ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 30–50

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.421.6

МОДЕЛИ ПАМЯТИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ: ОБЗОР И ТЕНДЕНЦИИ

© 2021 г. Е. А. Моисеенко[a,c,*], А. В. Подкопаев[b,c,], Д. В. Кознов[a,***]**

[a] *Санкт-Петербургский государственный университет*
198504 Санкт-Петербург, Петергоф, Университетский пр. 28, Россия
[b] *Национальный исследовательский университет "Высшая школа экономики"*
194100 Санкт-Петербург, Кантемировская ул. 3b. 1, Россия
[c] *JetBrains Research 197342 Санкт-Петербург, Кантемировская ул. 2, Россия*

**E-mail: e.moiseenko@2012.spbu.ru*

***E-mail: apodkopaev@hse.ru*

****E-mail: d.koznov@spbu.ru*

Поступила в редакцию 17.05.2021 г.
После доработки 13.06.2021 г.
Принята к публикации 13.07.2021 г.

Модель памяти языка программирования определяет семантику многопоточных программ, создаваемых на этом языке и оперирующих с разделяемой памятью. Наиболее известна модель последовательной согласованности, которая является слишком строгой, запрещая многие сценарии поведения, наблюдаемые при исполнении программ на современных процессорах. Попытки формально описать эти сценарии привели к возникновению так называемых слабых моделей памяти. В последние годы было предложено значительное количество слабых моделей памяти для различных языков программирования. Эти модели предлагают различные компромиссы относительно простоты/сложности рассуждений о поведении многопоточных программ и возможностей их оптимизации. Цель данной статьи заключается в обзоре существующих моделей памяти языков программирования и выработке общих рекомендаций по выбору/созданию модели памяти при создании/стандартизации языков программирования, а также при разработке компиляторов. Для данного обзора мы рассмотрели более 2000 статей, найденных по ключевым словам "Relaxed Memory Models", "Weak Memory Models", и "Weak Memory Consistency" поисковой системой Google Scholar. Используя разные критерии, мы сузили это множество до 40 статей, предлагающих и описывающих модели памяти языков программирования. Мы разделили эти модели на шесть классов и проанализировали их свойства и ограничения. В заключение мы показали, как дизайн языка программирования влияет на выбор модели памяти и обсудили возможные направления дальнейших исследований в этой области.

DOI: 10.31857/S0132347421060054

1. ВВЕДЕНИЕ

Многопоточное программирование активно используется в современном программировании, позволяя достичь существенного выигрыша в производительности системного программного обеспечения – ядер операционных систем, СУБД, высоконагруженных клиент-серверных приложений и пр. Главная трудность многопоточного программирования заключается в необходимости обеспечить корректную синхронизацию между различными потоками программы. Обычно это достигается при помощи специальных примитивов синхронизации, таких как блокировки, барьеры, каналы и т.д. Однако часто использование этих примитивов не позволяет достичь нужной производительности. Распространенным примером являются различные неблокирующие (lock-free) структуры данных. В подобных случаях необходимо обращаться к средствам низкоуровневого программирования и взаимодействовать с разделяемой потоками памятью напрямую. Здесь начинаются сложности.

Рассмотрим конкретный пример. Ниже представлена упрощенная версия алгоритма блокировки Деккера [1]:

$x := 1$	$y := 1$
$r_1 := y$	$r_2 := x$
if $r_1 := 0$ {	if $r_2 := 0$ {
//critical section	*//critical section*
}	}

(Dekker's Lock)

В этой программе два потока соревнуются за доступ к критической секции. Чтобы обозначить свое намерение войти в критическую секцию, потоки устанавливают значение переменных x и y соответственно[1]. Поток, который первым установит значение переменной и прочитает значение другой переменной до его установки, получает право войти в критическую секцию. Алгоритм полагается на тот факт, что оба потока не могут одновременно прочитать значение 0[2]. В противном случае два потока способны одновременно войти в критическую секцию, таким образом нарушая корректность алгоритма.

Естественно ожидать, что эта программа завершится с одним из следующих результатов: $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$ или $[r_1 = 1, r_2 = 1]$. Соответствующие сценарии поведения называются *последовательно согласованными* (*sequentially consistent*) [2], и это означает, что они могут быть получены в результате поочередного последовательного исполнения инструкций потоков.

Тем не менее не все сценарии поведения, наблюдаемые в многопоточных системах, являются последовательно согласованными. Например, если перевести псевдокод алгоритма Деккера на язык C, выполнить компиляцию полученной программы с помощью компилятора GCC и запустить получившийся код на процессорах семейства x86/x64, то можно наблюдать, помимо перечисленных выше результатов, еще и результат $[r_1 = 0, r_2 = 0]$. Сценарий поведения, который приводит к этому результату, является примером слабого (*weak*) поведения.

Слабые сценарии поведения появляются в результате оптимизаций программ компиляторами и процессорами. Например, рассматривая программу Dekker's Lock, оптимизатор может заметить, что запись в переменную x и чтение из y в левом потоке являются независимыми инструкциями и, следовательно, могут быть переупорядочены (заметим, что эта оптимизация является корректной для случая однопоточных программ). Для оптимизированной программы поведение с результатом $[r_1 = 0, r_2 = 0]$ является последовательно согласованным.

Множество допустимых сценариев поведения программы определяется семантикой многопоточной системы или *моделью памяти*. Модель *последовательной согласованности* (*sequential consistency*, SC) допускает только последовательно согласованные сценарии поведения. Модели памяти, которые допускают слабые сценарии поведения, называются, соответственно, *слабыми моделями памяти* (*weak memory models*).

Современные процессоры и языки программирования не ограничиваются моделью последовательной согласованности, так как она не позволяет многие важные оптимизации. Таким образом, важно понимать, на сколько слабой должна быть модель памяти. Более строгая модель допускает меньше сценариев поведения и предоставляет больше гарантии программисту. С другой стороны, более слабая модель позволяет выполнять большее количество оптимизаций, что повышает производительность программы.

Оказывается, что этот вопрос весьма сложен. Это привело к тому, что за последние 20 лет было предложено множество моделей для различных языков программирования, например, для Java [3, 4], C/C++ [5], LLVM [6], JavaScript [7], OCaml [3], Haskell [8] и т.д. Эти модели преследуют разные цели, делают различные компромиссы и имеют разнообразные ограничения. Более того, в данной области продолжают появляться новые исследования. По нашим подсчетам, за последние 10 лет в течение каждого года было опубликовано не менее 50 статей по этой тематике[3]. Тем не менее, несмотря на долгую историю и существенный прогресс, нет единого источника, который бы суммировал известную информацию и сравнивал существующие модели памяти различных языков программирования. Целью данной статьи является создание такого обзора.

Мы рассматриваем существующие модели памяти языков программирования, обсуждаем их дизайн, компромиссы и ограничения. Также мы сравниваем эти модели на предмет того, какие оптимизации они поддерживают и какие гарантии предоставляют программистам.

Мы надеемся, что наша работа будет полезна для исследователей в области языков программирования, желающих ознакомиться с темой слабых моделей памяти, а также для разработчиков компиляторов и виртуальных машин, которым необходимо выбрать модель памяти для их системы.

Данная статья организована следующим образом. В § 2 представлен обзор литературы. Далее мы описываем методологию нашего исследования § 3. Затем мы вводим критерии сравнения моделей памяти § 4, в частности, оптимальность схем компиляции, корректность преобразований программ и предоставляемые гарантии для рассуждения о поведении программ. Далее мы описываем способ, которым мы сравниваем модели § 5. В § 6 мы классифицируем модели на основе

[1] В этой статье переменные, разделяемые разными потоками программы, мы обозначаем следующим образом – x, y, z ..., а локальные переменные потока – r_1, r_2, r_3...

[2] Здесь и далее мы подразумеваем, что исходно переменные инициализированы значением 0, если иное не указано явно.

[3] Эти подсчеты подкреплены данными, полученными с помощью поисковой системы Google Scholar, подробности представлены 3.

их свойств и обсуждаем каждый класс в отдельности. В § 7 мы представляем набор рекомендации по выбору модели памяти на основе требований к языку программирования и рассматриваем их на примере языка Kotlin[4]. Наконец, в § 8 мы подводим итоги и обсуждаем возможные направления дальнейших исследований.

2. ОБЗОР ЛИТЕРАТУРЫ

Слабые модели памяти могут быть разделены на два класса: для архитектур процессоров и для языков программирования. Основная разница между ними заключается в том, что модели языков программирования должны поддерживать широкий класс оптимизаций, выполняемых компиляторами.

На сегодняшний день модели архитектур процессоров относительно хорошо изучены. Все основные архитектуры имеют формально определенные модели: x86 [9], IBM **POWER** [10–12], **Arm** [10, 12–15] и **RISC-V** [14]. Процессоры семейств x86 и **POWER** обладают стабильными моделями, которые не претерпевали изменений в течение последних нескольких лет, в то время как модель памяти процессоров семейства **Arm** существенно изменилась при переходе от **Armv7** [12] к **Armv8** [14] и была дополнена поддержкой новых инструкций для обращения к разделяемой памяти. Архитектура **RISC-V**[5] была представлена в 2010 году и недавно к ней была адаптирована модель [14], почти идентичная модели **Armv8**.

Все вышеупомянутые модели формализованы в *декларативном* (или *аксиоматическом*) стиле, ставшем стандартом для спецификации слабых моделей памяти и поддерживающим различные инструменты для тестирования и верификации моделей [12]. Модели некоторых языков программирования также используют декларативный стиль, например, C/C++ [5], JavaScript [7], Java [3] и другие. Тем не менее, декларативные модели не позволяют решить некоторые проблемы, характерные для моделей языков программирования, таких как C/C++, призванных обеспечить эффективную компиляцию в код целевой архитектуры и поддержку оптимизаций, которые потенциально могут удалять *синтаксические зависимости* между инструкциями (например, распространение констант).

Проблема заключается в том, что декларативные модели не способны отличить истинные *семантические зависимости* между инструкциями от ложных. Например, в примере ниже существует зависимость между инструкциями в левом потоке, в то время как зависимость между инструкциями в правом потоке является нет:

$$\begin{array}{l|l} r := x & r := x \\ y := r & y := r * 0 \end{array}$$

Для моделей памяти архитектур процессоров нет необходимости в таком различении, так как они сохраняют все синтаксические зависимости инструкций, в то время как оптимизирующий компилятор может, например, заменить выражение $r * 0$ на 0 и таким образом удалить соотвествующую зависимость. Таким образом, необходимо либо различать истинные и ложные зависимости, либо полностью игнорировать информацию о зависимостях (текущая формализация модели C/C++ использует второй подход). Но если игнорировать информацию о зависимостях в комбинации с поддержкой *буферизации операций чтения*, допустимых спецификацией процессоров **Arm** и **POWER**, то возникают так называемые *значения из воздуха* [16], нарушающие базовые гарантии поведения программ (эта проблема подробно обсуждается в § 4.3.4).

Для решения данной проблемы с сохранением поддержки различных оптимизаций было предложено множество моделей, использующих различные подходы: Java Memory Model (JMM) [3], Promising semantics [17, 18], Weakestmo [19], Modular Relaxed Dependencies (MRD) [20]. Некоторые другие модели, например, **RC11** [21] и модель памяти языка OCaml [22] избегают проблемы путем запрещения некоторых оптимизаций и предоставления больших гарантий [23].

Несмотря на то, что на сегодняшний день было предложено множество моделей языков программирования, которые делают различные компромиссы и поддерживают различные возможности, насколько нам известно, не существует детального обзора этих моделей. Этот факт мотивировал нашу работу над данной статьей.

3. МЕТОДОЛОГИЯ

Основной задачей нашей работы было изучение компромиссов в моделях памяти языков программирования. Мы хотим ответить на следующий вопрос.

• Как гарантии корректного поведения программ, предоставляемые моделью памяти, ограничивают возможности по оптимизации этих программ?

Для сравнения моделей в свете данного вопроса мы использовали стандартные критерии, встречающиеся в литературе.

C.1 *Оптимальность схемы компиляции.* Язык с моделью памяти, поддерживающей оптимальные схемы компиляции, может быть эффективно реализован на современных процессорах. Напротив, использование неоптимальных схем компиляции приводит к замедлению при исполнении програм-

[4] https://kotlinlang.org/
[5] https://riscv.org/

мы, но в то же время может предотвратить появление слабых сценариев поведения, допустимых спецификацией данной архитектуры.

C.2 *Корректность трансформаций над программным кодом*. При оптимизации компилятор применяет различные трансформации к исходному коду. Чем больше транформаций допускается моделью памяти языка программирования, тем больше оптимизаций потенциально может применить компилятор к программам на данном языке.

C.3 Наличие различных *гарантией для рассуждения о поведении программ* позволяет упростить выполнение доказательств корректности многопоточных программ на данном языке.

Чтобы отобрать модели памяти для нашего исследования, мы провели следующую процедуру поиска. На *первом этапе* мы вручную отобрали 10 рецензированных статей предлагающих новые модели памяти [3–5, 7, 17, 19–22, 24, 25], которые были представлены на высоко рейтинговых конференциях в области языков программирования, таких как “Symposium on Principles of Programming Languages” (POPL), “Conference on Programming Language Design and Implementation” (PLDI) и др. Затем мы взяли список ключевых слов из этих статей. Мы исключили ключевые слова, которые были слишком общими или, наоборот, слишком специфичными. В результате мы получили три ключевые фразы: Relaxed Memory Models, Weak Memory Models, Weak Memory Consistency.

На *втором этапе* мы использовали эти фразы в качестве поисковых запросов в Google Scholar[6]. По каждому запросу мы взяли первые 1000 результатов.[7] В итоге мы получили список из 2493 статей. Мы удостоверились, что каждая из 10 изначально выбранных статей попала в эту выборку.

На *третьем этапе* мы удалили из выборки дубликаты и нерецензированные статьи. Также мы удалили технические отчеты, диссертации, публикации не на английском языке и короткие статьи (меньше 4 страниц). В результате осталось 1077 статей.

На *четвертом этапе* мы продолжили отбор статей, изучив заголовки и аннотации. Мы оставили только те статьи, которые напрямую относятся к теме моделей памяти языков программирования, и, напротив, исключили статьи, которые только используют существующие результаты о моделях или статьи, относящиеся к смежным темам, таким как модели памяти архитектур процессоров, гетерогенных и распределенных систем; семантика транзакций и персистентности; методы верификации программ в контексте слабых моделей памяти. В результате осталось 105 статей.

На заключительном *пятом этапе* мы изучили содержание оставшихся статей. В итоговом списке мы оставили только те статьи, в которых основным результатом была либо новая модель памяти ЯП, либо изучение/уточнение существующей модели памяти ЯП. В итоге осталось 40 статей.

4. КРИТЕРИИ СРАВНЕНИЯ МОДЕЛЕЙ ПАМЯТИ

В этом разделе мы более подробно рассмотрим выбранные критерии сравнения моделей памяти языков программирования – оптимальность схем компиляции **C.1**, корректность трансформаций **C.2** и предоставляемые гарантии **C.3**. Эти критерии непосредственно связаны с примитивами, предоставляемыми абстракцией разделяемой памяти. Таким образом в начале нам необходимо рассмотерть эти примитивы.

Программные примитивы. Модель памяти определяет семантику разделяемой памяти программы при наличии параллельно исполняемых потоков. Разделяемая память состоит из переменных, каждая из которых имеет уникальный адрес.[8] Потоки могут обращаться к этим переменным, выполняя операции чтения и записи.

В большинстве языков программирования различатся следующие виды переменных: *неатомарные* (*non-atomic*), также именуемые как *обычные* (*plain*), и *атомарные* (*atomic*). Первые не должны использоваться для обращений из различных параллельно исполняемых потоков программы. В зависимости от конкретного языка параллельные обращения к неатомарным переменным либо полностью запрещены (например, в Haskell [8, 26] и Rust [27]), либо имеют неопределенное поведение (например, в C/C++ [5, 28]), либо обладают очень слабой семантикой, не предоставляющей гарантий о порядке, в котором потоки могут наблюдать эти обращения (например, в Java [3]).

В свою очередь, атомарные переменные как раз предназначены для параллельных обращений. Некоторые модели памяти вводят несколько типов обращений к атомарным переменным, аннотируя их *режимом доступа* (*access mode*). Например, языки C/C++ и Java (начиная с версии 9 [4]) имеют следующие режима доступа: ослабленный режим (*relaxed* или *opaque* в терминологии Java), режимы захвата и освобождения (*acquire/release*), последовательно согласованный режим (*sequentially consistent* или *volatile* в Java). Эти режи-

[6] https://scholar.google.com/

[7] Все поисквые запросы были выполнены 24 сентября 2020 года.

[8] В этой статье мы будем использовать термины “адрес” и “локация” как взаимозаменяемые.

мы обозначаются как **rlx**, **acq**, **rel** и **sc** соответственно. Заметим, что режим **acq** может быть применен только к операциям чтения, а режим **rel** – только к операциям записи. Неатомарные обращения иногда рассматриваются как дополнительный режим **na**, однако одновременное использование атомарных и неатомарных обращений к одной и той же переменной приводит к неопределенному поведению в программах на языке C/C++.

Режимы обращения упорядочены по гарантиям, которые они предоставляют, как показано на следующей диаграмме.

На одном конце спектра находятся последовательно согласованные обращения. При правильном использовании они гарантируют семантику последовательной согласованности (детали этого обсуждаются в разделе § 4.3.1). На другом конце спектра находятся неатомарные обращения, которые либо не дают никаких гарантий, либо предоставляют минимальные гарантии. Ослабленные обращения также имеют слабую семантику, тем не менее обычно они предоставляют свойство *когерентности* (см. раздел § 4.3.2). Наконец, в середине спектра находятся обращения, имеющие режимы захвата/освобождения. Они используются для поддержки идиомы передачи сообщений [29]. Поток, которому необходимо отправить сообщение, должен выполнить операцию освобождающей записи, а другой поток, ожидающий это сообщение, должен выполнить операцию захватывающего чтения. Если операция чтения наблюдает, что операция освобождающей записи выполнена, то два потока синхронизируются.

Модель памяти также может предоставлять атомарные операции *чтения-модификации-записи* (*read-modify-write*). Они включают в себя операции сравнения и замены (*compare-and-swap*), атомарного обмена (*exchange*) и разные вариации атомарного инкремента, например, *fetch-and-add*, *fetch-and-sub* и т.д. Операция сравнения и замены принимает на вход адрес разделяемой переменной, а также ожидаемое и желаемое значение. Она выполняет чтение переменной и сравнивает полученное значение с ожидаемым. Если они равны, то выполняется замена значения переменной на желаемое, а прочитанное значение возвращается как результат, вне зависимости от успеха проверки. Заметим, что описанные выше действия выполняются атомарно, т.е. ни одна другая операция записи не может выполниться между операциями чтения и записи. Операция обмена (*exchange*) атомарно заменит значение переменной и возвратит ее прежнее значение. Операция атомарного инкремента (*fetch-and-add* и др.) инкрементирует значение переменной и возвращает ее значение до модификации.

Некоторые модели памяти рассматривают блокировки (*locks*) как самостоятельный примитив [3]. Еще одним примитивом являются барьеры (*fence*) [5], которые соответствуют инструкциям барьеров памяти, выполняемых процессорами (см. раздел § 4.1).

Наконец, модель может рассматривать разделяемую память не как множество независимых типизированных переменных, а как последовательность байт, и допускать так называемые *смешанные* (*mixed-size*) параллельные обращения [30]. Например, в подобной модели операция чтения восьми байт может выполниться как два параллельных чтения по четыре байта.

4.1. Схема компиляции

Будем понимать под *схемой компиляции* отображение примитивов языка программирования в инструкции конкретного семейства процессоров. Мы будем рассматривать примитивы, представленные в разделе § 4. Процессоры обычно предоставляют инструкции для выполнения обычных операций чтения и записи[9], операции чтения-модификации-записи, а также различные типы барьеров памяти.

Схема компиляции должна быть корректной, т.е. обеспечивать, что множество сценариев поведения, допустимых моделью памяти процессора для скомпилированной программы, будет являться подмножеством сценариев поведения программы, допустимых моделью языка программирования.

Рассмотрим пример. Программа SB, представленная ниже, является фрагментом алгоритма Деккера, обсуждавшегося в § 1.

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB)}$$

Предположим, что язык программирования предоставляет модель последовательной согласованности, а программа должна быть скомпилирована для x86. Если компилировать операции чтения и записи обычным образом x86 [10], тогда, следуя спецификации модели памяти x86, допустимым будет следующий результат работы программы (он также будет наблюдаться и на практике):

[9] Некоторые архитектуры предоставляют дополнительные инструкции чтения и записи с определенным режим доступа, например, **Armv8** lda – захватывающее чтение (load acquire), stl – освобождающая запись (store release).

[10] В архитектуре x86 инструкция MOV используется для чтения и записи в память.

$[r_1 = 0, r_2 = 0]$. Данный результат может появиться вследствие *буферизации операций записи* – операция записи $x := 1$ может быть исполнена после выполнения всех остальных инструкций программы.

Отметим, что результат $[r_1 = 0, r_2 = 0]$ не является результатом последовательно согласованного сценария, и, следовательно, рассмотренная схема компиляции не является корректной. Как было продемонстрировано в разделе § 1, некорректность схемы компиляции может иметь негативные последствия и нарушать корректность программы.

Корректная схема компиляции для модели последовательной согласованности под архитектуру x86 может компилировать операцию записи как обычную инструкцию записи, за которой следует инструкция mfence [5, 9], как продемонстрировано ниже:

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ \texttt{mfence} & \texttt{mfence} \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB+MFENCE)}$$

Инструкция `mfence` является специальным барьером памяти в системе команд процессоров x86, который выполняет сброс буфера записей в основную память. Для программы SB+MFENCE результат $[r_1 = 0, r_2 = 0]$ запрещен моделью памяти x86.

Несмотря на то, что модифицированная схема компиляции является корректной, она не *оптимальна* [31] в том смысле, что она использует барьеры памяти, которые обычно являются причиной замедления программы на 10–30% на процессорах семейства x86 [32, 33]. К сожалению, современные процессоры не могут иметь одновременно *корректную и оптимальную* схему компиляции в рамках модели последовательной согласованности. Этот факт затрудняет использование модели **SC** для высокопроизводительных языков программирования и служит одним из стимулов к ослаблению моделей памяти.

В рамках этой статьи при обсуждении схем компиляции мы будем рассматривать процессоры семейств **x86**, **Armv7**, **Armv8** и **POWER** по следующим причинам. Во-первых, эти архитектуры наиболее распространены на сегодняшний день. Во-вторых, модели памяти для этих процессоров всесторонне изучены исследовательским сообществом, что привело к созданию строгих формальных спецификаций [9, 11, 14, 15].

4.2. Трансформации кода

Следующим критерием является **C.2** – корректность трансформаций, то есть правил переписывания исходного кода, применяемых в компиляторных оптимизациях.

Корректная трансформация должна сохранять семантику программы. В нашем контексте, как и в случае корректности схемы компиляции, это означает, что множество допустимых сценариев поведения программы после применения трансформации должно быть подмножеством допустимых сценариев поведения оригинальной программы.

Возвращаясь к примеру SB, снова рассмотрим модель последовательной согласованности и возьмем трансформацию, которая переставляет местами операции записи и чтения в левом потоке, предполагая, что они оперируют различными локациями в памяти:

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \quad \rightsquigarrow \quad \begin{array}{l|\!|l} r_1 := y & y := 1 \\ x := 1 & r_2 := x \end{array}$$

Для преобразованной версии программы (справа), результат $[r_1 = 0, r_2 = 0]$ является последовательно согласованным. Тем не менее, для оригинальной версии программы (слева) это неверно. Следовательно, вышеупомянутая трансформация является некорректной для модели SC.

В следующих разделах мы рассмотрим некоторые трансформации, обсуждаемые в различных исследованиях. Заметим, что этот список далеко не полон и не включает многие трансформации, выполняемые существующими оптимизирующими компиляторами [34]. Например, он не включает трансформации над циклами, так как в теории моделей памяти еще недостаточно проработаны темы гарантий прогресса (*liveness properties*) [35], которые необходимы для формального изучения этих трансформаций.

Трансформации, которые мы рассматриваем, разделены на два подкласса: *локальные* и *глобальные*. Локальные трансформации модифицируют небольшой участок кода в пределах одного потока; глобальные трансформации задействуют всю программу или ее большую часть, захватывая несколько потоков.

4.2.1. Локальные трансформации

Переупорядочивание независимых инструкций (Reordering of Independent Instructions). Эта трансформация переставляет местами две смежные инструкции, выполняющие обращение к различным адресам памяти. Выделяют четыре типа переупорядочивания: запись/чтение, запись/запись, чтение/чтение и чтение/запись.

$x := v; \quad r := y \rightsquigarrow r := y; \quad x := v$ store/load, SL

$x := v; \quad y := u \rightsquigarrow y := u; \quad x := v$ store/store, SS

$r := x; \quad s := y \rightsquigarrow s := y; \quad r := x$ load/load, LL

$r := x; \quad s := v \rightsquigarrow y := v; \quad r := x$ load/store, LS

Элиминация избыточного обращения (Elimination of Redundant Access). В паре двух смежных обраще-

ний к памяти одно из них может быть удалено, если его эффект покрывается другим. Например, две операции записи в одну и ту же переменную одного и того же значения могут быть заменены на одну операцию записи. Аналогично переупорядочиванию инструкций, выделяют четыре класса трансформации этого вида.

$x := v; \quad r := x \rightsquigarrow x := v; \quad r := v$ store/load, SL

$r := x; \quad s := x \rightsquigarrow r := x; \quad s := r$ store/store, SS

$r := x; \quad x := r \rightsquigarrow r := x$ load/store, LS

$x := v; \quad x := u \rightsquigarrow x := u$ store/store, SS

Элиминация нерелевантной операции чтения (Irrelevant Load Elimination, ILE). Эта трансформация удаляет инструкцию чтения, если ее результат не используется в программе.

$$r := x \rightsquigarrow \epsilon \mid r \text{ is never used}$$

Введение спекулятивной операции чтения (Speculative Load Introduction, SLI). Эта трансформация является обратной к предыдущей и вставляет инструкцию чтения в произвольное место программы.

$$\epsilon \rightsquigarrow r := x \mid r \text{ is never used}$$

В комбинации с элиминацей типа чтение/чтение эта трансформация может быть использована для того, чтобы вынести чтение из ветки условного оператора:

$$\texttt{if}\ (e)\ \texttt{then}\ \{r := x\} \rightsquigarrow s := x;\ \texttt{if}\ (e)\ \texttt{then}\ \{r := s\} \mid s\ \texttt{is never used}$$

Переупорядочивание с увеличением синхронизации (Roach Motel Reordering, RM). Этот класс трансформаций позволяет вносить дополнительные инструкции в блоки синхронизации. Например, инструкция записи может быть перенесена в критическую секцию, то есть переставлена за следующую операцию захвата блокировки. Интуитивно, такие перестановки могут только увеличить степень синхронизации в программе, то есть преобразованная программа должна обладать меньшим недетерминизмом и иметь меньшее количество допустимых сценариев поведения.

Неатомарные обращения могут быть внесены в критическую секцию без дополнительных предусловий. Кроме того, инструкция записи может быть перемещена после операции захвата блокировки, а инструкция чтения может быть перемещена до операции освобождения блокировки. Похожие правила применяются к перестановке операций вокруг захватывающих (*acquire*) и освобождающих (*release*) обращений к памяти.

$r :=_{\text{na}} x; \quad \text{lock}(l) \rightsquigarrow \text{lock}(l); \quad r :=_{\text{na}} x$

$x :=_{\text{o}} v; \quad \text{lock}(l) \rightsquigarrow \text{lock}(l); \quad r :=_{\text{o}} v$

$\text{unlock}(l); \quad x :=_{\text{na}} v \rightsquigarrow x :=_{\text{na}} v; \quad \text{unlock}(l)$

$\text{unlock}(l); \quad x :=_{\text{o}} x \rightsquigarrow r :=_{\text{o}} x; \quad \text{unlock}(l)$

Усиление обращений к памяти (Strengthening, S). Данная трансформация, подобно предыдущей, увеличивает степень синхронизации в программе путем усиления режима обращения к памяти. Например, неатомарное обращение к переменной может быть заменено на последовательно согласованное:

$$r :=_{o} x \rightsquigarrow r :=_{o'} x \mid o \sqsubset o'$$

$$r :=_{o} v \rightsquigarrow x :=_{o'} v \mid o \sqsubset o'$$

Трансформации, сохраняющие трассы (Trace Preserving Transformations, TP). Этот широкий класс трансформаций, который включает трансформации, не меняющие множество трасс потока [36]. Трассой называется последовательность видимых побочных эффектов, возникающих во время исполнения кода потока, при этом операции чтения и записи в разделяемую память тоже считаются эффектами. Классическим примером подобной трансформации является *распространение констант* [34, 37]. Ниже приведен пример применения данной трансформации.

$$x := 0 + v \rightsquigarrow x := v$$

Удаление общих подвыражений (Common Subexpression Elimination, CSE). CSE является еще одной классической трансформацией [34], которая выполняет поиск и удаление идентичных подвыражений. Вот пример выполнения этой трансформации:

$$r_1 := x + y; \quad r_2 := x + y \rightsquigarrow r_1 := x + y; \quad r_2 := r_1$$

4.2.2. Глобальные трансформации

Продвижение регистров (Register Promotion, RP). Если компилятор может определить, что обращения к разделяемой переменной происходят только из одного потока, тогда он может заменить все обращения к этой переменной на обращения к регистру.

$$x := v; \quad r := x \rightsquigarrow s := v; \quad r := s \mid \texttt{x}\ \text{is not accessed from other threads} \mid \texttt{s}\ \text{is a fresh register}$$

Слияние потоков (Thread Inlining, TI). Эта трансформация объединяет два потока в один. Оказывается, что эта на первый взгляд простая и очевидная трансформация не является корректной в некоторых моделях памяти.

$$P \parallel Q \rightsquigarrow P\ ;\ Q$$

Трансформации, основанные на анализе диапазона значений (Value Range Based Transformations, VR). Трансформации этого класса могут быть применены в случае, если программа удовлетворяет некоторому инварианту, выведенному с помощью

глобального анализа диапазона возможных значений переменных. Например, в программе ниже условный оператор может быть удален, так как статический анализ может вывести инвариант $x \geq 0$.

$$\begin{array}{l} r_1 := x \\ \text{if } (r_1 \geq 0) \text{ then} \\ y := 1 \end{array} \left\| \begin{array}{l} \\ r_2 := x \\ y := r_2 \end{array} \right. \quad \rightsquigarrow \quad \begin{array}{l} r_1 := x \\ y := 1 \end{array} \left\| \begin{array}{l} \\ r_2 := x \\ y := r_2 \end{array} \right.$$

4.3. Гарантии

Далее мы обсуждаем третий критерий **C.3** – гарантии о поведении программ, предоставляемые моделью памяти.

4.3.1. DRF-свойство

При рассуждении о многопоточных программах большинство программистов подразумевают модель последовательной согласованности. Действительно, было бы неправильно ожидать от программистов знания всех деталей слабых моделей, так как это только усложнило бы и без того непростую задачу проектирования и разработки многопоточных программ. Для того чтобы решить эту проблему, было предложено свойство *свободы от гонок* (*data-race freedom*, **DRF**) [3]. Это свойство гарантирует, что при наличии достаточной степени синхронизации программа будет иметь только последовательно согласованные сценарии поведения в слабой модели памяти. Другими словами, если слабая модель памяти обладает **DRF**-свойством, то при условии правильного использования примитивов синхронизации программист может не задумываться о слабых сценариях поведения и подразумевать модель последовательной согласованности.

Рассмотрим пример. Вернемся к программе SB из § 4.1. Как было продемонстрировано ранее, в слабой модели эта программа может допускать результат $[r_1 = 0, r_2 = 0]$. Тем не менее, семантика последовательной согласованности может быть восстановлена, например, при помощи блокировок, как показано в примере ниже:

$$\begin{array}{c} \texttt{lock}(l) \\ x := 1 \\ r_1 := y \\ \texttt{unlock}(l) \end{array} \left\| \begin{array}{c} \texttt{lock}(l) \\ y := 1 \\ r_2 := x \\ \texttt{unlock}(l) \end{array} \right. \qquad \text{(SB+LOCK)}$$

Совместимая с **DRF**-свойством слабая модель памяти должна гарантировать, что для программы выше допустимы только последовательно согласованные сценарии поведения с результатами $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$ или $[r_1 = 1, r_2 = 1]$.

Если модель предоставляет последовательно согласованный режим доступа, тогда программист также может аннотировать все обращения к переменным как последовательно согласованные и таким образом восстановить семантику SC:

$$\begin{array}{l} x :=_{\text{sc}} 1 \\ r_1 :=_{\text{sc}} y \end{array} \left\| \begin{array}{l} y :=_{\text{sc}} 1 \\ r_2 :=_{\text{sc}} x \end{array} \right. \qquad \text{(SB+SC)}$$

Более формально, **DRF**-свойство для слабой модели M утверждает, что если программа не содержит гонок в модели последовательной согласованности, тогда модель M допускает только последовательно согласованные сценарии поведения для этой программы.

Итак, свойство **DRF** позволяет свести рассуждения о поведении программы в слабой модели к рассуждениям в модели последовательной согласованности. Достаточно лишь показать, что программа не имеет гонок в модели SC, чтобы иметь факт, что она будет иметь только **SC** сценарии поведения в слабой модели.

Свойство **DRF** в приведенной выше формулировке иногда также называется *внешней свободой от гонок* (**eDRF**), чтобы отличать его от *внутренней свободы от гонок* (**iDRF**). **iDRF**-свойство гарантирует для программы семантику **SC** в слабой модели M только в случае, если программа не имеет гонок в самой модели M. Это свойство предоставляет более слабую гарантию по сравнению с внешней свободой от гонок. Оно не позволяет полностью избежать рассуждений в терминах слабой модели, так как сначала необходимо показать, что программа не имеет гонок именно в слабой модели. Как будет продемонстрировано далее (*см.* § 6.3), внутренняя свобода от гонок является компромиссом для определенного класса моделей, которые не могут предоставить внешнюю свободу от гонок.

4.3.2. Когерентность (COH)

Как было показано раньше, современные процессоры не гарантируют выполнимость модели последовательной согласованности. Тем не менее, обычно они предоставляют более слабую гарантию *последовательной согласованности по каждой локации в памяти*, именуемую также *когерентностью* [12]. Соответственно, модели памяти для языков программирования также зачастую предоставляют эту гарантию.

Когерентность гарантирует, что все операции записи по определенному адресу памяти будут полностью упорядочены, и что получающийся в результате *порядок когерентности* (*coherence order*) отражает порядок, в котором эффекты от операций записи, выполненных некоторым потоком, отражаются в основной памяти, и их результаты становятся видимыми для других потоков. В частности, из свойства когерентности следует, что программа, состоящая из обращений только к од-

ной локации в памяти, должна обладать семантикой последовательной согласованности. Например, рассмотрим следующую программу:

$$\begin{matrix} x := 1 \\ r_1 := x \end{matrix} \Bigg\| \begin{matrix} x := 2 \\ r_2 := x \end{matrix} \qquad \text{(COH)}$$

Наличие свойства когерентности предписывает модели памяти допускать для этой программы только последовательно согласованные сценарии поведения с результатами $[r_1 = 1, r_2 = 2], [r_1 = 1, r_2 = 1]$ или $[r_1 = 2, r_2 = 2]$. Для модели, не удовлетворяющей свойству когерентности, допустимым также является результат $[r_1 = 2, r_2 = 1]$. Например, модель памяти Java допускает подобный сценарий поведения [3].

4.3.3. Неопределенное поведение (no-UB)

Как мы уже кратко упоминали, некоторые модели памяти, например, C/C++, рассматривают программы с гонками на неатомарных обращениях как имеющие *неопределенное поведение* [28]. Другими словами, для таких программ любой сценарий поведения считается допустимым. Это свойство также иногда называется *возгорающейся семантикой* (*catch-fire semantics*).

Практическая польза такого подхода заключается в том, что он допускает оптимальную схему компиляции для неатомарных обращений и позволяет применять к ним любые оптимизации, корректные для последовательных программ. Дело в том, что эффекты от оптимизаций, осуществляемых процессором или компилятором, могут наблюдаться только при обращении к переменным из параллельных потоков. Если таким обращениям предписывается неопределенное поведение и на них не распространяются никакие гарантии, то тогда эффекты этих оптимизаций становятся неразличимы с точки зрения семантики программы.

4.3.4. Спекулятивное исполнение (In-Order) и значения из воздуха (no-OOTA)

Чтобы представить последние два свойства, а именно, наличие спекулятивного исполнения и значений из воздуха, мы рассмотрим еще один пример:

$$\begin{matrix} r_1 := x \\ y := 1 \end{matrix} \Bigg\| \begin{matrix} r_2 := y \\ x := r_2 \end{matrix} \qquad \text{(LB)}$$

Предположим, что модель памяти допускает результат $[r_1 = 1, r_2 = 1]$ для этой программы. Например, модели семейств процессоров **Armv7**, **Armv8** и **POWER** допускают сценарий поведения, ведущий к такому результату, и этот сценарий может наблюдаться на некоторых процессорах семейства **Armv7** [38].

Результат $[r_1 = 1, r_2 = 1]$ не может быть получен путем исполнения инструкции согласно их порядку внутри потоков. Чтобы получить подобное поведение, модель памяти должна использовать некоторую форму *спекулятивного исполнения* [16, 39]. Это означает, что операция чтения $r_1 := x$ должна быть буферизована, а операция записи $y := 1$ должна выполниться вне очереди (отсюда и название программы выше – буферизация операции чтения *load buffering*).

Однако неограниченные спекуляции могут привести к нежелательным последствиям. Операция записи, исполненная вне очереди, может обернуться самоисполняющимся пророчеством (self-fulfilling prophecy) [16]. Рассмотрим следующий вариант программы с буферизацией операции чтения:

$$\begin{matrix} r_1 := x \\ y := r_1 \end{matrix} \Bigg\| \begin{matrix} r_2 := y \\ x := r_2 \end{matrix} \qquad \text{(LB+data)}$$

Здесь гипотетическая абстрактная машина может спекулятивно исполнить операцию записи в переменную `y` значения `1` в левом потоке, затем прочитать это значение в правом потоке, записать его в переменную `x` и прочитать обратно из первого потока, таким образом сформировав парадоксальный цикл причинно-следственных связей. Значение `1` в примере выше появляется *из воздуха* (*out of thin-air*) и приводит к неожиданному результату $[r_1 = 1, r_2 = 1]$.

Как будет показано в § 6, возможность спекулятивного исполнение необходима для того, чтобы поддержать в модели памяти некоторый класс трансформаций программ. Тем не менее, спекулятивное исполнения необходимо ограничить должным образом, чтобы избежать появления значений из воздуха. В § 6.4–§ 6.6 мы рассмотрим то, как эта проблема решается в различных моделях памяти.

5. СРАВНЕНИЕ

Мы провели сравнение моделей памяти, отобранных с помощью процедуры описанной в § 3, по критериям, представленным в § 4. В этом разделе представлены результаты этого сравнения.

Сравнение моделей памяти было осложнено тем фактом, что рассмотренные статьи, зачастую, используют различную терминологию, предоставляют неполную информацию о моделях памяти, а в некоторых случаях статьи противоречат друг другу. Мы подошли к решению этих трудностей следующим образом. Во-первых, для обозначения свойств моделей памяти мы использовали единую терминологию, описанную нами в § 4. Во-вторых, мы дополняли информацию о каждой модели из разных источников. Если после этого

Таблица 1. Классы моделей памяти и их свойства

Class	# Models	Compilation				Transformations																Reasoning				
						Local														Global							
						Reordering				Elimination																
		x86	POWER	Armv7	Armv8	SL	SS	LL	LS	SL	SS	LL	LS	ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
Sequential Consistency	2	–	–	–	–	–	–	–	–	+	+	+	+	+	+	+	+	+	–	+	+	–	+	+	+	+	+
Total/Partial Store Order	2	+	–	–	–	+	–	–	+	+	+	+	+	–	+	+	–	–	+	–	–	–	+	+	+	+	+
Program Order Preserving	3	+	–	–	–	+	+	–	+	+	+	+	+	–	–	–	+	+	+	±	+	–	+	+	+	+	+
Syntax. Dep. Preserving	2	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	–	–	–	+	+	+	–	+
Semantic Dep. Preserving	7	+	+	+	+	+	+	+	+	+	+	+	+	±	+	+	+	+	+	±	–	+	+	+	+	–	+
Out of Thin-Air	5	+	+	+	+	+	+	+	+	+	+	+	+	–	–	–	+	+	+	–	+	–	–	+	+	–	–

наличие или отсутствие определенного свойства у модели все еще оставалось неясным, мы явно помечали этот факт.

Мы идентифицировали шесть классов моделей памяти: последовательно согласованные (sequentially consistent); модели с линейным/частичным порядком на операциях записи (total or partial store order); модели, сохраняющие программный порядок (program order preserving models); модели, сохраняющие синтаксические зависимости (syntactic dependency preserving); модели, сохраняющие семантические зависимости (semantic dependency preserving); модели допускающие значения из воздуха (out of thin-air). Модели из одного класса имеют схожие схемы компиляции, множество корректных трансформаций и предоставляемые гарантии. Сначала мы представляем результат сравнения различных классов (табл. 1), а затем – сравнение отдельных моделей (табл. 2).

В табл. 1 и 2 мы упорядочиваем модели по степени их ослабленности. Более строгие модели расположены в верхних строках, а более слабые – в нижних.

Колонки обоих таблиц соответствуют свойствам моделей памяти. Для краткости мы выбрали бинарную классификацию свойств, т.е. считаем, что модель либо удовлетворяет данному свойству, либо нет. Мы также разделили все свойства на несколько групп.

Первая группа свойств посвящена оптимальности схем компиляции для различных семейств процессоров. Мы классифицируем схему компиляции как оптимальную или неоптимальную следующим образом. Мы выбираем наиболее слабый режим доступа, поддерживаемый моделью памяти, и рассматриваем схему компиляции для обращений к памяти, аннотированных данным режимом. Для моделей, которые трактуют гонки на неатомарных переменных как неопределенное поведение, мы рассматриваем наиболее слабый режим доступа к атомарным переменным. Дело в том, что семантика с неопределенным поведением для программ с гонками тривиальным образом допускает оптимальную схему компиляции (*см.* § 4.3.3). Мы считаем схему компиляции *оптимальной*, если обращения к памяти, аннотированные выбранным режимом доступа, могут быть скомпилированы в обычные инструкции чтения и записи целевой архитектуры (т.е. без использования барьеров памяти или какого-либо другого дополнительного кода).

Вторая группа свойств посвящена корректности различных трансформаций. Здесь классификация также бинарная: конкретная трансформация либо является корректной в данной модели, либо нет. Мы вновь не рассматриваем все возможные комбинации трансформаций и режимов доступа. Вместо этого мы останавливаемся только на наиболее слабом режиме доступа с полностью определенной семантикой (т.е. без неопределенного поведения). Также мы разделяем трансформации на локальные и глобальные.

Третья группа свойств соответствует предоставляемым гарантиям о поведении программ. В частности, для каждой модели мы указываем, предоставляет ли она свойство внешней свободы от гонок **eDRF** (*см.* § 4.3.1), свойство когерентности (*см.* § 4.3.2), трактует ли она гонки на неатомарных переменных как неопределенное поведение (см. § 4.3.3), используется ли последовательное исполнение инструкций (*in-order execution*) или применяет спе-

Таблица 2. Модели памяти и их свойства

Class	Model	Compilation				Transformations																	Reasoning				
						Local														Global							
						Reordering				Elimination																	
		x86	POWER	Armv7	Armv8	SL	SS	LL	LS	SL	SS	LL	LS	ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
Sequential Consistency	SC [8, 32, 33, 40, 41]	–	–	–	–	+	–	–	–	+	+	+	–	+	+	+	+	+	–	+	+		+	+	+	+	+
	DRFx [42]	–	–	–	–	+	+	–	–	+	+	+	+	+	–	+	+	+	–	+	+		+	+	–	+	+
Total/Partial Store Order	BMM [43]	+	–	–	–	+	+	+	–	+	+	+	–	+	+	–		+	–		–		+	+	+	+	+
	RMMOA [44]	+	–	–	–	+	+	+	–														+		+	+	+
Program Order Preserving	RC11 [21, 45–47]	+	–	–	–	+	+	+	–	+	+	+	–		–	+	+	+	–	+	+		+	+	–	+	+
	OCMM [22]	+	–	–	–	+	+	+	–	+	+	+	–					+	+				+	+	–	+	+
	JAM [4]	+	–	–	–												+		–				+	+	+	+	+
Syntax. Dep. Preserving	LKMM [48]	+	+	+	+	+	+	+	+									–	–					+	+	–	+
	OHMM [49]					+	+	+	+	+	+	+	+	+	+	+		–					+		+	–	+
Semantic Dep. Preserving	JMM [3, 36, 50]	+	–	+	+	+	+	+	+	+	+	–	–	+	–	–		+	–		–		+	+	–	–	+
	PRM [17, 18]	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	+	+	+	–	–	+
	WMO [6, 19]	+	+	+	+	+	+	+	+	+	+	+	–		+	–	+		–		–		+	–	+	–	+
	CSRA [25]	+	+	–	–	+	+	+	+	+	+	+	–	+	–	+	–		–		–	+		+	+	–	+
	WJES [24]		–	–	–			–				–	–										+	–	+	–	+
	MRD [20]	+	+	+	+																		+	+	+	–	+
	GOS [51]																						+		+	–	+
Out of Thin-Air	C11 [5, 29, 30, 52–57]	+	+	+	+	+	+	+	+	+	+	+	–		–	+	+	+	–		+		–	+	–	–	–
	JSMM [7]	+	+	+	+																		–	–	+	–	–
	RMC [58]	+	+	+	+																		–	+	+	–	–
	RAO [59]																							–	+	–	–
	TSC [39]																						–		+	–	–

кулятивное исполнение (*speculative execution*), а также допускает ли данная модель значения из воздуха (см. § 4.3.4).

В табл. 1 каждая строка соответствует целому классу моделей. Клетка помечается символом “+” если большинство моделей данного класса обладают соответствующим свойством. Если рассматриваемым свойством обладает только некоторое количество моделей данного класса, не составляющих большинство, то клетка помечается символом “±”. Наконец, если ни одна из моделей данного класса не обладает данным свойством, тогда клетка помечается символом “–”. При подсчете большинства мы опускаем те модели данного класса, про которые неизвестно, обладают они данным свойством или нет. Также, если какое-то свойство вообще не изучалось в контексте определенного класса моделей, мы также помечаем клетку символом “–”. Таким образом, в табл. 1 символы “+” и “±” обозначают положительную информацию, а символ “–” обозначает как негативную информацию, так и отсутствие информации.

В табл. 2 каждая строка соответствует конкретной модели памяти, обозначаемой аббревиатурой, а каждая клетка описывает наличие или отсутствие определенного свойства этой модели. Мы помечаем клетку символом “+”, если рассматриваемая модель обладает данным свойством и символом “–” в противном случае. Если нам не удалось получить информацию о данном свойстве рассматриваемой модели, то соответствующая клетка заливается серым цветом ■.

Помимо табл. 1 и 2, которые описывают свойства моделей, мы также представляем табл. 3, содержащую список примитивов, поддерживаемых каждой моделью. Каждая строка данной таблицы соответствует отдельной модели памяти. Колонки соответствуют поддерживаемым примитивам. В частности, мы указываем, какие режимы обращений к переменным модель поддерживает: неатомарные (NA), ослабленные (RLX), захвата/освобождения (RA), последовательно согласованные (SC); какие типы барьеров поддержаны: захвата/освобождения (RA) и последовательно согласованные (F-SC); поддерживаются ли атомарные операции чтения-модификации-записи (RMW), поддерживаются ли явно операции блокировки (LK), и поддерживаются ли смешанные обращения (MIX).

6. АНАЛИЗ

В этом разделе мы обсуждаем результаты сравнения, представленные в предыдущем разделе. На основе данных из табл. 1 и 2 мы выводим взаимосвязи между оптимальностью схемы компиляции, корректностью трансформаций и предоставляемыми гарантиями. В частности, мы демонстрируем, как поддержка некоторых гарантий конфликтует с некоторыми трансформациями и как она влияет на оптимальность схемы компиляции. Мы начинаем с рассмотрения класса последовательно согласованных моделей § 6.1, затем переходим к моделям с линейным/частичным порядком на операциях записи § 6.2. После этого мы рассматриваем класс наиболее слабых моделей, допускающих значения из воздуха § 6.3, а далее переходим к обсуждению различных подходов к решению проблемы значений из воздуха и рассматриваем модели, сохраняющие программный порядок § 6.4, синтаксические § 6.5 и семантические § 6.6 зависимости. В § 6.7 мы отдельно обсуждаем некоторые конкретные свойства моделей, в частности, когерентность и возгорающуюся семантику. Факт наличия или отсутствия этих свойств ортогонален разбиению на вышеупомянутые классы. Тем не менее, наличие этих свойств у модели также влияет на корректность определенных трансформаций.

6.1. Модель последовательной согласованности

Данная модель является наиболее интуитивной моделью многопоточности. В рамках этой модели состояние памяти может быть представлено как отображение из адресов переменных в хранящиеся значения. Тогда каждый допустимый сценарий поведения программы может быть получен в результате поочередного последовательного исполнения инструкций потоков.

Многие распространенные трансформации оказываются некорректными в модели **SC**, включая все типы переупорядочивания операций, а также удаление общих подвыражений [32, 32]. Тот факт что переупорядочивание инструкций запрещено делает эту модель очень дорогостоящей при реализации на современных процессорах, так как даже относительно строгая модель памяти процессоров **x86** допускает переупорядочивание типа запись/чтение. Таким образом, чтобы гарантировать последовательную согласованность, компилятор вынужден вставлять в код тяжеловесные барьеры памяти между инструкциями записи и чтения, что приводит к неоптимальной схеме компиляции.

Однако в терминах предоставляемых программисту гарантий модель **SC** является весьма привлекательной. В частности, тривиальным образом гарантируются свойства **eDRF** и когерентности, так как модель присваивает программе только последовательно согласованные сценарии поведения.

Концептуальная простота и привлекательность модели **SC** вдохновила многих исследователей на попытки адаптации этой модели и смягчения накладываемых штрафов на время испол-

Таблица 3. Поддержка примитивов моделями памяти

Class	Model	Features								
		NA	RLX	RA	SC	F-RA	F-SC	RMW	LK	MIX
Sequential Consistency	EtE-SC [32, 40]	–	–	–	+	–	–	–	–	–
	VbD [33, 41]	+	–	–	+	–	–	+	+	–
	SC-Hs [8]	+	–	–	+	–	–	+		–
	DRFx [42]	+	–	–	+	–	–	–	–	–
Total/Partial Store Order	BMM [43]	+	–	–	+	–	–	–	–	–
	RMMOA [44]	+	–	–	–	–	–	–	+	–
Program Order Preserving	RC11 [21]	+	+	+	+	+	+	+	–	–
	ORC11 [47]	+	+	+	–	–	–	+	–	–
	RAR [46]	–	+	+	–	–	–	+	–	–
	CRC [45]	+	–	+	–	–	+	+	–	–
	OCMM [22]	+	–	–	+	–	–	+	–	–
	JAM [4]	+	+	+	+	+	+	+	–	–
Syntax. Dep. Preserving	LKMM [48]	–	+	+	–	+	+	+	–	–
	OHMM [49]	+	–	–	+	–	–	–	+	–
Semantic Dep. Preserving	JMM [3]	+	–	–	+	–	–	+	+	–
	PRM [17]	+	+	+	–	+	+	+	+	–
	WMO [19]	+	+	+	+	+	+	+	–	–
	CSRA [25]	+	+	+	–	–	–	+	+	–
	WJES [24]	–	+	–	–	–	–	+	+	–
	MRD [20]	–	+	–	–	–	–	–	+	–
	GOS [51]	+	–	–	–	–	–	–	+	–
Out of Thin-Air	C11 [5]	+	+	+	+	+	+	+	+	+
	JSMM [7]	+	–	–	+	–	–	+	+	+
	RMC [58]	–	+	+	+	+	+	+	–	–
	RAO [59]	+	–	–	+	–	–	–	–	–
	TSC [39]	+	–	–	+	–	–	–	+	–

нения программы. Общей идеей данных работ была попытка отделения локальных (доступных только одному потоку) и разделяемых переменных. Обращения к локальным переменным могут быть скомпилированы без добавления барьеров памяти, также к ним применим широкий спектр оптимизаций корректных для случая однопоточных программ. Чтобы безопасным образом классифицировать локальные и разделяемые переменные исследователи использовали системы типов [8], статический [40] или динамический анализ [41], поддержку на аппаратном уровне [40, 42], или различные комбинации вышеупомянутых методов.

Несмотря на эти усилия, модель **SC** все равно имеет существенные накладные расходы. Например, при компиляции на процессоры семейства **Armv8** замедление времени работы программ может достигать 70% [41]. Более того, хотя вышеупомянутые техники обычно уменьшают накладные расходы на локальные обращения (которые, зачастую, чаще встречаются в программах), они оказывают меньшее влияние на специфичные приложения, которые активно используют многопоточность, например, такие как неблокирующие структуры данных. Наконец, требуется значительное количество усилий и технической работы, чтобы модифицировать современные компиляторы для поддержки модели **SC** [32, 41].

6.2. Линейный/частичный порядок на операциях записи

Данный класс моделей был создан на основе моделей с *линейным порядком на операциях записей* (*total store order*, **TSO**) и *частичным порядком на операциях записей* (*partial store order*, **PSO**) [60]. Модели **TSO** и **PSO** являются моделями семейств процессоров **x86** [9] и **SPARC** [60] соответственно. В этих моделях потоки оснащены *буферами запи-*

сей. Все операции записи вначале попадают в эти буферы, а потом переносятся в основную память.

Для моделей этого класса схема компиляции для архитектуры **x86** является оптимальной, так как **x86** предоставляет модель **TSO**. Однако при компиляции под архитектуры с более слабой моделью памяти, например **POWER**, необходимо использовать практически такое же количество барьеров, как и при компиляции из модели SC [61].

Модели этого класса допускают большее количество трансформаций кода, чем **SC**. Использование буферов для операций записи позволяет выполнять переупорядочивание типа запись/чтение в случае **TSO** и запись/запись – в случае **PSO**.

Хотя модели **TSO** и **PSO** слабее, чем модель **SC**, тем не менее они все еще предоставляют довольно сильные гарантии, в частности, свойство **eDRF** и когерентность.

Таким образом, модели этого класса не имеют значительных преимуществ перед моделью **SC**, и при этом влекут соизмеримые накладные расходы при компиляции для архитектуры с более слабой моделью памяти, чем у **x86**. Следовательно, выбор этих моделей в качестве моделей для языка программирования оправдан только если предполагается поддержка компиляции исключительно для процессоров архитектуры **x86**.

6.3. Значения из воздуха

Далее мы переместимся на другой конец спектра моделей памяти и рассмотрим класс, в который входят наиболее слабые модели. Эти модели предоставляют оптимальные схемы компиляции и допускают, практически, любые разумные трансформации программ, но достигают этого ценой введения значений из воздуха (4).

Снова рассмотрим пример программы буферизации операции чтения:

$$\begin{array}{ll|l|ll} r_1 := x & r_2 := y & & y := 1 & r_2 := y \\ y := 1 & x := r_2 & \rightsquigarrow & r_1 := x & x := r_2 \end{array}$$

(LB) (LBtr)

Версия программы справа LBtr может быть получена из программы слева LB путем применения переупорядочивания инструкций типа чтение/запись. Результат $[r_1 = 1, r_2 = 1]$ является допустимым для программы LBtr. Тогда модель памяти, в которой переупорядочивание типа чтение/запись является корректной трансформацией, также должна допускать этот результат для программы LB. Как было продемонстрировано в § 4.3.4, для того, чтобы получить такой результат, необходимо применить спекулятивное исполнение.

Мы также обсуждали, что неограниченное спекулятивное исполнение может привести к появлению так называемых значений из воздуха, которые нарушают фундаментальные гарантии о поведении программ [16, 55], в частности, гарантии типобезопасности (type-safety) и композициональности. Также не выполняется и гарантия внешней свободы от гонок (**eDRF**). Чтобы убедиться в этом, рассмотрим еще один пример:

$$\begin{array}{l|l} r_1 := x & r_2 := y \\ \text{if } (r_1)\ \{ & \text{if } (r_2)\ \{ \\ \quad y := 1 & \quad x := 1 \\ \} & \} \end{array} \qquad \text{(LB+ctrl)}$$

Для модели памяти, допускающей значения из воздуха, результат $[r_1 = 1, r_2 = 1]$ также является допустимым (обоснование этого результата такое же, как и для примера LB+data из § 4.3.4). Однако этот результат не только не интуитивен, но и противоречит гарантии внешней свободы от гонок. Действительно, в модели **SC** единственный допустимый сценарий поведения этой программы ведет к результату $[r_1 = 0, r_2 = 0]$ и не содержит гонок, следовательно, в модели, предоставляющей гарантию **eDRF**, эта программа также должна иметь только этот единственный сценарий поведения.

Контр-интуитивное поведение моделей, допускающих значения из воздуха, а также тот факт, что они нарушают множество важных гарантий о поведении программ, привело к следующему консенсусу в исследовательском сообществе: эти модели не подходят на роль моделей памяти для языков программирования [16, 55]. Множество усилий было направлено на то, чтобы сделать невозможными значения из воздуха, но в то же время сохранить оптимальность схем компиляции и корректность как можно большего количества трансформаций. В оставшейся части этого раздела мы опишем различные способы преодоления проблемы значений из воздуха.

6.4. Сохранение программного порядка

Наиболее простой способ запретить значения из воздуха был предложен в работе [16]. Основная идея этого подхода – полностью запретить спекулятивное исполнение, что может быть достигнуто путем запрета переупорядочивания инструкций типа чтение/запись. Это решение позволяет не только восстановить свойство внешней свободы от гонок (**eDRF**) и другие гарантии [21], но также ведет к более простой модели. Абстрактная машина, реализующая данную модель, не нуждается в использовании спекулятивного исполнения и может выполнять инструкции потоков согласно их *программному порядку*, т.е. в том порядке, в котором они указаны. Память такой машины может быть организована как монотонно растущая история сообщений, где каждый поток имеет свое представление фронта данной истории [22, 46].

Данный подход был формализован в работе [21]. Там же было показано, что многие трансформации над кодом программ, за исключением переупорядочивания инструкций типа чтение/запись, остаются корректными в рамках моделей данного класса (см. табл. 1).

Схема компиляции программ для процессоров семейства **x86** является оптимальной, так как модель памяти данной архитектуры гарантирует сохранение порядка между операциями чтения и последующими операциями записи. Однако архитектуры с более слабыми моделями (**Arm**, **POWER**) не гарантируют сохранения этого порядка, и таким образом требуют принятия дополнительных мер при компиляции кода. В [16] было предложено компилировать инструкции ослабленного (**rlx**) чтения как обычные инструкции чтения, за которыми следует ложная инструкция условного ветвления (conditional jump), которая добавляет зависимость между операцией чтения и последующими операциями записи. Гарантируется, что процессоры семейств **Arm** и **POWER** сохранят такую зависимость и, таким образом, сохранят порядок между операцией чтения и последующими операциями записи. В работе [23] изучались накладные расходы этой схемы компиляции, при ее применении только к ослабленным (**rlx**) атомарным обращениям. При компиляции кода для процессоров семейства **Armv8** замедление времени работы составило 0% в среднем и 6.3% максимум на наборе тестов, реализующих различные многопоточные структуры данных, например, блокировки, стеки, очереди, деки, ассоциативные массивы и т.д. Заметим, что следует ожидать более существенного замедления времени работы программ при применении данной схемы компиляции также к неатомарным обращениям к памяти.

6.5. Сохранение синтаксических зависимостей

Альтернативное простое решение проблемы значений из воздуха заключается в сохранении *синтаксических зависимостей* между операциями обращения к памяти [16, 48]. В рамках этого подхода переупорядочивание инструкций типа чтение/запись разрешено, если переставляемые инструкции являются независимыми. Переупорядочивание запрещено, если операция записи зависит от значения, прочитанного операцией чтения, (в этом случае мы говорим что существует зависимость по данным, *data dependency*), или если это значение было использовано при вычислении адреса операции записи (зависимость по адресу (*address dependency*), или если путь исполнения программы, ведущей к операции записи, зависит от прочитанного значения (зависимость по управлению, (*control dependency*). Например, в программе LB+data существует зависимость по данным, так как инструкция $y := r_1$ записывает значение, прочитанное инструкцией $x := r_1$.

Заметим, что эти зависимости вычисляются следуя синтаксису программы (отсюда происходит и название), в противоположность *семантическим зависимостям*. Например, в модифицированной версии программы LB+data, представленной ниже, операция записи в переменную y в левом потоке имеет синтаксическую зависимость от предыдущей операции чтения.

$$\begin{array}{l} r_1 := x \\ y := 1 + 0 * r_1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := y \\ x := r_2 \end{array} \qquad \text{(LB+fakedata)}$$

В этом примере синтаксическая зависимость может быть удалена с помощью оптимизации распространения констант – подвыражение $1 + 0 * r_1$ может быть преобразовано в значение 1. Однако если модель памяти гарантирует сохранение синтаксических зависимостей, компилятору запрещено применять эту оптимизацию, так как после удаления зависимости ничто не мешает компилятору или процессору переставить операцию записи до предшествующей операции чтения.

На этом примере можно видеть главный недостаток моделей, сохраняющих синтаксические зависимости: различные оптимизации, сохраняющие трассы (например, распространение констант), оказываются некорректными в этих моделях. Распространение констант является одной из классических оптимизаций, и тот факт, что оно некорректно, препятствует применению моделей этого класса. Заметим, что модели памяти процессоров используют сходный подход и тоже сохраняют синтаксические зависимости между обращениями к разделяемой памяти [11, 12, 14]. Однако в этом случае это не является проблемой, так как процессоры во время исполнения программы не выполняют такие сложные оптимизации, как распространение констант.

В работе [23] изучалось замедление времени работы программ, накладываемое моделью памяти, сохраняющей синтаксические зависимости. Авторы модифицировали оптимизирующие проходы компилятора так, чтобы они сохраняли зависимости между неатомарными и ослабленными (**rlx**) атомарными обращениями. Затем они измерили время работы программ из тестового набора SPEC CPU2006, скомпилированных модифицированной версией компилятора LLVM для процессоров семейства **Armv8**, и сообщили об умеренном замедлении на 3.1% в среднем и 17.6% максимум.

6.6. Сохранение семантических зависимостей

Последний рассматриваемый нами подход к решению проблемы значений из воздуха заключается в построении понятия *семантических зави-*

симостей, которые могли бы точно характеризовать то, какие пары операций чтения/записи являются независимыми, и отфильтровали бы ложные зависимости как в примере LB+fakedata. Практическая ценность данного подхода заключается в том, что он не требует модификаций существующих компиляторов и процессоров и не накладывает дополнительных расходов на время исполнения скомпилированных программ. Конечной целью подхода является предоставление оптимальных схем компиляции, сохранение корректности большинства существующих трансформаций кода, и в то же время поддержка основных гарантий, таких как внешняя свобода от гонок (**eDRF**).

Оказывается, что эта задача является весьма трудной и на сегодняшний день не решена. Чтобы дать удовлетворительное определение семантических зависимостей, исследователи были вынуждены обратиться к концептуально сложным моделям памяти [17, 19, 20, 24, 25, 51]. Основная сложность работ в данном направлении – необходимость предоставления формальных доказательств того, что эти модели удовлетворяют предъявляемым требованиям, а именно поддерживают оптимальные схемы компиляции и широкий набор трансформаций, и в то же время сохраняют все важные гарантии о поведении программ.

На сегодняшний день наиболее полное решение данной проблемы предоставляет модель “обещающей” семантики (**Promising** semantics) [17, 18]. Было доказано, что эта модель допускает оптимальные схемы компиляции [62], разрешает применять большинство локальных и глобальных трансформаций (за исключением слияния потоков), и в то же время предоставляет свойство внешней свободы от гонок и другие гарантии.

6.7. Вспомогательная классификация

Теперь представим альтернативное разделение моделей на группы на основе того, обладают ли они конкретным свойством, в частности, когерентностью и возгорающейся семантикой. Продемонстрируем, как наличие этих свойств сказывается на оптимальности схемы компиляции и корректности некоторых трансформаций.

6.7.1. Когерентные модели

Свойство когерентности, которое также иногда называется последовательной согласованностью по каждой локации (§ 4.3.2), неожиданным образом взаимодействует с трансформацией удаления общих подвыражений (**CSE**). Впервые эта связь была замечена в контексте ранней версии модели памяти Java [63]. Чтобы увидеть проблему, рассмотрим программу ниже (слева) и ее версию после применения трансформации **CSE** (справа). Заметим, что оптимизация заменила второе обращение к переменной x присваиванием значения из регистра.

$$\begin{array}{l} r_1 := x \\ r_2 := y \\ r_3 := x \end{array} \;\Big\|\; y := 1 \qquad \rightsquigarrow \qquad \begin{array}{l} r_1 := x \\ r_2 := y \\ r_3 := r_1 \end{array} \;\Big\|\; y := 1$$

Предположим, что переменные x и y на самом деле указывают на одну и ту же ячейку памяти. При таком предположении можно сделать вывод о том, что сценарий выполнения программы с результатом $[r_1 = 0, r_2 = 1, r_3 = 0]$ должен быть запрещен моделью памяти, гарантирующей когерентность. В самом деле, когерентная модель памяти для программы, содержащей обращения только к одной локации в памяти, допускает только последовательно согласованные сценарии исполнения. Результат $[r_1 = 0, r_2 = 1, r_3 = 0]$ не может быть получен как последовательное чередование инструкций потоков, и, значит, он не является последовательно согласованным и должен быть запрещен. Тем не менее, данный результат допустим для оптимизированной версии этой программы.

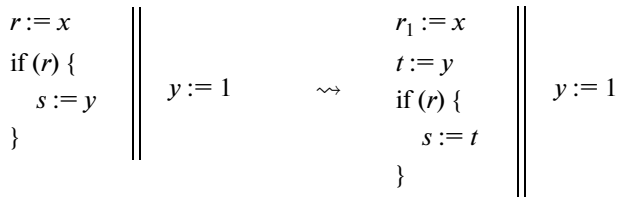
Заметим, что несмотря на вышесказанное, компилятор все равно может применить оптимизацию **CSE** к программе выше, но только в том случае, если он сможет вывести, что переменные x и y указывают на разные локации в памяти. Эта информация может быть получена при помощи анализа псевдонимов (alias analysis) [64]. По сути, в этом случае **CSE** может быть сведен к комбинации других трансформаций – переупорядочиванию и элиминации операций.

Таким образом, в общем случае свойство когерентность несовместимо с удалением общих подвыражений. Что касается оптимальности схем компиляции, то здесь когерентность не накладывает каких-то дополнительных ограничений, поскольку модели памяти процессоров гарантируют соблюдение когерентности [9, 11, 12, 21].

6.7.2. Модели с возгорающейся семантикой

Возгорающаяся семантика (*catch-fire semantics*) присваивает неопределенное поведение программам, содержащим гонки на неатомарных обращениях, и, таким образом, влияет на корректность трансформаций. Как было упомянуто ранее, наличие подобного неопределенного поведения позволяет использовать оптимальные схемы компиляции для неатомарных обращений и влечет корректность широкого класса трансформаций, корректных для случая однопоточных программ. Но помимо этого возгорающаяся семантика интересно взаимодействует с трансформацией введения спекулятивной операции чтения.

Рассмотрим следующий пример:



В § 4.2 мы упоминали, что спекулятивное введение операции чтения может быть использовано в комбинации с элиминацией типа чтение/чтение, чтобы вынести инструкцию чтения из ветки условного оператора. В частности, в примере выше спекулятивное введение операции чтения может добавить инструкцию $t := y$ перед условным оператором `if`, а затем элиминация типа чтение/чтение может заменить вторую операцию чтения присваиванием значения из регистра.

Тонкий момент тут заключается в том, что хотя программа слева не содержит гонок в модели последовательной согласованности, программа справа имеет гонку между операциями чтения и записи переменной y. В предположении, что все обращения в программе выше являются неатомарными, можно заключить, что возгорающаяся семантика должна рассматривать правую программу как имеющую неопределенное поведение. Другими словами, для программы справа допустим любой сценарий исполнения, в то время как для программы слева допустим только сценарий с результатом $[r = 0]$. Однако необходимость корректности трансформации требует, чтобы множество допустимых сценариев исполнения модифицированной программы было подмножеством сценариев поведения оригинальной программы. Можно видеть, что в этом примере условие корректности нарушается.

Говоря простыми словами, спекулятивное введение операции чтения в общем случае некорректно в моделях с возгорающейся семантикой, так как оно может привнести гонки в программы, которые их не содержали. Так как возгорающаяся семантика чувствительна к наличию или отсутствию гонок в программе, она не совместима с этой трансформацией.

Заметим, что эту проблему нельзя решить, запретив спекулятивное введение неатомарных операций чтения и разрешив вводить атомарные операции чтения. В самом деле, спекулятивно добавленная атомарная операция чтения может находиться в состоянии гонки с каким-то неатомарным обращением, расположенным в другом месте программы.

7. РЕКОМЕНДАЦИИ ПО ВЫБОРУ МОДЕЛИ

В этом разделе представлен краткий набор рекомендации для исследователей и системных разработчиков по выбору модели памяти для языка программирования.

Для языков, которые стремятся предоставить простую семантику и высокоуровневые абстракции ценой некоторых потерь в производительности (например, Haskell), целесообразно использовать простые модели памяти, такие как модель последовательной согласованности.

Языки программирования, фокусирующиеся на эффективности производимого кода, такие как C/C++, в свою очередь, вынуждены прибегать к наиболее слабым моделям, допускающим оптимальные схемы компиляции и широкий набор трансформаций кода. Для этих языков целесообразно использовать модели, сохраняющие семантические зависимости. Однако данные модели являются наиболее сложными, что существенно затрудняет рассуждения о корректности программ [65].

Между этими крайними вариантами находятся модели, сохраняющие программный порядок или синтаксические зависимости. Их целесообразно использовать для языков программирования, которые могут позволить умеренные накладные расходы на производительность в обмен на чуть более простое и предсказуемое поведение [23]. Примером такого языка может служить OCaml – высокоуровневый язык программирования, делающий упор на функциональной парадигме, который в то же время активно используется в областях, где критична производительность (разработка компиляторов и инструментов верификации программ).

Более того, модели, сохраняющие программный порядок, имеют дополнительное преимущество перед моделями, сохраняющими синтаксические или семантические зависимости: они не требуют использования спекулятивного исполнения. Этот факт еще больше упрощает рассуждение о корректности многопоточных программ. Ценой этой простоты является требование к моделям, сохраняющим программный порядок, использовать субоптимальные схемы компиляции для процессоров семейств **Arm** и **POWER**. С другой стороны, модели, сохраняющие синтаксические зависимости, позволяют эффективно компилировать код для процессоров **Arm** и **POWER**. Однако эти модели не поддерживают ряд трансформаций, сохраняющих трассы, например, распространение констант. Они также используют спекулятивное исполнение, что приводит к более сложной семантике.

Для языков программирования, использующих более строгие модели памяти, которые требуют использования неоптимальных схем компиляции и запрещают применение некоторых трансформаций, существует несколько общих

техник оптимизации, помогающих частично смягчить возникающие накладные расходы.

Система типов может существенно помочь в этом. Здесь преимущество имеют такие языки как Haskell, OCaml, Rust, которые статически различают и изолируют регионы памяти, к которым возможны обращения из параллельных потоков. Эти языки могут точно идентифицировать переменные, неизменяемые и локальные для одного потока, а затем компилировать обращения к таким переменным без использования барьеров. Более того, к этим обращениям можно применять широкий класс трансформаций кода, корректных для случая однопоточных программ.

Такие языки программирования как Java не могут использовать систему типов для предотвращения доступа к неатомарным переменным из параллельных потоков из-за обратной совместимости. Тем не менее подобные языки могут аппроксимировать множество локальных переменных потоков, используя консервативный статический анализ достижимости переменных (escape analysis) [66], или различные динамические техники [41], а затем оптимизировать обращения к этим переменным.

В функциональных языках программирования активно используются неизменяемые структуры данных. Этот стиль программирования минимизирует использование разделяемой памяти и помогает уменьшить накладные расходы, вызванные строгой моделью памяти [8].

Наконец, если язык допускает наличие неопределенного поведения как, например, C/C++, то альтернативой сложной модели, сохраняющей семантические зависимости, может служить более простая модель, сохраняющая программный порядок, и рассматривающая гонки на неатомарных обращениях как неопределенное поведение [16, 23]. В этом случае компилятор может использовать оптимальные схемы компиляции и широкий спектр трансформаций к неатомарным обращениям и в то же время предоставлять относительно простую семантику для атомарных обращений.

Выбор модели для языка Kotlin. Рассмотрим в качестве примера Kotlin[11], который является языком программирования общего назначения и еще не имеет стандартизированной модели памяти. На текущий момент Kotlin компилируется в байткод виртуальной машины Java, а также в код JavaScript в нативный код средствами LLVM (для платформ Linux, Windows, macOS, iOS и др.).

Kotlin не ориентирован на системное программирование, то есть он не обязан предоставлять абстракции с нулевой стоимостью для обращений к разделяемой памяти целевой архитектуры. Таким образом, модель памяти, сохраняющая программный порядок или синтаксические зависимости, подходит для Kotlin. Оба подхода ведут к умеренным накладным расходам на время работы программ. Однако модели памяти, сохраняющие программный порядок, лучше подходят для языков, допускающих неопределенное поведение для программ с гонками на неатомарных переменных [23], так как в этом случае можно компилировать неатомарные обращения как обычные инструкции обращения к памяти для архитектур процессоров **Arm** и **POWER**. Несмотря на то, что наличие неопределенного поведения в языке Kotlin в общем нежелательно, на практике это труднодостижимо, так как гонки на неатомарных переменных уже имеют неопределенное поведение в LLVM [6], а LLVM является одной из целевых платформ языка Kotlin.

Среди класса моделей, сохраняющих программный порядок, наиболее хорошо изучена и полна с точки зрения поддержки различных примитивов модель **RC11** [21]. Она является модифицированной версией модели **C11** [5], в которую было добавлено сохранение программного порядка. **RC11** поддерживает надмножество режимов обращения к разделяемой памяти, доступных в **JMM** [3] и ее расширении **JAM** [4]. Эта модель очень близка к моделям памяти JavaScript [7] и LLVM [6], так как обе эти модели основываются на **C11**.

Все это делает модель **RC11** хорошей отправной точкой для разработки модели памяти для Kotlin.

8. ЗАКЛЮЧЕНИЕ

В данной работе мы представили обзор существующих моделей памяти языков программирования. Мы сравнили эти модели по ряду критериев и идентифицировали шесть основных классов моделей памяти. Также, мы предложили рекомендации по выбору эффективной модели памяти для различных языков программирования. Мы надеемся, что наша работа будет полезна для исследователей и разработчиков в области языков программирования и послужит введением в сложную тематику слабых моделей памяти.

На основе нашего анализа мы также можем сделать следующие предположения о будущих направлениях работы в данной области.

Проблема оптимальности схем компиляции и корректности локальных трансформаций кода на сегодняшний день относительно хорошо изучена. Более новые модели, такие как RC11 [21], OCaml MM [22], **Promising** [17, 18] и **Weakestmo** [19], поддерживают широкий диапазон локальных трансформаций и имеют ясные компромиссы относительно оптимальности схем компиляции. Исклю-

[11] https://kotlinlang.org/

чением являются локальные трансформации, использующие циклы или рекурсию, так как их корректность все еще недостаточно изучена. Глобальным трансформациям также уделялось мало внимания, за важным исключением работ [18, 25]. Влияние этих трансформаций на дизайн модели памяти еще предстоит изучить.

Глобальное свойство свободы от гонок на настоящий момент детально изучено [3, 17, 21, 55]. Напротив, локальное свойство свободы от гонок [22] является новой концепцией. Стоит ожидать, что оно, как и другие локальные гарантии [45, 67, 68], будет активно исследоваться в ближайшем будущем.

Смешанные обращения [30], используемые в модели памяти JavaScript [7], а также в некоторых приложениях, например, в кодовой базе ядра Linux [30], на сегодняшний день недостаточно изучены даже в контексте моделей памяти процессоров. Таким образом, более глубокое понимание семантики смешанных обращений является еще одним важным направлением исследований.

Модели памяти, сохраняющие семантические зависимости, по-прежнему являются темой активных исследований [17–20, 67, 68]. Следует ожидать, что они будут более детально разрабатываться и уточняться в ближайшем будущем. Интересным направлением работ в этой области является разработка новых гарантий, помимо свойства свободы от гонок, которые помогут усовершенствовать мета-теорию этих моделей и упростить рассуждение о корректности программ.

Наконец, детальное исследование накладных расходов на время исполнения программ с различными моделями памяти также являются очень важной задачей. Несмотря на наличие здесь некоторого количества работ [8, 22, 23, 33, 40, 41], полная картина все еще остается недостаточно ясной.

БЛАГОДАРНОСТИ

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 20-31-90088.

СПИСОК ЛИТЕРАТУРЫ

1. *Dijkstra E.W.* “Cooperating sequential processes,” in The origin of concurrent programming. Springer, 1968. P. 65–138.
2. *Lamport L.* “How to make a multiprocessor computer that correctly executes multiprocess programs,” IEEE Trans. Computers. 1979. V. 28. № 9. P. 690–691.
3. *Manson J., Pugh W., Adve S.V.* “The Java memory model,” in POPL 2005. P. 378–391, ACM, 2005.
4. *Bender J., Palsberg J.* “A formalization of Java’s concurrent access modes,” Proceedings of the ACM on Programming Languages. 2019. V. 3. № OOPSLA. P. 1–28.
5. *Batty M., Owens S., Sarkar S., Sewell P., Weber T.* “Mathematizing C++ concurrency,” in POPL 2011. 2011. P. 55–66, ACM.
6. *Chakraborty S., Vafeiadis V.* “Formalizing the concurrency semantics of an LLVM fragment,” in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2017. P. 100–110.
7. *Watt C., Pulte C., Podkopaev A., Barbier G., Dolan S., Flur S., Pichon-Pharabod J., Guo S.-y.* “Repairing and mechanizing the JavaScript relaxed memory model,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020. P. 346–361.
8. *Vollmer M., Scott R.G., Musuvathi M., Newton R.R.* “SC-Haskell: Sequential consistency in languages that minimize mutable shared heap,” ACM SIGPLAN Notices. 2017. V. 52. № 8. P. 283–298.
9. *Sewell P., Sarkar S., Owens S., Nardelli F.Z., Myreen M.O.* “x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors,” Commun. ACM. 2010. V. 53. № 7. P. 89–97.
10. *Alglave J., Fox A., Ishtiaq S., Myreen M.O., Sarkar S., Sewell P., Nardelli F.Z.* “The semantics of Power and ARM multiprocessor machine code,” in Proceedings of the 4th workshop on Declarative aspects of multicore programming. 2009. P. 13–24.
11. *Sarkar S., Sewell P., Alglave J., Maranget L., Williams D.* “Understanding POWER multiprocessors,” in PLDI 2011. ACM, 2011. P. 175–186.
12. *Alglave J., Maranget L., Tautschnig M.* “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” ACM Trans. Program. Lang. Syst. 2014. V. 36. № 2. P. 7:1–7:74.
13. *Chong N., Ishtiaq S.* “Reasoning about the ARM weakly consistent memory model,” in Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08). 2008. P. 16–19.
14. *Pulte C., Flur S., Deacon W., French J., Sarkar S., Sewell P.* “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8,” Proceedings of the ACM on Programming Languages. 2018. V. 2. № POPL. P. 1–29.
15. *Flur S., Gray K.E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., Sewell P.* “Modelling the ARMv8 architecture, operationally: Concurrency and ISA,” in POPL 2016. 2016. P. 608–621, ACM.
16. *Boehm H.-J., Demsky B.* “Outlawing ghosts: Avoiding out-of-thin-air results,” in MSPC 2014. 2014. P. 7:1–7:6, ACM.
17. *Kang J., Hur C.-K., Lahav O., Vafeiadis V., Dreyer D.* “A promising semantics for relaxedmemory concurrency,” in POPL 2017, ACM, 2017.
18. *Lee S.-H., Cho M., Podkopaev A., Chakraborty S., Hur C.-K., Lahav O., Vafeiadis V.* “Promising 2.0: global optimizations in relaxed memory concurrency,” in Proceedings of the 41st ACM SIGPLAN Conference

on Programming Language Design and Implementation. 2020. P. 362–376.

19. *Chakraborty S., Vafeiadis V.* "Grounding thin-air reads with event structures," Proceedings of the ACM on Programming Languages. 2019. V. 3. № POPL. P. 1–28.
20. *Paviotti M., Cooksey S., Paradis A., Wright D., Owens S., Batty M.* "Modular relaxed dependencies in weak memory concurrency," in European Symposium on Programming. P. 599–625, Springer, Cham, 2020.
21. *Lahav O., Vafeiadis V., Kang J., Hur C.-K., Dreyer D.* "Repairing sequential consistency in C/C++11," in PLDI 2017, ACM, 2017.
22. *Dolan S., Sivaramakrishnan K., Madhavapeddy A.* "Bounding data races in space and time," ACM SIGPLAN Notices. 2018. V. 53. № 4. P. 242–255.
23. *Ou P., Demsky B.* "Towards understanding the costs of avoiding out-of-thin-air results," Proceedings of the ACM on Programming Languages. 2018. V. 2. № OOPSLA. P. 1–29.
24. *Jeffey A., Riely J.* "On thin air reads: Towards an event structures model of relaxed memory," in LICS 2016, IEEE, 2016.
25. *Pichon-Pharabod J. and Sewell P.*, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in POPL 2016. 2016. P. 622–633, ACM.
26. *Marlow S. et al.* "Haskell 2010 language report," Available on: https://www. haskell. org/onlinereport/haskell2010, 2010.
27. *Klabnik S., Nichols C.* The Rust Programming Language (Covers Rust 2018). No Starch Press, 2019.
28. *Boehm H.-J., Adve S.V.* "Foundations of the C++ concurrency memory model," ACM SIGPLAN Notices. 2008. V. 43. № 6. P. 68–78.
29. *Lahav O., Giannarakis N., Vafeiadis V.* "Taming release-acquire consistency," ACM SIGPLAN Notices. 2016. V. 51. № 1. P. 649–662.
30. *Flur S., Sarkar S., Pulte C., Nienhuis K., Maranget L., Gray K.E., Sezgin A., Batty M., Sewell P.* "Mixed-size concurrency: ARM, Power, C/C++ 11, and SC," ACM SIGPLAN Notices. 2017. V. 52. № 1. P. 429–442.
31. "C/C++11 mappings to processors," 2011. Available at https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html [Online; accessed 26-April-2021].
32. *Marino D., Singh A., Millstein T., Musuvathi M., Narayanasamy S.* "A case for an SC-preserving compiler," ACM SIGPLAN Notices. 2011. V. 46. № 6. P. 199–210.
33. *Liu L., Millstein T., Musuvathi M.* "A volatile-by-default JVM for server applications," Proceedings of the ACM on Programming Languages. 2017. V. 1. № OOPSLA. P. 1–25.
34. *Muchnick S.* Advanced compiler design and implementation. Morgan kaufmann, 1997.
35. *Lahav O., Namakonov E., Oberhauser J., Podkopaev A., Vafeiadis V.* "Making weak memory models fair," arXiv preprint arXiv:2012.01067, 2020.
36. *Ševčík J., Aspinall D.* "On validity of program transformations in the Java memory model," in European Conference on Object-Oriented Programming. P. 27–51, Springer, 2008.
37. *Wegman M.N., Zadeck F.K.* "Constant propagation with conditional branches," ACM Transactions on Programming Languages and Systems (TOPLAS). 1991. V. 13. № 2. P. 181–210.
38. *Maranget L., Sarkar S., Sewell P.* "A tutorial introduction to the ARM and POWER relaxed memory models," 2012. Available at https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf [Online; accessed 30-April-2021].
39. *Boudol G., Petri G.* "A theory of speculative computation," in European Symposium on Programming. Springer, 2010. P. 165–184.
40. *Singh A., Narayanasamy S., Marino D., Millstein T., Musuvathi M.* "End-toend sequential consistency," in 2012 39th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2012. P. 524–535.
41. *Liu L., Millstein T., Musuvathi M.* "Accelerating sequential consistency for Java with speculative compilation," in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019. P. 16–30.
42. *Marino D., Singh A., Millstein T., Musuvathi M., Narayanasamy S.* "DRFx: A simple and efficient memory model for concurrent programming languages," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2010. P. 351–362.
43. *Demange D., Laporte V., Zhao L., Jagannathan S., Pichardie D., Vitek J.* "Plan B: A buffered memory model for Java," in Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2013. P. 329–342.
44. *Boudol G., Petri G.* "Relaxed memory models: an operational approach," ACM SIGPLAN Notices. 2009. V. 44. № 1. P. 392–403.
45. *Dodds M., Batty M., Gotsman A.* "Compositional verification of compiler optimisations on relaxed memory," in European Symposium on Programming. Springer, 2018. P. 1027–1055.
46. *Doherty S., Dongol B., Wehrheim H., Derrick J.* "Verifying C11 programs operationally," in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. 2019. P. 355–365.
47. *Dang H.-H., Jourdan J.-H., Kaiser J.-O., Dreyer D.* "RustBelt meets relaxed memory," Proceedings of the ACM on Programming Languages. 2019. V. 4. № POPL. P. 1–29.
48. *Alglave J., Maranget L., McKenney P.E., Parri A., Stern A.* "Frightening small children and disconcerting grownups: Concurrency in the Linux kernel," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 2018. P. 405–418.
49. *Zhang Y., Feng X.* "An operational happens-before memory model," Frontiers of Computer Science. 2016. V. 10. № 1. P. 54–81.
50. *Huisman M., Petri G.* "The Java memory model: a formal explanation," VAMP. 2007. V. 7. P. 81–96.
51. *Jagadeesan R., Pitcher C., Riely J.* "Generative operational semantics for relaxed memory models," in European Symposium on Programming. Springer, 2010. P. 307–326.

52. *Sarkar S., Memarian K., Owens S., Batty M., Sewell P., Maranget L., Alglave J., Williams D.* “Synchronising C/C++ and POWER,” in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012. P. 311–322.
53. *Batty M., Memarian K., Owens S., Sarkar S., Sewell P.* “Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER,” ACM SIGPLAN Notices. 2012. V. 47. № 1. P. 509–520.
54. *Vafeiadis V., Balabonski T., Chakraborty S., Morisset R., Nardelli F.Z.* “Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it,” in POPL 2015. ACM, 2015. P. 209–220.
55. *Batty M., Memarian K., Nienhuis K., Pichon-Pharabod J., Sewell P.* “The problem of programming language concurrency semantics,” in ESOP. V. 9032 of LNCS. Springer, 2015. P. 283–307.
56. *Batty M., Donaldson A.F., Wickerson J.* “Overhauling SC atomics in C11 and OpenCL,” in Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2016. P. 634–648.
57. *Nienhuis K., Memarian K., Sewell P.* “An operational semantics for C/C++ 11 concurrency,” in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2016. P. 111–128.
58. *Crary K., Sullivan M.J.* “A calculus for relaxed memory,” in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2015. P. 623–636.
59. *Saraswat V.A., Jagadeesan R., Michael M., von Praun C.* “A theory of memory models,” in Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. 2007. P. 161–172.
60. *Inc S.I., Weaver D.L.* The SPARC architecture manual. Prentice-Hall, 1994.
61. *Lustig D., Trippel C., Pellauer M., Martonosi M.* “ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures,” in Proceedings of the 42nd Annual International Symposium on Computer Architecture. 2015. P. 388–400.
62. *Podkopaev A., Lahav O., Vafeiadis V.* “Bridging the gap between programming languages and hardware weak memory models,” Proceedings of the ACM on Programming Languages. 2019. V. 3. № POPL. P. 1–31.
63. *Pugh W.* “Fixing the Java memory model,” in Proceedings of the ACM 1999 conference on Java Grande. 1999. P. 89–98.
64. *Diwan A., McKinley K.S., Moss J.E.B.* “Type-based alias analysis,” ACM Sigplan Notices. 1998. V. 33. № 5. P. 106–117.
65. *Svendsen K., Pichon-Pharabod J., Doko M., Lahav O., Vafeiadis V.* “A separation logic for a promising semantics,” in Programming Languages and Systems (A. Ahmed, ed.), (Cham). P. 357–384, Springer International Publishing, 2018.
66. *Choi J.-D., Gupta M., Serrano M., Sreedhar V.C., Midkiff S.* “Escape analysis for Java,” Acm Sigplan Notices. 1999. V. 34. № 10. P. 1–19.
67. *Jagadeesan R., Jeffrey A., Riely J.* “Pomsets with preconditions: a simple model of relaxed memory,” Proceedings of the ACM on Programming Languages. 2020. V. 4. № OOPSLA. P. 1–30.
68. *Cho M., Lee S.-H., Hur C.-K., Lahav O.* “Modular data-race-freedom guarantees in the Promising semantics,” in Proceedings of the 42st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2021.