Обзор технологий JIT-компиляции

С.С. Постнов

***Аннотация*—Настоящий обзор посвящён принципам JIT-компиляции («just-in-time» или компиляции «на лету») программ. Кратко рассматривается история развития технологий динамической компиляции программ в разных языках программирования. Подробно рассматривается реализация технологии JIT-компиляции программ в виртуальных машинах Java. Приводится классификация JIT-компиляторов. Описывается базовая технология компиляции «на лету», её особенности и примеры её реализации в компиляторах для языка Java. Обсуждаются потенциально достижимая сложность и степень оптимизации порядка компиляции. Проводится сравнение JIT-компиляции и бинарной динамической компиляции. Уделяется внимание рискам безопасности, присущим JIT-компиляторам. В частности, анализируются уязвимости JIT-компиляторов в отношении атак внедрения и переиспользования кода. Описываются меры противодействия атакам данного типа. Даётся обзор современных принципов разработки JIT-компиляторов, в том числе с использованием методов машинного обучения.**

***Ключевые слова*—JIT компилятор, динамическая компиляция, виртуальная машина Java.**

I. Введение

JIT-compiler или JIT-компилятор представляет собой динамический компилятор, реализующий технологию «Just-in-Time»-компиляции или компиляции программ «на лету». Если говорить более строго, JIT-компилятор транслирует промежуточное представление программы в объектный код во время исполнения программы [1, с. 353].

Принято считать [2], что впервые технология JIT-компиляции была реализована в динамическом компиляторе для языка LISP в 1960 году [3]. В этой работе обсуждается компиляция функций в машинный язык (набор инструкций), настолько быстрая, что она проходит в процессе выполнения и не возникает необходимости сохранения результата работы компилятора.

Имеются также публикации, в которых упоминается о том, что в 1966 г. в исполнительной системе для IBM 7090 транслятор и загрузчик могли функционировать в процессе исполнения [2, 4]. В 1968 г. Для ЭВМ IBM 7094 была реализована компиляция регулярных выражений в машинный код, который тут же исполнялся [5]. Также в 1968 г. был создан экспериментальный язык диалоговых вычислений LC^2 для интерактивного программирования в рамках учебного процесса [6].

В 1970 г. технология JIT-компиляции была реализована в языке APL, причём в двух разновидностях [2, 7]. Первый компилятор выполнял трансляцию программы на языке APL в постфиксный код для специальной машины, поддерживающей буфер отложенных инструкций. Эта машина выполняла роль второго динамического компилятора, который выполнял оптимизацию кода, при необходимости вызывая машину-исполнителя и передавая ей буферизованные инструкции. Позднее эти технологии использовались в программном обеспечении HP APL\3000 [8, 9].

Во второй половине 70-х гг. XX века были созданы JIT-компиляторы для языка FORTRAN [10, 11]. В работе [10] был предложен подход к оптимизации “горячих участков” кода на стадии выполнения, основанный на анализе частот выполнения разных блоков кода. В работе [11] был предложен компилятор, переводящий выполняемые функции в «псевдоинструкции», которые в дальнейшем будут интерпретированы.

Первым объектно-ориентированным языком, в котором были реализованы принципы динамической компиляции, считается язык Smalltalk, появившийся в 1983 году. В этом языке осуществлялась трансляция кода виртуальной машины в машинный (нативный) код, в ходе которой происходила “ленивая” компиляция процедур в момент входа в них исполнителя. При этом выполнялось кэширование машинного кода для последующего использования [12]. Появившийся впоследствии язык программирования Self имел динамическую типизацию и содержал «наиболее агрессивную и амбициозную» (т.е., сильно изменяющую заметную часть кода) [2, p. 101] по тем временам реализацию JIT-компилятора. Было разработано три разных поколения компилятора, который всегда вызывался динамически после вызова метода [13].

Принцип JIT-компиляции был реализован и в языке Оберон, когда исследователи стремились преодолеть проблемы переносимости программного обеспечения. Если программы выполнялись в гетерогенном окружении, то могли возникать коллизии и ошибки из-за того, что компьютеры (процессоры) разной архитектуры требовали разные бинарные исполняемые файлы. Для предотвращения этих ошибок и коллизий был предложен принцип динамической кодогенерации на основе гибких (облегчённых) бинарных файлов («slim binaries») [14, 15]. Такие файлы содержали высокоуровневое машинно-независимое представление программного модуля, фактически являющееся абстрактным синтаксическим деревом (по современной терминологии). При загрузке модуля для него на лету выполнялась генерация исполняемого кода, который предположительно был адаптирован к текущей среде

Статья получена 20 июля 2020 .
С. С. Постнов – МГУ им. М.В. Ломоносова (e-mail: sspostnov@gmail.com).

исполнения.

В 90-е гг. XX века были также попытки реализации технологий динамической компиляции для языков C, Erlang, Prolog и др. [2]. Появившийся в те же годы язык Java изначально характеризовался низкой скоростью трансляции байт-кода в исполняемый машинный код [2]. Причина этого состояла в том, что первые реализации виртуальной машины Java (JVM) представляли собой простые интерпретаторы. Соответственно, решение данной проблемы разработчики увидели в создании JIT-компилятора байт-кода. Считается, что именно тогда термин «just-in-time-компиляция» получил широкое распространение, будучи заимствован Джеймсом Гослингом из профессионального языка промышленников и производителей (где «just-in-time» означало «точно в срок»). В данный момент JIT-компиляторы используются во всех реализациях JVM.

Стоит отметить, что примерно в те же годы идея использования промежуточного представления была реализована в России в многопроцессорном вычислительном комплексе «Эльбрус». Объектный код этой ЭВМ фактически представлял собой постфиксную запись. В комплексе «Эльбрус» в ходе исполнения программы реализовывалась аппаратная JIT-компиляция упомянутого объектного кода в обычный трехадресный регистровый код [1, с. 353].

Первая реализация JIT-компилятора для JVM напрямую использовала байт-код как промежуточное представление для JIT-компилятора [16]. В дальнейшем от этой идеи отошли и стали использовать преобразование байт-кода JVM в промежуточное регистровое представление [17].

Интересным направлением создания JIT-компиляторов для Java стали работы, в которых предлагались компиляторы, выполняющие не только трансляцию байт-кода в нативный код, но и оптимизацию кода [18-22]. В работе [19] была реализована “compile-only” стратегия JIT-компилятора без интерпретатора.

В работе [23] было предложено использовать аннотации для того, чтобы перенести оптимизацию кода преимущественно на этап выполнения. Механизм реализации этой идеи состоял в том, что необходимые для эффективной JIT-оптимизации данные предварительно вычислялись и вносились в байт-код в виде аннотаций, которые потом использовались JIT-компилятором.

Также стоит упомянуть работу [24], где была предложена идея «непрерывной компиляции», в ходе которой интерпретатор и компилятор работают в конкуррентном режиме, желательно на отдельных процессорах.

Далее в обзоре будут рассмотрены современные технологии JIT-компиляции и существующие JIT-компиляторы.

II. JIT-КОМПИЛЯЦИЯ: ПРИНЦИПЫ И ОСОБЕННОСТИ

A. Общее описание

Как правило, реализация технологии JIT применяется для компиляции промежуточного представления программы (байт-кода) непосредственно в машинный код. Компиляция же исходного кода приложения в байт-код может выполняться и не на лету, посредством статической или AoT-компиляция (AoT – «ahead-of-time»).

В такой двухэтапной схеме часто удаётся скомпенсировать лишние затраты времени при запуске приложения более быстрой его работой за счёт JIT-компилятора. При этом высокая производительность статической компиляции сочетается с быстротой и гибкостью динамической. В самом деле, на этапе статической компиляции выполняется большинство «тяжеловесных» операций (парсинг исходного кода, выполнение базовых процедур оптимизации и т.п.), в результате чего формируется байт-код, как известно, более унифицированный и переносимый. Далее компиляция байт-кода и синтез машинного кода выполняются уже динамически (JIT), что занимает гораздо меньше времени. При этом среда исполнения может осуществлять контроль за выполнением скомпилированного байт-кода и запускать приложение в безопасном режиме.

Как уже отмечалось, базовая оптимизация кода выполняется статическим компилятором при синтезе байт-кода. JIT-компилятор также выполняет оптимизацию на стадии работы с байт-кодом, причём, как правило, эти процедуры оптимизации могут быть выполнены только во время исполнения кода на его «горячих» участках. К таким процедурам можно отнести анализ данных, перевод стековых операций в регистровые, снижение времени доступа к памяти за счет использования выделенных регистров, исключение общих (повторяющихся) «подвыражений» и т. д. При этом реализуется более рациональное использование кэша. «Продвинутые» реализации JIT-компилятора могут собирать статистику выполнения разных участков кода и кэшировать результаты компиляции часто вызываемых методов. Такие компиляторы могут работать в многопроходном режиме, выполняя оптимизацию исполняемого кода несколько раз за раунд исполнения [25].

JIT-компилятор может не только собирать статистические данные и использовать их в своей работе, но и предоставлять их среде исполнения, которая также может проводить анализ статистики исполнения конкретной программы. Кроме того, такие данные среда исполнения может предоставлять статическим компиляторам, учитывающим информацию о предыдущих запусках приложения. Также благодаря динамическому компилятору оказывается возможным выполнять процедуры глобальной оптимизации кода (например, встраивание библиотечных функций в код), не привнося лишних накладных расходов, характерных для статических компиляторов.

JIT-компиляция может реализовываться непосредственно для конкретной платформы (процессора и операционной системы), на которой выполняется приложение. Например, JIT-компилятор может анализировать возможность поддержки целевым процессором векторных SSE2-расширений и

использовать их в работе. Такой уровень оптимизации сейчас не очень широко используется, хотя даёт вполне сравнимые со статическими компиляторами результаты. Дело в том, что для реализации такого подхода требуется либо обеспечивать поддержку бинарных файлов, специфичных для каждой из целевых платформ, либо комплектовать библиотеки оптимизаторами, специфичными для каждой из целевых платформ.

B. Классификация

Как и в любой предметной области, для JIT-компиляторов можно строить разные системы классификации с разными наборами признаков и способами их оценки. Далее будет описана одна из таких систем, отличающаяся простотой и предложенная уже достаточно давно [2].

Предполагается, что JIT-системы можно классифицировать по следующим трём признакам.

(1) Вызов (инициирование работы). JIT-компилятор является вызываемым явно, если пользователь должен предпринять некоторые действия, вызывающие компиляцию во время выполнения. Неявно вызываемый JIT-компилятор является прозрачным для пользователя.

(2) Исполняемость. JIT-системы обычно включают в себя два языка: исходный язык для перевода с одного, исходного, языка и целевого языка для перевода на другой, результирующий (хотя эти языки могут быть одинаковыми, если, например, JIT-система выполняет только оптимизацию на ходу). JIT-система называется моноисполняемой, если она может выполнять только один из этих языков, и полиисполняемой, если может выполнять более одного. Полиисполняемые JIT-системы могут решать, когда вызов компилятора необходим, так как можно использовать любое представление программы.

(3) Параллелизм. Это свойство характеризует то, как выполняется JIT-компилятор относительно самой программы. Если выполнение программы приостанавливается по её «собственному желанию», чтобы разрешить компиляцию, то такой режим не обеспечивает параллелизма. JIT-компилятор в этом случае может быть вызван с помощью вызова подпрограммы, передачи сообщения или передачи управления сопрограмме. В отличие от этого, параллельный JIT-компилятор может работать так, что программа выполняется одновременно: в отдельном потоке или процессе, или даже на другом процессоре.

Можно также выделить в отдельный класс JIT-системы, функционирующие в жестком режиме реального времени. Кроме того, в последнее время уделяется внимание языку реализации JIT-компиляторов, в частности, компиляторам для JVM, написанным на Java (см. ниже).

C. Особенности выполнения

Часто отмечается, что типичной особенностью работы JVM и вообще JIT-систем является ощутимая задержка при запуске JIT-компилятора, обусловленная временными затратами на загрузку среды и компиляцию приложения в машинный код. Как правило, чем больше степень оптимизации кода, выполняемой динамическим компилятором на лету, тем итоговая задержка больше. В случае, когда временной ресурс ограничен, перед разработчиками JIT-компиляторов встаёт проблема грамотного компромисса между качеством генерируемого кода и допустимыми временными затратами на запуск приложения. Стоит, однако, помнить, что «узкое горло» в ходе компиляции может быть обусловлено не работой самого компилятора, а задержками системы ввода-вывода.

В качестве альтернативы, используемой для уменьшения времени, затрачиваемого на запуск приложения, применяется так называемая технология pre-JIT. Преимуществом этой технологии является то, что код компилируется до запуска, что заметно снижает вышеупомянутую временную задержку. Но скомпилированный в результате код является гораздо менее оптимизированным и, в целом, менее качественным по сравнению с полноценной JIT-компиляцией.

Отдельным фактором, обусловливающим затраты времени на компиляцию, является порядок, в котором выполняется компиляция или перекомпиляция функций (вообще говоря – единиц компиляции) программы. В работе [26] проводится теоретический анализ потенциального предела оптимизации порядка компиляции. В частности, доказывается сильная NP-полнота исследуемой задачи. Кроме того, предлагается квазиоптимальный алгоритм построения порядка компиляции. В работе [27] предложен алгоритм построения порядка компиляции с учётом размера единицы копиляции.

Проанализируем далее особенности JIT-компиляции по сравнению с бинарной динамической компиляцией (т.е., динамической компиляцией бинарного исполняемого кода) [28].

В бинарной динамической компиляции состояние выполнения включает в себя содержимое машинных регистров и программный счетчик. В JIT-компиляции состояние выполнения зависит от используемой виртуальной машины. Например, в Java состояние выполнения включает в себя содержимое стека среды исполнения Java.

Рассмотрим теперь особенности обработки исключений. Если исключение возникает, когда элемент управления находится внутри кэша скомпилированного кода, то текущее состояние выполнения может не соответствовать ни одному допустимому состоянию в исходной программе. Обработчик исключений в такой ситуации может отказать или отработать неадекватно, если ему было передано состояние выполнения, которое было каким-то образом изменено с помощью динамической компиляции. Ситуация еще более усложняется, если динамический компилятор выполнил оптимизацию динамически скомпилированного кода (т.е., динамическая компиляция проводится в многопроходном режиме).

Сложная ситуация, когда динамическая компиляция изменила состояние выполнения, может возникнуть в результате выполнения оптимизации, которая устраняет

определённые участки кода, перезаполняет регистры и/или переупорядочивает инструкции. В случае JIT-компилятора языка Java это также включает в себя передачу расположений стека в машинные регистры. Чтобы восстановить исходное состояние выполнения фрагмент кода должен быть деоптимизирован. Эта проблема похожа на ту, что возникает при отладке оптимизированного кода, когда исходное неоптимизированное состояние пользователя должно быть представлено программисту при достижении точки останова. Методы деоптимизации для динамической компиляции обсуждаются в литературе [28]. Каждая стратегия оптимизации требует своей собственной стратегии деоптимизации и, более того, не все стратегии (и отдельные процедуры) оптимизации являются деоптимизируемыми. Например, переупорядочение двух операций загрузки памяти не может быть отменено после того, как переупорядоченная предыдущая загрузка была выполнена и вызвала исключение.

Для деоптимизации таких преобразований, как устранение неисполняемого кода, можно использовать несколько подходов. Динамический компилятор может сохранять существенную информацию в каждой точке оптимизации динамически скомпилированного кода. Когда возникает исключение к этой сохраненной информации обращаются, чтобы построить «компенсирующий» (замещающий) код, необходимый для отмены изменений, произведённых в ходе оптимизации, и воспроизведения исходного состояния выполнения. Для устранения неисполняемого кода код компенсации может быть так же прост, как выполнение удалённых инструкций. Хотя этот подход обеспечивает быстрое восстановление состояний на момент исключения, он может потребовать значительного объема памяти для данных деоптимизации.

JIT-компиляция и бинарная динамическая компиляция имеют ряд общих, весьма важных, характеристик. В обоих случаях управление кэшем скомпилированного кода имеет решающее значение, т.к. подобно бинарному динамическому компилятору, JIT-компилятор может использовать профилирование для поэтапной компиляции и оптимизации усилий в нескольких режимах, от быстрого базового режима компиляции без оптимизации до «агрессивно оптимизирующего» режима. Некоторые важные различия между JIT и бинарной динамической компиляцией обусловлены различными уровнями абстракции в их входных данных. Для облегчения выполнения в виртуальной машине промежуточный код обычно снабжен семантической информацией, такой как таблицы символов и констант. JIT-компилятор имеет в данном случае преимущество, поскольку может воспользоваться доступной семантической информацией. Таким образом, JIT-компиляция в данном контексте больше напоминает процесс статической компиляции, чем двоичная перекомпиляция.

Код виртуальной машины, с которым работает JIT-компилятор, обычно не зависит от расположения, а информация о программных компонентах, таких как процедуры или методы, является доступной. В отличие от этого, бинарные динамические компиляторы работают с полностью связанным двоичным кодом и обычно сталкиваются с проблемой восстановления кода. Чтобы распознать поток управления, способы компоновки кода, которые были реализованы при создании двоичного файла, к этому файлу нужно применить механизмы обратного проектирования, и полное восстановление кода в общем случае невозможно. Из-за проблемы восстановления кода бинарные динамические компиляторы более ограничены в выборе единицы компиляции. Они обычно выбирают простые кодовые единицы, такие как линейные кодовые блоки, трассировки или древовидные области. JIT-компиляторы, с другой стороны, могут распознавать высокоуровневые кодовые конструкции и глобальный поток управления. Они обычно выбирают целые методы или процедуры в качестве единицы компиляции, как это сделал бы статический компилятор. Однако в последнее время было признано, что существуют и другие преимущества рассмотрения единиц компиляции с иной степенью детализации, чем вся процедура, такие как уменьшение размеров компилируемого кода [28].

Наличие семантической информации в JIT-компиляторе также позволяет расширить «репертуар оптимизации», т.е. набор используемых методов и процедур. За исключением накладных расходов, JIT-компилятор так же способен оптимизировать код, как и статический компилятор. Как уже отмечалось, JIT-компиляторы могут даже выходить за пределы возможностей статического компилятора, используя преимущества динамической информации о коде. В отличие от этого, бинарный динамический оптимизатор более ограничен низкоуровневым представлением и отсутствием глобального представления о программе. Проблема наложения гораздо серьёзнее в бинарной динамической компиляции, поскольку информация более высокого уровня, которая может помочь устранить неоднозначность ссылок на память, недоступна. Кроме того, отсутствие глобального представления программы вынуждает бинарный динамический компилятор делать наихудшие предположения в точках входа и выхода текущего обработанного фрагмента кода, что может препятствовать безопасной оптимизации.

Рассмотрим теперь основные отличия JIT-компиляции от бинарной динамической компиляции. JIT-компилятор, очевидно, способен производить гораздо более оптимизированный код, чем двоичный компилятор. Однако, если рассматривать сценарии, где целью является не качество (степень оптимизации) кода, а скорость компиляции, то однозначное преимущество JIT-компилятора становится уже не таким очевидным. Ряд решений по компиляции и генерации кода, таких как выделение регистров и отбор инструкций, уже были сделаны в двоичном коде и часто могут быть повторно использованы во время динамической компиляции. Например, двоичные трансляторы обычно строят фиксированное отображение между машинными регистрами гостевой и основной (хостовой) систем. Рассмотрим ситуацию, когда гостевая архитектура имеет меньше регистров, например 32, чем хост-

архитектура, например. 64, так что 32 гостевых регистра могут быть сопоставлены с первыми 32 хост-регистрами. При трансляции инструкций *opcode, op1, op2* транслятор может использовать фиксированное отображение для прямой трансляции операндов из регистров гостевой машины в регистры хост-машины. Таким образом, переводчик может создавать код с глобально выделенными регистрами без какого-либо анализа, просто повторно используя решения о выделении регистров из гостевого кода.

Напротив, JIT-компилятор, работающий с промежуточным кодом, должен выполнять потенциально трудозатратный глобальный анализ, чтобы достичь того же уровня выделения регистров. Таким образом, то, что кажется ограничением, может оказаться полезным в зависимости от сценария компиляции.

D. Риски безопасности

В последние годы всё более актуальными становятся вопросы информационной безопасности программного обеспечения, в том числе, безопасности данных при компиляции и потенциальных уязвимостей компиляторов. Основными типами атак при этом являются атаки внедрения кода и атаки повторного использования кода [29, 30].

Рассмотрим первый из упомянутых типов атак. При JIT-компиляции результат записывается в память и исполняется сразу же, не используя диск и не вызывая код как отдельную программу. В современных архитектурах для обеспечения безопасности вводится разграничение и произвольные участки памяти не могут быть исполнены как машинный код. Для этого память должна быть помечена, как исполняемая (NX bit), причём для усиления безопасности используется следующее правило. Области памяти, в которые записывается исполняемый код, должны быть предварительно помечены как исполняемые, при этом флаг исполнения может ставиться только после снятия флага разрешения записи, когда память становится помечена как доступная только для чтения. Это позволяет избежать появления уязвимостей, связанных с наличием участков перезаписываемой и исполняемой памяти, и называется в литературе «Write XOR Execute» или $W \oplus X$-политикой работы с памятью [29, 31-32]. Другой способ предупреждения атак внедрения кода – организация «песочницы» (sandbox), облегченной виртуальной машины уровня процесса, симулирующей функции JIT-компилятора с помощью динамической двоичной трансляции. При этом ограничивается доступ симулируемых инструкций к кэшу кода и буферу данных. Доступ к внешним данным или коду может быть реализован с помощью вызова специальных библиотечных функций-обёрток, использование которых гарантирует, что программа не будет выполнять вредоносных действий [29].

Атака JIT-Spray заключается в использовании выражений с константами на языке высокого уровня, скомпилированных в машинный код, для встраивания байтов вредоносного кода во время выполнения [33]. Это обходит механизм предупреждения исполнения данных (Data Execution Prevention – DEP), поскольку данные здесь косвенно внедряются как код. Кроме того, если злоумышленнику удастся создать много областей такого кода, их местоположение станет предсказуемым.

Примером атаки повторного использования кода является атака JIT-ROP (ROP – Return Oriented Programming) [29, 30], в ходе которой злоумышленники многократно используют уязвимости раскрытия кода и анализируют смещение инструкций передачи от первоначального «утекшего» фрагмента кода, чтобы найти другие фрагменты кода, пока, наконец, не соберут достаточное количество фрагментов кода для завершения атаки. Для противодействия данной атаке предложены два защитных механизма: рандомизация времени жизни (Lifetime Randomization) и целостность указателя кода (Code Pointer Integrity – CPI) [29, 30]. Первый из этих механизмов основан на мелкомасштабной («fine-grained») рандомизации. Кроме того, запущенный процесс может непрерывно изменять порядок внутренних инструкций в своем пространстве памяти, чтобы обеспечить недействительность информации о расположении «утекшего» кода на момент захвата потока управления. Такой метод обеспечивает высокую эффективность противодействия атаке JIT-ROP, но характеризуется невысокой масштабируемостью и производительностью. Основу механизма CPI составляет то, что все указатели кода в области памяти должны быть изолированы или зашифрованы. Когда программе необходимо разыменовать эти указатели, она должна использовать закрытый ключ, сохраненный в закрытом регистре, для расшифрования. Таким образом обеспечивается гарантия, что информация о указателях кода не компрометируется. Данный механизм является более практичным и эффективным.

Другой тип потенциальных уязвимостей связан с вызовом компилятора, производимым приложением во время выполнения. Например, большинство реализаций Common Lisp содержат функцию *compile*, которая в ходе исполнения может создать функцию; в Python это функция *eval*. Это удобно для программиста, поскольку позволяет отслеживать, какие части кода действительно подлежат компиляции. Таким же образом можно компилировать динамически сгенерированный код, что может обеспечить лучшую производительность, чем реализация в статически скомпилированном коде. Однако такие функции могут быть потенциально опасны в ситуациях, когда работа идёт с данными, переданными из недоверенных источников.

III. Тенденции развития технологий JIT-компиляции

В развитии современных средств динамической компиляции можно выделить две основных тенденции, которые будут рассмотрены ниже: разработка технологий многоуровневой (многоэтапной) компиляции и использование технологий обучения машин в работе компиляторов.

В процессе многоуровневой компиляции выделяется

обычно три этапа: низкоуровневая интерпретация, обеспечивающая быстрый запуск, затем вызов «простого» компилятора, а затем – компилятора, выполняющего агрессивную оптимизацию. При этом каждый из этих этапов может включать дополнительные подэтапы (ступени) и допускает выполнение компиляции в параллельных нитях (“compilation threads”) [34]. Например, в работе [34] выделяется пять основных ступеней в многоуровневой компиляции с участием двух типовых компиляторов – C1 и C2.

Важным процессом в ходе многоуровневой компиляции является замена исполняемого в текущий момент кода другой его версией, в том числе деоптимизированной (OSR – On-Stack Replacement). Существует множество подходов к организации такой замены [35, 36].

Концепция многоуровневой динамической компиляции оказывается весьма актуальной в области высокопроизводительных вычислений в связи с задачами адаптации и непрерывной оптимизации программного кода с учётом изменений среды исполнения и гетерогенности платформы [37]. При решении этих задач может использоваться подход, предполагающий генерацию нескольких версий кода вычислительного ядра (или другого интенсивно используемого кода) и последующий выбор наилучшей (в текущем состоянии среды исполнения, платформы и окружения) версии кода при каждом вызове ядра. Такой выбор может быть частью алгоритма автонастройки, используемого как для настройки параметров программного обеспечения, так и для поиска для оптимальных решений в пространстве допустимых оптимизаций компилятора [37, 38].

В настоящее время также разработаны подходы к построению так называемых «ленивых» (“lazy”) JIT-компиляторов для ситуаций, когда ресурсы разработки сильно ограничены и реализация процедур многоступенчатой компиляции не представляется возможной [39, 40]. Эти подходы позволяют выполнять компиляцию непосредственно из абстрактного синтаксического дерева исходной программы в целевой машинный код, в то же время обеспечивая его оптимизацию и не используя какого-либо промежуточного представления. В работах [39, 40] описана реализация этих подходов в JIT-компиляторе для языка Scheme.

Рассмотрим теперь использование методов обучения машин, широко применяемых сегодня для построения новых, высокоэффективных алгоритмов оптимизации, в области создания новых компиляторов, в том числе JIT-компиляторов.

В работе [41] описано применение методов обучения с учителем к решению задачи планирования порядка выполнения инструкций для JIT-компилятора Java. Для каждого блока кода строился прогноз относительно необходимости его планирования (включения в список планируемых или непланируемых блоков). Это было одним из первых применений методов обучения машин в задачах автонастройки компиляторов, т.е. выбора стратегии оптимизации при заданном пространстве параметров [42]. В работе [43] для обучения моделей автонастройки JIT-компилятора IBM Testarossa использовались алгоритмы машин опорных векторов.

В работах [44-45] описан самонастраивающийся и «самооптимизирующийся» («self-optimized») адаптивный компилятор MILEPOST GCC. В этом компиляторе для обучения используется набор тренировочных программ, в результате чего создаётся обобщённая модель программы. Эта модель строится на основе признаков, описывающих различные структуры данных программы, и объединяемых в вектор признаков. Далее проводится вероятностное обучение, в ходе которого выделяются оптимальные или квазиоптимальные (высокоэффективные) стратегии оптимизации. Критерием качества оптимизации при этом может быть время компиляции, длина программы и т.п. (возможно и использование составных критериев на основе нескольких показателей) В итоге строится предиктор (в том числе, с использованием технологии деревьев решений), позволяющий прогнозировать оптимальную стратегию компиляции для новой (не входившей в обучающий набор) программы.

В работе [46] описано применение нейросетевых алгоритмов обучения в задаче выбора порядка применения процедур оптимизации к анализируемому блоку кода. Авторами использовалась технология NEAT (Neuro-Evolution for Augmenting Topologies), реализованная применительно к JIT-компилятору Jikes RVM. В этой работе создавалось множество нейронных сетей, каждая из которых использовалась для прогнозирования следующей процедуры оптимизации, наиболее подходящей для применения к текущему фрагменту кода, на основе статических признаков этого кода. Далее реализовывалась эволюция этого множества сетей, в результате чего вырабатывалась наиболее подходящая последовательность процедур оптимизации, применяемых в процессе компиляции к заданному фрагменту кода.

IV. JIT-КОМПИЛЯТОРЫ

Здесь коротко будут описаны конкретные реализации JIT-компиляторов, используемые на сегодня.

Выше отмечалось, что популярной стратегией оптимизации является компиляция только тех участков кода приложения, которые используются чаще всего. Такой подход реализован в PyPy и HotSpot JVM компании Sun Microsystems, входит в состав Open JDK. В качестве эвристики здесь может использоваться счётчик запусков участков приложения, размер байт-кода или детектор циклов.

В HotSpot JVM имеется два режима работы — клиент и сервер. Им соответствуют два JIT-компилятора – клиентский C1 и серверный C2 или *opto* [34]. В режиме клиента количество компиляций и процедур оптимизации минимизировано для более быстрого запуска, а в режиме сервера, наоборот достигается максимальная производительность, но, как следствие, растёт величина задержки запуска. Изначально идея состояла в том, что клиентский компилятор ориентирован на настольные приложения, для которых нежелательны длительные паузы из-за работы JIT-

компилятора. Серверный же компилятор был ориентирован на «долгоиграющие» серверные приложения в которых затраты времени на компиляцию менее ощутимы [34, 47, 48]. На сегодня эти два подхода комбинируются в рамках технологии ступенчатой компиляции (tiered compilation) так, чтобы сначала код компилировался с помощью C1, а затем, если он продолжает интенсивно выполняться и имеет смысл затратить дополнительное время, — с помощью C2 [34, 47, 48].

В настоящее время набирает популярность новый JIT-компилятор для Java – Graal [47]. Его отличительной чертой является то, что он написан на Java, в отличие от компиляторов C1 и C2, написанных на C++. Отмечается, что новый компилятор за счёт реализации на Java позволит обеспечить больший уровень безопасности, реализацию обработки исключений, обусловливающую невозможность аварийных завершений программ, отсутствие реальных утечек памяти или «висячих» указателей. Кроме того, данный компилятор имеет современные вспомогательные средства (отладчики, профилировщики, и инструменты вроде *VisualVM*) и поддержку IDE [47].

Технологии JIT-компиляции реализованы также в известной библиотеке LLVM, представляющей собой программную инфраструктуру для создания компиляторов и сопутствующих им утилит. Данная библиотека написана на языке C++ и включает набор компиляторов для языков высокого уровня (включая C++, C#, Java, Scala, Kotlin, Rust, Haskel и многие другие), систем оптимизации, интерпретации и компиляции в машинный код [49]. Инфраструктура MLIR [50] во многом аналогична LLVM, но считается «более чистой» средой для создания компиляторов.

Более низкоуровневый подход реализован в библиотеке GNU Lightning [51], написанной на языке C. В этой библиотеке, в отличие от LLVM, не предполагается использование какого-либо промежуточного представления программ, вроде байткода, а осуществляется трансляция инструкций некоторого абстрактного, машинно-независимого, ассемблера виртуального RISC-процессора в машинный код одной из поддерживаемых архитектур. Кроме того, данная библиотека не предоставляет средств оптимизации.

В начале этого года появилась информация специалистов компании Red Hat о проводимой ими разработке нового легковесного JIT-компилятора MIR, способного обеспечивать выполнение кода, предварительно преобразованного в промежуточное представление MIR (Medium Internal Representation) [52]. Код проекта написан на языке Си и пока позволяет осуществлять трансляцию в MIR для языка Си и биткода LLVM, при этом планируется расширить возможность генерации MIR, в том числе реализовав их для байткода Java, CIL (Common Intermediate Language), и C++. Особенностями MIR являются строгая типизация, поддержка модулей и функций, предоставление набора инструкций для преобразования типов, сравнения, арифметических и логических операций, ветвления и т.п. Модули, включающие набор функций, преобразованных в формат MIR, могут загружаться в форме библиотек, а также загружать внешний код на языке Си. В качестве ключевого достоинства нового компилятора отмечается возможность выполнения промежуточного кода в JIT вместо компиляции в нативные исполняемые файлы, позволяющая формировать компактные файлы, выполняемые без пересборки на разных аппаратных архитектурах (x86, ARM, PPC, MIPS).

Наконец, стоит отметить JIT-компилятор, предназначенный для выполнения кода не на центральном процессоре, а на графическом сопроцессоре [53]. Этот компилятор реализует функции контекстно-свободного микроуправления ресурсами графического сопроцессора, заключающегося в объединении и переупорядочении ядер, занятых выполнением команд, в реальном времени. Таким образом реализуется одновременно пространственное и временное перераспределение ресурсов, позволяющее оптимизировать загрузку ядер сопроцессора и время выполнения программ.

V. Заключение

В настоящем обзоре описана история возникновения и развития технологий JIT-компиляции, приведено описание данной технологии и существующая классификация JIT-компиляторов. Описаны особенности JIT-компиляции как подкласса методов динамической компиляции кода и проведено сравнение с другими методами этого класса, в частности, с бинарной динамической компиляцией. Упомянуты современные тенденции развития JIT-компиляторов, в частности, методы многоуровневой компиляции и методы на основе технологий машинного обучения. Отмечены особенности JIT-компиляции, связанные с возникновением рисков безопасности. Наконец, описаны конкретные реализации JIT-компиляторов.

Благодарности

Автор выражает благодарность Д.Е. Намиоту за предложение написать данный обзор и полезные советы по его содержанию.

Библиография

[1] С.З. Свердлов, *Конструирование компиляторов*. Saarbrucken: Lambert Academic Publishing, 2015.

[2] J. Aycock, “A Brief History of Just-In-Time”, *ACM Computing Surveys*, vol. 35, No. 2, pp. 97–113, June 2003.

[3] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part I”, *Commun. ACM*, vol. 3, No. 4, pp. 184–195, 1960.

[4] *University of Michigan Executive System for the IBM 7090 Computer*, Vol. 1-2, University of Michigan: Ann Arbor, MI, 1966.

[5] K. Thompson, “Regular expression search algorithm”, *Commun. ACM*, vol. 11, No. 6, pp. 419–422, June 1968.

[6] J. G. Mitchell, A. J. Perlis, H. R. van Zoeren, “LC^2: A language for conversational computing”, *Interactive Systems for Experimental Applied Mathematics*, M. Klerer and J. Reinfelds, Eds. New York: Academic Press, 1968.

[7] P. S. Abrams, “An APL machine”, Ph.D. dissertation, Stanford University, Stanford, CA, Stanford Linear Accelerator Center (SLAC), Rep. 114, 1970.

[8] R.L. Johnston, “The dynamic incremental compiler of APL\3000”, in *APL’79 Conference Proceedings, APL Quote Quad 9*, 4 (June), Pt. 1, pp.82–87, 1977.

[9] E. J. van Dyke, “A dynamic incremental compiler for an interpretive language”, *Hewlett-Packard J.*, vol. 28, No. 11, pp. 17–24, July 1977.

[10] G.J. Hansen, “Adaptive systems for the dynamic run-time optimization of programs”, Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA, 1974.

[11] T.S. Ng, A. Cantoni, “Run time interaction with FORTRAN using mixed code”, *The Comput.J.,*.vol. 19, No. 1, pp. 91–92, 1976.

[12] L.P. Deutsch, A.M. Schiffman, “Efficient implementation of the Smalltalk-80 system”, in *Proceedings of POPL ’84*, pp. 297–302, 1984.

[13] U. Hölzle, “Adaptive optimization for Self: Reconciling high performance with exploratory programming”, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1994.

[14] M. Franz, “Code-generation on-the-fly: A key to portable software”, Ph.D. dissertation. ETH Zurich, Zurich, Switzerland., 1994.

[15] M. Franz, T. Kistler, “Slim binaries”, *Commun. ACM*, vol. 40, No. 12, pp. 87–94, Dec. 1997.

[16] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko, “Compiling Java just in time”, *IEEE Micro*, vol. 17, No. 3, pp. 36–43, May/June 1997.

[17] *The Java HotSpot virtual machine*, White paper, Sun Microsystems, Santa Clara, CA, 2001.

[18] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V.M. Parikh, J.M. Stichnoth, “Fast, effective code generation in a just-in-time Java compiler”, *PLDI ’98*, pp. 280–290, 1998.

[19] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, “The Jalape ˜ no dynamic optimizing compiler for Java”, *Proceedings of JAVA* ’99, pp. 129–141, 1999.

[20] A. J. C. Bik, M. Girkar, M.R. Haghighat, “Experiences with Java JIT optimization”, In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, , Los Alamitos, CA: IEEE Computer Society Press, pp. 87–94, 1999.

[21] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, T. Nakatani, “Design, implementation, and evaluation of optimizations in a just-in-time compiler”, *Proceedings of JAVA ’99*, pp. 119–128, 1999.

[22] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcio˘glu, E. Altman, “LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation”, In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Los Alamitos, CA : IEEE Computer Society Press,. pp. 128–138, 1999.

[23] A. Azevedo, A. Nicolau, J. Hummel, “Java annotation-aware just-in-time (AJIT) compilation system”, *Proceedings of JAVA ’99*, pp.142–151, 1999.

[24] M.P. Plezbert, R. K. Cytron, “Does “just in time” = “better late then never”?”, *Proceedings of POPL ’97*, pp. 120–131, 1997.

[25] M. Arnold, S.J. Fink, D. Grove, M. Hind, P.F. Sweeney, “A Survey of Adaptive Optimization in Virtual Machines”, *Proc. IEEE*, vol. 93 , Is. 2, pp. 449 – 466, Feb. 2005.

[26] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, X. Shen, “Finding the Limit: Examining the Potential and Complexityof Compilation Scheduling for JIT-Based Runtime Systems”, *SIGARCH Comput. Archit. News*, vol. 42, No. 1, pp. 607–622, Feb. 2014.

[27] J. Brock, C. Ding, X. Xu, Y. Zhang, “PAYJIT: Space-Optimal JIT Compilation and Its Practical Implementation”, *Proc. 27th International Conference on Compiler Construction (CC 2018)*, New York, NY, USA: ACM, pp. 71–81, 2018.

[28] *The Compiler design handbook: optimizations and machine code generation*, Ed. by Y.N. Srikant, P. Shankar, 2nd ed., CRC Press, 2018.

[29] B. Tang, H Ying, W. Wang, H. Tang, “Eternal War in Software Security: A Survey of Control Flow Protection”, *Advances in Computer Science Research*, vol. 59, pp. 716-725, 2017.

[30] Z. Shen, W. Chen, “A Survey of Research on Runtime Rerandomization Under Memory Disclosure”, *IEEE Access*, vol. 7, pp. 105432-105440, 2019.

[31] https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/

[32] https://www.phoronix.com/scan.php?page=news_item&px=W-XOR-E-Firefox

[33] R. Gawlik, T. Holz, “SoK: Make JIT-Spray Great Again”, In Proc. Of 12th {USENIX} Workshop on Offensive Technologies ({WOOT}), Baltimore, MD, paperID 220582, 2018.

[34] S. Oaks, *Java Performance: The Definitive Guide*, O'Reilly Media, Inc., 2014.

[35] https://www.cs.purdue.edu/homes/rompf/papers/essertel-preprint201907.pdf

[36] D. Cono D’Elia, C. Demetrescu, “On-stack replacement, distilled”, *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 166–180, June 2018.

[37] M. Festa, N. Gervasoni, S. Cherubin, G. Agosta, “Continuous Program Optimization via Advanced Dynamic Compilation Techniques”, *Proc. 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2019)*, New York, NY, USA, ACM, Article 2, pp. 1–6, 2019.

[38] S. Benkner, F. Franchetti, H.M. Gerndt, J.K. Hollingsworth, “Automatic Application Tuning for HPC Architectures”, *Dagstuhl Reports*, vol. 3, No. 9, pp. 214–244, 2014.

[39] B. Saleil, M. Feeley, “Building JIT compilers for dynamic languages with low development effort”, *Proc. 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2018)*, New York, NY, USA, ACM, pp. 36–46, 2018.

[40] B. Saleil, “Simple Optimizing JIT Compilation of Higher-Order Dynamic Programming Languages”, Ph.D. dissertation, Université de Montréal, Montréal, 2019.

[41] J. Cavazos, J.E.B. Moss, ‘Inducing heuristics to decide whether to schedule”, *ACM SIGPLAN Not.*, vol. 39, is. 6, pp. 183–194, May 2004.

[42] A.H. Ashouri, W. Killian, J. Cavazos, G. Palermo, C. Silvano, “A Survey on Compiler Autotuning using Machine Learning”, *ACM Comput. Surv.*, vol. 51, No. 5, Article 96, pp. 96:1-96:8, September 2018.

[43] R.N. Sanchez, J.N. Amaral, D. Szafron, M. Pirvu, M. Stoodley, “Using machines to learn method-specific compilation strategies”, *Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’11)*, IEEE Computer Society, USA, pp. 257–266, 2011.

[44] G. Fursin et al., “MILEPOST GCC: machine learning based research compiler”, *Proc. GCC Developers' Summit 2008*, Ottawa, Canada, p. 1-6, 2008. (https://hal.inria.fr/inria-00294704/)

[45] G. Fursin et al., “Milepost GCC: Machine Learning Enabled Self-tuning Compiler”, *Int. J. Parallel Prog.*, vol. 39, pp. 296–327, 2011.

[46] S, Kulkarni, J, Cavazos, “Mitigating the compiler optimization phase-ordering problem using machine learning”, *ACM SIGPLAN Not.*, vol. 47, No. 10, pp. 147–162, October 2012.

[47] https://chrisseaton.com/truffleruby/jokerconf17/

[48] https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/

[49] LLVM http://llvm.org/

[50] C. Lattner et al., “MLIR: A Compiler Infrastructure for the End of Moore’s Law”, arXiv:2002.11054v2, March 2020.

[51] https://www.gnu.org/software/lightning/

[52] https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project/

[53] J. Paras, X. Mo, A. Jain, A. Tumanov, J.E. Gonzalez, I. Stoica, “The OoO VLIW JIT Compiler for GPU Inference”, arXiv:1901.10008, January 2019.

A Survey of JIT-Compilation Technologies

Sergey Postnov

***Abstract*— This survey focuses on the principles of JIT compilation ("just-in-time" compilation) of programs. Briefly discusses the history of the development of technologies for dynamic compilation of programs in different programming languages. The implementation of JIT compilation technology in Java virtual machines is discussed in detail. The classification of JIT compilers is given. The basic JIT compilation technology, its features, and examples of its implementation in compilers for the Java language are described. The potentially achievable complexity and degree of optimization of the compilation order are discussed. A comparison is made between JIT compilation and binary dynamic compilation. Attention is paid to the security risks inherent in the JIT compiler. In particular, we analyze vulnerabilities of JIT compilers in relation to code implementation and reuse attacks. Measures to counter this type of attack are described. An overview of modern principles for developing JIT compilers, including using machine learning methods, is given.**

***Keywords*—JIT compiler, dynamical compilation, Java Virtual Machine.**

REFERENCES

[1] S.Z. Sverdlov, Konstruirovanie kompiljatorov. Saarbrucken: Lambert Academic Publishing, 2015.
[2] J. Aycock, “A Brief History of Just-In-Time”, ACM Computing Surveys, vol. 35, No. 2, pp. 97–113, June 2003.
[3] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part I”, Commun. ACM, vol. 3, No. 4, pp. 184–195, 1960.
[4] University of Michigan Executive System for the IBM 7090 Computer, Vol. 1-2, University of Michigan: Ann Arbor, MI, 1966.
[5] K. Thompson, “Regular expression search algorithm”, Commun. ACM, vol. 11, No. 6, pp. 419–422, June 1968.
[6] J. G. Mitchell, A. J. Perlis, H. R. van Zoeren, “LC2: A language for conversational computing”, Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds, Eds. New York: Academic Press, 1968.
[7] P. S. Abrams, “An APL machine”, Ph.D. dissertation, Stanford University, Stanford, CA, Stanford Linear Accelerator Center (SLAC), Rep. 114, 1970.
[8] R.L. Johnston, “The dynamic incremental compiler of APL\3000”, in APL’79 Conference Proceedings, APL Quote Quad 9, 4 (June), Pt. 1, pp.82–87, 1977.
[9] E. J. van Dyke, “A dynamic incremental compiler for an interpretive language”, Hewlett-Packard J., vol. 28, No. 11, pp. 17–24, July 1977.
[10] G.J. Hansen, “Adaptive systems for the dynamic run-time optimization of programs”, Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA, 1974.
[11] T.S. Ng, A. Cantoni, “Run time interaction with FORTRAN using mixed code”, The Comput.J.,.vol. 19, No. 1, pp. 91–92, 1976.
[12] L.P. Deutsch, A.M. Schiffman, “Efficient implementation of the Smalltalk-80 system”, in Proceedings of POPL ’84, pp. 297–302, 1984.
[13] U. Hölzle, “Adaptive optimization for Self: Reconciling high performance with exploratory programming”, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1994.
[14] M. Franz, “Code-generation on-the-fly: A key to portable software”, Ph.D. dissertation. ETH Zurich, Zurich, Switzerland., 1994.
[15] M. Franz, T. Kistler, “Slim binaries”, Commun. ACM, vol. 40, No. 12, pp. 87–94, Dec. 1997.
[16] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko, “Compiling Java just in time”, IEEE Micro, vol. 17, No. 3, pp. 36–43, May/June 1997.
[17] The Java HotSpot virtual machine, White paper, Sun Microsystems, Santa Clara, CA, 2001.
[18] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V.M. Parikh, J.M. Stichnoth, “Fast, effective code generation in a just-in-time Java compiler”, PLDI ’98, pp. 280–290, 1998.
[19] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, “The Jalape ˜ no dynamic optimizing compiler for Java”, Proceedings of JAVA ’99, pp. 129–141, 1999.
[20] A. J. C. Bik, M. Girkar, M.R. Haghighat, “Experiences with Java JIT optimization”, In Innovative Architecture for Future Generation High-Performance Processors and Systems, , Los Alamitos, CA: IEEE Computer Society Press, pp. 87–94, 1999.
[21] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, T. Nakatani, “Design, implementation, and evaluation of optimizations in a just-in-time compiler”, Proceedings of JAVA ’99, pp. 119–128, 1999.
[22] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcio˘glu, E. Altman, “LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation”, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Los Alamitos, CA : IEEE Computer Society Press,. pp. 128–138, 1999.
[23] A. Azevedo, A. Nicolau, J. Hummel, “Java annotation-aware just-in-time (AJIT) compilation system”, Proceedings of JAVA ’99, pp.142–151, 1999.
[24] M.P. Plezbert, R. K. Cytron, “Does “just in time” = “better late then never”?”, Proceedings of POPL ’97, pp. 120–131, 1997.
[25] M. Arnold, S.J. Fink, D. Grove, M. Hind, P.F. Sweeney, “A Survey of Adaptive Optimization in Virtual Machines”, Proc. IEEE, vol. 93 , Is. 2, pp. 449 – 466, Feb. 2005.
[26] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, X. Shen, “Finding the Limit: Examining the Potential and Complexityof Compilation Scheduling for JIT-Based Runtime Systems”, SIGARCH Comput. Archit. News, vol. 42, No. 1, pp. 607–622, Feb. 2014.
[27] J. Brock, C. Ding, X. Xu, Y. Zhang, “PAYJIT: Space-Optimal JIT Compilation and Its Practical Implementation”, Proc. 27th International Conference on Compiler Construction (CC 2018), New York, NY, USA: ACM, pp. 71–81, 2018.
[28] The Compiler design handbook: optimizations and machine code generation, Ed. by Y.N. Srikant, P. Shankar, 2nd ed., CRC Press, 2018.
[29] B. Tang, H Ying, W. Wang, H. Tang, “Eternal War in Software Security: A Survey of Control Flow Protection”, Advances in Computer Science Research, vol. 59, pp. 716-725, 2017.
[30] Z. Shen, W. Chen, “A Survey of Research on Runtime Rerandomization Under Memory Disclosure”, IEEE Access, vol. 7, pp. 105432-105440, 2019.
[31] https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/
[32] https://www.phoronix.com/scan.php?page=news_item&px=W-XOR-E-Firefox
[33] R. Gawlik, T. Holz, “SoK: Make JIT-Spray Great Again”, In Proc. Of 12th {USENIX} Workshop on Offensive Technologies ({WOOT}), Baltimore, MD, paperID 220582, 2018.
[34] S. Oaks, Java Performance: The Definitive Guide, O'Reilly Media, Inc., 2014.
[35] https://www.cs.purdue.edu/homes/rompf/papers/essertel-preprint201907.pdf
[36] D. Cono D’Elia, C. Demetrescu, “On-stack replacement, distilled”, Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 166–180, June 2018.
[37] M. Festa, N. Gervasoni, S. Cherubin, G. Agosta, “Continuous Program Optimization via Advanced Dynamic Compilation Techniques”, Proc. 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2019), New York, NY, USA, ACM, Article 2, pp. 1–6, 2019.
[38] S. Benkner, F. Franchetti, H.M. Gerndt, J.K. Hollingsworth, “Automatic Application Tuning for HPC Architectures”, Dagstuhl Reports, vol. 3, No. 9, pp. 214–244, 2014.
[39] B. Saleil, M. Feeley, “Building JIT compilers for dynamic languages with low development effort”, Proc. 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate

Languages (VMIL 2018), New York, NY, USA, ACM, pp. 36–46, 2018.
[40] B. Saleil, “Simple Optimizing JIT Compilation of Higher-Order Dynamic Programming Languages”, Ph.D. dissertation, Université de Montréal, Montréal, 2019.
[41] J. Cavazos, J.E.B. Moss, ‘Inducing heuristics to decide whether to schedule”, ACM SIGPLAN Not., vol. 39, is. 6, pp. 183–194, May 2004.
[42] A.H. Ashouri, W. Killian, J. Cavazos, G. Palermo, C. Silvano, “A Survey on Compiler Autotuning using Machine Learning”, ACM Comput. Surv., vol. 51, No. 5, Article 96, pp. 96:1-96:8, September 2018.
[43] R.N. Sanchez, J.N. Amaral, D. Szafron, M. Pirvu, M. Stoodley, “Using machines to learn method-specific compilation strategies”, Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’11), IEEE Computer Society, USA, pp. 257–266, 2011.
[44] G. Fursin et al., “MILEPOST GCC: machine learning based research compiler”, Proc. GCC Developers' Summit 2008, Ottawa, Canada, p. 1-6, 2008. (https://hal.inria.fr/inria-00294704/)
[45] G. Fursin et al., “Milepost GCC: Machine Learning Enabled Self-tuning Compiler”, Int. J. Parallel Prog., vol. 39, pp. 296–327, 2011.
[46] S, Kulkarni, J, Cavazos, “Mitigating the compiler optimization phase-ordering problem using machine learning”, ACM SIGPLAN Not., vol. 47, No. 10, pp. 147–162, October 2012.
[47] https://chrisseaton.com/truffleruby/jokerconf17/
[48] https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/
[49] LLVM http://llvm.org/
[50] C. Lattner et al., “MLIR: A Compiler Infrastructure for the End of Moore’s Law”, arXiv:2002.11054v2, March 2020.
[51] https://www.gnu.org/software/lightning/
[52] https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project/
[53] J. Paras, X. Mo, A. Jain, A. Tumanov, J.E. Gonzalez, I. Stoica, “The OoO VLIW JIT Compiler for GPU Inference”, arXiv:1901.10008, January 2019.