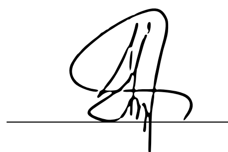
МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# ИССЛЕДОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ БИБЛИОТЕК JIT КОМПИЛЯЦИИ НА ЯЗЫКЕ C

АВТОРЕФЕРАТ МАГИСТЕРСКОЙ РАБОТЫ

студента 2 курса 273 группы
направления 02.04.03 — Математическое обеспечение и администрирование информационных систем
факультета КНиИТ
Слуцкого Алексея Дмитриевича

Научный руководитель
к. ф.-м. н., доцент  Г. Г. Наркайтис

Заведующий кафедрой
к. ф.-м. н., доцент ______________ А. С. Иванов

Саратов 2020

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ . . . 3
1 Постановка задачи . . . 4
2 Основы JIT-компиляции . . . 6
3 Анализ результатов . . . 13
ЗАКЛЮЧЕНИЕ . . . 15
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . 16

# ВВЕДЕНИЕ

Оптимизация компьютерных программ необходима для того, чтобы приложение работало эффективно и могло сохранять разумную скорость работы при увеличении размера входных данных. В англоязычной литературе такая способность программ называется «scale».

Одним из основных способов оптимизации программы является нахождение наиболее часто вызываемого программного кода и замена этого кода более оптимальным решением, начиная от улучшения алгоритма и заканчивая заменой языка программирования. Эффективным решением такой оптимизации является генерация низкоуровневых команд в рантайме. Такая задача может быть решена посредством JIT (just in time компиляция).

Например, в Java генерацией JIT занимается виртуальная машина Hotspot. Она помимо непосредственной интерпретации байт-кода может выполнять компиляцию байт-кода (отдельных методов целиком) в машинные инструкции для ускорения процесса выполнения. Эта компиляция происходит во время исполнения, после того как метод уже был выполнен несколько раз. Ожидание фактического использования метода дает возможность Java HotSpot VM сделать более точное решение о том, как оптимизировать код путем компиляции. И наибольшей эффективности она достигает тогда, когда она может собрать достаточно статистики, чтобы принять разумное решение о том, что скомпилировать. [1]

В данной работе будет рассмотрена оптимизация в виде Just In Time компиляции.

## 1 Постановка задачи

Несмотря на более чем полувековую историю вычислительной техники, формально годом рождения теории компиляторов можно считать 1957, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом. Лишь появление теории формальных языков и строгих математических моделей позволило перейти от «творчества» к «науке». Именно благодаря этому, стало возможным появление сотен новых языков программирования.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их компиляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью существующих компиляторов. Поэтому основы теории языков и формальных грамматик, а также практические методы разработки компиляторов актуальны по сей день и открывают широкий горизонт для исследования.

Говоря об эффективности, скорость работы программы всегда будет оставаться важной характеристикой. Например, не беспричинно все большую популярность набирает такое понятие как BigData. С развитием технологий на аппаратном уровне, естественно возникает желание обработки больших объемов данных. Однако за счет одних только оптимизаций аппаратного уровня добиться разумного времени работы неэффективной программы не удастся. Улучшение алгоритмической составляющей играет не менее важную роль.

В качестве одной из оптимизаций при построении компилятора можно рассматривать Just in time компиляцию, суть которой состоит в генерации машинных команд во время исполнения программы для тех участков кода, которые выполняются наиболее часто. Существуют разные библиотеки, предоставляющие функционал для реализации такого рода оптимизаций. Но какова разница в производительности программ, использующих эти библиотеки? Какая из библиотек позволяет добиться минимального времени исполнения парсинга выражения?

Целью данной дипломной работы является исследование прироста производительности при использовании JIT компиляции на примере арифметических выражений. В рамках данной работы будут реализованы разные версии LR-анализаторов с использованием разных библиотек, позволяющих генерировать машинные команды в рантайме. Созданные программы будут исследованы на предмет скорости работы с целью выявления наиболее оптимальной библиотеки для решения данной задачи. В качестве исследуемых реализаций рассматриваются абстрактное синтаксическое дерево, Libgccjit, Libjit, LLVM.

## 2 Основы JIT-компиляции

Компиляция программного кода в машинный код на этапе работы программы называется JIT компиляцией. Она может быть применена как ко всей программе, так и к её отдельным частям. С AOT-компиляцией такое сделать не представляется возможным для случаев, когда данные предоставляются во время исполнения программы, а не в момент компиляции.

Большинство реализаций JIT имеют последовательную структуру: сначала приложение компилируется в байт-код виртуальной машины среды исполнения (AOT-компиляция), а потом JIT компилирует байт-код непосредственно в машинный код (рис. 1). В итоге при запуске приложения тратится лишнее время, что впоследствии компенсируется более быстрой его работой. [1]

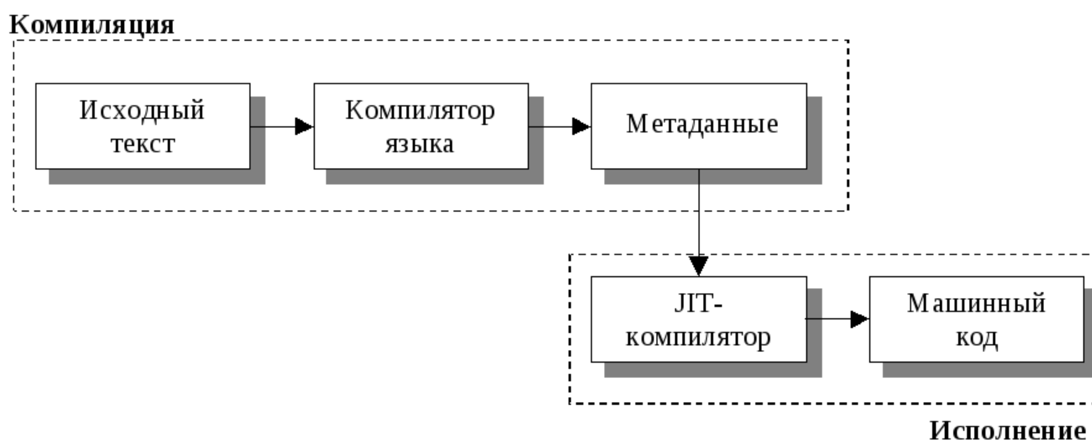



Рисунок 1 – Схема JIT компиляции

Самую первую реализацию JIT можно отнести к LISP, написанную McCarthy в 1960 году. В его книге «Recursive functions of symbolic expressions and their computation by machine» он упоминает функции, компилируемые во время выполнения, тем самым избавив от надобности вывода работы компилятора на перфокарты.

Другой ранний пример упоминания JIT можно отнести к Кену Томпсону, который в 1968 году впервые применил регулярные выражения для поиска подстрок в текстовом редакторе QED. Для ускорения алгоритма Томпсон реализовал компиляцию регулярных выражений в машинный код IBM 7094.

Важный метод получения скомпилированного кода был предложен Митчелом в 1970 году, когда он реализовал экспериментальный язык $LC^2$. [2]

Язык Smalltalk, появившийся в 1983 году был пионером в области JIT-

технологий. Трансляция в машинный код выполнялась по требованию и кэшировалась для дальнейшего использования. Когда память кончалась, система могла удалить некоторую часть кэшированного кода из оперативной памяти и восстановить его, когда он снова потребуется. Язык программирования Self некоторое время был самой быстрой реализацией Smalltalk и работал всего лишь в два раза медленней C, будучи полностью объектно-ориентированным.

В последствии Self был заброшен Sun, но исследования продолжились в рамках языка Java. Термин «Just-in-time компиляция» был заимствован из производственного термина «Точно в срок» и популяризован Джеймсом Гослингом, использовавшим этот термин в 1993. В данный момент JIT используется почти во всех реализациях Java Virtual Machine.

Также большой интерес представляет диссертация, защищённая в 1994 году в Университете ETH (Швейцария, Цюрих) Михаэлем Францем «Динамическая кодогенерация» — ключ к переносимому программному обеспечению» и реализованная им система Juice динамической кодогенерации из переносимого семантического дерева для языка Оберон. Система Juice предлагалась как плагин для интернет-браузеров.

В языках, таких как Java, PHP, C#, Lua, Perl, GNU CLISP, исходный код транслируется в одно из промежуточных представлений, называемое байт-кодом. Байт-код не является машинным кодом какого-либо конкретного процессора и может переноситься на различные компьютерные архитектуры и исполняться точно так же. Байт-код интерпретируется (исполняется) виртуальной машиной. JIT читает байт-код и компилирует его в машинный код. Это может быть файл, функция или любой фрагмент кода. Однажды скомпилированный код может кэшироваться и в дальнейшем повторно использоваться без перекомпиляции. [3]

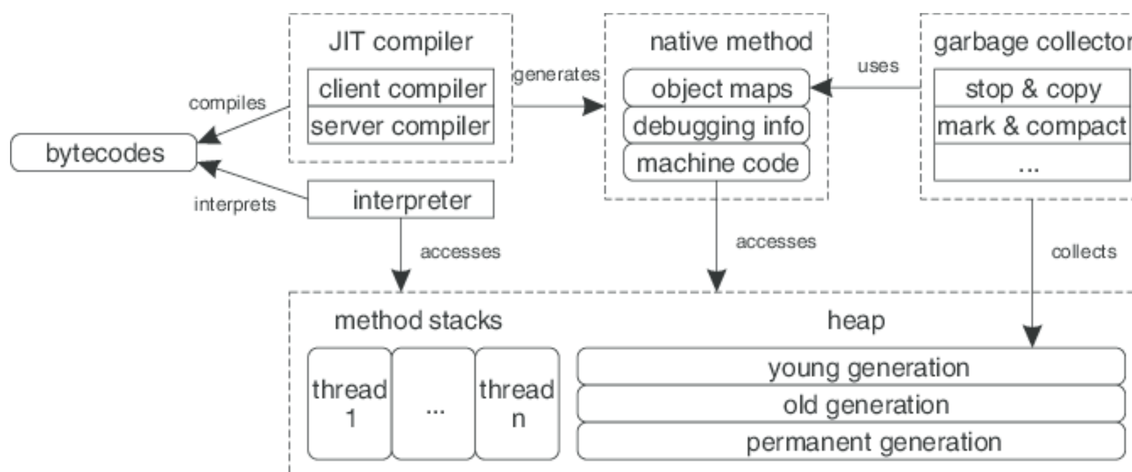



Рисунок 2 – Java Hotspot

Например, в Java генерацией JIT занимается виртуальная машина Hotspot (рис. 2). Она составляет основу как для Java (JVM), так и для OpenJDK (проект с открытым исходным кодом). Как и все виртуальные машины Java HotSpot VM обеспечивает необходимую среду для выполнения байт-кода. На практике она отвечает за три основные функции:

— интерпретация байт-кода
— поиск, загрузка и проверка типов (так называемая загрузка классов)
— управление памятью

Java HotSpot VM помимо непосредственной интерпретации байт-кода может выполнять компиляцию байт-кода (отдельных методов целиком) в машинные инструкции для ускорения процесса выполнения.

Если виртуальной машине передать особый параметр «$-XX : +PrintCompilation$», то можно увидеть как были скомпилированы методы. Эта компиляция происходит во время исполнения, после того как метод уже был выполнен несколько раз. Ожидание фактического использования метода дает возможность Java HotSpot VM сделать боле точное решение о том, как оптимизировать код путем компиляции.

Наибольшей эффективности Java HotSpot VM достигает тогда, когда она может собрать достаточно статистики, чтобы принять разумное решение о том, что скомпилировать. Если порог компиляции слишком низок, Java HotSpot VM может потратить огромное количество времени компилируя методы, которые выполняются не так часто. Некоторые оптимизации выполняются только тогда, когда было собрано достаточное количество статистики. Так что код может быть не таким оптимальным каким бы он мог быть. [4]

А вот в PyPy, например, существует специальный JIT-генератор. Python, как и большинство динамических языков программирования, традиционно отдает предпочтение гибкости в обмен на снижение производительности. Архитектура PyPy, обладая особенной гибкостью и широким спектром абстракций, затрудняет реализацию возможности очень быстрой интерпретации. Мощные абстракции пространств объектов и мультиметодов в стандартном пространстве объектов не могут быть реализованы без последствий. В результате производительность не модифицированного интерпретатора PyPy будет в четыре раза ниже производительности интерпретатора CPython. Для того, чтобы не создавать репутацию медленного языка не только для данной реализации, но и для языка Python в общем, в рамках PyPy был реализован динамический компилятор (рис. 3). С помощью JIT-компилятора часто используемые пути исполнения кода преобразуются в ассемблерное представление в процессе исполнения программы.

JIT-компилятор из состава PyPy использует преимущества уникальной архитектуры процесса преобразования кода в PyPy. На самом деле PyPy не использует Python-специфичный JIT-компилятор; вместо него используется JIT-генератор. Генерация JIT-кода реализована просто в виде еще одной дополнительной фазы преобразования кода. Интерпретатор, желающий провести генерацию JIT-кода, должен осуществить два вызова специальных функций, называемых указаниями jit (jit hints).

JIT-генератор из состава PyPy является трассирующим JIT-генератором (tracing JIT). Это значит, что он определяет часто используемые циклы с целью их оптимизации путем компиляции в ассемблерный код. В момент, когда JIT-генератор принимает решение приступить к компиляции кода цикла, он записывает операции в рамках одной итерации цикла и этот процесс называется трассировкой (tracing). Эти операции впоследствии компилируются в машинный код. [5]

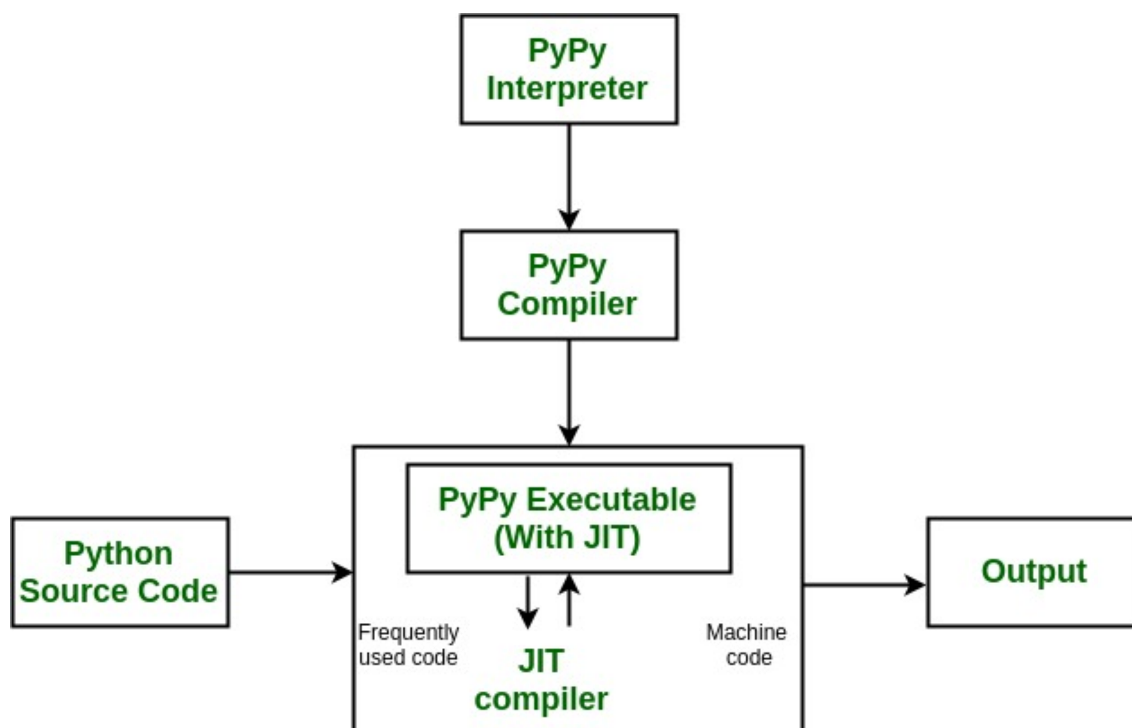



Рисунок 3 – PyPy

Динамически компилируемая среда — это среда, в которой компилятор может вызываться приложением во время выполнения. Например, большинство реализаций Common Lisp содержат функцию compile, которая может создать функцию во время выполнения; в Python это функция eval. Это удобно для программиста, так как он может контролировать, какие части кода действительно подлежат компиляции. Также с помощью этого приёма можно компилировать динамически сгенерированный код, что в некоторых случаях приводит даже к лучшей производительности, чем реализация в статически скомпилированном коде. Однако стоит помнить, что подобные функции могут быть опасны, особенно когда данные передаются из недоверенных источников.

Вопрос безопасности и возможных уязвимостей играет далеко не последнюю роль. JIT составляет исполняемый код из данных, а JIT компиляция включает в себя компиляцию исходного кода или байт-кода в машинный код и его выполнение. Как правило, результат записывается в память и исполняется сразу же, без промежуточного сохранения на диск или его вызов как отдельной программы. В современных архитектурах для повышения безопасности произвольные участки памяти не могут быть исполнены как машинный код (NX bit). Для корректного запуска регионы памяти должны быть предварительно помечены как исполняемые, при этом для большей безопасности флаг

исполнения может ставиться только после снятия флага разрешения записи.

Другой острый вопрос — это задержка при запуске. Типичная причина задержки при запуске JIT-компилятора — расходы на загрузку среды и компиляцию приложения в машинный код. В общем случае, чем лучше и чем больше оптимизаций выполняет JIT, тем дольше получается задержка. Поэтому разработчикам JIT приходится искать компромисс между качеством генерируемого кода и временем запуска. Однако, часто оказывается так, что узким местом в процессе компиляции оказывается не сам процесс компиляции, а задержки системы ввода-вывода (так, например, rt.jar в Java Virtual Machine (JVM) имеет размер 40 МБ, и поиск метаданных в нём занимает достаточно большое количество времени).

Ещё одно средство оптимизации — компилировать только те участки приложения, которые используются чаще всего. Этот подход реализован, например, в приведенных выше PyPy и HotSpot Java Virtual Machine.

В качестве эвристики может использоваться счётчик запусков участков приложения, размер байт-кода или детектор циклов.

Порой достаточно сложно найти правильный компромисс. Так, например, JVM имеет два режима работы — клиент и сервер. В режиме клиента количество компиляций и оптимизаций минимально для более быстрого запуска, в то время как в режиме сервера достигается максимальная производительность, но из-за этого увеличивается время запуска.

Также существует техника, называемая pre-JIT, которая компилирует код до запуска. Преимуществом данной техники является уменьшенное время запуска, в то же время недостатком является плохое качество скомпилированного кода по сравнению с runtime JIT.

Основная цель использования JIT — достичь и превзойти производительность статической компиляции, сохраняя при этом преимущества динамической компиляции:

— Большинство тяжеловесных операций, таких как парсинг исходного кода и выполнение базовых оптимизаций, происходит во время компиляции (до развёртывания), в то время как компиляция в машинный код из байт-кода происходит быстрее, чем из исходного кода.
— Байт-код более переносим (в отличие от машинного кода).
— Компиляторы из байт-кода в машинный код легче в реализации, так как

большинство работы по оптимизации уже было проделано компилятором.

— Среда может контролировать выполнение байт-кода после компиляции, поэтому приложение может быть запущено в песочнице (стоит отметить, что для нативных программ такая возможность тоже существует, но реализация данной технологии сложнее).

JIT, как правило, эффективней, чем интерпретация кода. К тому же в некоторых случаях JIT может показывать большую производительность по сравнению со статической компиляцией за счёт оптимизаций, возможных только во время исполнения:

— Компиляция может осуществляться непосредственно для целевого процессора и операционной системы, на которой запущено приложение. Например, JIT может использовать векторные SSE2 расширения процессора, если он обнаружит их поддержку. Однако, до сих пор нет основных реализаций JIT, где этот подход бы использовался, ведь чтобы обеспечить подобный уровень оптимизации, сравнимый со статическими компиляторами, потребовалось бы либо поддерживать бинарный файл под каждую платформу, либо включать в одну библиотеку оптимизаторы под каждую платформу.
— Среда может собирать статистику о работающей программе и производить оптимизации с учётом этой информации. Некоторые статические компиляторы также могут принимать на вход информацию о предыдущих запусках приложения.
— Среда может делать глобальные оптимизации кода (например, встраивание библиотечных функций в код) без потери преимуществ динамической компиляции и без накладных расходов, присущих статическим компиляторам и линкерам.
— Более простое перестраивание кода для лучшего использования кэша. [6]

## 3 Анализ результатов

В качестве тестовых данных выбраны выражения, содержащие $x$. Они имеют разную длину. Интерес представляет зависимость времени работы от длины выражения. Поэтому в данном эксперименте, программы запускаются для выражений разных длин с целью получения графического представления этой зависимости.

Помимо программ, использующих JIT компиляцию, также будет замерено время работы алгоритма на основе AST дерева. В нем структура выражения рассматривается только один раз, по этой структуре строится абстрактное дерево, представляющее собой выражение, разобранное на основе правил грамматики. Некоторые листья содержат числовые значения, некоторые — $x$. Таким образом, для каждого нового параметра, его значение подставляется в соответствующие листья, после чего производится подъем по дереву. В корне лежит ответ.

Одним из примеров было вычисление абсолютно произвольных выражений с бинарными операциями из не более 20 операндов, содержащих в себе параметр, значение которого варьировалось от 0 до $10^7$. В коде данные условия регулируются константами `MAX_STATEMENT_SIZE` и `X_RANGE` соответственно.

Результаты можно видеть на следующем графике.

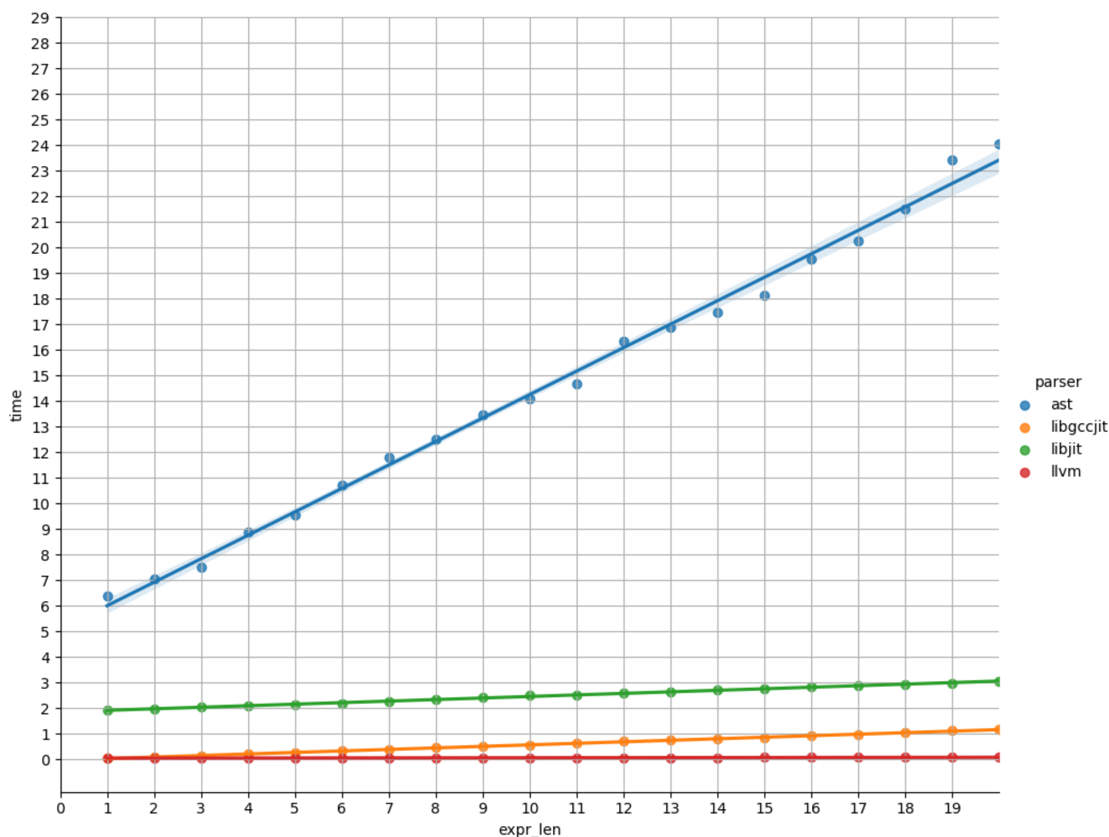



Рисунок 4 – Сравнение с AST деревом

Из проведенных выше замеров становится понятно, что использование JIT дает существенный прирост производительности даже по сравнению с оптимизациями на уровне алгоритмов. Так AST дерево хоть и оказалось гораздо быстрее простого алгоритма повторного разбора выражения, значительно проиграло в скорости реализациям на основе JIT.

Это заключение оправдывает целесообразность исследования оптимизаций подобного вида. Улучшение алгоритмов открывает большие границы для усовершенствования программ, однако оптимизация на уровне компилятора языка выводит производительность на совершенно другой уровень.

Группа из трех JIT реализаций совместно находится в нижней части графика. Между тремя библиотеками виден лидер — это LLVM. Чуть медленнее оказался Libgccjit. Следом идет Libjit.

Соответственно в среднем случае LLVM работает быстрее. Из чего можно сделать вывод, что LLVM предоставляет наиболее быстрый интерфейс для решения подобных задач.

# ЗАКЛЮЧЕНИЕ

В данной работе все поставленные задачи были успешно выполнены, был проведен сравнительный анализ реализаций разного рода реализаций LR-парсера для многократного пересчета выражения с параметром.

В результате вычислительного эксперимента было установлено, что для решения поставленной задачи наиболее оптимальным решением является использование Just in time компиляции, которая обеспечивает наибольшее ускорение. Среди библиотек, предоставляющих интерфейс генерации JIT команд, наиболее эффективной оказалась LLVM.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1 *Muldner, T.* C for Java Programmers / T. Muldner. — UK.: Pearson, 2000.

2 *Guerron, P.* Scientific computing languages / P. Guerron // *University of Pennsylvania*. — March 2019. — Vol. 61.

3 *Chris Newland James Gough, B. J. E.* Optimizing Java / B. J. E. Chris Newland, James Gough. — Sebastopol, CA: O'Reilly, 2018.

4 *Эванс, Б.* Java. Новое поколение разработки / Б. Эванс. — СПб.: Питер, 2014.

5 The JIT compiler [Электронный ресурс]. — URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html (Дата обращения 12.03.2019). Загл. с экр. Яз. англ.

6 Just In Time Compiler [Электронный ресурс]. — URL: https://www.geeksforgeeks.org/just-in-time-compiler/ (Дата обращения 10.12.2019). Загл. с экр. Яз. англ.