в том, что подобные опросы почти никогда невозможно оптимизировать, а проводить их необходимо часто. По этой причине текущая реализация safepoint Poll в HotSpot JVM заключается в следующем:

1. Виртуальная машина по заранее известному адресу (называемому «Polling Page») запрет на чтение;

2. Когда поток достигает safepoint, он пытается прочитать это значение;

3. При чтении происходит segmentation Fault, к которому виртуальная машина готова, и по-особому обрабатывает его. Это сигнализирует о ей готовности потока остановиться [2].

Результаты В Zulu Prime JVM обработка safepoint poll происходит иначе, нежели в HotSpot JVM, следовательно, генерация соответствую-

щего кода требует изменения. Было принято решение о порождении на месте, где должен быть сделан опрос, определенного числа последовательно идущих инструкций («площадки») «NOPW». Далее, на стороне JVMCI, увидев известный паттерн, инструкции «NOPW» будут обработаны так, чтобы порождать код, который ожидает увидеть HotSpot JVM для обработки safepoint poll. Более детально, в ходе работы было выполнено следующее:

1. В функции по генерации safepoint poll создаются if-else блоки с незначащим условием — это не позволяет другим методам Falcon оптимизировать их до того, как safepoint poll будут созданы;

2. В каждом из этих блоков создаются два одинаково пустых вызова с сохранением состояния виртуальной машины;

3. Путём прикрепления определённых метаданных к данным вызовам, Falcon получает сигнал о том, что тот из них, который будет в итоге вызван, в ассемблере должен быть приведен к площадке из определённого числа последовательно идущих инструкций «NOPW».

4.2. Организация виртуальных вызовов

Реализация в OpenJDK В OpenJDK существует два типа виртуальных вызовов:

1. Оптимизированные — виртуальные вызовы, для которых существует одна конкретная реализация метода. При дизассемблировании кода, сгенерированного OpenJDK, с такими методами они помечены как «optimized virtual call» [10]. В скомпилированном коде такой вызов является статическим;

2. Обычные — виртуальные вызовы, делающиеся с использованием inline caching [10]. Первым аргументом при подготовке генерации таких виртуальных вызовов ожидается значение «−1» [16].

Результаты Помимо двух вышеописанных типов, Falcon так же поддерживает вызов через таблицу виртуальных методов («vtable»). Таблица виртуальных методов — это структура для поиска соответствующих реализаций виртуальных методов во время исполнения. Решение о генерации вызова через таблицу или inline cache принимается согласно эвристике. Поэтому одной из подзадач в случае компиляции для OpenJDK было гарантировать выбор данной эвристикой inline cache call.
Таким образом, необходимо изменить генерацию виртуальных вызовов на более подходящую.
В ходе работы были выполнены следующие подзадачи:

- Проведён анализ проверок для генерации inline cache вызовов, в результате чего теперь для каждого вызова с соответствующими метаданными (специальный атрибут у виртуального вызова) гарантируется генерация inline caching оптимизации;
- Переделано порождение LLVM IR в создании inline cache вызовов: генерируется значение «−1», которое на этапе порождения нативного кода попадает в регистр «%rax» (согласно правилам, описанным в Falcon).

4.3. Деоптимизации

Реализация в OpenJDK Требуется, чтобы при деоптимизациях вызов функций производился по непосредственному адресу [23].

Результаты В текущем варианте порождаемого кода создаётся лишний для OpenJDK указатель. Для форсирования нужной генерации был создан соответствующий атрибут, проверка на который делается в момент порождения машинных инструкций для создания узла глобального адреса в графе выбора инструкций[7].

[7] https://llvm.org/docs/CodeGenerator.html#introduction-to-selectiondags

5. Эксперименты

5.1. Сравнение генерируемого ассемблерного кода

В результате внесённых в Falcon изменений генерируется правильный код, согласованный (где это необходимо[8]) с тем, что генерируется при работе с OpenJDK. Ассемблерный код получался с использованием опций «-XX:+UnlockDiagnosticVMOptions» и «-XX:+PrintAssembly».

5.1.1. Организация виртуальных вызовов

Тест «Test15.java». Код, генерировавшийся Falcon до изменений, приведён в листинге 2. Код, порожденный Falcon после внесения изменений совпал с кодом, порождаемым OpenJDK и приведён в листинге 2.

Листинг 2: Организация виртуальных вызовов. Код, сгенерированный Falcon до внесения изменений

```
and  $0x1fffffff,%eax
nopl (%rax)
data16 data16 data16 data16 data16 cs nopw 0x200(%rax,%
   rax,1)
```

Листинг 3: Организация виртуальных вызовов. Код, сгенерированный Falcon после внесения изменений, а также порождаемый OpenJDK

```
movabs $0xffffffffffffffff,%rax
call   0x00007f87d4112480          ; ImmutableOopMap {}
                                   ;*invokevirtual
                                      testCountUpdater {
                                      reexecute=0
                                      rethrow=0
                                      return_oop=0}
                                   ; - Test15::test@2 (
                                      line 12)
```

[8]Подробнее в разделе «Реализация»

```
                                        ;   {virtual_call}
```

5.1.2. Деоптимизации

Тест «Test08.java». Код, генерировавшийся Falcon до изменений, приведён в листинге 4. Код, порожденный Falcon после внесения изменений приведён в листинге 5. Код, генерируемый OpenJDK приведён в листинге 6.

Листинг 4: Деоптимизации. Код, генерировавшийся Falcon до внесения изменений

```
movabs $0x0,%rax

....

call   *%rax
```

Листинг 5: Деоптимизации. Код, сгенерированный после внесения Falcon

```
call   99 <static int Test08.test(jobject, int)1001+0x99>

...

RELOCATION RECORDS FOR [.text]:
0000000000000095 R_X86_64_PLT32    __llvm_deoptimize-0
   x0000000000000004
```

Листинг 6: Деоптимизации. Код, сгенерированный OpenJDK

```
call   0x00007f8aac71627a          ; ImmutableOopMap {rsi=Oop
                                   ;*iload_1 {reexecute=1 reth
                                   ; - (reexecute) Test08::tes
                                   ;   {runtime_call Deoptimiz
```

5.1.3. Барьеры на запись

Тест «SelfWrittenVersion.java». Код, сгенерированный OpenJDK приведён, в листинге 7. Код, порожденный Falcon после внесения изменений, приведён в листинге 8.

Листинг 7: Безусловная запись в card table. Код, сгенерированный OpenJDK

```
shr    $0x9,%r11   ; r11 == object to mark
movabs $0x7f1c3757f000,%rbx
mov    %r12b,(%rbx,%r11,1)  ; r12b == 0
```

Листинг 8: Безусловная запись в card table. Код, сгенерированный Falcon

```
mov    %rdx,%rsi    ; rdx == object to mark
shr    $0x9,%rsi
movb   $0x0,(%r8,%rsi,1)   ; r8 == card table base (0
   x7f1c3757f000)
```

5.1.4. Safepoint poll

Тест «Test47.java». Код, генерировавшийся Falcon до внесения изменений, приведён в листинге 9. Код, порожденный Falcon после внесения изменений, приведён в листинге 10.

Листинг 9: Safepoint poll. Код, сгенерированный до внесения изменений Falcon

```
testl  $0x1,0x348(%r15) ; the poll

...

call   2e <static int Test47.test(long, int)1001+0x2e> ;
   transfers control to vm
```

Листинг 10: Safepoint poll. Код, сгенерированный после внесения изменений Falcon

```
data16 data16 data16 data16 data16 nopw cs:0x200(%rax,%
    rax,1)
```

Также, после интеграции с модулем JVMCI, были успешно пройдены соответствующие тесты[9]. Коды тестов для примеров выше дополнительно приведены в приложении.

[9]Код всех тестов приведён в Google Drive

6. Заключение

6.1. Выполненные задачи

- Произведён обзор элементов реализации HotSpot JVM;
- Произведена модификация JIT-компилятора Falcon для согласования с реализацией HotSpot JVM следующих концепций:
  - сборки мусора;
  - виртуальных вызовов через «Inline Cache»;
  - деоптимизаций.

Исходный код находится под соглашением о неразглашении.

7. Приложение

Листинг 11: Исходный код теста «Test15»

```
// call with inline cache to method, with transition from monomorphic to
   megamorphic state
public class Test15 {
    public static int countInTest15AbstractClassClass1 = 0;
    public static int countInTest15AbstractClassClass2 = 0;

    public void _testInner(int c1, int c2) {
        countInTest15AbstractClassClass1 += c1;
        countInTest15AbstractClassClass2 += c2;
    }

    public void test(Test15AbstractClass obj) {
        obj.testCountUpdater(this);
    }

    public static void main(String[] args) {
        Test15 invokingObject = new Test15();
        Test15AbstractClass obj = new Test15AbstractClassClass1()
           ;
        int iterationsTest15 = 0;
        int sum = 0;
        long minDuration = 1000 * Integer.getInteger(''duration'',
           5);
        long swapTime = minDuration + System.currentTimeMillis();
        long stopTime = minDuration * 2 + System.
           currentTimeMillis();

        do {
            for (int i = 5000000; --i ≥ 0; ) {
                iterationsTest15++;
```

```
            if (i % 2 == 0) obj = new
               Test15AbstractClassClass1();
            else obj = new Test15AbstractClassClass2();
            invokingObject.test(obj);
         }
      } while (minDuration != 0 && System.currentTimeMillis() <
          stopTime);
      if (countInTest15AbstractClassClass1 +
         countInTest15AbstractClassClass2 != iterationsTest15
         || Math.abs(countInTest15AbstractClassClass1 -
         countInTest15AbstractClassClass2) > 2) {
          System.err.printf("FAILED: "
             countInTest15AbstractClassClass1 = %d, "
             countInTest15AbstractClassClass2 = %d, expected "
             iterationsTest15 = %d",
             countInTest15AbstractClassClass1,
             countInTest15AbstractClassClass2, iterationsTest15
             );
          System.exit(1);
      }
      System.err.printf("PASSED: %d %d",
         countInTest15AbstractClassClass1,
         countInTest15AbstractClassClass2);
   }

   static public abstract class Test15AbstractClass {

      abstract public void testCountUpdater(Test15 obj);

   }

   static public class Test15AbstractClassClass1 extends
      Test15AbstractClass {
```

```
        @java.lang.Override
        public void testCountUpdater(Test15 obj) {
            obj._testInner(1, 0);
        }
    }

    static public class Test15AbstractClassClass2 extends
       Test15AbstractClass {
        @java.lang.Override
        public void testCountUpdater(Test15 obj) {
            obj._testInner(0, 1);
        }
    }
}
```

Листинг 12: Исходный код теста «Test08»

```
// real virtual call test
public class Test08 {
    public static final int EXPECTED = 1642668640;
    public static int testField = 0;
    public static int countInOverridenMethod = 0;
    public static String str = ''greater'';

    public void _testInner(String value) {
        str = value;
    }

    public static int test(Test08 obj, int newValue) {
        testField = newValue;
        if (newValue > 0) obj._testInner(''greater'');
        else obj._testInner(''less'');
        return newValue;
    }
```

```
public static void main(String[] args) {
    Test08 obj = new Test08();
    int iterationsOverriden = 0;
    int sum = 0;
    long minDuration = 1000 * Integer.getInteger(''duration'',
       10);
    long swapTime = minDuration + System.currentTimeMillis();
    long stopTime = minDuration * 2 + System.
       currentTimeMillis();
    do {
        sum = 0;
        for (int i = 5000000; --i ≥ 0; ) {
            int last = sum;
            if (i > 5000000 / 2) obj = new Test08();
            else {
                iterationsOverriden++;
                obj = new Test08OverrideClass();
            }
            sum = test(obj, sum + i);
            if (sum > 0 && str.equals(''less'') || sum ≤ 0 &&
               str.equals(''greater'') || testField != sum) {
                System.err.println(''FAILED:␣str␣'' + str + ''␣
                   with␣sum␣'' + sum + ''␣and␣testField␣'' +
                   testField + ''␣and␣i␣'' + i + ''␣and␣sum␣on␣
                   previus␣iteration␣'' + last + ''␣and␣sum␣+␣i
                   ␣'' + (last + i));
                System.exit(1);
            }
        }
    } while (minDuration != 0 && System.currentTimeMillis() <
        stopTime);
    if (testField != EXPECTED || testField > 0 && str.equals(
```

```
            ''less'') || testField ≤ 0 && str.equals(''greater'') ||
            countInOverridenMethod != iterationsOverriden) {
                System.err.println(''FAILED'');
                System.exit(1);
            }
            System.err.println(''PASSED'');
        }

        static public class Test08OverrideClass extends Test08 {
            @java.lang.Override
            public void _testInner(String value) {
                countInOverridenMethod = countInOverridenMethod + 1;
                str = value;
            }
        }
}
```

Листинг 13: Исходный код теста «SelfWrittenVersion»

```
import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.Optional;

public class SelfWrittenVersion {

    private static final String launcher_name =
            ''openjdk'';

    // This field is read by HotSpot
    private static final String java_version =
            ''17.0.5-internal'';
```

```
private static final String java_version_date =
        ''2022-10-18'';

// This field is read by HotSpot
private static final String java_runtime_name =
        ''OpenJDK␣Runtime␣Environment'';

// This field is read by HotSpot
private static final String java_runtime_version =
        ''17.0.5-internal+0-adhoc.student.jdk17u'';

private static final String VERSION_NUMBER =
        ''17.0.5'';

private static final String VERSION_SPECIFICATION =
        ''17'';

private static final String VERSION_BUILD =
        ''0'';

private static final String VERSION_PRE =
        ''internal'';

private static final String VERSION_OPT =
        ''adhoc.student.jdk17u'';

private static final boolean isLTS =
        ''adhoc.student.jdk17u''.startsWith(''LTS'');

private static final String CLASSFILE_MAJOR_MINOR =
        ''61.0'';

private static final String VENDOR =
```

```
        ''N/A'';

private static final String VENDOR_URL =
        ''https://openjdk.java.net/'';

// The remaining VENDOR_* fields must not be final,
// so that they can be redefined by jlink plugins

// This field is read by HotSpot
private static String VENDOR_VERSION =
        '''';

private static String VENDOR_URL_BUG =
        ''https://bugreport.java.com/bugreport/'';

// This field is read by HotSpot
@SuppressWarnings(''unused'')
private static String VENDOR_URL_VM_BUG =
        ''https://bugreport.java.com/bugreport/crash.jsp'';


private static final int SLEEP_TIME = 1 * 60 * 1000;

private static void print(boolean err, boolean newln) {
    PrintStream ps = err ? System.err : System.out;
    if (err) {
        ps.println(''openjdk␣version␣\''17.0.5-internal\''␣
           2022-10-18'' + (isLTS ? ''␣LTS'' : ''''));
    } else {
        ps.println(''openjdk␣17.0.5-internal␣2022-10-18'' + (
           isLTS ? ''␣LTS'' : ''''));
    }
```

```
        String jdk_debug_level = System.getProperty(''jdk.debug'',
           ''release'');
        if (''release''.equals(jdk_debug_level)) {
            jdk_debug_level = '''';
        } else {
            jdk_debug_level = jdk_debug_level + ''␣'';
        }

        String vendor_version = VENDOR_VERSION.isEmpty() ? '''' : ''
           ␣'' + VENDOR_VERSION;
        ps.println(''OpenJDK␣Runtime␣Environment'' + vendor_version
            + ''␣('' + jdk_debug_level + ''build␣'' + ''17.0.5-
           internal+0-adhoc.student.jdk17u'' + '')'');
        String java_vm_name = System.getProperty(''java.vm.name'');
        String java_vm_version = System.getProperty(''java.vm.
           version'');
        String java_vm_info = System.getProperty(''java.vm.info'');
        ps.println(java_vm_name + vendor_version + ''␣('' +
           jdk_debug_level + ''build␣'' + java_vm_version + '',␣'' +
           java_vm_info + '')'');
    }

    public static void main(String[] args) throws Exception {
        System.out.println(''Start␣of␣printing␣version␣with␣sleep␣
           time:␣'' + SLEEP_TIME);
        System.out.println(''Sleep␣start'');
        Thread.sleep(SLEEP_TIME);
        System.out.println(''Sleep␣finish'');
        print(true, false);
    }

}
```

Листинг 14: Исходный код теста «Test47»

```
public class Test47 {
    public static String testFailed = null;
    public static int criticalSectionCount = 0;
    public static int fieldConstant = 500;

    static public void test() {
        for(int i = 0; i < fieldConstant; i++) {
            if(i % 2 == 0) fieldConstant++;
            else fieldConstant--;
        }
    }

    public static void main(String[] args) throws
       InterruptedException {
        Test47 invokingObject = new Test47();
        long minDuration = 1000 * Integer.getInteger(''duration'',
           5);
        long stopTime = minDuration * 2 + System.
           currentTimeMillis();

        do {
            for (int i = 50000; --i ≥ 0; ) {
                test();
            }
        } while (System.currentTimeMillis() < stopTime);

        System.err.println(''PASSED'');
    }
}
```

Список литературы

[1] A. Shipilev. JVM Anatomy Quark — 13: Intergenerational Barriers. — URL: https://shipilev.net/jvm/anatomy-quarks/13-intergenerational-barriers/ (дата обращения: 01.05.23).

[2] A. Shipilev. JVM Anatomy Quark — 22: Safepoint Polls. — URL: https://shipilev.net/jvm/anatomy-quarks/22-safepoint-polls/ (дата обращения: 01.05.23).

[3] A. Shipilev. JVM Anatomy Quark — 29: Uncommon Traps. — URL: https://shipilev.net/jvm/anatomy-quarks/29-uncommon-traps/ (дата обращения: 12.12.22).

[4] Azul. Azul Zulu Prime Builds of OpenJDK. — URL: https://www.azul.com/products/components/azul-zulu-prime-builds-of-openjdk/ (дата обращения: 12.12.22).

[5] Berkeley. 11 Most In-Demand Programming Languages in 2022. — URL: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/ (дата обращения: 12.12.22).

[6] Constant Fine. Comparing performance of JVM implementations. — URL: https://www.fineconstant.com/posts/comparing-jvm-performance/ (дата обращения: 15.12.22).

[7] D. Csákvári. Profile-based optimization techniques in the JVM. — URL: https://advancedweb.hu/profile-based-optimization-techniques-in-the-jvm/ (дата обращения: 12.12.22).

[8] Demange D. Yon Fernández de Retana Y. Pichardie D. Semantic reasoning about the sea of nodes. — URL: https://hal.inria.fr/hal-01723236/file/sea-of-nodes-hal.pdf (дата обращения: 15.12.22).

[9] Duboscq G. Stadler L. Würthinger T. Simon D. Wimmer C. Mössenböck H. Graal IR: An Extensible Declarative Intermediate Representation. — URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6688214dab5456c75c99f8171846242e09d4f5e3 (дата обращения: 03.05.23).

[10] Duke J. (анонимный автор). Overview of CompiledIC and CompiledStaticCall. — URL: https://wiki.openjdk.org/display/HotSpot/Overview+of+CompiledIC+and+CompiledStaticCall (дата обращения: 15.12.22).

[11] Fink S. Qian F. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. — URL: https://ieeexplore.ieee.org/document/1191549 (дата обращения: 03.05.23).

[12] Kotzmann T. Wimmer C. Mossenbock H. Rodriguez T. Russel K. Cox D. Design of the Java HotSpot Client Compiler for Java 6. — URL: https://dl.acm.org/doi/pdf/10.1145/1369396.1370017 (дата обращения: 02.05.23).

[13] LLVM. LLVM's Analysis and Transform Passes. — URL: https://llvm.org/docs/Passes.html (дата обращения: 04.01.23).

[14] M. Bernstein. Inline caching. — URL: https://bernsteinbear.com/blog/inline-caching/ (дата обращения: 12.12.22).

[15] Microsystems Sun. Memory Management in the Java HotSpot Virtual Machine. — URL: https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf (дата обращения: 01.05.23).

[16] Oracle. AMD64 Call. — URL: https://github.com/oracle/graal/blob/0dd159f485b668977d33bd07d2d8b4575541aa54/compiler/src/org.graalvm.compiler.lir.amd64/src/org/graalvm/compiler/lir/amd64/AMD64Call.java#L235 (дата обращения: 12.12.22).

[17] Oracle. HotSpot Virtual Machine Garbage Collection Tuning Guide. — URL: https://docs.oracle.com/en/java/javase/11/gctuning/garbage-collector-implementation.html#GUID-23844E39-7499-400C-A579-032B68E53073 (дата обращения: 01.05.23).

[18] Oracle. Java Garbage Collection Basics. — URL: https://www.oracle.com/technetwork/tutorials/tutorials-1873457.html (дата обращения: 01.05.23).

[19] Paleczny M. Click C. Vick C. The Java HotSpot Server Compiler. — URL: https://www.researchgate.net/publication/220817735_The_Java_HotSpot_Server_Compiler (дата обращения: 07.05.23).

[20] S. Trefilov. Использование JVMCI для интеграции стороннего компилятора в Java 17. — URL: https://se.math.spbu.ru/thesis/texts/Trefilov_Stepan_Zaharovich_Bachelor_Report_2022_text.pdf (дата обращения: 15.12.22).

[21] WikiBooks. Dynamic Class Loading. — URL: https://en.wikibooks.org/wiki/Java_Programming/Reflection/Dynamic_Class_Loading (дата обращения: 12.12.22).

[22] Wolf M. Tene G. Iyengar B. C4: The Continuously Concurrent Compacting Collector. — URL: https://www.azul.com/sites/default/files/images/c4_paper_acm.pdf (дата обращения: 03.05.23).

[23] openjdk. Shared Runtime X86_64. — URL: https://github.com/openjdk/jdk17u-dev/blob/master/src/hotspot/cpu/x86/sharedRuntime_x86_64.cpp#L2540 (дата обращения: 12.12.22).