Обещающая компиляция в ARMv8.3

[1,2]А.В.Подкопаев <apodkopaev@2009.spbu.ru>
[3] О.Лахав <orilahav@tau.ac.il>
[4]В.Вафеядис <viktor@mpi-sws.org>
[1] Санкт-Петербургский Государственный Университет,
199034, Россия, Санкт-Петербург, Университетская набережная, д. 7–9
[2]JetBrains Research, 199034, Россия, Санкт-Петербург,
Университетская набережная, д. 7–9–11/5А
[3] Тель-Авивский Университет, 39040, Израиль, Тель-Авив 69978
[4]Институт им. Макса Планка: Программные Системы,
67663, Германия, Кайзерслаутерн, ул. Пауль-Эрлих G26

Аннотация. Поведение многопоточных программ не может быть промоделировано попеременным последовательным исполнением различных потоков на одном вычислительном узле. На данный момент полное и корректное описание поведения многопоточных программ является открытой теоретической проблемой. Одним из перспективных решений этой проблемы является „обещающая“ модель памяти. Для того, чтобы некоторая модель могла быть использована в стандарте некоторого промышленного языка программирования, должна быть доказана корректность компиляции из этой модели в модель памяти целевой процессорной архитектуры. Данная статья представляет доказательство корректности компиляции из подмножества обещающей модели в модели памяти процессора ARMv8.3. Главной идеей доказательства является введение промежуточного операционного варианта модели ARMv8.3, поведение которого может быть симулировано обещающей моделью.

Ключевые слова: многопоточность; корректность компиляции; слабые модели памяти.

DOI: 10.15514/ISPRAS-2017-29(5)-9

Для цитирования: Подкопаев А.В., Лахав О., Вафеядис В. Обещающая компиляция в ARMv8.3. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 149-164. DOI: 10.15514/ISPRAS-2017-29(5)-9

1. Введение

Распространенным заблуждением является уверенность, что поведение многопоточных программ может быть описано попеременным исполнением инструкций потоков одним процессором, то есть в рамках модели *последовательной консистентности* [1]. В качестве контрпримера рассмотрим следующую программу:

a := [*x*];	‖	b := [*y*];	**(LB)**
[*y*] := 1;	‖	[*x*] := 1;	

В этой программе каждый поток сначала выполняет чтение из некоторой локации (*x* или *y*), а потом записывает значение 1 в другую локацию. Пусть изначально в локации *x* и *y* записан 0. Тогда попеременным исполнением инструкций потоков можно получить три следующих результата: 1) a = 0, b = 0; 2) a = 0, b = 1; 3) a = 1, b = 0. Тем не менее, данная программа, будучи запущенной на процессорах семейств Power или ARM, может завершиться с результатом a = 1, b = 1. Такой результат не может быть получен в модели последовательной консистентности, так как требует выполнения обеих операций записи до любого чтения.

Поведение многопоточной программы, выходящее за пределы последовательной консистентности, называют *слабым поведением* (weak behavior). Слабое поведение является результатом применения оптимизаций со стороны компилятора и/или процессора. Так в случае программы **LB**, компилятор может переставить инструкции первого потока, например, как часть планирования кода [2], или процессор может буферизировать чтение во втором потоке и выполнить инструкцию [*x*] := 1 до завершения исполнения инструкции чтения. В обоих случаях результат a = 1, b = 1 станет возможным.

В современных системах используют оптимизации, которые таким странным образом меняют семантику многопоточной программы, потому что, во-первых, они ускоряют выполнение программ, а, во-вторых, для большинства многопоточных систем справедливо, что слабое поведение не проявляется у программ с корректными блокировками, т. е. у программ без гонок по памяти[1].

В связи с тем, что слабое поведение программ бывает очень сложным, и требуется гарантировать надёжность прикладному программисту, научное сообщество в сотрудничестве с индустрией разрабатывает формальные семантики для систем со слабыми поведениями, или *слабые модели памяти* (weak memory models). Слабые модели существуют для таких распространенных процессорных платформ, как x86 [3], Power [4,5], ARM [6,7], а также для языков программирования C/C++ [8] и Java [9].

Отметим, что, с одной стороны, прикладному программисту хотелось бы иметь как можно больше гарантий, что означает уменьшение числа возможных слабых поведений. С другой стороны, модель памяти языка должна предоставлять возможность эффективной компиляции в целевые платформы, что ведёт к увеличению числа слабых поведений.

На данный момент не существует стандартизованной модели памяти промышленного языка программирования, которая удовлетворяет всем

1 В программе имеется гонка по памяти (data race), если в ней есть два конкурентных обращения к одной и той же ячейке памяти, при этом как минимум одно из обращений является записью в ячейку [10].

требованиям. Так известно, что модель памяти Java не допускает некоторые оптимизации, которые используются даже в официальном компиляторе Java [11], а модель памяти C/C++11 допускает т. н. значения из воздуха (out-of-thin-air values) [12]. Тем не менее, недавно было представлено перспективное решение для данной проблемы – «обещающая» модель памяти [13].

Для того, чтобы обещающая модель памяти смогла войти в стандарт промышленного языка программирования, про модель должно быть доказано несколько утверждений, в том числе возможность её эффективной и корректной компиляции. Данная статья посвящена корректности компиляции из обещающей модели памяти в аксиоматическую модель памяти процессора ARMv8.3 [7].

Основным результатом работы является доказательство корректности компиляции из подмножества обещающей модели памяти без сертификации [13] в модель ARMv8.3. Рассмотренное подмножество обещающей модели состоит из расслабленных (relaxed) операций чтения и записи, а также высвобождающих (release) и приобретающих (acquire) барьеров памяти. Кроме того, в доказательстве используется новый подход, который потенциально может быть использован для других подобных доказательств: вводится промежуточная операционная семантика, которая осуществляет обход исполнения аксиоматической семантики (в нашем случае, ARMv8.3), при этом данный обход может быть симулирован обещающей моделью. В отличии от метода, использованного ранее в [13], наш подход вводит меньше ограничений на целевую платформу и не опирается на представимость целевой модели как набора оптимизаций над более строгой моделью [14].

Статья организована следующим образом. Мы вводим понятие корректности компиляции и описываем существующие работы в соответствующей области (раздел 2), обсуждаем базовые концепции модели памяти ARMv8.3 и обещающей модели на примерах (разделы 3 и 4), описываем общую структуру доказательства (раздел 5). В разделе 6 подводятся итоги и приводятся возможные направления для дальнейшей работы.

Отметим, что в данной статье приводится лишь верхнеуровневое описание доказательства. Подробное формальное доказательство приведено в [15].

2. Корректность компиляции. Существующие работы

Выберем два языка – HL (high level) и LL (low level) – а также схему компиляции *comp* из HL в LL, т. е. функцию, преобразующую программу на языке HL в язык LL. Схема компиляции *comp* считается корректной, если для любой программы *Prog* и её скомпилированной версии *comp(Prog)* выполняется следующее: любое поведение, разрешенное для *comp(Prog)* в рамках модели языка LL, также разрешено для *Prog* в рамках модели HL.

На данный момент существует множество работ, посвящённых корректности компиляции. Наиболее известными являются [16] и [17], представляющие

верифицированные компиляторы для языков C и CakeML соответственно. Данные компиляторы сравнимы с промышленными по качеству генерируемого кода. Существенным ограничением данных работ является то, что они рассматривают только однопоточные программы. Существует и расширение работы [16] для случая слабой памяти [18], но в [18] рассматривается только достаточно простая и ограниченная модель памяти TSO (total store order).

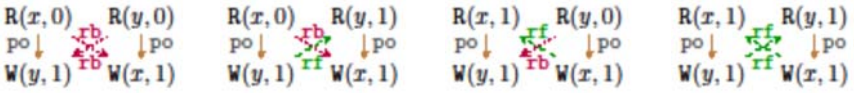
Имеются работы, посвящённые компиляции из одной слабой модели памяти в другую [8, 13, 19, 20, 21]. В них отсутствует полноценная компиляция промышленного языка программирования и рассматривается лишь подмножество языка, имеющее отношение к многопоточности. Как следствие, исходный и целевой языки обычно считаются совпадающими с точностью до модификаторов на операциях записи, чтения и барьеров памяти, которые имеют отношение к многопоточности. Этим же путём следует и представляемая работа.

Наиболее близкими к нашей работе являются [13] и [21]. В [13] доказывается корректность компиляции из обещающей модели в модели процессоров x86-TSO [3] и Power [4]. При этом авторы [13] использовали результат [14], который утверждает, что все исполнения, возможные в рамках моделей x86-TSO [3] и Power [4], могут быть представлены как некоторое число оптимизаций над исполнением в более строгих моделях, которые не нуждаются в механизме обещаний. Далее авторы доказывают, что упомянутые оптимизации являются корректными в рамках обещающей модели, т. е. семантика оптимизированной программы является подмножеством семантики изначальной программы, и показывают корректность компиляции для упрощенных моделей. К сожалению, модель памяти ARMv8.3 [7] не может быть представлена как набор упомянутых оптимизаций над более строгой моделью памяти. Поэтому в нашей работе нам пришлось использовать другой подход.

В работе [21] показана корректность компиляции из обещающей модели в модель памяти ARMv8 POP [6]. В работе использован метод *запаздывающей симуляции*, который является специальной формой индукции по исполнению программы в целевой машине. Данный подход возможен, т. к. обе модели представлены *операционно*, т. е. в терминах некоторой абстрактной машины. Поскольку модель памяти процессора ARMv8.3 задана *аксиоматически*, т.е. она представляет семантику программы как множество графов, где каждый конкретный граф является одним из возможных исполнений программы, то доказательство с помощью симуляции напрямую не построить. Чтобы преодолеть данное ограничение и все-таки использовать симуляцию, мы вводим операционную семантику обхода графа, представляющего исполнение в модели ARMv8.3. Данный обход может быть напрямую симулирован абстрактной машиной, представляющей обещающую модель памяти.

3. Описание модели памяти ARMv8.3 на примерах

В этом разделе мы рассмотрим модель памяти процессора ARMv8.3 на примере программы **LB** из раздела 1 и вариации этой программы, для которой запрещено слабое поведение.



а) б) в) г)

*Рис. 1. Возможные исполнения программы **LB** в модели ARMv8.3*
*Fig. 1. Executions of program **LB**, which are allowed by model ARMv8.3*

Поскольку модель ARMv8.3 является аксиоматической, то семантика программы в этой модели представляется множеством графов, где вершины представляют *события*, т. е. операции над памятью, такие как чтение, запись или барьер памяти. На рис. 1 мы можем видеть все четыре возможных исполнения программы **LB** в рамках этой модели. Исполнения а), б) и в) соответствуют результатам (a = 0, b = 0), (a = 0, b = 1) и (a =1, b = 0), в то время как исполнение г) является слабым и представляет результат (a = 1, b = 1). Обратим внимание на исполнение б) и разберемся с обозначениями.

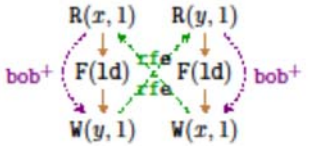
В случае исполнения б) в графе четыре операции, из которых две являются чтением (R(x, 0), R(y, 1)), а две – записью (W(y, 1), W(x, 1)). Также в каждом графе присутствуют инициализирующие записи, но в рис. 1 мы их опустили для краткости. Рёбрами в графе являются различные отношения между операциями. Так рёбра, подписанные po (program order), обозначают программный порядок между операциями, ребро rf (read from) ведёт в операцию чтения из соответствующей операции записи, а ребро rb (read before) – из операции чтения в операцию записи, когда чтение происходит из «более ранней» записи. Для определения «более ранних» записей используется отношение co (coherence order), которое является полным порядком на записях в одну локацию, но поскольку на рис. 1 нет двух записей в одну локацию, то отношение co не представлено.

Для того, чтобы запретить результат (a = 1, b = 1), в программу могут быть вставлены *барьеры памяти*, которые запрещают процессору выполнение инструкций не по порядку. В рамках архитектуры ARMv8 существует несколько различных видов барьеров, в том числе полный барьер памяти, имеющий модификатор **SY**, и барьер на чтение, имеющий модификатор **LD**, который является ослабленной версией полного барьера. Существуют и другие модификаторы, но они не используется в схеме компиляции из обещающей модели памяти, рассмотренной в этой работе.

В случае программы **LB** достаточно вставить барьер на чтение в оба потока между операциями, чтобы запретить слабое поведение:

a := [x];	b := [y];		**(LB-LD)**
fence(**LD**);	fence(**LD**);		
[y] := 1;	[x] := 1;		

Рассмотрим, каким образом модель памяти ARMv8.3 запрещает поведение (a = 1, b = 1) для программы **LB-LD**. Ниже представлен соответствующий граф исполнения:



В этом графе есть несколько изменений по сравнению с графом на рис. 1, г). Во-первых, в графе появились вершины, соответствующие барьерам памяти. Во-вторых, появились ребра, соответствующие отношению bob+, которое является транзитивным замыканием для bob. Отношение bob (barrier order before) связывает операцию чтения с операцией барьера по чтению, если барьер следует за чтением в программном порядке, и любую операцию, следующую за барьером. В-третьих, подписи отношения rf были заменены на подписи rfe, где e (external) обозначает, что данное ребро связывает события из разных потоков.

В результате в графе присутствует цикл из рёбер отношений bob и rfe. Такие циклы запрещены в модели памяти ARMv8.3, поэтому поведение (a = 1, b = 1) в целом не является *ARM-согласованным*. Под ARM-согласованным поведением мы понимаем граф, который не противоречит аксиомам модели ARMv8.3. Полный список аксиом приведён в [7].

4. Описание обещающей модели памяти на примерах

В этом разделе мы, аналогично разделу 3, рассмотрим обещающую модель памяти на примере программы **LB**, а также на двух вариациях этой программы, которые содержат барьеры памяти.

Обещающая модель памяти задана операционно, т. е. в терминах некоторой абстрактной машины, которую дальше мы будет называть *обещающей машиной*. Состояние этой машины включает *память* – множество сообщений, каждое из которых соответствует какой-то записи в локацию. Изначально память (M) содержит по инициализирующему сообщению на каждую локацию.

Каждое сообщение в памяти состоит из трёх компонент: целевой локации, значения и *метки времени*. Метки времени служат для того, чтобы упорядочивать сообщения, которые относятся к одной и той же локации, и задаются положительными рациональными числами.

Так, перед исполнением программы **LB**, память выглядит следующим образом:

$$M = \{\langle x{:}0@0_\tau\rangle,\ \langle y{:}0@0_\tau\rangle\},$$

где 0 – это записанное значение, а 0_τ – метка времени.

Исполнение программы в обещающей машине представляется как некоторое попеременное исполнение потоков. На каждом конкретном шаге активный поток может сделать одно из двух действий: либо выполнить следующую в программе инструкцию, либо *пообещать* сделать запись в память.

Для того, чтобы получить результат (a = 1, b = 1), в программе **LB** один из потоков должен начать свою работу с обещания. Пусть это будет левый поток. После того, как левый поток обещает сделать запись $\langle y{:}1@2_\tau \rangle$, соответствующее сообщение добавляется во множество обещаний, еще не выполненных левым потоком, а также в память. После этого память содержит три сообщения:

$$M = \{\langle x{:}0@0_\tau \rangle,\ \langle y{:}0@0_\tau \rangle,\ \langle y{:}1@2_\tau \rangle\}.$$

Теперь это сообщение является доступным для чтения другим потокам, поскольку оно находится в памяти, но не для левого потока, т. к. оно находится в его множестве еще не выполненных обещаний. После этого, правый поток может быть исполнен в обычном порядке.

Первым действием он производит чтение из нового сообщения в памяти, а вторым – записывает новое сообщение в локацию *x*. Теперь память содержит четыре сообщения:

$$M = \{\langle x{:}0@0_\tau \rangle,\ \langle y{:}0@0_\tau \rangle,\ \langle y{:}1@2_\tau \rangle, \langle x{:}1@1_\tau \rangle\}.$$

Далее левый поток читает из этого сообщения и в конце выполняет своё обещание, исполняя инструкцию записи в локацию *y*.

Точно так же, как и в случае модели ARMv8.3, для того, чтобы запретить результат (a = 1, b = 1), в программу могут быть добавлены барьеры памяти. Виды барьеров в обещающей семантике отличаются, т. к. обещающая модель памяти придерживается синтаксиса, описанного в стандартах C/C++11.

В рассматриваемом нами подмножестве обещающей машины есть два типа барьеров: *высвобождающий* (rel, release) и *приобретающий* (acq, acquire). При эффективной компиляции программы из синтаксиса обещающей модели в синтаксис ARMv8 высвобождающий барьер переходит в полный барьер памяти ARMv8, а приобретающий – в барьер по чтению.

Программа, результатом компиляции которой является **LB-LD**, выглядит так[2]:

a := [*x*];	b := [*y*];	
fence(**ACQ**);	fence(**ACQ**);	**(LB-ACQ)**
[*y*] := 1;	[*x*] := 1;	

Несмотря на то, что после компиляции данная программа не может иметь слабое поведение (a = 1, b = 1), сама программа **LB-ACQ** в рамках обещающей модели имеет данное поведение. Это не противоречит корректности

2 Заметим, что для обоих программ используется одна и та же нотация, несмотря на то, что программа **LB-LD** рассматривается как программа на целевом языке компиляции, а программа **LB-ACQ** – на исходном языке.

компиляции, т. к. последняя утверждает, что любое поведение скомпилированной программы должно быть поведением изначальной программы, при этом обратное не утверждается.

Для того, чтобы запретить результат (a = 1, b = 1), нужно использовать высвобождающие барьеры:

a := [*x*];	b := [*y*];	
fence(**REL**);	fence(**REL**);	**(LB-REL)**
[*y*] := 1;	[*x*] := 1;	

Данный тип барьера запрещает потоку делать обещания через него, т. е. в момент исполнения самого барьера множество ещё не выполненных обещаний данного потока должно быть пусто. Поэтому при исполнении программы как минимум одно из чтений будет выполнено до того, как в памяти появится хотя бы одно сообщение со значением 1.

При компиляции высвобождающие барьеры переходят в полные барьеры, которые так же, как и барьеры по чтению, гарантируют отсутствие слабого поведения для соответствующей программы в рамках модели ARMv8.3.

5. Доказательство корректности компиляции

Основным результатом данной статьи является доказательство следующей теоремы.

Теорема 1. Для любой программы *Prog*, результата её компиляции *Prog*ARM и ARM-согласованного исполнения *G* программы *Prog*ARM существует исполнение *Prog* обещающей машиной т.ч. финальное состояние памяти машины совпадает с состоянием памяти *G*.

Под финальным состоянием памяти обещающей машины имеется ввиду подмножество сообщений, каждое из которых имеет максимальную метку времени среди сообщений, относящихся к той же локации. Состоянием памяти исполнения *G* считается множество событий-максимумов по отношению co, которое является аналогом меток времени в ARM-согласованных исполнениях.

5.1. Структура доказательства

Как видно из разделов 3 и 4, ARMv8.3 и обещающая модели заданы в совершенно разных стилях. Нужно это различие преодолеть для того, чтобы иметь возможность доказать теорему 1.

Наше доказательство базируется на введении промежуточной операционной семантики, которая совершает обход графа ARM-согласованного исполнения в специальном порядке, гарантирующим возможность обещающей машине исполнять соответствующие инструкции одновременно с обходом. Имея упомянутую промежуточную семантику, мы можем показать, что обещающая машина может симулировать семантику обхода.

Метод симуляции является прямой индукцией по исполнению целевой машины. Первым шагом метода является определение отношения симуляции, которое тесно связывает состояния машин обоих моделей. Вторым шагом доказывается, что если целевая машина находится в состоянии *t*, а исходная — в состоянии *s*, и эти состояния связаны отношением симуляции, тогда для любого состояния *t'*, в которое может перейти целевая машина, существует состояние *s'*, в которое может перейти исходная машина, т.ч. *t'* и *s'* опять связаны отношением симуляции.

5.2. Обход ARM-согласованных исполнений

Данный обход мы представляем в виде операционной семантики некоторой абстрактной машины. Состояние машины состоит из пары множеств *(C, I)* и называется *конфигурацией обхода*. При этом пара *(C, I)* считается корректной конфигурацией обхода ARM-согласованного исполнения *G*, если выполняется утверждения, представленные ниже.

1. *C* и *I* являются подмножествами событий в графе *G*.
2. *C* является замкнутым по префиксу отношения po (программного порядка), т. е. если некоторое событие *e* принадлежит *C*, то и любое событие *u*, т.ч. из *u* есть po-ребро в *e*, тоже принадлежит *C*.
3. Все события из множества *I* являются событиями записи, и все события записи из множества *C* также являются элементами *I*.
4. Все события, которые являются инициализирующими записями в локации, являются элементами *C*.

Далее множество *C* мы будем называть множеством *покрытых* элементов (covered), а множество *I* – множеством *выпущенных* элементов (issued).

У абстрактной машины, соответствующей обходу исполнения, есть два правила перехода из одной конфигурацию в другую: *покрытие события* и *выпуск события*.

Правило покрытия позволяет для некоторого события *e* перейти из конфигурации *(C, I)* в конфигурацию *(C ∪{e}, I)*. При этом для события *e* должны выполняться условия, представленные ниже.

1. Все события, po-предшествующие *e*, должны принадлежать *C*.
2. Событие *e* не принадлежит C.
3. Если событие *e* является операцией чтения, то событие записи *w*, из которого читает *e* (т. е. между *w* и *e* есть ребро отношения rf), должно быть элементом множества *I*.
4. Если событие *e* является операцией записи, то оно должно быть элементом множества *I*.

Правило выпуска добавляет событие *e* во множество выпущенных, т. е. соответствует переходу из *(C, I)* в *(C, I ∪{e})*. Событие *e* должно удовлетворять следующим ограничениям:

1. Событие *e* является операцией записи.
2. Событие *e* не является элементом множества *I*.
3. Все po-предшествующие барьеры памяти должны быть покрыты.
4. Все записи других потоков, от которых зависит событие *e*, являются выпущенными[3].

Так как оба правила только наращивают компоненты конфигурации, а любое исполнение мы считаем конечным графом, то для того, чтобы доказать наличие полного обхода ARM-согласованного исполнения *G*, достаточно показать, что из любой конфигурации *(C, I),* т.ч. *C* не совпадает со множеством событий *G*, существует переход к новой конфигурации.

Лемма 2. Пусть *(C, I)* является корректной конфигурацией ARM-согласованного исполнения *G*, и *C* не совпадает со множеством событий *G*. Тогда существуют *C`* и *I`*, т.ч. есть правило перехода из *(C, I)* в *(C`, I`)*.

Идея доказательства. Если *C* не совпадает со множеством событий исполнения *G*, то множество не покрытых событий, т.ч. для каждого события из множества все po-предшествующие ему события покрыты, не пусто. Обозначим данное множество *Next*. Если в *Next* есть элемент, удовлетворяющий остальным требованиям правила покрытия, то существует соответствующий переход в новую конфигурацию. Если в этом множестве есть событие записи, то оно может быть выпущено согласно другому правилу обхода. Иначе все элементы *Next* соответствуют операциям чтения из ещё не выпущенных событий. В этом случае можно показать, что либо существует запись, которая может быть выпущена, либо в графе исполнения *G* существует цикл, который противоречит требованиям ARM-согласованности [7]. □

При симуляции обещающей машиной семантики обхода шаг обхода, являющийся покрытием события, будет соответствовать исполнению инструкции в обещающей машине (выполнение инструкции чтения, барьера памяти или сделанного ранее обещания), тогда как выпуск события будет соответствовать обещанию сообщения соответствующим потоком.

5.3. Симуляция обхода ARMv8.3 исполнения обещающей машиной

Как было замечено выше, доказательство корректности компиляции через симуляцию является индукционным доказательством по исполнению целевой машины. Важным элементом такого доказательства является отношение симуляции на состояниях соответствующих машин. Это отношение должно выполняться для начальных состояний, что соответствует базе индукции, а

3 Формально, событие *e* является зависимым от записи другого потока *w*, если существует путь в графе исполнения *G* от *w* к *e* такой, что первое ребро данного пути является ребром отношения rfe, а все остальные ребра являются ребрами отношения dob (dependency ordered before), определение которого приведено в [7].

также должно сохраняться при индукционном переходе. Из того, что отношение выполняется для финального состояния целевой машины, должно следовать заключение корректности компиляции, в нашем случае совпадение состояний памяти обещающей машины и ARM-согласованного исполнения.

Отношение симуляции *Inv* (инвариант, invariant) между состоянием обещающей машины *S* и корректной конфигурацией обхода *(C, I)* ARM-согласованного исполнения *G* выполняется, если память обещающей машины соответствует множеству выпущенных событий *I*, и для каждого события, которое может быть покрыто в текущей конфигурации, существует соответствующая инструкция в программе[4].

Более подробно, должны выполняться три следующих условия:

1. Для любого события записи *w* из множества выпущенных событий *I* существует сообщение *m* в памяти *S*, т.ч. *m* относится к той же локации, что и *w*, записывает то же значение и метка времени сообщения совпадает с порядковым номером *w* в отношении со (coherence order). При этом, если *w* также является покрытым (т. е. является элементом *C*), то сообщение *m* является выполненным, иначе – *m* еще не выполненное обещание.
2. Аналогично, для любого сообщения *m* из памяти *S* существует соответствующее ему событие во множестве выпущенных событий *I*.
3. Для цепочки событий каждого потока в исполнении *G* существует соответствующая серия переходов в графе потока управления программы. Кроме того, каждый поток обещающей машины уже выполнил часть цепочки, которая относится к покрытым событиям из *C*.

Имея отношение симуляции, нужно показать, что оно сохраняется при индукционном переходе. Из него напрямую следует утверждение теоремы 1, т. е. корректность компиляции.

Лемма 3. Пусть выполняется отношение симуляции *Inv(S, C, I, G)* и существует переход из конфигурации *(C, I)* в конфигурацию *(C`, I`)*, то существует шаг обещающей машины из состояния *S* в такое состояние *S`*, что выполняется *Inv(S`, C`, I`, G)*.

Идея доказательства. Переход из конфигурации *(C, I)* в конфигурацию *(C`, I`)* может быть либо покрытием события, либо выпуском события. Рассмотрим данные варианты и начнем с покрытия.

Если переход в *(C`, I`)* является покрытием некоторого события *e*, то $C` = C \cup \{e\}$ и $I` = I$. Согласно отношению симуляции, поток обещающей машины, к которому относится событие *e*, может сделать соответствующий переход. Если *e* является операцией чтения, то согласно требованию на покрываемость

4 В отношении симуляции присутствуют и другие требования, относящиеся к компонентам обещающей машины, которые были опущены в данной статье для краткости.

события оно читает из события записи, которое уже выпущено, т. е. является элементом *I*. Согласно отношению симуляции, в памяти обещающей машины есть соответствующее сообщение, и оно может быть прочитано при переходе от *S* к *S`*. Если *e* является операцией записи, то оно является выпущенным согласно условию на покрытие, а соответствующее сообщение находится в памяти обещающей машины, т.о. при переходе от *S* к *S`* обещающая машина выполняет связанное обещание. Если *e* является операцией барьера, то, согласно условию выпускаемости события, не существует po-последующего события записи, которое выпущено. Из этого следует, что соответствующий поток обещающей машины не имеет невыполненных обещаний и может выполнить следующую инструкцию барьера памяти. Таким образом, обещающая машина может симулировать переход покрытия.

Если переход в *(C`, I`)* является выпуском некоторого события записи *w*, то $C` = C$ и $I` = I \cup \{w\}$. Согласно требованию на выпускаемость события *w*, все po-предшествующие барьеры памяти покрыты, а значит, по отношению симуляции сообщение, соответствующее *w*, может быть обещано в рамках обещающей машины. Отношение симуляции при этом переходе, очевидно, сохранится. □

6. Заключение

Мы доказали корректность компиляции из подмножества обещающей модели памяти в модель памяти ARMv8.3. Для этого мы использовали новый метод обхода графов исполнений, что позволило нам базировать доказательство на симуляции. Данный метод накладывает меньше ограничений на целевую платформу по сравнению с методом, использованном для доказательства корректности компиляции в модели x86-TSO и Power в [13]. Последнее означает, что предложенный метод потенциально проще переиспользовать в последующих доказательствах корректности из обещающей модели памяти.

В нашей работе было рассмотрено базовое подмножество обещающей машины, состоящее из расслабленных чтений и записей, а также приобретающих и высвобождающих барьеров памяти. Нашей следующей задачей является расширение доказательства до полной обещающей модели. Для этого нужно будет рассмотреть более строгие варианты чтений и записей, инструкции атомарного чтения и записи (read-modify-write), частным случаем которых является операция чтения с возможным обменом (CAS, compare-and-swap), и полные барьеры памяти (SC fences). Далее мы опишем, какие модификации в доказательстве нужны для поддержки упомянутых инструкций.

Для того, чтобы добавить более строгое чтение, а именно приобретающее чтение (acquire read), в доказательство не нужно вносить никаких изменений. Это следует из того, что для обещающей модели памяти была доказана корректность оптимизации, которая заменяет приобретающее чтение на

приобретающий барьер памяти и расслабленное чтение, а данные инструкции рассмотрены в нашем доказательстве.

Для поддержки более строго варианта инструкции записи (высвобождающей записи, release write) нужно показать, что в обходе ARM-согласованного исполнения при выпуске события, соответствующего высвобождающей записи, это событие может быть сразу же покрыто. Данное утверждение нужно, поскольку обещающая машина должна одновременно пообещать и выполнить обещание, относящееся к высвобождающей записи. При этом само утверждение может быть тривиальным образом показано.

Аналогичное утверждение о покрытии сразу двух событий должно быть доказано и для поддержки инструкций атомарного чтения и записи (CAS), т. к. в модели ARMv8.3 такие инструкции представляются двумя непосредственно следующими друг за другом событиями, тогда как в обещающей модели памяти атомарная инструкция выполняется за один шаг.

Полные барьеры памяти (SC fences) в обещающей модели памяти требуют некоторого фиксированного полного порядка на их исполнении. Для поддержки таких барьеров в доказательстве нужно расширить отношение симуляции, чтобы в нем появилась глобальная компонента, отражающая данный порядок. Также в обход ARM-согласованного исполнения нужно добавить ограничение на покрытие полных барьеров, гарантирующее их покрытие в том же порядке, в котором они могут быть исполнены в обещающей модели памяти.

Список литературы

[1]. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

[2]. Aho A.V., Sethi R., Ullman J.D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[3]. Sewell P., Sarkar S., Owens S., Zappa Nardelli F., and Myreen M. O. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.

[4]. Alglave J., Maranget L., and Tautschnig M.. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014. doi:10.1145/2627752.

[5]. Sarkar S., Sewell P., Alglave J., Maranget L., and Williams D. Understanding POWER multiprocessors. In PLDI 2011, pages 175–186. ACM, 2011. doi:10.1145/1993498.1993520.

[6]. Flur S., Gray K. E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., and Sewell P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In POPL 2016, pages 608–621. ACM, 2016. doi:10.1145/2837614.2837615.

[7]. Pulte C., Flur S., Deacon W., French J., Sarkar S., and Sewell P. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. URL: http://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf. July 2017.

[8]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency. In POPL 2011, pages 55–66. ACM, 2011. doi:10.1145/1925844.1926394.

[9]. Manson J., Pugh W., and Adve S. V. The Java memory model. In POPL 2005, pages 378–391. ACM, 2005. doi:10.1145/1040305.1040336.

[10]. Трифанов В.Ю. Динамическое обнаружение состояний гонки в многопоточных Java-программах. Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2013, 112 стр.

[11]. Ševčík J. and Aspinall D. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22nd European conference on Object-Oriented Programming*(ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 27-51. doi: 10.1007/978-3-540-70592-5_3.

[12]. Boehm H.-J. and Demsky B. Outlawing ghosts: Avoiding out-of-thin-air results. In MSPC 2014, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.

[13]. Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. A promising semantics for relaxed-memory concurrency. In POPL 2017. ACM, 2017. doi:10.1145/3009837.3009850.

[14]. Lahav O. and Vafeiadis V. Explaining relaxed memory models with program transformations. FM 2016. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.

[15]. Обещающая компиляция в ARMv8.3. Формальное доказательство. Октябрь 2017. URL: https://podkopaev.net/armpromise

[16]. Leroy X. A formally verified compiler back-end. J. Autom. Reasoning, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[17]. Kumar R., Myreen M. O., Norrish M., and Owens S.. CakeML: A verified implementation of ML. In POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 179–191. ACM Press, January 2014. doi:10.1145/2535838.2535841.

[18]. Sevcík J., Vafeiadis V., Zappa Nardelli F., Jagannathan S., and Sewell P. Relaxed-memory concurrency and verified compilation. In Proceedings of the 38th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 43–54, 2011. doi:10.1145/1926385.1926393.

[19]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency: The post-Rapperswil model. technical report n3132, iso iec jtc1/sc22/wg21. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf.

[20]. Batty M., Memarian K., Nienhuis K, Pichon-Pharabod J., and Sewell P. The problem of programming language concurrency semantics. In ESOP, volume 9032 of LNCS, pages 283–307. Springer, 2015. doi:10.1007/978-3-662-46669-8_12.

[21]. Podkopaev A., Lahav O., and Vafeiadis V. Promising compilation to ARMv8 POP. In ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22.

Promising Compilation to ARMv8.3

[1,2] *A.V.Podkopaev <apodkopaev@2009.spbu.ru>*
[3] *O.Lahav <orilahav@tau.ac.il>*
[4] *V.Vafeiadis <viktor@mpi-sws.org>*
[1] *St. Petersburg University,*
199034, Russia, St. Petersburg, Universitetskaya emb. 7–9
[2] *JetBrains Research,*
199034, Russia, St. Petersburg, Universitetskaya emb. 7–9-11/5A
[3] *Tel-Aviv University, 39040, Israel, Tel Aviv 69978*
[4] *Max Planck Institute for Software Systems,*
67663, Germany, Kaiserslautern, Paul-Ehrlish str. G26

Abstract. Concurrent programs have behaviors, which cannot be explained by interleaving execution of their threads on a single processing unit due to optimizations, which are performed by modern compilers and CPUs. How to correctly and completely define a semantics of a programming language, which accounts for the behaviors, is an open research problem. There is an auspicious attempt to solve the problem – promising memory model. To show that the model might be used as a part of an industrial language standard, it is necessary to prove correctness of compilation from the model to memory models of target processor architectures. In this paper, we present a proof of compilation correctness from a subset of promising memory model to an axiomatic ARMv8.3 memory model. The subset contains relaxed memory accesses and release and acquire fences. The proof is based on a novel approach of an execution graph traversal.

Keywords: concurrency; compilation correctness; weak memory models.

DOI: 10.15514/ISPRAS-2017-29(5)-9

For citation: Podkopaev A.V., Lahav O., Vafeiadis V. Promising Compilation to ARMv8.3. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 149-164 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-9

References

[1]. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.
[2]. Aho A.V., Sethi R., Ullman J.D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[3]. Sewell P., Sarkar S., Owens S., Zappa Nardelli F., and Myreen M. O. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.
[4]. Alglave J., Maranget L., and Tautschnig M.. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014. doi:10.1145/2627752.
[5]. Sarkar S., Sewell P., Alglave J., Maranget L., and Williams D. Understanding POWER multiprocessors. In PLDI 2011, pages 175–186. ACM, 2011. doi:10.1145/1993498.1993520.

[6]. Flur S., Gray K. E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., and Sewell P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In POPL 2016, pages 608–621. ACM, 2016. doi:10.1145/2837614.2837615.
[7]. Pulte C., Flur S., Deacon W., French J., Sarkar S., and Sewell P. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. URL: http://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf. July 2017.
[8]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency. In POPL 2011, pages 55–66. ACM, 2011. doi:10.1145/1925844.1926394.
[9]. Manson J., Pugh W., and Adve S. V. The Java memory model. In POPL 2005, pages 378–391. ACM, 2005. doi:10.1145/1040305.1040336.
[10]. Trifanov V.Yu. Dynamic detection of race conditions in multithreaded Java programs. Dissertation, ITMO University, 2013, p. 112 (in Russian).
[11]. Ševčík J. and Aspinall D. On Validity of Program Transformations in the Java Memory Model. In Proceedings of the 22nd European conference on Object-Oriented Programming(ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 27-51. doi: 10.1007/978-3-540-70592-5_3.
[12]. Boehm H.-J. and Demsky B. Outlawing ghosts: Avoiding out-of-thin-air results. In MSPC 2014, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.
[13]. Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. A promising semantics for relaxed-memory concurrency. In POPL 2017. ACM, 2017. doi:10.1145/3009837.3009850.
[14]. Lahav O. and Vafeiadis V. Explaining relaxed memory models with program transformations. FM 2016. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.
[15]. [Promising compilation to ARMv8.3. A formal proof]. October 2017. URL: https://podkopaev.net/armpromise
[16]. Leroy X. A formally verified compiler back-end. J. Autom. Reasoning, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.
[17]. Kumar R., Myreen M. O., Norrish M., and Owens S.. CakeML: A verified implementation of ML. In POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 179–191. ACM Press, January 2014. doi:10.1145/2535838.2535841.
[18]. Sevcík J., Vafeiadis V., Zappa Nardelli F., Jagannathan S., and Sewell P. Relaxed-memory concurrency and verified compilation. In Proceedings of the 38th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 43–54, 2011. doi:10.1145/1926385.1926393.
[19]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency: The post-Rapperswil model. technical report n3132, iso iec jtc1/sc22/wg21. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf.
[20]. Batty M., Memarian K., Nienhuis K, Pichon-Pharabod J., and Sewell P. The problem of programming language concurrency semantics. In ESOP, volume 9032 of LNCS, pages 283–307. Springer, 2015. doi:10.1007/978-3-662-46669-8_12.
[21]. Podkopaev A., Lahav O., and Vafeiadis V. Promising compilation to ARMv8 POP. In ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22.