МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет Механико-математический факультет
Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Салимов Рустам Аскарович
(Фамилия, Имя, Отчество автора)

Тема работы: Эффективная интерпретация языков высокого уровня с помощью предварительного переписывания байт-кода

«К защите допущена»
Вр.и.о. зав. кафедрой,
к.ф.-м.н.,

Емельянов П.Г. /_____
(фамилия , И., О.) / (подпись, МП)

«…»……20…г.

Научный руководитель
Старший преподаватель
ММФ НГУ

Соловьев В.В. /_____
(фамилия , И., О.) / (подпись, МП)

«…»……20…г.

Дата защиты: «…»……20…г.

Новосибирск, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Существуют различные подходы к реализации языков программирования. Основными из них являются AOT-компиляция, интерпретация и JIT-компиляция. Данная работа посвящена интерпретации.

Интерпретаторы могут разрабатываться с различными целями, и обеспечение высокой производительности не всегда является приоритетной задачей. В рамках данной работы под термином «эффективный интерпретатор» понимается интерпретатор, для которого достижение высокого уровня производительности является ключевой целью.

Интерпретаторы широко используются в системах с JIT-компиляцией. Использование интерпретатора обеспечивает быстрый запуск приложения: пользователю не требуется ожидать завершения компиляции, и он может сразу начать работу с программой. Если интерпретатор достаточно эффективен, пользователь может не заметить, что программа выполняется без компиляции, либо снижение скорости исполнения по сравнению со скомпилированным кодом будет незначительным. Это позволяет дольше оставаться в режиме интерпретации и собирать более полный профиль исполнения программы, что, в свою очередь, способствует повышению качества JIT-компиляции. Наличие более точного профиля исполнения позволяет генерировать более эффективный машинный код.

Интерпретаторы также могут быть полезны при отладке приложений. Отладчики обычно используют интерпретаторы для получения различной информации о состоянии программы, включая значения переменных, содержимое стека вызовов и последовательность выполняемых инструкций [1,2]. Следовательно, производительность отладчика напрямую зависит от производительности интерпретатора.

Интерпретация является наиболее переносимым способом реализации языков программирования. При использовании оптимизаций на уровне машинных инструкций большая часть реализации может быть выполнена независимо от платформы исполнения, а платформозависимой останется лишь небольшая специфическая часть.

Однако, несмотря на удобство и широкую применимость интерпретаторов, они обладают существенным недостатком по сравнению, например, с нативным кодом — более низкой производительностью.

Для интерпретации программы обычно используется некоторое промежуточное представление. Эффективные интерпретаторы, чаще всего, используют представление программы в виде инструкций байт-кода [3]. Каждая инструкция байт-кода, как правило, содержит код операции и аргументы, если они необходимы. Байт-код не зависит от архитектуры процессора и является переносимым форматом.

Для исполнения байт-кода необходимо выполнить следующие этапы:

а) получить код текущей операции;

б) перейти к обработчику соответствующей операции;

в) вычислить аргументы операции;

г) выполнить операцию;

д) перейти к шагу а).

Выполнение самой операции, как правило, не занимает значительного машинного времени, тогда как диспетчеризация, переход к обработчику и получение операндов требуют обращений к памяти и выполнения косвенных переходов, что создаёт существенные накладные расходы на уровне процессора. Таким образом, интерпретаторы значительную часть времени тратят не на выполнение полезной работы, а на организацию исполнения.

Наибольшие накладные расходы возникают на этапе диспетчеризации (пункты *а)* и *б)*). Получение кода операции представляет собой обращение к памяти, в которой хранится байт-код. Пункт *б)* соответствует косвенному переходу. Такой переход называется косвенным, поскольку целевой адрес заранее неизвестен и может быть вычислен только во время исполнения. Следовательно, для исполнения каждой инструкции байт-кода необходимо выполнять косвенный переход. Эффективность выполнения косвенных переходов существенно зависит от качества их предсказания процессором.

Существуют различные подходы к реализации диспетчеризации. Наиболее простым является использование конструкции `switch`. Иллюстрация представлена на рисунке 1. Такой подход обладает

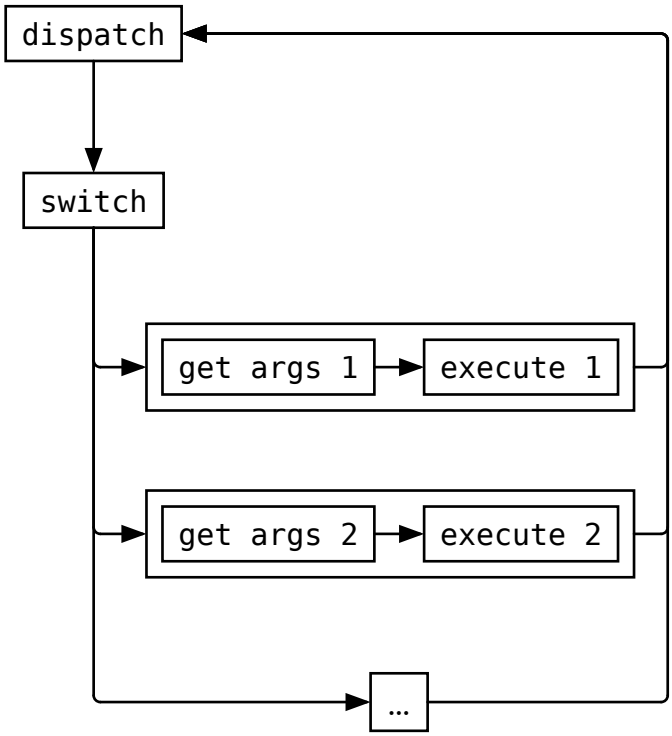


Рисунок 1 — схема кода при использовании switch

существенным недостатком: низкой производительностью вследствие большого количества ошибок предсказания переходов.

Предсказатель переходов — специализированный модуль процессора. С помощью различных эвристик, например истории предыдущих переходов, шаблонов исполнения циклов и статистики наиболее частых направлений перехода, он пытается определить, по какому пути будет продолжено исполнение программы. При использовании метода, представленного на рисунке 1, переходы происходят из одной точки — оператора `switch`. В такой ситуации предсказателю переходов сложнее определить дальнейший путь исполнения [4]. При использовании данной схемы диспетчеризации предсказателю фактически требуется определить, какая инструкция будет выполнена следующей, что в общем случае является сложной задачей.

Для упрощения задачи предсказания переходов применяется метод шитого кода [4,5]. Суть данной оптимизации заключается в том, что по коду операции получается прямой адрес обработчика соответствующей операции. Получение адреса может происходить различными способами. Например, возможно использование специальной структуры для сопоставления кода операции и обработчика. В таком случае метод называется *методом косвенного шитого кода*. Также возможно использование самого адреса обработчика в качестве кода операции. В этом случае метод называется *методом прямого шитого кода*. При этом сами обработчики, вместо возврата к оператору выбора, получают адрес следующего обработчика и выполняют косвенный переход к нему. Иллюстрация представлена на рисунке 2.

При использовании такого подхода переходы выполняются непосредственно из каждого обработчика. Благодаря большему числу участков кода, из которых осуществляются переходы, предсказатель получает больше контекста о путях исполнения программы. Например, в случае цикла предсказатель может «запомнить» последовательность инструкций, что позволяет снизить количество ошибок предсказания. Согласно результатам симуляций, приведённым в [4], интерпретаторы на основе метода шитого кода демонстрируют существенно меньшее

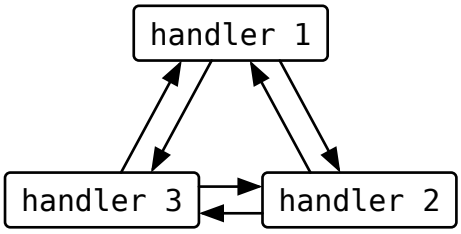


Рисунок 2 — диспетчеризация при использовании шитого кода

количество ошибок предсказания переходов по сравнению с реализациями на основе оператора switch.

Одним из подходов к повышению производительности интерпретаторов является дополнительный этап обработки — переписывание байт-кода. Байт-код современных высокоуровневых языков программирования может требовать дополнительных вычислений, выполняемых непосредственно перед интерпретацией или в процессе её выполнения.

Во время переписывания байт-код преобразуется в форму, более удобную для исполнения. Например, обращения к полям объекта могут дополняться заранее вычисленной информацией, содержащей смещение поля внутри объекта. Также возможно создание специальной таблицы методов для замены символьной информации, хранящейся в инструкциях, индексами методов из данной таблицы.

Такое переписывание позволяет сократить объём вычислений во время исполнения, благодаря чему уменьшаются накладные расходы и повышается общая производительность системы.

Целью квалификационной работы является исследование возможности использования механизма предварительной обработки байт-кода не только для решения задач позднего связывания, но и для формирования нового представления байт-кода, ориентированного на более эффективную интерпретацию. Предлагаемый формат должен обеспечивать возможность применения метода прямого шитого кода для реализации диспетчеризации инструкций.

Для достижения поставленной цели сначала будет реализован интерпретатор на языке высокого уровня, затем — эквивалентный по функциональности интерпретатор на языке ассемблера архитектуры ARMv8. После этого будет разработан новый формат байт-кода, выполнена его интеграция в механизм предварительной обработки и реализован соответствующий интерпретатор на языке ассемблера ARMv8.

Объектом исследования являются интерпретаторы байт-кода виртуальных машин. Предметом исследования являются методы повышения производительности интерпретаторов. Целью данной работы является создание экспериментальной реализации механизма переписывания байт-кода и интерпретации нового байт-кода. Для достижения поставленной цели необходимо выполнить следующие задачи:

- Провести анализ подходов к построению эффективных интерпретаторов.
- Разработать базовую реализацию интерпретатора на языке высокого уровня, без применения специальных средств оптимизации.
- Разработать базовую реализацию интерпретатора байт-кода на языке низкого уровня.
- Сравнить получившиеся реализации интерпретатора.
- Предложить новый формат байт-кода и механизмы переписывания.

– Разработать эталонную реализацию интерпретатора байт-кода на языке низкого уровня с учетом исследованных факторов повышения производительности.
– Провести сравнение производительности.
– Сформулировать выводы о применимости предложенного подхода и степени достигнутой производительности.

1 Обзор предшествующих работ

1.1 Проблематика интерпретации программ

1.1.1 Роль интерпретаторов

Существует два основных подхода к реализации языков программирования: компиляция и интерпретация. При компиляции исходный код программы транслируется во промежуточное представление компилятора. На этом представлении могут выполняться различные оптимизации, направленные на повышение эффективности выполнения программы. После этого оптимизированное представление преобразуется в машинный код, зависящий от конкретной аппаратной платформы.

В случае интерпретации обычно используется промежуточное представление и виртуальная машина. Программа на исходном языке транслируется в промежуточное представление, после чего выполняется (интерпретируется) виртуальной машиной. Промежуточное представление, как правило, является платформенно-независимым, что делает данный подход более переносимым способом реализации языка программирования. Проведение сложных оптимизаций непосредственно в интерпретаторе затруднено, поскольку такие оптимизации требуют анализа и преобразования значительных участков программы, что создаёт дополнительные накладные расходы во время исполнения. Интерпретатор же ориентирован на немедленное выполнение инструкций виртуальной машины с минимальными задержками.

Для добавления поддержки новой платформы, в случае компилируемого языка необходимо учесть все её особенности, поскольку компиляция происходит сразу в машинный код процессора. Для поддержки интерпретируемого языка на новой платформе необходимо поддержать исполнитель промежуточного представления, то есть виртуальную машину. Вторая задача, как правило, проще.

Интерпретируемые языки также удобнее для отладки [1]. Для анализа исполнения необходимо лишь подключиться к виртуальной машине, которая может предоставить много различной информации о состоянии исполняемой программы. Благодаря этому такие языки более предпочтительны для обучения и создания небольших прототипов.

Благодаря использованию виртуальной машины можно гарантировать бóльшую безопасность во время исполнения [6,7]. Этого можно добиться, ограничив доступ исполняемого приложения к памяти и контролируя процесс исполнения.

Еще одним немаловажным фактором является скорость запуска приложения. В случае компилируемых языков необходимо дождаться завершения компиляции, тогда как для интерпретируемых выполнение может начаться практически сразу. Интерпретаторы эффективнее в сценариях, где JIT-компиляция не окупает своих затрат, например, в коротких скриптах или при написании приложения для командной строки. Поэтому многие скриптовые языки являются таковыми.

Несмотря на меньшую производительность, интерпретаторы всё ещё являются актуальным инструментом для реализации языков

программирования и играют важную роль в экосистеме, благодаря преимуществам в переносимости, безопасности, удобстве разработки и скорости запуска.

1.1.2 Основные проблемы производительности интерпретаторов

Хотя интерпретаторы сохраняют преимущества в переносимости, безопасности и быстроте запуска, их производительность обычно ниже по сравнению с JIT-компиляцией и исполнением нативного кода. В дальнейшем будем рассматривать интерпретаторы, использующие байт-код в качестве промежуточного представления.

Существует несколько факторов, которые имеют ключевое влияние на производительность:

а) Переключение между инструкциями;

б) Слабая локальность данных и инструкций;

в) Доступ к стеку виртуальной машины;

г) Динамические проверки.

Рассмотрим некоторые из них подробнее.

1.1.2.1 Стоимость переключения

Для обработки каждой инструкции байт-кода необходимо пройти несколько этапов:

а) Чтение инструкции из памяти;

б) Декодирование операндов;

в) Переход к реализации инструкции.

Такой процесс занимает больше времени, чем аналогичный для машинной инструкции. Наиболее затратными операциями в этой цепочки являются: доступ к памяти при чтении байт-кода, и косвенный переход к обработчику инструкции.

Косвенный переход является затратным за счёт конвейеризации исполнения машинной инструкции. Для исполнения инструкции необходимо выполнить несколько действий: получение инструкции, декодирование — то есть определение типа инструкции и данных, необходимых для её исполнения, доступ к памяти, выполнение самой инструкции, запись результата. Всё это происходит в конвейере процессора. Современный процессор обычно имеет не менее пяти этапов.

Во время исполнения программы, представленной в виде линейной последовательности, инструкции поочерёдно поступают в конвейер и проходят все его стадии. Если возникает необходимость перехода, требуется загрузить инструкции по новому адресу. Эти инструкции могут находиться в кеше, и в зависимости от его уровня загрузка займёт от 4 до 40 циклов процессора. Если же инструкции отсутствуют в кеше и должны быть загружены из оперативной памяти, доступ к ним может занимать от 200 циклов [8, ch. 4.7]. Это время будет потеряно для процессора, поскольку он будет вынужден ожидать новых инструкций.

Если переход является условным, возникает вероятность ошибки. Процессор предполагает, что исполнение продолжится по одному пути, однако фактическое выполнение может произойти по другому.

В случае неверного предположения приходится не только дожидаться новых инструкций, но и отменять исполнение уже выполненных. Очистка конвейера в подобных ситуациях может занимать до 20 циклов [8, ch. 4.8].

В случае с косвенными переходами задача усложняется. Необходимо не просто определить, произойдёт ли переход, но и предсказать адрес следующей инструкции.

Таким образом, ошибки в предсказании оказывают существенное влияние на производительность интерпретатора.

1.1.2.2 Слабая локальность

Нативный код обычно компактно размещается в памяти, благодаря чему обладает хорошей локальностью инструкций. В случае интерпретатора исполнение, наоборот, постоянно переключается между различными участками кода. Это связано с необходимостью выбирать обработчики инструкций, считывать байткод и операнды. Кроме того, интерпретатору приходится постоянно обращаться к своим внутренним структурам данных. В стековой виртуальной машине это приводит к частому взаимодействию со стеком, а в регистровой — к обращениям к виртуальным регистрам.

Многие оптимизации, доступные для машинного кода, невозможно эффективно применить во время интерпретации. Поэтому возникают дополнительные вызовы функций и обращения к памяти при работе с объектами и данными. Все это повышает вероятность кэш-промахов, что существенно снижает производительность.

1.1.2.3 Остальные факторы

Помимо перечисленных выше, существуют и другие факторы, влияющие на производительность интерпретаторов. К ним относятся, например, стоимость динамической типизации, необходимость управления памятью и работы сборщика мусора, а также накладные расходы, связанные с особенностями объектной модели и системы типов языка программирования.

Устранение данных факторов, как правило, требует изменений на уровне виртуальной машины или самого языка программирования, например изменения модели представления объектов, системы типов или стратегии управления памятью. Поскольку настоящая работа посвящена повышению эффективности исполнения байт-кода на уровне интерпретатора, указанные аспекты не рассматриваются подробно.

1.1.3 Основные направления оптимизации

Одним из главных направлений оптимизации является уменьшение количества ошибок предсказаний. Для реализации этого направления существуют разные подходы, например использование метода шитого кода вместо конструкции выбора (switch).

Использование суперинструкций снижает количество переходов в целом, что, как следствие, уменьшает число ошибочных предсказаний. Такие инструкции являются объединением нескольких инструкций, что позволяет удалить переключения.

Дупликация инструкций, при правильном использовании, увеличивает количество мест, из которых совершаются косвенные переходы, что даёт предсказателю больше контекста и помогает делать верные предсказания.

Поскольку инструкции байт-кода могут быть недостаточно конкретными (например, доступ к полю объекта или вызов виртуального метода), в интерпретаторах может применяться метод специализации инструкций. Он заключается в замене общей инструкции на более специализированный вариант, ориентированный на конкретный тип данных, структуру объекта или часто встречающийся сценарий исполнения.

Например, универсальная инструкция доступа к полю объекта может быть заменена инструкцией, предназначенной для работы с объектами определённого класса и содержащей заранее вычисленное смещение поля в памяти. Аналогично, вызов виртуального метода может быть преобразован в специализированную инструкцию, не требующую повторного поиска метода во время исполнения.

Ещё одним подходом к повышению производительности является JIT-компиляция. Несмотря на недостатки, такие как специализация на архитектуры, усложнение отладки, временные затраты на компиляцию, она даёт значительный прирост производительности [9].

В данной работе не будет рассматриваться JIT-компиляция. Основное внимание будет уделено остальным перечисленным методам оптимизации.

1.2 Метод шитого кода

В статье [4] подробно рассматриваются методы диспетчеризации инструкций и их влияние на эффективность интерпретаторов. Авторы этой статьи определяют эффективный интерпретатор, таким они называют интерпретатор, основной целью разработки которого ставилось эффективное исполнение.

Авторы статьи замечают что у всех эффективных интерпретаторов есть схожие черты, которые мешают эффективному исполнению.

1.2.1 Проблема косвенных переходов в интерпретаторах

Поскольку все эффективные интерпретаторы используют байт-код, возникает проблема диспетчеризации инструкций. Она заключается в том, что независимо от типа диспетчеризации и при отсутствии дополнительных оптимизаций для исполнения одной инструкции байт-кода необходим один косвенный переход.

Это объясняется тем, что для выполнения каждой инструкции её необходимо сначала считать, а затем вызвать соответствующий обработчик. Хотя методы диспетчеризации могут различаться, адрес перехода всегда определяется во время работы программы. Поэтому такой переход обязательно является косвенным.

Таким образом, невозможно полностью избавиться от подобных переходов выбором метода диспетчеризации. Выбор метода диспетчеризации может лишь повлиять на вероятность правильного предсказания такого перехода.

1.2.2 Основные методы диспетчеризации

Существует два основных метода диспетчеризации.

1.2.2.1 Диспетчеризация на основе оператора выбора

Этот тип диспетчеризации является самым простым для реализации. Такой метод заключается в использовании конструкции `switch-case` для выбора обработчика очередной инструкции.

При использовании такого типа диспетчеризации путь исполнения, как правило, соответствует схеме, представленной на рисунке 1. Сначала происходит чтение кода инструкции, после чего выполняется переход к соответствующему обработчику. Заметим, что из этой точки может перейти в любой из обработчиков. По завершении обработки управление возвращается к коду, который осуществляет чтение следующей инструкции.

Рассмотрим тривиальный пример с циклом и использованием простейшего предсказателя. Такой предсказатель будет предполагать, что в очередной раз исполнение перейдёт туда же, что и в прошлый. Будем считать, что псевдобайт-код выглядит как на листинге 1.

В идеальном случае хотелось бы получить следующий результат: первая итерация цикла будет состоять из неудачных предсказаний, а уже со следующей итерации все предсказания будут удачные.

```
loop:
  inc a
  jmp loop
```

Листинг 1 — псевдо байт-код простейшего цикла

Однако при использовании такого метода диспетчеризации неудачные предсказания будут продолжаться. Предположим, что прошла одна итерация цикла, тогда предсказатель запомнил, что в последний раз управление перешло к обработчику инструкции `jmp`, но очередная инструкция окажется `inc`. Затем он запомнит, что последний переход был на инструкцию `inc`, но следующая инструкция будет `jmp`, и так далее. Предсказателю не хватает информации, поскольку все переходы совершаются из одного места.

1.2.2.2 Диспетчеризация на основе метода шитого кода

Как видно из примера, необходимо увеличить количество мест, из которых выполняется косвенный переход. Имея больше таких точек, предсказатель получает больше контекста о структуре исполняемого кода.

Для того чтобы увеличить количество таких мест, можно дублировать участки, отвечающие за получение кода инструкции и переход к соответствующему обработчику, в конец обработчиков инструкций. Тогда пусть исполнение будет соответствовать схеме, предоставленной на рисунке 2.

В этом случае использование оператора выбора становится менее удобным, и возникает идея использовать указатели на соответствующие обработчики. То есть для получения очередной инструкции нужно прочитать её байт-код, по которому можно получить указатель на обработчик, и выполнить переход по адресу. Переход

остаётся косвенным, однако теперь такие переходы осуществляются непосредственно из обработчиков.

Снова рассмотрим пример с циклом. На первой итерации происходят неудачные предсказания, а уже начиная со второй предсказания становятся верными. Предсказатель запомнит, что после инструкции `inc` следует `jmp`, а после `jmp` — `inc`, и будет давать верные предсказания.

Этот пример сознательно упрощён: в реальных программах инструкции внутри циклов могут повторяться, а используемые предсказатели значительно сложнее. Однако такой пример отражает основную идею.

У метода шитого кода есть два подхода к реализации. Существует прямой шитый код и косвенный шитый код. Оба подхода реализуют вышеописанную идею, но по-разному.

1.2.2.2.1 Прямой шитый код

В прямом шитом коде (*Direct Threaded Code*, сокращенно *DTC*) в качестве байт-кодов используются непосредственно адреса обработчиков инструкций. Такой метод является более быстрым, поскольку для перехода к обработчику необходимо сделать лишь одно разыменование. Однако он имеет свои недостатки, например, для его использования требуется больше памяти, чем в случае косвенного шитого кода. Именно этот метод рассматривается в статье [4].

1.2.2.2.2 Косвенный шитый код

Косвенный шитый код (*Indirect Threaded code*, сокращенно *ITC*), метод, описанный в [5], использует представление, при котором хранится не адрес обработчика, а указатель на структуру в таблице. Такая структура хранит указатель на обработчик инструкции, параметры и данные, необходимые для выполнения операции. Таким образом, для перехода к обработчику инструкции необходимо два разыменования: сначала нужно получить адрес структуры, а затем адрес соответствующего обработчика.

Преимущества данного метода заключаются в том, что обработчики инструкций могут быть реализованы в виде архитектурно-зависимых библиотечных функций, а байт-код при этом не зависит от конкретной платформы. Это происходит благодаря тому, что компилятор байт-кода записывает лишь индекс соответствующего элемента таблицы, а не адрес машинного обработчика. Если таблицы будут унифицированы для всех архитектур, то один и тот же байт-код можно будет исполнять на разных архитектурах, используя скомпилированные, под конкретную архитектуру, обработчики.

1.2.3 Сравнение

В этой части будут рассмотрены различные виды интерпретаторов и проведено сравнение методов шитого кода и конструкции выбора. Вся статистика взята из работы [4].

Таблица 1 — типы интерпретаторов

Система	**Тип виртуальной машины**	**Тип диспетчеризации**	**Проверка типов времени исполнения**
Gforth 0.4.9	Stack	ITC	Нет
Ocaml 2.04	Stack	DTC / switch	Нет
Scheme48 0.53	Stack	switch	Да
Yap 4.3.2	RAM	DTC / switch	Да
Perl (SPEC95)	—	list IR	Нет
Xlisp (SPEC95 li)	—	tree IR	Нет

1.2.3.1 Сравнение архитектур

В таблице 1 приведены различные виды интерпретаторов, указаны типы их виртуальных машин, методы получения очередной инструкции, а также информация о наличии проверок типов во время исполнения.

Заметим, что последние две системы значительно отличаются от остальных: они не используют виртуальные машины. В качестве промежуточного представления используются древовидное и списочное. Остальные используют стековые или регистровые виртуальные машины. В качестве метода получения очередной инструкции применяется метод шитого кода, прямой шитый код (DTC) или косвенный шитый код (ITC).

Некоторые интерпретаторы поддерживают как метод шитого кода, так и оператор выбора (switch). Это необходимо, поскольку, если следовать стандарту ANSI C, выразить метод шитого кода на языке C невозможно. Однако некоторые компиляторы предоставляют расширения, позволяющие реализовать такой метод. Тогда во время компиляции интерпретатора, если есть возможность, будет использован метод шитого кода, иначе — оператор выбора.

Таблица 2 — результаты измерения производительности

interpreter	workload	inst. exec.	loads	stores	branches	indirect-branches		inst./ind.
gforth	benchgc	64396699	40.7%	10.5%	14.5%	13.0%	8380089	7.6
gforth	prims2x	94962783	39.4%	8.8%	18.4%	10.3%	9841564	9.6
ocaml	ocamlc	66439963	26.4%	10.2%	17.5%	6.3%	4204772	15.8
ocaml (switch)	ocamlc	91550037	21.8%	6.1%	21.5%	4.5%	4204772	21.7
ocaml	ocamllex	69738725	29.5%	10.7%	19.8%	11.3%	7918321	8.8
ocaml (switch)	ocamllex	122558868	22.2%	5.1%	24.3%	6.4%	7918321	15.4
scheme48	build	100000003	27.9%	12.1%	20.0%	3.2%	3275171	30.5
yap	boyer	68153580	32.9%	11.7%	19.7%	5.4%	3681492	18.5
yap (switch)	boyer	97326209	24.8%	8.2%	24.2%	3.7%	3681492	26.4
yap	chat	15510382	33.4%	14.5%	17.1%	5.5%	864703	17.9
yap (switch)	chat	22381662	25.6%	10.0%	22.4%	3.8%	864703	25.8
perl	jumble	2391904893	25.8%	17.8%	19.3%	0.7%	17090181	140.0
xlisp	boyer	173988551	26.1%	16.8%	22.7%	1.1%	1858367	93.6

Рассмотрим последний столбец таблицы 1. Проверка типов во время исполнения означает, что интерпретатор проверяет типы объектов перед выполнением операции. Такие проверки представляют собой косвенные переходы, но поскольку в тестах обычно все объекты имеют один тип, эти переходы легче предсказать.

1.2.3.2 Сравнение исполняемых инструкций

В работе приведено сравнение некоторых интерпретаторов по их поведению, а точнее — по количеству исполняемых инструкций ветвления. Для получения данных использовалась симуляция [4].

Рассмотрим таблицу 2. Она содержит следующие столбцы: `interpreter` — название интерпретатора, `workload` — тип бенчмарка, `inst. exec.` — количество исполненных и вступивших в силу инструкций. Далее идут столбцы, которые содержат пропорции инструкций загрузки,

сохранения, ветвления, а также косвенных инструкций ветвления (не считая возвратов) ко всем исполненным инструкциям. Следующий столбец содержит абсолютное количество исполненных косвенных переходов, после чего вычислена пропорция исполненных инструкций на один косвенный переход.

Количество инструкций на косвенный переход у `perl` и `xlisp` значительно отличается от остальных. Это происходит из-за того, что они не используют байт-код.

Если рассмотреть интерпретаторы, поддерживающие два метода диспетчеризации, наблюдается рост количество исполненных инструкций, при использовании оператора switch. Также возрастает количество инструкций на косвенный переход. Это объясняется наличием дополнительных проверок, которые необходимо совершить для перехода к обработчику.

Для `Yap` и `Scheme48` количество инструкций на косвенный переход выше. Это объясняется тем, что эти языки имеют проверки типов во время исполнения. А также `Yap` имеет регистровую виртуальную машину, а такой тип виртуальной машины предполагает бóльшие расходы на декодирование аргументов по сравнению со стековой [10].

1.2.4 Ускорение непрямых переходов

В этой части будут рассмотрены результаты симуляции с использованием различных типов предсказателей. Вся статистика заимствована из работы [4].

1.2.4.1 Виды предсказателей

- **Нет предиктора**;
- **BTB** — сохраняет результат последнего перехода;
- **Profile Guided** — всегда предсказывает адрес, который был самым частым в тренировочном запуске;
- **BTB 2** — вариант BTB, который даёт возможность на две ошибки перед сменой адреса перехода;
- **2-level** с бесконечным размером истории. Однако в режиме cycle-accurate возникает проблема: процессор не знает реальный адрес перехода и поэтому не может использовать его как ключ для предсказания. В связи с этим используются следующие варианты предикторов:
  - **2-Level-speculative** — регистр истории предсказаний хранит n последних предсказанных адресов. То есть предсказания обновляются спекулятивно, когда исполнение переходит на предсказанный адрес. Этот адрес может быть неверным, предсказания при этом не удаляются.
  - **2-Level-commited** — регистр истории хранит n последних подтверждённых адресов. То есть значение заносится в историю после подтверждения. В таком случае история может отставать от исполнения, поскольку подтверждение может происходить во время предсказания следующего адреса.

1.2.4.2 Результаты тестирования

Закономерности:

- **Threaded Code**: PG даёт результат в 71–88% неудач, BTB — 57–63%, BTB2 — 50–61%;
- **Switch based**: PG даёт результат в 84–99% неудач, BTB и BTB2 — 98% для `ocaml` и `scheme48`, 81–86% для `Yap`;
- **lisp**: BTB и BTB2 дают результат в 14–21% неудач. Такая высокая эффективность связана с тем, что большинство косвенных вызовов в тесте были связаны с динамическим определением типов, а не с dispatcher интерпретатора;
- **perl**: PG, BTB и BTB2 дают около 75%;
- **2-level** показал хорошие результаты (менее 10% неудач).

Разница в результатах для шитого кода и конструкции выбора заключается в том, что для каждой инструкции виртуальной машины есть свой путь диспетчеризации. Таким образом, имеется больше контекста для предсказания. При использовании конструкции выбора необходимо предсказывать каждый раз, имея один и тот же контекст. То есть профилирование предсказывает самую частую инструкцию, BTB — последнюю исполненную. И такие предсказания более верны для RAM-машин.

В таблице 3 представлена разность в количестве процессорных циклов, затрачиваемых на диспетчеризацию байт-кода, при использовании различных предикторов, в случае штрафа за неверное предсказание в 4 цикла.

Таблица 3 — разность в скорости исполнения между шитым кодом и конструкцией выбора

predictor	**cycles**
none	3.9-4.5
PG	4.4-5.4
BTB	5.3-6.3
BTB2	5.5-6.6
2-level	2.2-3.1

Разность между двумя типами диспетчеризации возникает из-за разницы в частоте ошибочных предсказаний и разницы в количестве инструкций, которые нужно исполнить для диспетчеризации. Большая разница между профилированием, BTB, BTB2 и отсутствием предсказателя возникает из-за высокого уровня ошибочных предсказаний для оператора выбора при использовании таких предсказателей. Для 2-level предсказателя разность получается небольшой из-за схожего уровня ошибочных предсказаний, а верные предсказания помогают сгладить разницу в количестве инструкций, необходимых для диспетчеризации.

В случае если время выполнения инструкции виртуальной машины достаточно мало, использование оператора выбора оказывается медленнее до 2,02 раза.

2 Разработка базовой реализации интерпретатора на языке высокого уровня

Для получения базовых значений производительности в рамках данной работы была разработана реализация интерпретатора байт-кода на языке высокого уровня, используемая в качестве базовой для последующего сравнения с оптимизированными версиями интерпретатора.

Использование языка высокого уровня накладывает ограничения на управление памятью и потоками исполнения. В частности отсутствует возможность явного выбора места размещения отдельных значений, что приводит к дополнительным накладным расходам. Вместе с тем, разработка такой реализации позволяет выявить основные источники потерь производительности, которые могут быть устранены в последующих версиях интерпретатора.

2.1 Описание окружения

Работа проводится в рамках исследовательской многоязыковой виртуальной машины Huawei. Байт-кодом данной виртуальной машины является формат CBC (Cangjie Byte Code).

В качестве языка реализации был выбран язык AJ, представляющий собой расширение языка Java с добавлением специализированных аннотаций, предназначенных для разработки системных приложений. Использование AJ позволяет интегрировать

интерпретатор в существующий инструментальный конвейер и использовать возможности платформы выполнения.

2.1.1 Формат байт-кода CBC

Рассмотрим особенности представления инструкций в байт-коде CBC. Инструкции в данном формате имеют переменную длину. Часть инструкций кодируется 16 битами, в то время как другие могут иметь длину 24 бита, некоторые 32.

Виртуальная машина предоставляет набор регистров, включающий:

- 13 основных регистров общего назначения;
- 17 дополнительных регистров общего назначения;
- нулевой регистр;
- регистры для операций с плавающей точкой (16 основных и 16 дополнительных).

Для кодирования номера основного регистра достаточно 4 бит, в то время как для обращения к дополнительным регистрам используется специальный префикс.

В рамках данной работы, для упрощения реализации интерпретатора, используются только основные регистры общего назначения, что не оказывает принципиального влияния на рассматриваемые аспекты производительности.

Пример структуры инструкции байт-кода приведён на рисунке 3. Первые 8 бит определяют код операции, после чего следуют два операнда по 4 бита каждый.

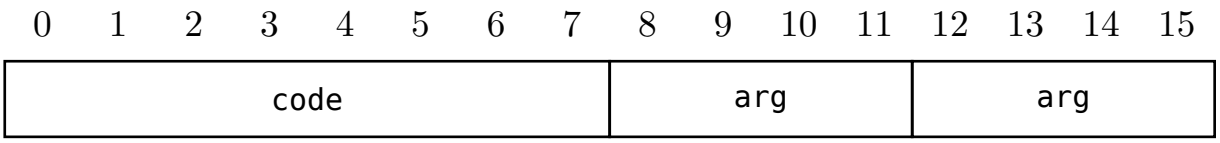


Рисунок 3 — пример структуры инструкции байт-кода

2.1.2 Переписыватель байт-кода

Для упрощения интерпретации байт-кода используется специальный переписыватель байт-кода. Сейчас в его задачи входит разрешение ссылок и вычисление смещения полей объектов. Однако возможно использовать переписыватель для модификации байт-кода в целях ускорения интерпретации.

2.2 Структура

Структура интерпретатора была выбрана с учётом преимуществ и ограничений языка высокого уровня. Интерпретатор содержит следующие поля:

– Регистры виртуальной машины.

– Поток байтов инструкций виртуальной машины.

– Фрейм метода исполняемой функции.

В момент перехода управления к интерпретатору происходит инициализация всех полей, а также фрейма метода.

Затем происходит переход к чтению первых восьми бит инструкции и определение необходимого обработчика. Выбор обработчика делается при помощи конструкции `switch`. Затем в обработчике происходит дополнительное чтение аргументов и/или типа операции. После чего

происходят обращения к регистрам виртуальной машины и/или вызов необходимых методов.

Для того чтобы избежать копирования кода и увеличить скорость разработки и поддержки новых инструкций, были созданы обобщенные методы вычисления операций. Такие методы принимают операнды и тип операции, возвращая результат.

При помощи этих методов упрощается поддержка разных типов инструкций, поскольку множество инструкций содержит одинаковые операции с разными аргументами. Например, операция сложения представлена в виде трехадресной инструкции, двухадресной инструкции, инструкции с константой. Также использование обобщенных методов упрощает поддержку новых операций.

2.3 Выявленные проблемы

В результате изучения получившегося объектного файла были выявлены недостатки такого подхода. Недостатки, в основном, связаны с ограничениями языка высокого уровня. Рассмотрим каждый из них подробнее.

2.3.1 Обращения к памяти

В силу того, что позиция байт-кода является полем интерпретатора, её обновление всегда влечёт за собой обновление соответствующего поля. Обновление поля объекта приводит к прямой записи в память — ресурсоёмкой операции с высокой задержкой.

Для доступа к самому байт-коду также требуются два разыменования: одно — для получения адреса, по которому расположен байт-код, другое — для чтения очередных инструкций.

2.3.2 Подсчет общих выражений

Поскольку после диспетчеризации по первым восьми битам может выполняться либо дальнейшая диспетчеризация, либо чтение аргументов инструкции, компилятор заранее добавил вычисление новых позиций в байт-коде ещё до завершения первой диспетчеризации.

В результате вычисление обновленных позиций байт-кода выполняется независимо от того, потребуется ли оно в дальнейшем. Таким образом, для части инструкций вычисляются значения, которые впоследствии не используются.

2.3.3 Генерация серии условных выражений

Поскольку выбор обработчика инструкции представлен в виде одной большой конструкции выбора. В рассматриваемом случае компилятор генерирует последовательность условных переходов. При такой реализации поиск обработчика имеет сложность $O(N)$, где N — количество обработчиков. В общем случае компилятор может использовать и другие схемы диспетчеризации. Вероятно, компилятор не использовал таблицу переходов из-за низкой плотности ключей: в рассматриваемом случае используется лишь около 20% пространства возможных кодов инструкций.

В рассматриваемом модельном примере это не оказывает существенного влияния на производительность, однако при значительном количестве поддерживаемых инструкций такие накладные расходы могут стать заметными.

Тем не менее, подобная структура, вероятно, обладает и неожиданным преимуществом. Последовательность условных переходов может оказаться более удобной для предсказателя переходов. Во время исполнения большинство проверок завершается результатом `false`, и только последняя приводит к переходу на обработчик инструкции. Поскольку в данном случае используются условные, а не косвенные переходы, предсказателю, возможно, проще выявлять закономерности исполнения и адаптироваться к ним.

2.4 Некоторые оптимизации

Для того чтобы сделать базовую версию более конкурентноспособной и проверить свои предположения, были применены некоторые оптимизации. Некоторые из оптимизаций потребовали небольших изменений в компиляторе AJ и добавления специальных аннотаций.

2.4.1 Оптимизация вызовов

Для более оптимальной обработки вызовов было принято решение во время инициализации интерпретатора выделить место для фреймов вызовов. Таким образом происходит экономия на вызовах `malloc`. При каждом вызове на заранее выделенном стеке выделяется нужное

количество памяти. Эта операция имеет значительно меньшие накладные расходы.

2.4.2 Оптимизация диспетчеризации инструкции

Для того чтобы избежать последовательного перебора инструкций, можно использовать выбор инструкции при помощи таблицы диспетчеризации. После прочтения ключа инструкции, интерпретатору нужно лишь посмотреть соответствующую ячейку в таблице, для того чтобы перейти к нужному обработчику.

Поскольку ключи реализованных обработчиков покрывают лишь небольшую часть пространства возможных значений и имеют значительный разброс, компилятор не генерировал таблицу переходов автоматически. Вероятно, с точки зрения компилятора такая реализация была нецелесообразной из-за низкой плотности ключей. Для возможности принудительного использования таблицы диспетчеризации в компилятор AJ была добавлена специальная аннотация `@TableJump`.

3 Реализация базовой версии интерпретатора на языке низкого уровня

В качестве низкоуровневого языка был выбран ассемблер архитектуры ARMv8. Данный выбор обусловлен необходимостью прямого контроля над выполнением инструкций и использованием особенностей целевой архитектуры при реализации интерпретатора.

3.1 Описание окружения

Для упрощения разработки и запуска интерпретатора было принято решение встроить его в уже существующую платформу. Это позволяет повторно использовать процессы сборки, загрузки и выполнения программ, а также упростит интеграцию с существующим инструментарием. Язык AJ содержит специальные аннотации, которые упрощают интеграцию с программой, написанной на ассемблере.

3.2 Выбранные оптимизации

При разработке базовой реализации интерпретатора были выбраны известные методы повышения производительности интерпретаторов байт-кода. Основной целью данных оптимизаций является снижение накладных расходов, вызванных диспетчеризацией инструкций, а также уменьшение количества обращений к памяти.

В частности, были применены следующие методы:

– табличная диспетчеризация инструкций (table switch);

– устранение обратных переходов в цикле интерпретации за счёт дублирования кода диспетчеризации;

– размещение локальных переменных интерпретатора в регистрах процессора;
– использование стека для сохранения фреймов вызываемых методов.

Далее рассмотрим реализацию каждого из этих методов в разработанном интерпретаторе.

3.2.1 Табличная диспетчеризация инструкций

Для того чтобы избавиться от большого количества последовательных if-else, используется табличная диспетчеризация. Для диспетчеризации используется таблица переходов размером 2^{12}. Индексом в таблице является двенадцатибитный код инструкции, а в качестве значений хранятся указатели на соответствующие обработчики.

Для перехода к нужному обработчику необходимо лишь прочитать очередные 16 бит инструкции, замаскировать старшие 4 бита и перейти к соответствующему обработчику, адрес которого находится в таблице по соответствующему индексу.

3.2.2 Устранение обратных переходов

Однако добавление только табличной диспетчеризации может оказать негативное влияние на производительность. Эта оптимизация является лишь одной из составляющих метода шитого кода. Для реализации метода шитого кода, необходимо также вставить диспетчеризацию в конце каждого обработчика. Таким образом будут удалены обратные дуги на единый код диспетчеризации.

3.2.3 Размещение локальных переменных интерпретации в регистрах

Для того чтобы избежать большого количества доступов к памяти необходимо разместить основные переменные интерпретации на регистрах. Следующие переменные были размещены на регистрах:

– текущая позиция байт-кода и адрес конца байт-кода;

– адрес структуры содержащей регистры виртуальной машины;

– адрес таблицы обработчиков.

При помощи такой оптимизации удалось значительно сократить количество обращений к памяти. Чтение новых инструкций теперь требует лишь одного разыменования, обновление позиции не требует разыменований. Адрес таблицы обработчиков вычисляется один раз при запуске интерпретатора и затем хранится в регистре.

3.2.4 Использование стека для сохранения фреймов вызываемых методов

Поскольку машинный стек не использовался, было принято решение использовать его для реализации вызовов. Для этого при создании интерпретатора на стек кладётся флаг, означающий, что больше фреймов на стеке нет.

При вызове очередной функции неволатильные регистры виртуальной машины складываются на стек, также сохраняется текущая позиция байт-кода. После этого происходит вызов метода интерпретатора, в котором выполняются действия, необходимые для

получения новой позиции байт-кода. После возврата осуществляется обновление текущей и конечной позиций в регистрах.

Когда необходимо выполнить возврат из метода, происходит проверка флага на стеке. Если флаг не установлен (на стеке есть фреймы), происходит восстановление неволатильных регистров виртуальной машины. Если же на стеке установлен флаг, необходимо выполнить возврат из интерпретатора.

3.3 Дополнительные оптимизации

После внедрения основных оптимизаций, описанных в предыдущем разделе, были проанализированы объектные файлы, полученные после ассемблирования. Анализ показал наличие ряда недочётов, допущенных при разработке интерпретатора, которые могут оказывать влияние на его производительность.

3.3.1 Изменения в коде диспетчеризации

Поскольку код диспетчеризации выполняется для каждой инструкции, его размер необходимо минимизировать. В связи с этим была удалена проверка границ байт-кода. Ранее перед выполнением каждой инструкции выполнялось сравнение указателя текущей позиции байт-кода с указателем его конца. В новой версии предполагается, что при подаче на вход интерпретатора некорректного байт-кода его поведение считается неопределённым.

Также был изменён инвариант, связанный с указателем на текущую позицию байт-кода. Ранее указатель ссылался на следующий

непрочитанный байт, что требовало его инкремента после чтения каждого байта. Теперь указатель указывает на начало исполняемой инструкции. Это позволяет выполнять инкремент только при переходе между инструкциями, причём он может совмещаться (в рамках одной инструкции) с чтением первых 16 бит следующей инструкции.

3.3.2 Выравнивание

В ходе анализа объектных файлов было выдвинуто предположение о целесообразности выравнивания. При этом выравниванию могут подвергаться как обработчики инструкций, так и таблица указателей на них. Выравнивание таблицы указателей потенциально может повысить производительность за счёт улучшения локальности данных. Аналогично, выравнивание обработчиков инструкций может уменьшить количество пересечений границ кэш-линий и упростить предвыборку инструкций процессором. Однако эффективность подобных оптимизаций существенно зависит от архитектуры процессора и структуры исполняемого кода.

3.3.3 Знаковое расширение

Поскольку операнды часто хранятся в более компактном виде, чем используются (например, аргумент задаётся двумя байтами, а используется как восьмибайтовое значение), возникает необходимость выполнения знакового расширения. Значительную часть таких операций можно исключить, если выполнять чтение сразу с расширением.

Данная оптимизация позволила сократить количество инструкций в ряде обработчиков.

4 Сравнение эталонной реализации на языке низкого уровня и базовой реализации на языке высокого уровня

В этом разделе проводится сравнение различных вариаций интерператоров. Для каждой реализации были проведены измерения производительности на наборе тестовых сценариев. В качестве метрики было выбрано время выполнения тестовой программы в секундах. Тестирование проводилось на системе со следующими характеристиками:

- операционная система: Ubuntu 20.04.6 LTS (Focal Fossa);
- процессор: HiSilicon Kunpeng 920 (архитектура ARMv8), 128 вычислительных ядер;
- объём кэш-памяти: L1d — 8 MiB, L1i — 8 MiB, L2 — 64 MiB, L3 — 128 MiB;
- объём оперативной памяти: 125 GB.

4.1 Вариации интерпретаторов

Рассматриваются следующие вариации интерпретатора:

`aj-stack` — Реализация на языке AJ, с используется заранее выделенный стек для реализации вызовов.

`asm-base` — Реализация на языке ассемблера архитектуры ARMv8 с оптимизациями, описанными в разделе 3.2.

`asm-align-all` — Реализация на языке ассемблера архитектуры ARMv8 с оптимизациями, описанными в разделе 3.3.

asm-align-table — Реализация на языке ассемблера архитектуры ARMv8 с оптимизациями, описанными в разделе 3.3, однако выравнивание было применено только для таблицы с указателями на обработчики.

asm-no-align — Реализация на языке ассемблера архитектуры ARMv8 с оптимизациями, описанными в разделе 3.3, без применения выравниваний.

Вариация `aj-stack` была выбрана, поскольку базовая реализация показала низкую эффективность при большом количестве вызовов, а оптимизация, описанная в пункте 2.4.2, оказалась неэффективной. Дело в том, что без данной оптимизации генерируется последовательность условных переходов, которые выполняют диспетчеризацию инструкции. В этом случае, в рамках цикла, предсказатель переходов вероятно обучается и начинает корректно предсказывать, когда результат проверки будет положительным, а когда отрицательным.

Использование таблицы диспетчеризации может приводить большому количеству ошибок в предсказаниях, поскольку в этом случае существует только одна точка диспетчеризации инструкции. И хотя асимптотически этот метод лучше, в случае малого количества обработчиков он, по-видимому, уступает.

4.2 Вариации тестовых программ

Для проведения экспериментального сравнения были разработаны наборы тестовых программ (бенчмарков), предназначенные для

выявления различий в производительности различных вариаций интерпретаторов в разных сценариях.

Каждая тестовая программа представляет собой отдельную функцию. Во всех тестах, за исключением `bench_call_mess`, основная нагрузка формируется внутри цикла, в теле которого выполняется последовательность инструкций. Структура последовательности инструкций различается между тестами и описана ниже.

Под типом инструкции понимается операция виртуальной машины. При этом из-за особенностей реализаций интерпретаторов на языке высокого уровня и на языке ассемблера, одни и те же инструкции могут проходить разный путь диспетчеризации.

`call_mess` — Тест, состоящий преимущественно из вызовов функций. Предназначен для оценки оптимизаций, направленных на ускорение вызовов внутри виртуальной машины.

`flip` — Цикл содержит несколько инструкций одного типа, после чего следует большое количество инструкций другого типа.

`flip_large` — В цикле последовательно выполняются группы инструкций нескольких типов. В тесте представлено 3 типа, в пропорции 1:1:2

`pred` — В цикле поочередно выполняются инструкции нескольких типов.

`unpred` — В цикле последовательно выполняются инструкции двух типов, в пропорции 1:2.

seq — В теле цикла последовательно выполняются группы инструкций различных типов. Всего используется шесть типов инструкций.

4.3 Методика измерений

Для каждой вариации интерпретатора и тестовой программы проводилась серия из 48 запусков. Первые восемь запусков использовались для разогрева и не учитывались при расчёте итоговых показателей. Таким образом, в статистическую выборку включались результаты оставшихся 40 запусков ($n = 40$).

Для каждой серии измерений вычислялись среднее значение времени выполнения, а также выборочное стандартное отклонение, определяемое по формуле:

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2}. \tag{1}$$

Для построения 95% доверительного интервала использовалось распределение Стьюдента с $n - 1 = 39$ степенями свободы. Квантиль уровня 0.975 для данного распределения равен

$$t_{0.975,39} \approx 2.02. \tag{2}$$

Границы доверительного интервала вычислялись по формуле

$$\mathrm{CI} = \overline{x} \pm t_{0.975,39} \cdot \frac{s}{\sqrt{n}}. \tag{3}$$

Доверительные интервалы использовались для качественной оценки устойчивости различий между результатами измерений.

Формальные проверки статистических гипотез в рамках работы не проводились.

4.4 Результаты

На рисунке 4 представлены результаты тестирования в виде диаграммы. Вертикальная ось отражает время выполнения программы; значения нормированы относительно версии `aj-stack` (в процентах от неё). На концах столбцов показаны границы доверительных интервалов.

Рассмотрим тенденции, наблюдаемые на графике. Заметно, что использование ассемблера при реализации интерпретатора оказывает значительное влияние на скорость интерпретации (ускорение от 1,8 до 2,5 раза). Также наблюдается неожиданный результат при сравнении `asm-align-table` и `asm-align-all`: во всех случаях выравнивание обработчиков негативно влияет на производительность. Вероятно, это связано с

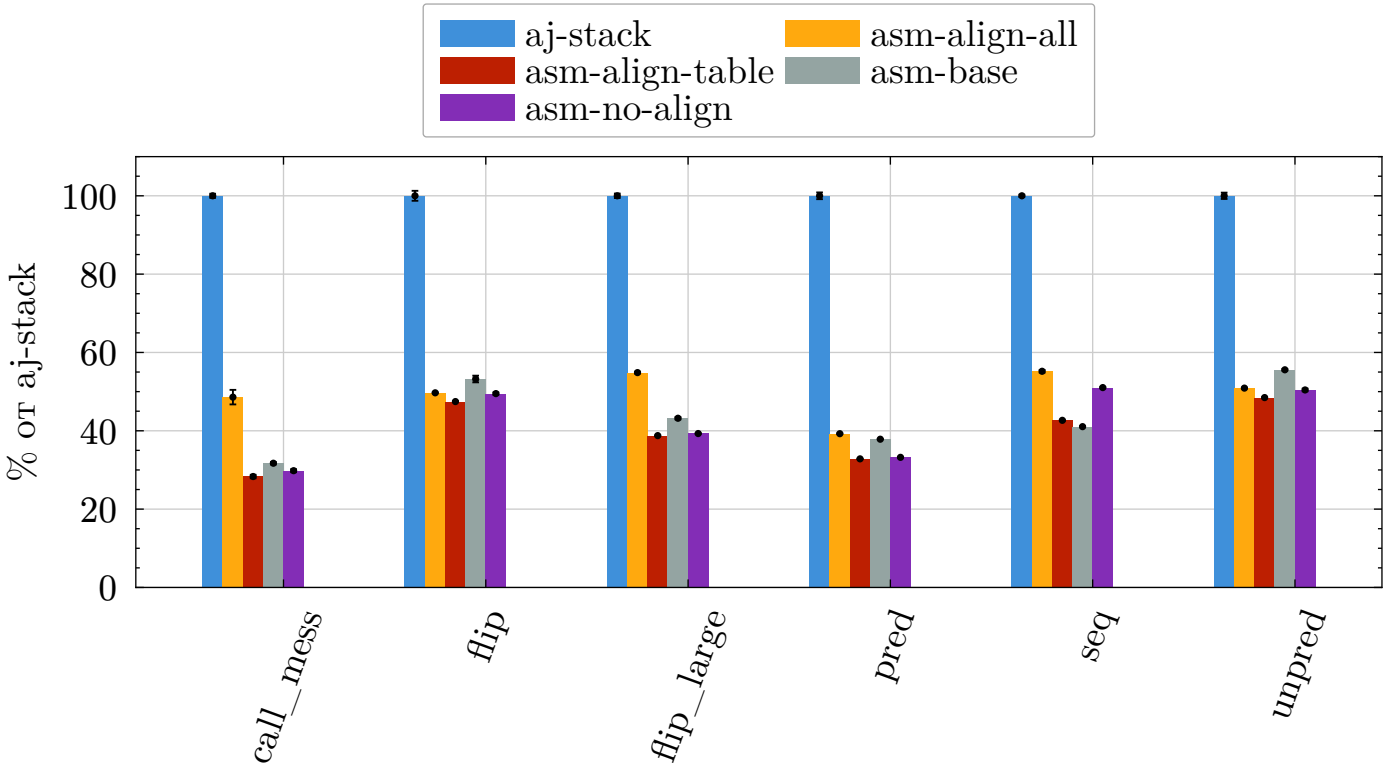


Рисунок 4 — результаты тестирования

увеличением объёма кода обработчиков из-за вставки nop-инструкций, что может ухудшать локальность инструкций. Во всех случаях, кроме `seq`, `asm-align-table` оказался лучше, чем `asm-base`.

5 Новый формат байт-кода

Для повышения производительности было принято решение использовать переписыватель для изменения формата исполняемого байт-кода. В новом формате байт-кода реализован метод прямого шитого кода, что позволило исключить избыточное разыменование при диспетчеризации инструкций.

5.1 Переписыватель

В силу того, что интерпретатор реализует язык высокого уровня, имеющий множество динамических возможностей, необходимо проводить разрешение ссылок непосредственно перед интерпретацией или во время неё. До изменений переписыватель использовался в основном для разрешения позднего связывания. Однако была выдвинута идея использовать его также для изменения формата байт-кода.

Для облегчения внедрения новых типов переписывателей в процесс интерпретации уже имеется вся необходимая инфраструктура. Необходимо лишь поддержать нужные инструкции и определить необходимые релокации.

5.2 Формат инструкций

Для повышения эффективности интерпретатора было принято решение оптимизировать байт-код. Оптимизация будет заключаться в следующем: на пути диспетчеризации будет удалена одна косвенность. То есть для перехода к обработчику инструкции необходимо будет выполнить лишь одно разыменование.

Для реализации такой оптимизации потребуется изменить способ представления инструкций в байт-коде. Структуру данных, содержащую обработчики инструкций, назовём сегментом обработчиков. На начало каждого обработчика будет указывать соответствующая метка, а метка первого обработчика будет также называться базой. Предполагается заменить двенадцатибитный ключ инструкции на смещение адреса обработчика относительно базы обработчиков. Вычисление смещений будет производиться перед началом переписывания.

Изменение кодировок будет происходить непосредственно перед началом исполнения очередной функции. Переписывание будет выполняться лениво, не более одного раза для каждой функции, и затрагивать только те функции, которые точно будут исполняться.

5.2.1 Выбор параметров нового формата

Необходимо описать параметры нового формата байт-кода. Сначала необходимо определить ширину смещения до обработчика. Для каждого обработчика это смещение должно иметь одинаковый размер. Было принято значение смещения в 16 бит.

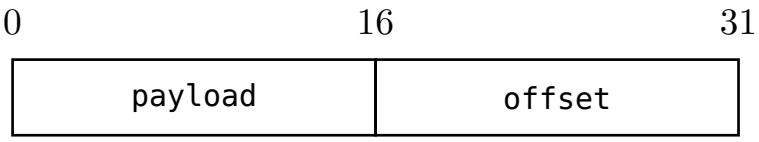
Такое значение было выбрано по следующим соображениям: 16 бит адресуют пространство в 65536 байт. Для архитектуры ARMv8 это 16384 инструкции. Средний размер обработчика составляет около 10 инструкций. Таким образом, имеется около 1600 обработчиков.

Принято решение взять размер одной инструкции, кратный 4 байтам. При необходимости за инструкцией могут следовать дополнительные аргументы размером 4 или 8 байт.

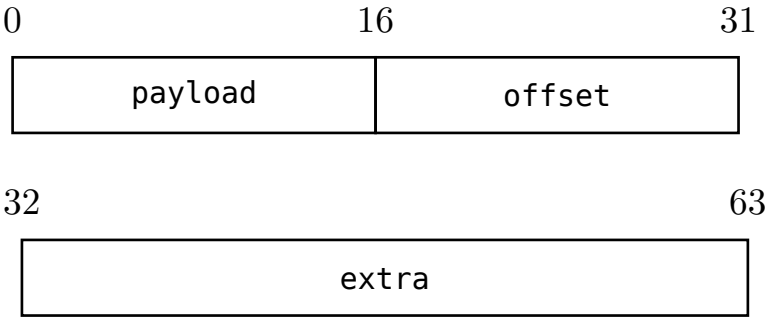
5.2.2 Инструкции

Введем следующие классы инструкций:

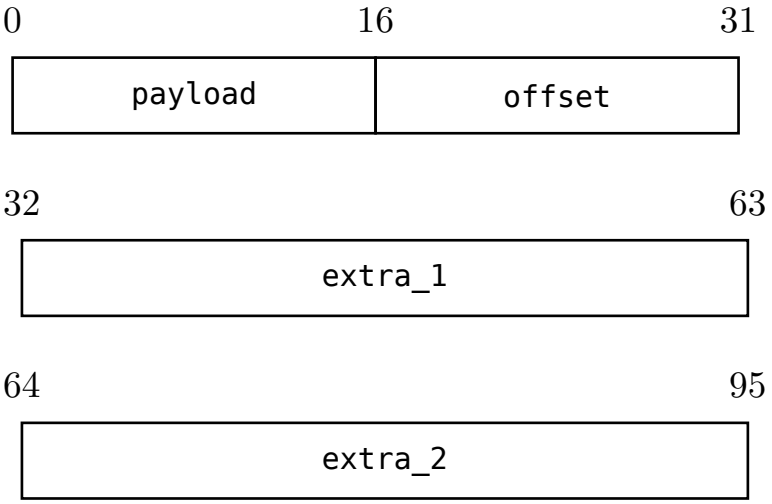
а) Base (базовый). Имеют размер 4 байта и структуру:



б) Base Extended (расширенный). Имеют размер 8 байт и структуру:



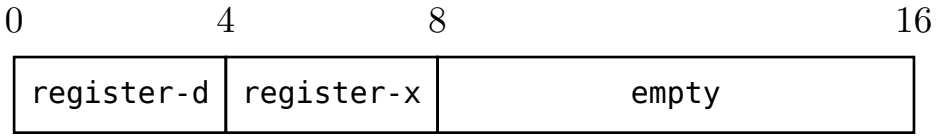
в) Base Extended Extended (длинный). Имеют размер 12 байт и структуру:



5.2.2.1 Инструкции базового размера

5.2.2.1.1 Бинарные арифметические двухадресные инструкции

Полезная нагрузка:



Семантика:

`register-d` ← `register-d op register-x`

Возможные операции:

– сложение;

– вычитание;

– умножение;

– побитовая конъюнкция;

– побитовая дизъюнкция;

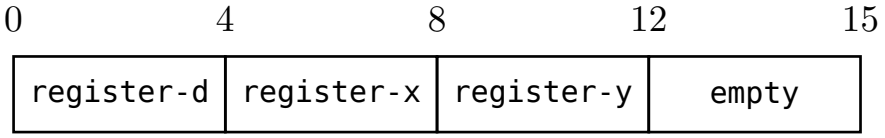
– побитовая исключающее или;

Отдельно стоит отметить операцию копирования значения. Она использует другую семантику:

`register-d` ← `register-x`

5.2.2.1.2 Бинарные арифметические трёхадресные инструкции

Полезная нагрузка:



Семантика:

`register-d` ← `register-x op register-y`

Возможные операции:

– сложение;

– вычитание;

– умножение;

– побитовая конъюнкция;

– побитовая дизъюнкция;

– побитовая исключающее или;

– знаковое деление;

– остаток знакового деления;

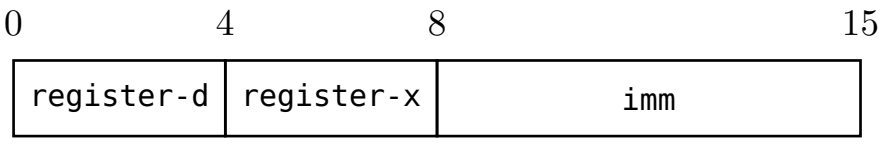
– логический сдвиг вправо;

– логический сдвиг влево;

– арифметический сдвиг вправо;

5.2.2.1.3 Бинарные операции с регистром и 8 битной константой

Полезная нагрузка:



Семантика:

`register-d` ← `register-x op imm`

Возможные операции:

– сложение;

– вычитание;

– умножение;

– побитовая конъюнкция;

– побитовая дизъюнкция;

– побитовая исключающее или;

– знаковое деление;

– остаток знакового деления;

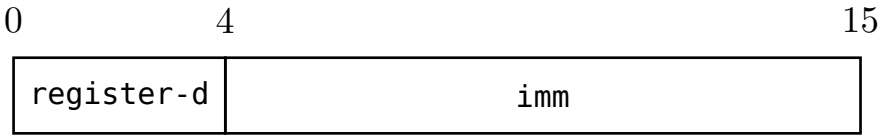
– логический сдвиг вправо;

– логический сдвиг влево;

– арифметический сдвиг вправо;

5.2.2.1.4 Операция копирования значения с 12 битной константой

Полезная нагрузка:

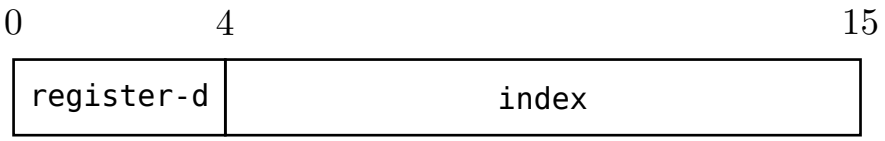


Семантика:

`register-d` ← `imm`

5.2.2.1.5 Инструкция вызова

Полезная нагрузка:

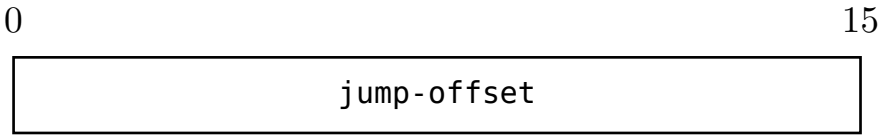


Где поле `index` есть 12ти битный индекс в таблице, содержащей указатели на байт-код методов.

Семантика: происходит вызов метода по индексу `index`. После возврата из вызванной функции значение регистра `ir1` складывается на регистр `register-d`.

5.2.2.1.6 Инструкция перехода с 16 битным смещением

Полезная нагрузка:

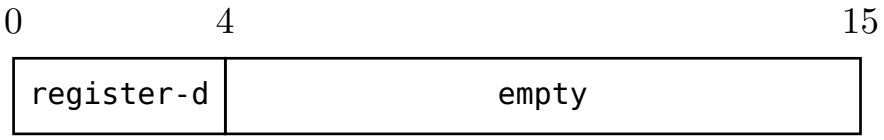


Семантика:

$$\texttt{ip} \leftarrow \texttt{ip} + (\texttt{jump-offset} \ll 2)$$

5.2.2.1.7 Инструкция возврата

Полезная нагрузка:

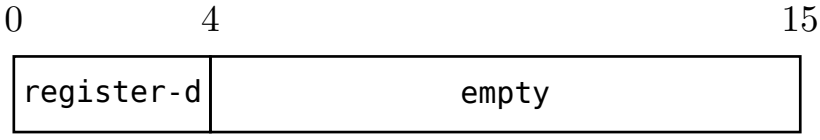


Семантика: Значение регистра `register-d` складывается в регистр `ir1`, после чего осуществляется возврат из текущей функции.

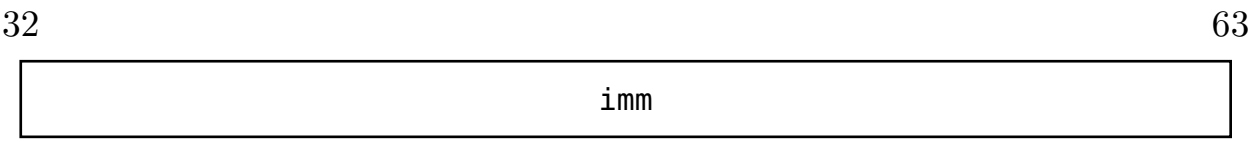
5.2.2.2 Расширенные инструкции

5.2.2.2.1 Бинарные операции с регистром и 32 битной константой

Полезная нагрузка:



Дополнительные аргументы:



Семантика:

$$\texttt{register-d} \leftarrow \texttt{register-x op imm}$$

Возможные операции:

– сложение;

– вычитание;

– умножение;

– побитовая конъюнкция;

– побитовая дизъюнкция;

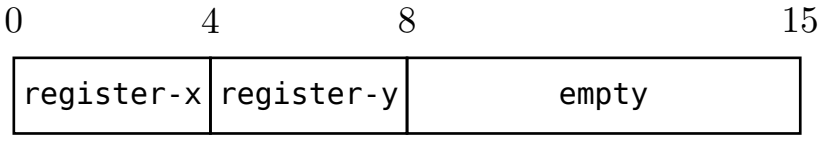
– побитовая исключающее или;

– знаковое деление;

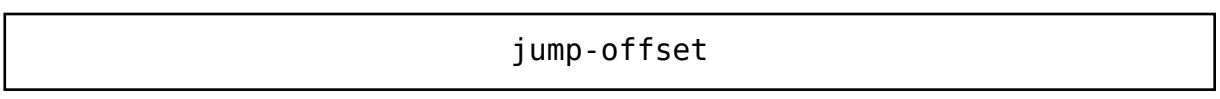
– остаток знакового деления;

5.2.2.2.2 Инструкции ветвления с двумя регистрами

Полезная нагрузка:



Дополнительные аргументы:



Семантика:

$$\text{if } (\texttt{register-x cc register-y})\ \texttt{ip} \leftarrow \texttt{ip} + (\texttt{jump-offset} \ll 2)$$

Возможные условия:

– проверка на равенство;

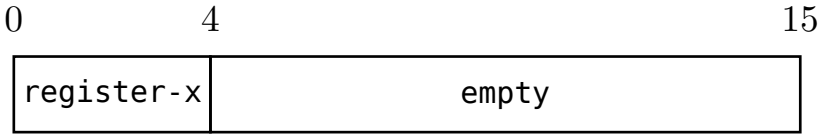
– проверка на неравенство;

– проверка на строго меньше;

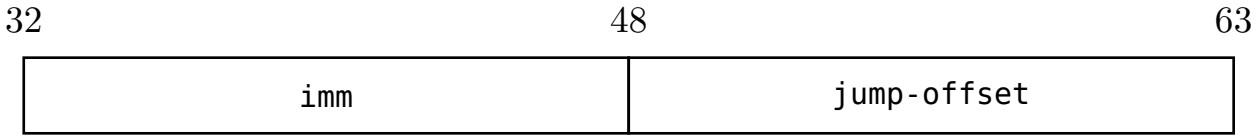
– проверка на больше или равно.

5.2.2.2.3 Инструкции ветвления с регистром и 16 битной константой

Полезная нагрузка:



Дополнительные аргументы:



Семантика:

$$\text{if } (\texttt{register-x cc imm})\ \texttt{ip} \leftarrow \texttt{ip} + (\texttt{jump-offset} \ll 2)$$

Возможные условия:

– проверка на равенство;

– проверка на неравенство;

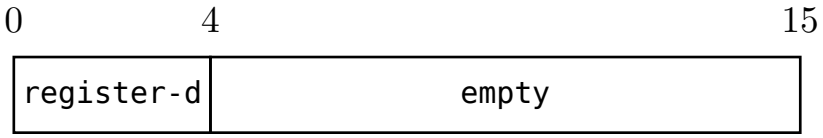
– проверка на строго меньше;

– проверка на больше или равно.

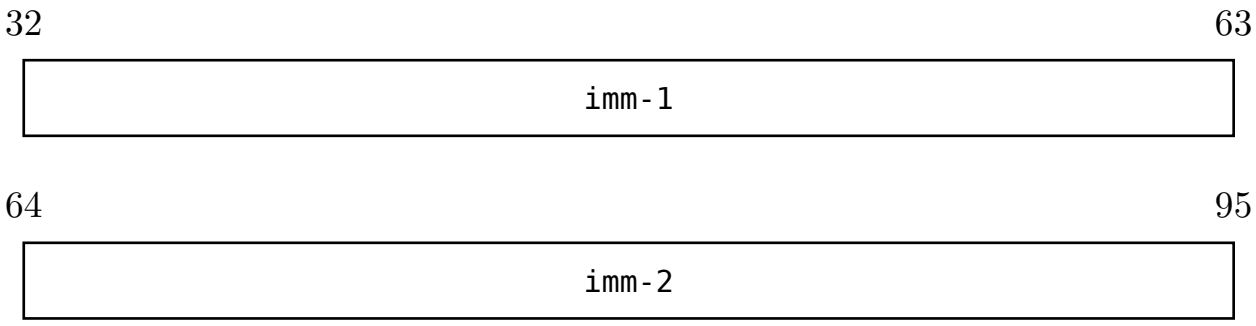
5.2.2.3 Длинные инструкции

5.2.2.3.1 Бинарные операции с регистром и 64 битной константой

Полезная нагрузка:



Дополнительные аргументы:



Семантика:

$$\texttt{register-d} \leftarrow \texttt{register-x op } ((\texttt{imm-2} \ll 32) \mid \texttt{imm-1})$$

Возможные операции:

– сложение;

– вычитание;

– умножение;

– побитовая конъюнкция;

– побитовая дизъюнкция;

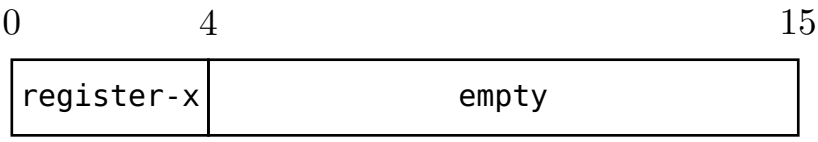
– побитовая исключающее или;

– знаковое деление;

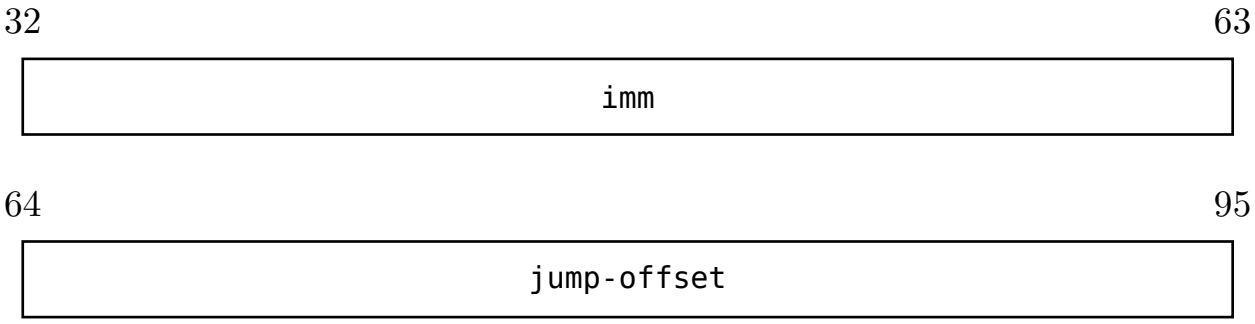
– остаток знакового деления;

5.2.2.3.2 Инструкции ветвления с регистром и 32 битной константой

Полезная нагрузка:



Дополнительные аргументы:



Семантика:

$$\text{if } (\texttt{register-x cc imm})\ \texttt{ip} \leftarrow \texttt{ip} + (\texttt{jump-offset} \ll 2)$$

Возможные условия:

– проверка на равенство;

– проверка на неравенство;

– проверка на строго меньше;

– проверка на больше или равно.

5.3 Интерпретатор

Для исполнения нового формата инструкций был создан новый интерпретатор. Для удобства написания интерпретатора была проведена ручная аллокация регистров. В таблице 4 приведено распределение регистров.

Таблица 4 — таблица распределения регистров

Глобальные переменные		
ip	x21	Текущая позиция в потоке байткода.
registers	x23	Указатель на массив регистров.
interpreter	x23	Указатель на объект интерпретатора.
base	x24	Адрес базы сегмента обработчиков.
Переменные для передачи значения		
v_op	x7	Содержит первые 4 байта инструкции.
v_op_w	w7	Содержит младшие 4 байта v_op.
lv	x3	Левое значение.
rv	x5	Правое значение.
dri	x2	Индекс регистра назначения.
tip	x2	Целевой ip для перехода.
Временные		
t1	x1	Временное значение.
t1_w	w1	Временное значение.
t2	x4	Временное значение.
t2_w	w4	Временное значение.
t3	x6	Временное значение.
t3_w	w6	Временное значение.

5.3.1 Типичный обработчик

Типичный обработчик инструкции представлен на листинге 2. Метка обработчика называется `BE_BRANCH_R_R_EQ`, где `BE` означает `Base Extended`, `BRANCH` — что это инструкция ветвления, `R_R` — что инструкция имеет два регистра в качестве аргументов, `EQ` — что значения регистров будут сравниваться на равенство.

Обработчик имеет следующую структуру: получение аргументов, выполнение операции, переход к следующему обработчику. Рассмотрим листинг 3. На нём представлен макрос, обеспечивающий переключение между обработчиками инструкций. Поскольку сохраняется инвариант

```
BE_BRANCH_R_R_EQ:
        GET_ARGS_BRANCH_BE_R_R

        DO_BRANCH eq

        NEXT 12
```

Листинг 2 — код типичного обработчика инструкции

```
.macro NEXT incamount # incamount -- amount bytes to move ip

  ldr   v_op_w, [ip, \incamount]!
  # Load 4 bytes to v_op: | offset | payload |
  add   t1, base, v_op, lsr 16
  br    t1

  # At the end of macro v_op:
  # | offset | payload |
  # 32      16         0

.endm
```

Листинг 3 — код переключения между обработчиками

— по указателю находится адрес начала обрабатываемой инструкции, макрос принимает один аргумент — размер текущей инструкции.

Сначала происходит чтение 4 байт с предварительным смещением указателя на текущую инструкцию. После чего выполняется получение смещения обработчика при помощи сдвига с одновременной компенсацией базы. Затем выполняется переход по вычисленному адресу.

Рассмотрим листинг 4. На нём представлен код макроса диспетчеризации аргументов инструкции ветвления с двумя регистрами.

Сначала вычисляются индексы регистров операндов. Они лежат в нижних 8 битах слова, прочитанного в макросе NEXT (листинг 3). После этого происходит доступ к значениям, находящимся в регистрах. Затем

```
.macro GET_ARGS_BRANCH_BE_R_R
        and    t1, v_op, 0xF    # Get lv register index
        ubfx   t2, v_op, 4, 4   # Get rv register index

        ldr    lv, [registers, t1, lsl 3] # Load lv value
        ldr    rv, [registers, t2, lsl 3] # Load rv value

        ldrsw  t1, [ip, 4]
        add    tip, ip, t1, lsl #2

        # At the end of macro:
        # tip, lv, rv
.endm
```

Листинг 4 — код диспетчеризации аргументов для инструкции ветвления

считывается и знаково расширяется смещение прыжка: оно занимает 4 байта. Наконец, смещение складывается с текущим указателем на байт-код одновременно со сдвигом влево (семантика описана в 5.2.2.2.2) для получения целевого адреса.

5.3.2 Обработчики вызова и возврата

Теперь рассмотрим обработчики инструкций вызова и возврата. В настоящий момент интерпретатор поддерживает вызовы только интерпретируемых функций. В дальнейшем возможна поддержка переходов из интерпретатора в предварительно скомпилированный машинный код. Возврат может осуществляться как обратно в интерпретатор (если вызов был выполнен из интерпретатора), так и из интерпретатора (в этом случае интерпретация завершается).

Рассмотрим листинг 5. На нём представлен код обработчика вызова метода. Сначала происходит диспетчеризация аргументов, после

чего выполняется чтение волатильных регистров для последующего сохранения их на стек. Затем на стек сохраняется регистр, в который необходимо вернуть значение, а также флаг, который означает, что в данный момент на стеке лежит кадр некоторого метода. После этого в регистр `x0` помещается указатель на объект интерпретатора и происходит вызов метода `PREPARE_FOR_CALL`, который устанавливает новое значение указателя на байт-код внутри объекта. Затем новое значение указателя сохраняется в регистр, после чего происходит переход к исполнению байт-кода вызванного метода.

Отметим, что поскольку новый указатель уже выставлен на начало первой инструкции вызванного метода, аргумент макроса `NEXT` должен равняться нулю.

Теперь рассмотрим листинг 6. На нём представлен код обработчика инструкции возврата. Сначала считывается индекс регистра, значение которого необходимо вернуть, после чего получается значение и сохраняется в регистр `ir1`. Затем со стека считываются два значения: регистр, в который необходимо вернуть значение, и флаг. Флаг равен 1, если необходимо совершить возврат из интерпретатора. В этом случае исполнение переходит на метку `ret_end`, где осуществляется вызов метода, подготавливающего интерпретатор к возврату. После этого исполнение переходит на метку `return_from_interpret`, в которой происходит возврат из интерпретатора.

Иначе возврат происходит обратно в интерпретатор. Для этого необходимо восстановить `ip`, волатильные регистры, и положить

```
B_CALL:
  and     x2,  v_op, 0xF   # Return reg
  ubfx    x1,  v_op, 4, 12 # Table index

  # Save volatile registers (IR8 - IR13)
  # It seems faster to just save all of them
  ldp     x0, x3, [registers, #64]
  ldp     x4, x5, [registers, #80]
  ldp     x6, x7, [registers, #96]

  # Store flag that there is a Frame
  stp     x2, xzr, [sp, #-80]!
  stp     ip, xzr, [sp, #16]
  stp     x0, x3, [sp, #32]
  stp     x4, x5, [sp, #48]
  stp     x6, x7, [sp, #64]

  # Put interpreter as arg for call
  mov     x0, interpreter
  bl      PREPARE_FOR_CALL

  # Stream has changed so it is necassary to load new ip
  ldr ip, [interpreter, #120]

  # Current ip is already at the start of first
instruction of callee
  NEXT 0
```

Листинг 5 — код обработчика вызова метода

возвращаемое значение в регистр, полученный из стекового кадра (этот регистр был сохранён во время вызова). После чего необходимо перейти к обработке следующей инструкции. В этом случае аргумент макроса `NEXT` равен 4, поскольку указатель был выставлен на начало инструкции вызова.

```
B_RET:
  // register to ret
  and   dri, v_op, 0xF
  // register to ret
  ldr   x2, [registers, dri, lsl #3]

  // put value in IR1
  str   x2, [registers, #8]

  // Get return register in x1 and flag in x0 (is there frame)
  ldp   x1, x0, [sp]

  // If Flag is set to 1 => return => goto ret_end
  cbnz  x0, ret_end
  // Else restore frame

  ldp     ip, xzr, [sp, #16]
  // Load volatile registers (IR8 - IR13)
  ldp     x0, x3, [sp, #32]
  ldp     x4, x5, [sp, #48]
  ldp     x6, x7, [sp, #64]

  stp     x0, x3, [registers, #64]
  stp     x4, x5, [registers, #80]
  stp     x6, x7, [registers, #96]
  add     sp, sp, #80

  // Load return value to retRegister (retRegister is from call)
  str   x2, [registers, t1, lsl #3]

  // ip was at the start of call instruction
  NEXT 4

ret_end:
// put intetpreter as arg
  mov   x0, interpreter
  bl    PREPARE_FOR_RETURN
  b     return_from_interpret
```

Листинг 6 — код обработчика возврата

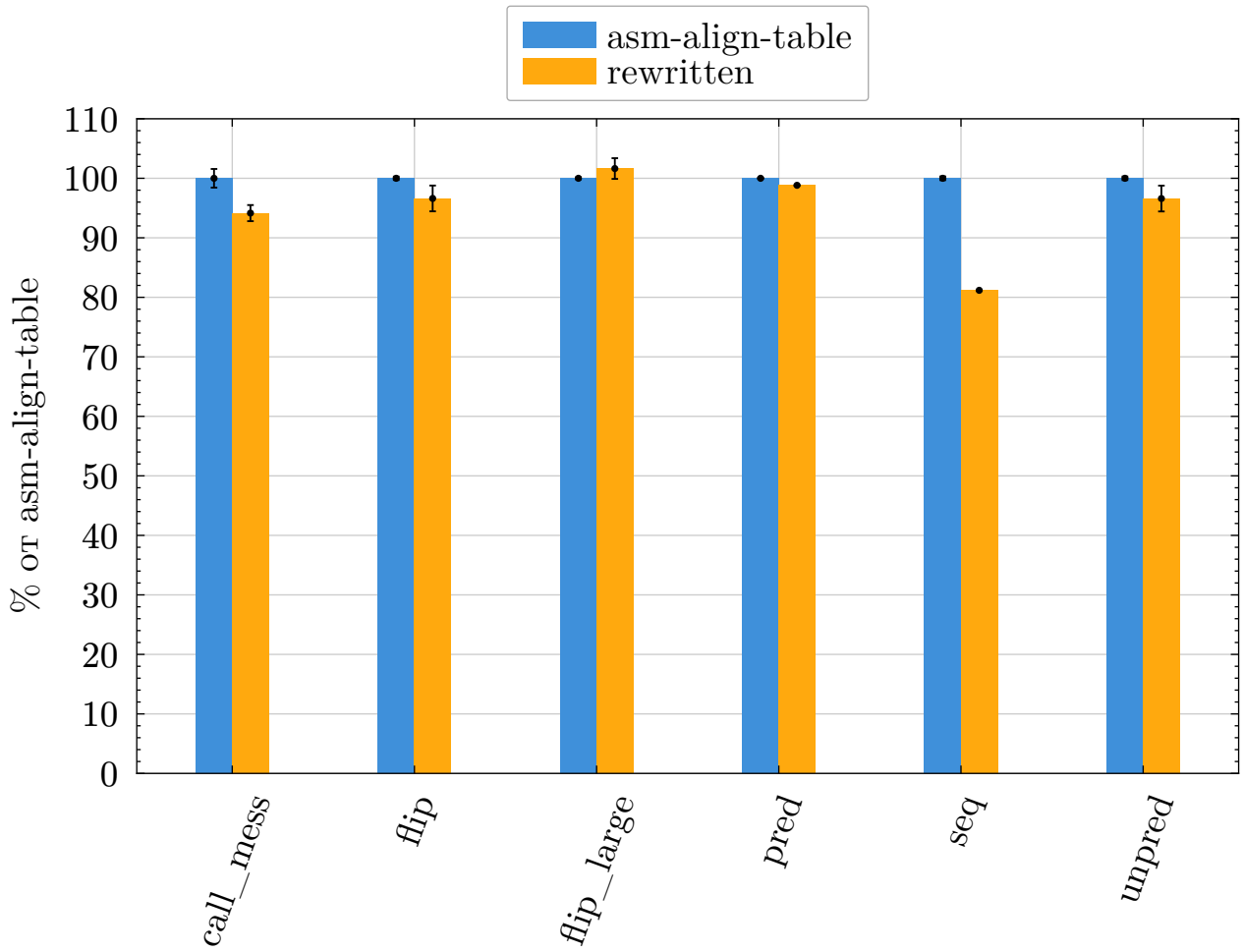


Рисунок 5 — результаты тестирования интерпретатора нового формата

6 Производительность интерпретатора байт-кода нового формата

Методика сравнения и описание тестовых программ представлены в разделе 4. На рисунке 5 представлены результаты тестирования (меньше — лучше). Результаты нормированы в процентах от `asm-align-table`.

В среднем изменение формата байт-кода и использование переписывателя позволило получить улучшение порядка 2–5%. Небольшой прирост производительности объясняется тем, что дополнительное разыменование не вносит основного вклада в накладные расходы при диспетчеризации. Несмотря на уменьшение стоимости

диспетчеризации, существенная часть времени исполнения по-прежнему приходится на косвенные переходы.

Наиболее заметное изменение наблюдается в тесте `seq`: ускорение составляет около 19%. Данный результат может быть обусловлен структурой теста. Последовательное выполнение групп инструкций формирует более предсказуемый поток исполнения, что повышает точность предсказания переходов. В сочетании с уменьшением накладных расходов при диспетчеризации инструкций это может приводить к более выраженному эффекту.

В тесте `flip_large`, наоборот, наблюдается небольшое ухудшение производительности, однако доверительные интервалы перекрываются. Вероятно, в данном случае уменьшение накладных расходов при диспетчеризации не компенсирует влияние особенностей расположения обработчиков в памяти и структуры переходов между инструкциями.

7 Сравнение с индустриальным интерпретатором

В данном разделе представлено сравнение с интерпретатором HotSpot JVM.

7.1 Вариации интерпретаторов

default-hotspot — стандартный интерпретатор HotSpot, также называемый **template interpreter**.

zero-hotspot — интерпретатор HotSpot, разработанный с целью максимальной переносимости. В нём не применяются оптимизации на уровне машинного кода, благодаря чему он является переносимым.

Также представлены разработанные варианты интерпретаторов: **aj-stack**, **asm-align-table**, **rewritten**.

7.2 Вариации тестовых программ

Для сравнения были выполнены следующие тесты:

seq_java — тест, совпадающий с **seq** по исходному коду. Код теста был переписан на язык Java.

seq_jasm — тест, семантически совпадающий с **seq**, в котором инструкции виртуальной машины были выбраны вручную.

seq_like — тест, семантически не совпадающий с **seq**, но сходный с ним по структуре байт-кода. В нём содержится такое же количество блоков инструкций виртуальной машины, как и в **seq**,

а также совпадает количество инструкций в каждом блоке. Как и в прошлой вариации выбор инструкций виртуальной машины был произведен вручную.

7.3 Сравнение

На рисунке 6 представлены результаты тестирования (меньше — лучше); методология описана в разделе 4. В качестве метрики было выбрано время выполнения тестовой программы (в секундах). Тестирование интерпретаторов **HotSpot JVM** проводится с опцией `-Xint`, которая выключает JIT компиляцию.

Сравнение **seq** и **seq_java** демонстрирует различия моделей исполнения и принципах на которых основаны интерпретаторы.

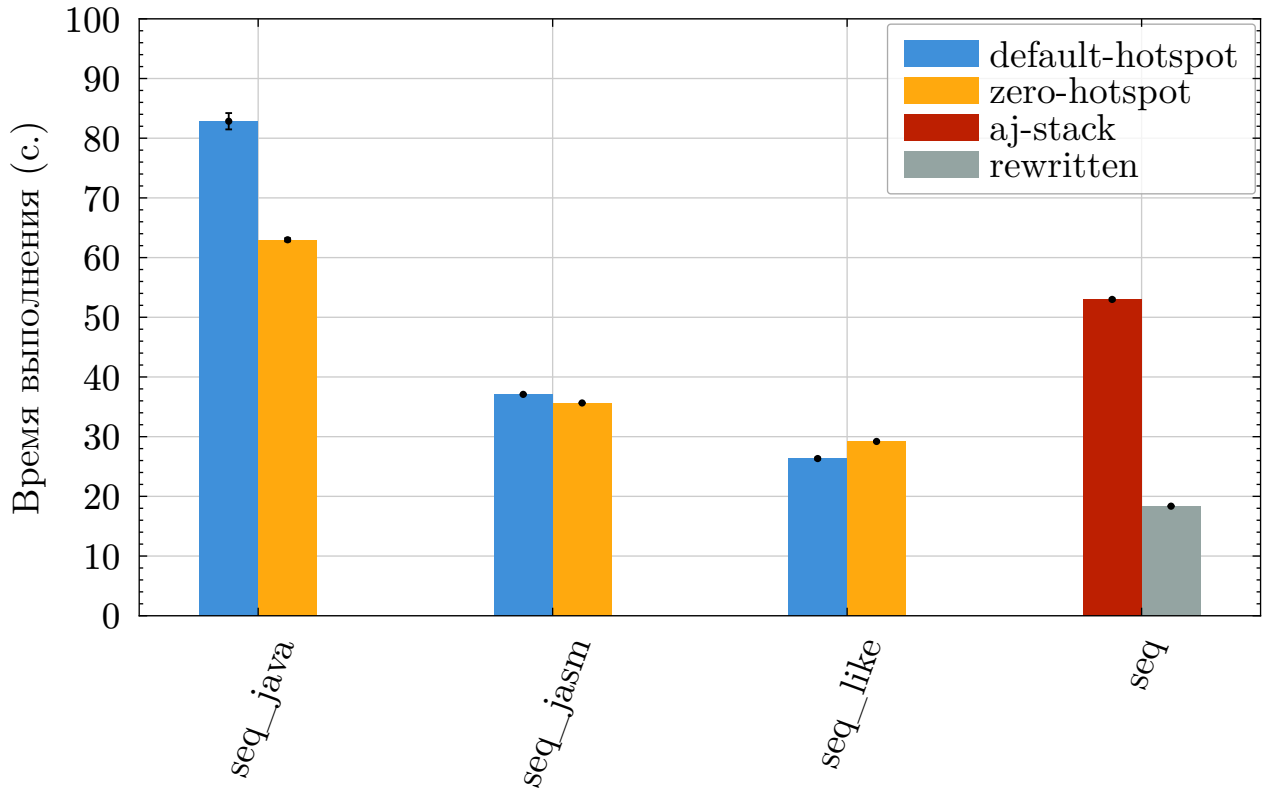


Рисунок 6 — сравнение результатов тестирования с интерпретатором OpenJDK

Виртуальная машина HotSpot является стековой, и вследствие такой архитектуры для тестов данного типа требуется значительно больше инструкций. В силу принципов, на которых основывается HotSpot, от компилятора байт-кода не ожидается значительных оптимизаций. При анализе байт-кода, полученного после компиляции исходного кода, было установлено, что компилятор не использовал основное преимущество стековой машины.

Как было описано в 4.2, **seq** состоит из цикла, в котором представлены несколько групп различных инструкций. В исходном коде теста используются 4 переменные, с которыми поочерёдно выполняются 6 типов операций. Сначала выполняются операции с первой переменной: она умножается на некоторое константное значение, затем выполняется побитовая конъюнкция с некоторой константой, после чего — дизъюнкция и так далее. Затем аналогичные операции выполняются со следующей переменной.

Компилятор виртуальной машины Huawei переупорядочил операции таким образом, что одинаковые инструкции располагаются последовательно. Сначала выполняются все операции умножения, затем — все операции конъюнкции, после чего — дизъюнкции и так далее. Вероятно это помогло предсказателю переходов. Такая перестановка стала возможной благодаря тому, что модель исполнения предполагает регистровую архитектуру, в которой это не вызывает затруднений.

В случае компилятора виртуальной машины HotSpot перестановка операций не была выполнена, при этом компилятор также не

воспользовался преимуществами стековой модели. Перед каждой операцией значение локальной переменной загружается на стек, а после выполнения операции сохраняется обратно в память. При этом практически после каждого сохранения следует повторная загрузка той же переменной на стек.

Именно этим объясняется наблюдаемая существенная разница в результатах. default-hotspot уступает aj-stack примерно на 56%, а по сравнению с asm-align-table разница достигает 3 раз.

Тест **seq_jasm** семантически совпадает с **seq**, однако байт-код был сформирован вручную. Были устранены избыточные инструкции загрузки и сохранения локальных переменных, при этом порядок операций был сохранён. Сначала выполняются все операции над одной переменной, затем — над следующей и так далее. Таким образом, стек виртуальной машины используется для хранения промежуточных результатов, а операции сохранения и загрузки выполняются только при смене обрабатываемой переменной. При этом общее количество инструкций оказалось больше, чем в байт-коде для виртуальной машины Huawei. По структуре данный тест ближе к исходному коду, чем к байт-коду для **seq**.

В данном случае **aj-stack** начинает уступать более оптимизированным интерпретаторам. Сравнение default-hotspot и asm-align-table показывает прирост производительности около 40%.

Тест **seq_like** сходен с **seq** по структуре байт-кода, однако отличается по семантике вычислений. В данном тесте последовательно

выполняются операции одного типа, затем другого и так далее. Именно этот вариант позволяет оценить эффективность интерпретаторов. Поскольку в тестах используются простые операции (арифметические операции, загрузка локальной переменной на стек), основное время интерпретации затрачивается на диспетчеризацию. Версии asm-align-table и rewritten выполняются быстрее default-hotspot на 15% и 31% соответственно.

7.4 Вывод

В условиях отключённого JIT-компилятора преимущество получает специализированная реализация, минимизирующая накладные расходы диспетчеризации. Интерпретатор HotSpot ориентирован на поддержку полной семантики языка Java и поэтому несёт дополнительные накладные расходы, как минимум связанные с поддержкой сборки мусора.

Также, вероятно, заметное влияние оказывает реализация инструкций в интерпретаторе HotSpot. Хотя простые арифметические операции занимают немного меньше инструкций, чем в нашей реализации (например, для инструкции `add`: 7 против 9), разница обуславливается возможностью оптимизации вершины стека. Интерпретатор хранит вершину стека в регистре, за счёт чего удаётся исключить инструкции получения индекса регистра и обращения к нему.

Инструкции управления потоком, напротив, занимают значительно больше инструкций, а также содержат дополнительную внутреннюю логику, отсутствующую в нашем интерпретаторе. Например, инструкция

`ifgt`, отвечающая за ветвление, содержит около 22 инструкций, и её выполнение зависит от условия. Если условие выполнено, будет исполнено 12 инструкций, не считая эффектов конвейера и предсказателя переходов, иначе — 7. Инструкция `ldc2_w`, выполняющая загрузку константы на стек, состоит из 84 инструкций. В лучшем случае будут исполнены 20 инструкций, а также совершены два перехода внутри обработчика. В то время как в нашем наборе инструкций константы кодируются непосредственно внутри инструкции.

8 Ограничения предложенного подхода

Предложенный подход ориентирован на повышение производительности интерпретаторов за счёт предварительного переписывания байт-кода и использования метода шитого кода. Однако данный подход имеет ряд ограничений.

Эффективность предложенного подхода существенно зависит от особенностей аппаратной архитектуры, на которой предполагается исполнение. Производительность интерпретатора определяется поведением предсказателя переходов процессора, организацией кэш-памяти и механизмами выборки инструкций. В связи с этим результаты, полученные на системе под управлением процессора архитектуры ARMv8, могут не воспроизводиться на других архитектурах и моделях процессоров.

Переписывание требует дополнительного времени, однако данный этап уже присутствовал в исходной реализации интерпретатора. Предлагаемые изменения изменяют характер выполняемого переписывания, но, вероятно, не приводят к существенным дополнительным накладным расходам. В рамках данной работы влияние переписывания на время запуска программ отдельно не исследовалось, поскольку основной целью было изучение влияния нового формата байт-кода на эффективность интерпретации. Кроме того, использование переписывания усложняет реализацию и сопровождение интерпретатора.

Новый формат байт-кода ориентирован в первую очередь на повышение эффективности интерпретации. Во время переписывания

размер байт-кода возрастает, что приводит к увеличению потребления оперативной памяти.

Также важно отметить, что для тестирования использовался специализированный тестовый набор. Тесты, представленные в работе, преимущественно ориентированы на исследование накладных расходов диспетчеризации и не полностью отражают поведение интерпретатора на реальных прикладных нагрузках.

ЗАКЛЮЧЕНИЕ

В рамках данной выпускной квалификационной работы были исследованы подходы к повышению производительности интерпретации байт-кода виртуальных машин. Актуальность данной задачи обусловлена тем, что интерпретаторы продолжают играть важную роль при реализации языков программирования. В частности, это связано с их переносимостью, возможностью быстрого запуска приложений, удобством отладки, а также возможностью совместной работы с JIT-компилятором для сбора профиля исполнения и более эффективной компиляции с учётом собранного профиля.

В ходе работы были исследованы факторы, ограничивающие эффективность интерпретаторов. Показано, что существенное влияние на скорость исполнения оказывают накладные расходы, связанные с диспетчеризацией инструкций. Основной вклад в эти накладные расходы вносят косвенные переходы, необходимые для исполнения байт-кода, ошибки предсказания переходов и стоимость обращений к памяти. Проведён анализ существующих методов повышения производительности интерпретаторов, включая табличную диспетчеризацию инструкций и метод шитого кода.

Для проведения экспериментальной части работы была реализована базовая версия интерпретатора байт-кода на языке системного программирования AJ. Полученная реализация позволила выявить основные источники потери производительности, возникающие вследствие использования языка высокого уровня.

Затем была реализована базовая версия интерпретатора на языке ассемблера архитектуры ARMv8. В данной реализации были применены методы оптимизации, направленные на снижение накладных расходов интерпретации: табличная диспетчеризация инструкций, устранение обратных переходов в цикле интерпретации, размещение состояния интерпретатора в регистрах процессора, а также использование машинного стека для хранения фреймов вызовов. Дополнительно были исследованы оптимизации, связанные с организацией памяти и уменьшением количества инструкций в критических участках кода.

Экспериментальное сравнение показало, что реализация интерпретатора на языке ассемблера обеспечивает существенный прирост производительности. В зависимости от характера тестовой нагрузки ускорение составило от 1,8 до 2,5 раза. Также было выявлено, что отдельные низкоуровневые оптимизации оказывают негативное влияние на производительность интерпретатора. В частности, выравнивание обработчиков инструкций привело к ухудшению производительности.

Основным результатом работы стала разработка нового формата байт-кода и механизмов его переписывания перед началом исполнения. Предложенный подход позволяет использовать механизмы предварительной обработки байт-кода не только для разрешения позднего связывания, но и для преобразования инструкций в форму, более удобную для эффективной интерпретации. Благодаря реализации механизмов переписывания удалось применить метод прямого шитого

кода, что позволило исключить дополнительное разыменование при диспетчеризации инструкций.

В результате проведённого исследования было показано, что применение механизмов переписывания байт-кода позволяет повысить эффективность интерпретации. Полученные результаты подтверждают практическую применимость предложенного подхода при разработке высокопроизводительных интерпретаторов для языков высокого уровня.

Дальнейшее развитие работы может быть связано с полной поддержкой инструкций байт-кода. Также возможно исследование более сложных методов специализации байт-кода, внедрения суперинструкций, а также автоматической генерации обработчиков инструкций. Кроме того, представляет интерес интеграция с системами JIT-компиляции. Дополнительно возможно исследование рассмотренного подхода на других архитектурах процессоров.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Bernstein R. Debuggers in dynamic languages. June, 2010.
2. Singh A. и др. Debugging Dynamic Language Features in a Multi-tier Virtual Machine // Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. New York, NY, USA: Association for Computing Machinery, 2023. Сс. 18–28.
3. Gosling J. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95) // SIGPLAN Not. New York, NY, USA: Association for Computing Machinery, 1995. Т. 30, № 3. Сс. 111–118.
4. Romer T.H. и др. The structure and performance of interpreters // Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 1996. Сс. 150–159.
5. Bell J.R. Threaded code // Commun. ACM. New York, NY, USA: Association for Computing Machinery, 1973. Т. 16, № 6. Сс. 370–372.
6. Grimmer M. и др. Memory-safe Execution of C on a Java VM // Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security. New York, NY, USA: Association for Computing Machinery, 2015. Сс. 16–27.
7. Hartel P.H., Moreau L. Formalizing the safety of Java, the Java virtual machine, and Java card // ACM Comput. Surv. New York, NY, USA: Association for Computing Machinery, 2001. Т. 33, № 4. Сс. 517–558.

8. Bakhvalov D. Performance Analysis and Tuning on Modern CPUs: Squeeze the Last Bit of Performance from Your Application. Independently published, 2020.
9. Kulkarni P.A. JIT compilation policy for modern machines // SIGPLAN Not. New York, NY, USA: Association for Computing Machinery, 2011. Т. 46, № 10. Сс. 773–788.
10. Shi Y. и др. Virtual machine showdown: Stack versus registers // ACM Trans. Archit. Code Optim. New York, NY, USA: Association for Computing Machinery, 2008. Т. 4, № 4.