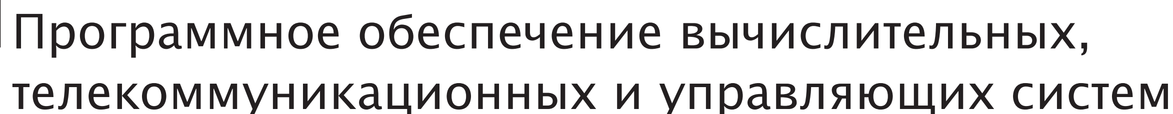


Программное обеспечение вычислительных, телекоммуникационных и управляющих систем

DOI: 10.18721/JCSTCS.10405
УДК 004.4'422

О КОРРЕКТНОСТИ КОМПИЛЯЦИИ ПОДМНОЖЕСТВА ОБЕЩАЮЩЕЙ МОДЕЛИ ПАМЯТИ В АКСИОМАТИЧЕСКУЮ МОДЕЛЬ ARMv8.3

А.В. Подкопаев[1], О. Лахав[2], В. Вафеядис[3]

[1]Санкт-Петербургский государственный университет,
Санкт-Петербург, Российская Федерация;
[2]Тель-Авивский университет, Тель-Авив, Израиль;
[3]Институт имени Макса Планка: Программные системы, Кайзерслаутерн, Германия

«Обещающая» модель памяти является перспективным решением проблемы задания семантики многопоточности в контексте императивных языков программирования, таких как C/C++ и Java. Естественным требованием, которое ставится перед моделью памяти языка программирования, является наличие эффективной и корректной схемы компиляции для распространенных процессорных архитектур. Ранее для обещающей модели была показана корректность компиляции в архитектуры x86, Power и для операционной модели памяти ARMv8 POP. В статье приведено доказательство корректности компиляции в аксиоматическую модель ARMv8.3. В доказательстве использован новый метод обхода исполнений аксиоматических моделей памяти. Этот метод является более общим, чем использованные ранее подходы, и может использоваться в последующих доказательствах корректности компиляции из обещающей модели памяти.

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

Ссылка при цитировании: Подкопаев А.В., Лахав О., Вафеядис В. О корректности компиляции подмножества обещающей модели памяти в аксиоматическую модель ARMv8.3 // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 2017. Т. 10. № 4. С. 51–69. DOI: 10.18721/JCSTCS.10405

ON COMPILATION CORRECTNESS FOR A SUBSET OF A PROMISING MEMORY MODEL TO THE ARMv8.3 MEMORY MODEL

A.V. Podkopaev [1], O. Lahav [2], V. Vafeiadis [3]

[1]St. Petersburg State University, St. Petersburg, Russian Federation;
[2]Tel Aviv University, Tel Aviv, Israel;
[3]Max Planck Institute for Software Systems, Kaiserslautern, Germany

A ‘promising’ memory model is an auspicious solution to the problem of defining semantics for an imperative language with concurrency, such as C/C++ or Java.

An essential requirement for such a memory model is the existence of an effective and correct compilation scheme from the language to its target platforms. There are compilation correctness proofs from the promising model to x86 and Power as well as to an operational model ARMv8 POP. This paper presents such proof for an axiomatic memory model for ARMv8.3. In the proof, we use a new method of execution traversal, which might be used in other compilation correctness proofs for the promising memory model.

Keywords: concurrency; compilation correctness; weak memory models; ARM; semantics.

Citation: Podkopaev A.V., Lahav O., Vafeiadis V. On compilation correctness for a subset of a Promising memory model to the ARMv8.3 memory model. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems, 2017, Vol. 10, No. 4, Pp. 51–69. DOI: 10.18721/JCSTCS.10405

1. Введение

Параллельные вычисления давно являются естественной частью прикладного программирования. Несмотря на это, до сих пор ведется активная работа по разработке формальных семантик для языков программирования и процессоров с многопоточностью, или *моделей памяти* [4, 7, 9, 13, 17, 20]. Под моделью памяти можно понимать функцию, которая по программе выдает множество возможных *поведений*, т. е. состояний системы, исполняющей программу, после выполнения программы. Чаще всего речь идет об оперативной памяти и значении регистров.

Наиболее известная модель памяти для многопоточных программ – модель *последовательной консистентности* (sequential consistency – SC) [14]. Любое поведение программы в данной модели может быть получено как некоторое попеременное исполнение команд разных потоков на одноядерном процессоре. К сожалению, такая модель памяти не способна описывать поведения, которые наблюдаются у программ, оптимизированных компилятором и запущенных на современных процессорах. Так, аналог следующей программы, написанный на C++, обработанный компилятором gcc и запущенный на процессоре семейства x86, может завершиться с результатом $a = b = 0$[1]:

$$\begin{array}{r\|l} [x] := 1; & [y] := 1; \\ a := [y]; //0 & b := [x]; //0 \end{array} \qquad \text{(SB)}$$

Данное поведение, которое в литературе именуется буферизацией записи (store buffering – SB) [23], не может быть получено поочередным исполнением потоков, т. к. при таком исполнении одна из записей ($[x] := 1$ и $[y] := 1$) должна случиться раньше любого из чтений ($a := [y]$ и $b := [x]$). Поведение параллельной программы, которое выходит за рамки последовательной консистентности, называют слабым. В свою очередь, слабая модель памяти – это модель, допускающая слабые поведения.

Первая причина появления слабых поведений у многопоточных программ – компиляторные оптимизации. Например, компилятор gcc с включенным режимом оптимизации переставляет независимые обращения к памяти в рамках планирования кода. Соответствующая перестановка наблюдается для аналогов программы SB. Второй причиной является оптимизирующее поведение современных процессоров: процессоры могут исполнять инструкции не по порядку, а обращения к памяти могут быть переупорядочены за счет использования локальной памяти ядра.

Компиляторные и процессорные оптимизации стали широко применяться задолго до появления многопоточных процессоров; они дают существенный прирост производительности. Как следствие, большинство экспертов сходится во мнении, что лучше разрешить слабые поведения программ, чем запретить оптимизации.

В последние годы были созданы сла-

[1] Здесь и во всех последующих примерах мы предполагаем, что x, y являются различными адресами в памяти, в которые изначально записаны нули, а a, b являются локальными переменными (регистрами).

бые модели памяти для самых распространенных архитектур: x86 [23], Power [3, 21] и ARM [9, 20]. Эти модели разработаны в сотрудничестве с компаниями, создающими данные архитектуры. Они описывают не только поведения, наблюдаемые с помощью тестирования существующих процессоров, но и оставляют пространство для оптимизаций, планируемых в новых версиях архитектур.

Существуют слабые модели и для распространенных языков программирования, таких как Java [17] и C/C++ [7]. Данные модели входят в соответствующие стандарты языков. Источником слабых поведений в контексте моделей памяти языков программирования являются оптимизации, которые создатели стандарта хотят разрешить для реализации в компиляторах, а также слабые поведения процессоров, которые являются целевыми для языков и их платформ.

У существующих моделей памяти Java и C/C++ имеются недостатки: разрешение «значений из воздуха» (out-of-thin-air values) [8], некорректность широко применяемых на практике оптимизаций [24] и слишком строгие требования к семантике процессоров в контексте используемых схем компиляции [13, 16]. Недавно была представлена новая модель памяти [10], т. н. «обещающая» (promising) модель памяти, призванная решить упомянутые проблемы.

Потенциально, обещающая модель может стать частью стандартов языков Java и C/C++; обсуждение этого уже начинается в соответствующих комитетах. Для того чтобы обещающая модель смогла стать частью стандартов Java и C/C++, должна быть доказана корректность существующих (эффективных) схем компиляции [2] из обещающей модели в модели памяти основных процессорных архитектур (x86, Power, ARM).

Под корректностью компиляции схемы *compile* из модели *A* в модель *B* мы подразумеваем следующее:

$$\forall Prog \in A. [\![compile(Prog)]\!]_B \subseteq [\![Prog]\!]_A,$$

а именно, что для любой программы *Prog* в языке определения модели *A* и ее скомпилированной версии *compile*(*Prog*) в языке определения модели *B* выполняется, что любое поведение *compile*(*Prog*) в *A* также является поведением *Prog* в *A*. Это естественное требование, позволяющее программисту быть уверенным в том, что написанная им программа работает только так, как описано в стандарте языка ее реализации, и никак иначе.

В работе [10] доказана корректность компиляции из обещающей модели в модели x86-TSO [23] и Power [3]; в [19] – корректность компиляции из подмножества обещающей модели в операционную модель памяти ARMv8 POP [9]. В апреле 2017 года была предложена новая модель памяти для архитектуры ARM, модель ARMv8.3 [1]. Как следствие, актуальна задача доказательства корректности компиляции в новую модель ARMv8.3. Решению этой задачи посвящена данная статья.

В статье показана корректность компиляции для подмножества обещающей модели без сертификации [10] в модель ARMv8.3. Рассматриваемое подмножество состоит из ослабленных (relaxed) обращений памяти (записей и чтений), высвобождающих (release) и приобретающих (acquire) барьеров памяти. Несмотря на то, что мы не доказываем корректность компиляции для всей модели, мы уверены, что доказательство может быть расширено, что является нашей дальнейшей работой. Но в связи с тем, что текущее доказательство достаточно объемное и сложное, а результат является базовым для дальнейшей работы, мы представляем доказательство корректности компиляции для подмножества модели.

Распространенным способом показать корректность компиляции из абстрактной машины *A* в машину *B* является доказательство симуляции между ними [18]. Для этого определяется отношение симуляции, связывающее состояния машин, и доказывается, что если текущие состояния машин *A* и *B* связаны отношением симуляции, и машина *B* делает шаг, то машина *A* может сделать ноль и более шагов так, чтобы новые состояния машин были также связаны отношением симуляции. Напрямую данный подход не применим для

доказательства корректности компиляции из обещающей модели в модель ARMv8.3, т. к. модель памяти ARMv8.3 является аксиоматической, т. е. семантика конкретной программы в этой модели есть множество графов, каждый из которых представляет некоторый конкретный запуск программы. Это означает, что в модели ARMv8.3 запуск программы определяется сразу и целиком, а не по шагам, как это происходит в обещающей модели, заданной в виде абстрактной машины. Чтобы преодолеть данное ограничение, мы разработали способ обхода графов модели ARMv8.3, который представлен в виде операционной семантики. Этот обход и используется при доказательстве симуляции между обещающей и ARMv8.3 моделями. Данный метод может применяться для доказательства корректности компиляции из обещающей в аксиоматические модели памяти и является более общим, чем использованные ранее методы (см. раздел 9), базирующиеся на особых свойствах целевых аксиоматических моделей.

2. Обещающая модель на примерах

В этом разделе мы приведем неформальное описание обещающей модели памяти [10] на нескольких примерах.

2.1. Базовые элементы обещающей модели. Рассмотрим программу MP (передача сообщения, message passing):

$$\begin{array}{l} [x] := 1; \\ [y] := 1; \end{array} \left\| \begin{array}{l} a := [y]; \ //1 \\ b := [x]; \ //0 \end{array} \right. \qquad \text{(MP)}$$

Эта программа является упрощенным вариантом шаблона, используемого для передачи данных между потоками. Первый поток записывает данные в локацию x и потом выставляет флаг (локация y), что данные подготовлены; в свою очередь, второй поток проверяет флаг, а потом читает данные. Модель последовательной консистентности гарантирует, что если второй поток увидел, что флаг выставлен ($a = 1$), то он увидит и подготовленные данные ($b = 1$). Данное заключение неверно для многих слабых моделей, в том числе для обещающей и ARMv8.3 моделей. Рассмотрим, как слабое поведение $a = 1$, $b = 0$ достигается при исполнении MP в обещающей модели памяти.

Обещающая модель памяти задана операционным способом. Это означает, что существует некоторая абстрактная машина, связанная с моделью, которая исполняет программы по шагам, и множество всех конечных состояний этой машины для некоторой программы является семантикой программы в рамках данной модели. Далее мы будем использовать термин *обещающая машина* для обозначения упомянутой абстрактной машины.

Состояние обещающей машины включает в себя множество сообщений, которое называется *памятью*. Мы считаем, что память перед исполнением программы содержит по сообщению со значением ноль для каждой локации. Так, для приведенной выше программы начальная память будет состоять из двух сообщений:

$$M_{init} = \{\langle x : 0 @ 0\rangle, \langle y : 0 @ 0\rangle\}.$$

Сообщение – это тройка из локации, значения и метки времени (timestamp, в приведенном примере – 0 после @). Метки времени используются для упорядочивания сообщений, связаных с одной и той же локацией.

Теперь покажем некоторое исполнение MP, которое приводит к $a = 1$, $b = 0$. Сначала левый поток программы MP выполняет обе записи, после чего память обещающей машины содержит четыре сообщения:

$$M = \{\langle x : 0 @ 0\rangle, \langle y : 0 @ 0\rangle, \\ \langle x : 1 @ 5\rangle, \langle y : 1 @ 3.5\rangle\}.$$

Заметим, что метки времени новых сообщений должны быть больше, чем метки инициализирующих сообщений, но между собой они никак не связаны и могли бы совпасть — выбор меток времени в обещающей машине недетерминирован. Так, конкретное значение метки времени сообщения не имеет значения, важно лишь соотношение между метками сообщений, относящихся к той же локации.

Следующим шагом в выбранном запуске, приводящем к $a = 1$, $b = 0$, правый поток выполняет чтение из локации y. Обещающая машина разрешает потоку произвести чтение из любого сообще-

ния локации y в памяти, $\langle y:0@0\rangle$ или $\langle y:1@3.5\rangle$. Поскольку мы хотим получить $a = 1$, правый поток производит чтение из сообщения с меткой времени 3.5. Далее, если бы поток еще раз читал из локации y, ему было бы запрещено производить чтение из сообщений с меткой времени меньше 3.5. Интуитивно, если поток увидел более новое значение, то он больше не должен видеть более старое значение.

Для того чтобы обеспечить данное свойство, у каждого потока обещающей машины есть фронт – взгляд (view, viewfront) на память. Это функция, которая по локации возвращает метку времени последнего сообщения, относящегося к этой локации, которое было увидено потоком. Так, перед тем как левый поток выполнил инструкции записи, фронты левого и правого потоков указывают на изначальные сообщения:

$$T1.view_{\mathrm{cur}} = [x@0, y@0]$$
$$T2.view_{\mathrm{cur}} = [x@0, y@0].$$

После того как левый поток выполнил обе инструкции записи, он больше не может видеть более старые значения. Это гарантируется тем, что его фронт по каждой локации увеличился и стал равным метки времени соответствующего сообщения:

$$T1.view_{\mathrm{cur}} = [x@5, y@3.5]$$
$$T2.view_{\mathrm{cur}} = [x@0, y@0].$$

После того как правый поток выполнил чтение из сообщения y с меткой времени 3.5, его фронт увеличивается по y (становится равным максимуму из старого значения по y, 0, и метки времени сообщения, 3.5):

$$T1.view_{\mathrm{cur}} = [x@5, y@3.5]$$
$$T2.view_{\mathrm{cur}} = [x@0, y@3.5].$$

В программе остается невыполненной только второе чтение из локации x в правом потоке. Это чтение правый поток может совершить из сообщения с меткой времени 0, так как $T2.view_{\mathrm{cur}}(x) \le 0$.

2.2 Барьеры памяти. Как же гарантировать передачу сообщения между потоками в программе MP, а именно запретить результат $a = 1$, $b = 0$? Для этого нужно добавить в программу MP несколько *барьеров памяти* (memory fences) — специальных инструкций, которые в реальных системах запрещают некоторые оптимизации со стороны компилятора и процессора. Например, *высвобождающий* (release, rel) барьер используется для того, чтобы запретить переупорядочивание инструкций записи, а *приобретающий* (acquire, acq) — инструкций чтения. Так, версия программы MP с высвобождающим и приобретающим барьерами в левом и правом потоках соответственно не может завершиться с результатом $a = 1$, $b = 0$:

$$\begin{array}{r||l}
[x] := 1; & a := [y]; \ //1 \\
\mathrm{fence(rel)}; & \mathrm{fence(acq)}; \\
[y] := 1; & b := [x]; \ //0
\end{array} \qquad \text{(MP-rel-acq)}$$

Для поддержки барьеров памяти нужно внести некоторые изменения в память обещающей машины – у каждого сообщения появляется свой фронт. Интуитивно, фронт сообщения *msg* указывает на те сообщения в памяти, которые становятся видны потоку, который читает из сообщения *msg* и после этого выполняет приобретающий барьер. Так, после выполнения инструкций записи левым потоком, память машины выглядит следующим образом:

$$M = \{\langle x:0@0,[x@0]\rangle, \langle y:0@0,[y@0]\rangle,$$
$$\langle x:1@5,[x@5, y@0]\rangle,$$
$$\langle x:1@3.5,[x@5, y@3.5]\rangle,$$

В данном случае интерес представляет фронт второго сообщения локации y – [x@5, y@3.5]. После того как левый поток выполнит чтение из этого сообщения и исполнит приобретающий барьер, его фронт будет равен [x@5, y@3.5], что запретит ему чтение из сообщения $\langle x:0@0,[x@0]\rangle$.

Для поддержки барьеров нам также нужно добавить по два дополнительных фронта потокам – высвобождающий и приобретающий. Рассмотрим тот же запуск программы MP. Перед исполнением программы все три фронта левого потока равны [x@0, y@0]. После выполнения инструкции записи в локацию x, базовый и приобретающий фронты левого потока соответственным образом растут, тогда как высвобож-

дающий остается без изменений:

$$T1.view_{\text{cur}} = [x@5, y@0]$$
$$T1.view_{\text{acq}} = [x@5, y@0]$$
$$T1.view_{\text{rel}} = [x@0, y@0].$$

После выполнения барьера высвобождающий фронт становится равным базовому:

$$T1.view_{\text{cur}} = [x@5, y@0]$$
$$T1.view_{\text{acq}} = [x@5, y@0]$$
$$T1.view_{\text{rel}} = [x@5, y@0].$$

Теперь, когда левый поток добавляет новое сообщение локации y в память, фронт этого сообщения равен высвобождающему фронту потока, увеличенному на [y@3.5] — метку времени самого сообщения.

Перейдем к правому потоку. Его фронты также изначально равны [x@0, y@0]. После того как поток читает из сообщения $\langle y : 1@3.5, [x@5, y@3.5]\rangle$, его базовый фронт увеличивается на метку времени самого сообщения, [y@3.5], тогда как приобретающий — на фронт сообщения:

$$T2.view_{\text{cur}} = [x@0, y@3.5]$$
$$T2.view_{\text{acq}} = [x@5, y@3.5]$$
$$T2.view_{\text{rel}} = [x@0, y@0].$$

Выполнение барьера приравнивает базовый фронт к приобретающему:

$$T2.view_{\text{cur}} = [x@5, y@3.5]$$
$$T2.view_{\text{acq}} = [x@5, y@3.5]$$
$$T2.view_{\text{rel}} = [x@0, y@0].$$

После этого правый поток не может прочитать сообщение $\langle x : 0@0, [x@0]\rangle$, так как $T2.view_{\text{cur}} > 0$.

2.3 Механизм обещаний. Как мы только что убедились, обещающая модель позволяет задавать поведения многопоточной программы, выходящие за пределы простого попеременного исполнения потоков над обычной памятью. Это достигается за счет устройства памяти обещающей машины, которая может хранить более одного сообщения для каждой локации. Тем не менее этого недостаточно для выражения всех поведений, наблюдаемых на современных архитектурах. Например, следующая программа LB (load buffering) может завершиться с результатом $a = b = 1$:

$$\begin{array}{r} a := [x];\ //1 \\ [y] := 1; \end{array} \Bigg\| \begin{array}{l} b := [y];\ //1 \\ [x] := 1 \end{array} \qquad \text{(LB)}$$

Такое поведение не может наблюдаться в версии обещающей машины, описанной выше, т. к. для получения $a = b = 1$ первой инструкцией должна выполниться одна из инструкций записи. Архитектурные модели памяти, такие как ARMv8 POP [9], явным образом разрешают исполнение инструкций не по порядку (out of order execution). В обещающей модели нет такой возможности. Зато любой поток в любой момент исполнения программы может пообещать сделать запись некоторого сообщения в память в будущем. Например, первым шагом обещающей модели для программы LB может быть обещание левым потоком сделать запись в локацию y. После этого в памяти машины появляется соответствующее сообщение:

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

Теперь правый поток может выполнить свои инструкции: сначала прочитать из нового сообщения, а потом выполнить инструкцию записи обычным способом, без обещаний. После этого память будет содержать четыре сообщения:

$$M = \{\langle x : 0@0, [x@0]\rangle, \langle y : 0@0, [y@0]\rangle, \langle y : 0@0, [y@0]\rangle, \langle x : 1@2, [x@2, y@0]\rangle\},$$

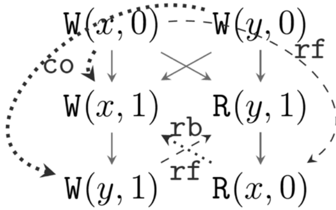
и левый поток может прочитать из сообщения $\langle x : 1@2\rangle$ и после этого выполнить обещание записать сообщение $\langle y : 1@2, [y@2]\rangle$ в память.

Как чтение из памяти потоком ограничено его базовым фронтом, так и для обещаний существует ограничение: поток не может читать из сообщений, которые он пообещал, но еще не выполнил. Для поддержания этого инварианта у каждого потока имеется компонента состояния, которая хранит множество еще не выполненных обещаний.

3. Модель ARMv8.3 на примерах

Модель памяти ARMv8.3 представляет семантику программы как множество гра-

фов, каждый из которых соответствует некоторому конкретному запуску программы [20]. Так, например, для программы MP, которую мы уже рассматривали в разделе 2, исполнение программы с результатом $a = 1$, $b = 0$ выглядит следующим образом:



В графовом представлении каждый узел представляет некоторое событие (event) — операцию над памятью. Событие может быть записью (например, W(*x*, 1)), чтением (например, R(*y*, 1)) или барьером памяти (например, F(acq)). В представленном выше графе W(*x*, 0) и W(*y*, 0) обозначают инициализацию, а остальные вершины — операции левого и правого потоков. Далее в статье мы будем использовать W, R и F как для обозначения меток конкретных событий, так и для обозначения подмножества событий соответствующего типа.

Ребра в графе используются для обозначения различных отношений между событиями. Стрелки без подписи обозначают po — отношение программного порядка (program order). Программный порядок связывает инициализирующие записи со всеми остальными событиями в графе, а также является полным порядком на событиях одного и того же потока[2].

События записи в одну и ту же локацию тотально упорядочены с помощью co — отношения согласованности (coherence order). Это отношение выполняет ту же функцию, что и метки времени в рамках обещающей машины.

Отношение rf («читает из», reads from) связывает события записи с событиями чтения, причем у каждого чтения должна существовать одна и только одна запись, с которой оно (чтение) связано данным отношением, и целевые локации и значения событий должны совпадать.

Отношение rb («читает ранее», reads before) является производным от rf и co:

$$\mathrm{rb} \triangleq \mathrm{rf}^{-1};\mathrm{co},$$

и связывает чтение *a* с записью, которая co-следует за записью, из которой читает *a*. Здесь и далее композиция отношений ‘;’ задается так:

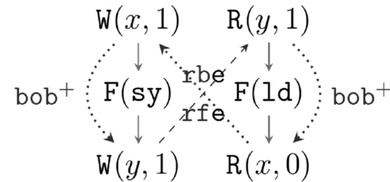
$$A;B \triangleq \{\langle a,b\rangle \mid \exists c.\langle a,c\rangle \in A \wedge \langle c,b\rangle \in B\}.$$

Как и в случае обещающей семантики, добавление в программу MP барьеров памяти запрещает результат $a = 1$ и $b = 0$:

$$\begin{array}{l||l} [x] := 1; & \mathrm{a} := [y]; //1 \\ \mathrm{fence(sy)}; & \mathrm{fence(ld)}; \\ [y] := 1; & b := [x]; //0 \end{array} \qquad \text{(MP-sy-ld)}$$

Данная программа не соответствует один в один программе MP-rel-acq из раздела 2, но является результатом компиляции MP-rel-acq в ассемблер ARMv8.3.

Каким образом модель ARMv8.3 запрещает поведение $a = 1$ и $b = 0$ для программы MP-sy-ld? Рассмотрим граф, который соответствует данному поведению:



В этом графе мы опустили инициализирующие записи и добавили отношение барьеров bob (barrier-ordered-before), заданное так:

$$\mathrm{bob} \triangleq \mathrm{po};[\mathrm{F(sy)}] \cup [\mathrm{F(sy)}];\mathrm{po} \cup [\mathrm{R}];\mathrm{po};[\mathrm{F(ld)}] \cup [\mathrm{F(ld)}];\mathrm{po},$$

где $[A] \triangleq \{(a,a) \mid a \in A\}$. Кроме того, мы заменили ребра отношений rf и rb ребрами отношений rfe и rbe, где e (external) означает, что соответствующее отношение связывает только события разных потоков.

В приведенном графе есть цикл из ребер bob, rfe и rbe, что противоречит одной из аксиом, которая используется в модели ARMv8.3 для определения ARM-согласованного исполнения (подробнее в

---

[2] Программный порядок транзитивен, но в приведенном графе мы оставили только непосредственные ребра.

разделе 6). Таким образом граф не является ARM-согласованным, и поведение $a = 1$ и $b = 0$ запрещено для программы MP-sy-ld в рамках модели.

4. Структура доказательства корректности компиляции

Центральным результатом данной работы является доказательство следующей теоремы.

Теорема 4.1. Для любых программы *Prog*, результата ее 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.

Здесь под «финальным состоянием памяти» мы понимаем значение последних записей в локации, т. е. записей с наибольшими метками времени, в случае обещающей семантики, и событий-максимумов по отношению частичного порядка G.co, в случае ARMv8.3. Обещающая и ARMv8.3 модели памяти заданы в существенно разных стилях и имеют множество концептуальных отличий. Для доказательства теоремы 4.1 необходимо преодолеть данные различия.

Главным отличием является то, что обещающая модель памяти представлена операционно, в терминах шагов некоторой абстрактной машины, тогда как аксиоматическая модель памяти ARMv8.3 задает семантику конкретного исполнения программы в виде графа, для которого выполняются определенные свойства. Для решения данной проблемы мы вводим операционную семантику обхода графа ARM-согласованного исполнения, которому обещающая машина может следовать (раздел 7).

Другим существенным отличием является то, как модели запрещают «плохие» поведения программ: например, чтение потоком старой записи в x, если поток уже читал из более новой записи в x. Для этого модель памяти ARMv8.3 использует аксиомы, которые проверяют отсутствие циклов в графе, тогда как в обещающей модели для этого используются фронты потоков и сообщений. Для того чтобы обойти описанное различие между моделями, мы вводим дополнительные отношения на вершинах ARM согласованного исполнения, которые тесно связаны с фронтами обещающей модели (раздел 8). Используя описанные выше идеи, мы доказываем корректность компиляции между обещающей и ARMv8.3 моделями с помощью симуляции (раздел 8).

5. Формальное определение обещающей модели

В этом разделе мы формально описываем подмножество обещающей модели памяти [10].

Состояние обещающей машины $\mathrm{State}_{\mathrm{Promise}}$ является парой $\langle \mathcal{TS}, M \rangle$. При этом $\mathrm{M} \subset \mathrm{Msg}$ – это память машины, являющаяся множеством сообщений вида $\langle \ell : \mathrm{val} @ \tau >, \mathrm{view} \rangle$: Msg, состоящих из целевой локации, ℓ: Loc, значения, val : Val, метки времени, τ: Time = Q, и фронта сообщения, view : V = Loc → Time. $\mathcal{TS}$: Tid → TS — это функция, которая по идентификатору потока возвращает его состояние. Состояние потока, TS : TS, является тройкой $\langle \sigma, V, \mathrm{P} \rangle$; σ – локальное состояние потока в рамках описанной ниже помеченной системы переходов (labeled transition system – LTS); $V = \langle view_{cur}, view_{acq}, view_{rel} \rangle$: $V \times V \times V$ — базовый, приобретающий и высвобождающий фронты потока; $\mathrm{P} \subset \mathrm{Msg}$ – множество сообщений, которые были обещаны потоком, но еще не выполнены.

Как было отмечено выше, обещающая модель памяти задана на помеченной системе переходов, а не на программах непосредственно. Сами метки в системе переходов могут быть: (1) операцией чтения из локации ℓ значения $\nu(\mathrm{R}(\ell, \nu))$; (2) операцией записи в локацию ℓ значения $\nu(\mathrm{W}(\ell, \nu))$; (3) барьером памяти с модификатором fmod (F(fmod)) или (4) внутреннему переходу (ϵ). Последний тип описывает действия потока, которые не затрагивают память, например, присваивание в локальную переменную и исполнение оператора условного перехода.

Перед исполнением любой программы память обещающей машины состоит из записей во все локации, $\mathrm{M}^{\mathrm{init}} \triangleq \{\langle \ell : 0 @ 0, view^{\mathrm{init}} \mid \ell \in \mathrm{Loc}\}$, где

(READ-HELPER)
$$\frac{\mathsf{cur}(\ell) \le t \qquad \mathsf{cur}' = \mathsf{cur} \sqcup [\ell@\tau] \qquad \mathsf{acq}' = \mathsf{acq} \sqcup \mathsf{cur}'}{\langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle \xrightarrow{\mathtt{R}(\ell,\tau,R)} \langle \mathsf{cur}', \mathsf{acq}', \mathsf{rel} \rangle}$$

(WRITE-HELPER)
$$\frac{\mathsf{cur}(\ell) < t \qquad R = \mathsf{rel} \sqcup [\ell@\tau] \qquad \mathsf{cur}' = \mathsf{cur} \sqcup [\ell@\tau] \qquad \mathsf{acq}' = \mathsf{acq} \sqcup \mathsf{cur}'}{\langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle \xrightarrow{\mathtt{W}(\ell,\tau,R)} \langle \mathsf{cur}', \mathsf{acq}', \mathsf{rel} \rangle}$$

(READ)
$$\frac{\langle \ell : v@\tau, R \rangle \in M \qquad \sigma \xrightarrow{\mathtt{R}(\ell,v)} \sigma' \qquad \mathcal{V} \xrightarrow{\mathtt{R}(\ell,\tau,R)} \mathcal{V}'}{\langle\langle \sigma, \mathcal{V}, P \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}', P \rangle, M\rangle}$$

(WRITE)
$$\frac{\sigma \xrightarrow{\mathtt{W}(\ell,v)} \sigma' \qquad \mathcal{V} \xrightarrow{\mathtt{W}(\ell,\tau,R)} \mathcal{V}' \qquad m = \langle \ell : v@\tau, R \rangle \qquad \forall v', view'.\ \langle \ell : v'@\tau, view' \rangle \notin M \qquad M' = M \cup \{m\} \qquad P' = P \setminus \{m\}}{\langle\langle \sigma, \mathcal{V}, P \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}', P' \rangle, M'\rangle}$$

(ACQ-FENCE)
$$\frac{\sigma \xrightarrow{\mathtt{F(acq)}} \sigma'}{\langle\langle \sigma, \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle, P \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \langle \mathsf{acq}, \mathsf{acq}, \mathsf{rel} \rangle, P \rangle, M\rangle}$$

(REL-FENCE)
$$\frac{\sigma \xrightarrow{\mathtt{F(rel)}} \sigma'}{\langle\langle \sigma, \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle, \emptyset \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \langle \mathsf{cur}, \mathsf{acq}, \mathsf{cur} \rangle, \emptyset \rangle, M\rangle}$$

(SILENT)
$$\frac{\sigma \xrightarrow{\epsilon} \sigma'}{\langle\langle \sigma, \mathcal{V}, P \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}, P \rangle, M\rangle}$$

(PROMISE)
$$\frac{\forall v, view.\ \langle m.\ell : v@m.\tau, view \rangle \notin M \qquad P' = P \cup \{m\} \qquad M' = M \cup \{m\} \qquad \forall \ell.\ \exists v, view.\ \langle \ell : v@m.\mathtt{view}(\ell), view \rangle \in M'}{\langle\langle \sigma, \mathcal{V}, P \rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma, \mathcal{V}, P' \rangle, M'\rangle}$$

(GLOBAL)
$$\frac{\langle TS, M \rangle \xrightarrow[\text{Promise } tid]{} \langle TS', M' \rangle}{\langle \mathcal{TS}[tid \mapsto TS], M \rangle \xrightarrow[\text{Promise}]{} \langle \mathcal{TS}[tid \mapsto TS'], M' \rangle}$$

Переходы обещающей машины

$view^{\mathrm{init}} \triangleq \lambda\ell.0$. В целом, изначальное состояние машины выглядит следующим образом:

$$\mathrm{p}^{\mathrm{init}} \triangleq \langle \lambda tid. \langle \sigma = \sigma^{\mathrm{init}},$$
$$V = \langle view^{\mathrm{init}}, view^{\mathrm{init}}, view^{\mathrm{init}} \rangle, \mathrm{P} = \varnothing \rangle, \mathrm{M}^{\mathrm{init}} \rangle.$$

Перейдем к описанию шагов исполнения в обещающей машине (см. рисунок). Единственным правилом, оперирующим на всем состоянии машины, является правило (GLOBAL). В версии обещающей машины без сертификации [10] оно не несет смысловой нагрузки, а только позволяет записать остальные правила локально для потоков (см. описание ниже).

Выполнение приобретающего барьера (ACQ-FENCE) делает базовый фронт потока $view^{\mathrm{cur}}$ равным приобретающему фронту потока $view^{\mathrm{acq}}$, где приобретающий фронт – объединение фронтов сообщений[3], прочитанных потоком к этому моменту, и меток времени записей, произведенных потоком.

Выполнение высвобождающего барьера (REL-FENCE) делает высвобождающий фронт потока $view^{\mathrm{rel}}$ равным базовому фронту потока $view^{\mathrm{cur}}$. Как результат, фронты сообщений, которые поток добавит в память после выполнения высвобождающего барьера, будут содержать информацию о записях, произведенных потоком

[3] Под объединением фронтов A и B мы понимаем фронт $C \triangleq \lambda\ell.\ \max(A(\ell), B(\ell))$.

до выполнения высвобождающего барьера. Кроме того, переход, соответствующий высвобождающему барьеру, имеет дополнительное ограничение: в момент его выполнения у потока не должно быть невыполненных обещаний, т. е. P должен быть равен пустому множеству.

Чтение из локации ℓ *(READ)* поток выполняет следующим образом. Поток выбирает некоторое сообщение $\langle \ell : val @ \tau, view \rangle$ из памяти машины. При этом метка времени записи τ должна быть больше, чем значение базового фронта потока $view^{cur}(\ell)$, а само сообщение не должно быть среди обещанных, но не выполненных потоком сообщений P. Это правило увеличивает базовый фронт потока на $[\ell @ \tau]$, а приобретающий – на *view*.

Обещание записи $\langle \ell : val @ \tau, view \rangle$ *(PROMISE)* добавляет сообщение в память машины и во множество P. При этом целевая локация ℓ и значение *val* могут быть произвольными, т. е. они не зависят от локального состояния потока σ. Метка времени должна быть уникальной среди сообщений локации ℓ, уже находящихся в памяти. Этот переход не обновляет фронты потока. Также легко заметить, что этот переход существенно недетерминирован, однако он ограничен сертификацией.

Выполнение обещания $\langle \ell : val @ \tau, view \rangle$ *(WRITE)* удаляет сообщение из множества невыполненных обещаний P. При этом данный переход должен быть возможен в рамках помеченной системы переходов локальных состояний, т. е. должно существовать состояние σ', такое что σ связан с σ' переходом $\mathrm{W}(\ell, \mathrm{val})$. При этом метка времени τ должна быть больше значения базового фронта $view^{cur}(\ell)$, и фронт сообщения view должен быть равен композиции высвобождающего фронта $view^{rel}$ и $[\ell @ \tau]$. Переход также увеличивает базовый и приобретающий фронты потока на $[\ell @ \tau]$.

Внутренний переход (SILENT) соответствует шагу исполнения потока, который не взаимодействует с памятью – присваивание в локальную переменную, исполнение условного перехода или пустой операции. Данный переход меняет только локальное состояние потока.

6. Формальное определение модели ARMv8.3

В этом разделе модель ARMv8.3 описывается формально, в соответствии с [20]. Мы не будем явным образом использовать язык ARM-ассемблера и будем считать, что он совпадает с исходным языком с точностью до модификаторов барьера памяти:

$$fmod_{\mathrm{ARM}} ::= \mathrm{sy} \mid \mathrm{ld}.$$

Также мы будем считать, что результат компиляции любой программы *Prog* на исходном языке является той же программой, в которой модификаторы rel заменены на sy, а acq – на ld.

Определение. Исполнение G – это граф, который состоит из следующих компонент.

1. Конечное множество событий $E \subseteq \mathbb{Z}$, которое включает выделенное множество начальных событий $E_0 = \{a_0^x \mid x \in \mathrm{Loc}\}$. Мы используем *a*, *b*, ... как переменные для обозначения событий.

2. Функция tid, которая по событию из E возвращает номер потока, породившего событие. При этом начальные события $a_0^x \in E_0$ мы считаем относящимися к потоку с номером 0, $\mathrm{tid}(a_0^x) = 0$.

3. Функция lab, которая присваивает метку каждому событию из E. Метки могут обозначать следующие операции:

- чтение, $\mathrm{R}(x, v)$, где $x \in \mathrm{Loc}$ – локация, из которой событие читает значение $v \in \mathrm{Val}$;
- запись, $\mathrm{W}(x, v)$, где $x \in \mathrm{Loc}$ – локация, в которую событие записывает значение $v \in \mathrm{Val}$;
- барьер памяти, $\mathrm{F}(o)$, где $o \in \{\mathrm{ld}, \mathrm{sy}\}$ – тип барьера, причем sy-барьеры строже, чем ld, $\mathrm{ld} \sqsubset \mathrm{sy}$.

Каждое начальное событие – это инициализирующая запись в некоторую локацию $\forall a_0^x \in E_0.\mathrm{lab}(a_0^x) = \mathrm{W}(x, 0)$. Функция lab естественным образом определяет следующие частично определенные функции, которые по событиям возвращают:

- typ – тип события (R, W или F);
- mod – тип барьера;
- loc – целевую локацию;
- $\mathrm{val_r}$ – прочитанное значение;

• $\mathrm{val_w}$ — записанное значение.

Далее мы будем использовать R, W и F также и для обозначения соответствующих множеств событий (например, R для обозначения $\{e \in E \mid \mathrm{typ}(e) = \mathrm{R}\}$).

4. Строгий частичный порядок на событиях $\mathrm{po} \subseteq E \times E$, называемый программным порядком, который является объединением непересекающихся множеств $\{\mathrm{po}_i\}_{i \in \{0\} \cup \mathrm{Tid}}$, где $\mathrm{po}_0 = E_0 \times (E \setminus E_0)$ и для каждого потока $i \in \mathrm{Tid}$ отношение po_i является полным порядком на множестве событий этого потока $\{a \in E \mid \mathrm{tid}(a) = i\}$.

5. Отношения $\mathrm{data}, \mathrm{addr}, \mathrm{ctrl} \subseteq \mathrm{po} \setminus \mathrm{po}_0$, которые удовлетворяют следующим ограничениям:

• $\mathrm{data} \subseteq \mathrm{R} \times \mathrm{W}$;

• $\mathrm{addr} \subseteq \mathrm{R} \times (\mathrm{R} \cup \mathrm{W})$;

• $\mathrm{ctrl}; \mathrm{po} \subseteq \mathrm{ctrl}$.

Они представляют зависимости по данным, по целевому адресу инструкции и по потоку управления соответственно.

6. Отношение $\mathrm{rf} \subseteq [\mathrm{W}]; =_{\mathrm{loc}}; [R]$, которое удовлетворяет следующим ограничениям:

• $\mathrm{val_w}(a) = \mathrm{val_r}(b)$ для любой пары $\langle a, b \rangle \in \mathrm{rf}$;

• $a_1 = a_2$, если $\langle a_1, b \rangle, \langle a_2, b \rangle \in \mathrm{rf}$.

Мы также вводим функцию rf^{-1}, которая определена на области отображения rf, $\mathrm{codom(rf)}$, и для любого a из $\mathrm{codom(rf)}$ верно, что $\langle \mathrm{rf}^{-1}(a), a \rangle \in \mathrm{rf}$.

7. Строгий частичный порядок co на элементах W, который является объединением отношений $\{\mathrm{co}_x\}_{x \in \mathrm{Loc}}$, где co_x — полный порядок на элементах $\mathrm{W}_x = \{e \in \mathrm{W} \mid \mathrm{loc}(e) = x\}$. Мы также используем некоторые производные отношения. Так, $\mathrm{rb} \triangleq \mathrm{rf}^{-1}$; co — это отношение связывает событие чтения с событием записи, которое co-следует за прочитанным событием записи. С помощью суффиксов i и e обозначаются подмножества отношений, которые связывают события одного и разных потоков соответственно. Например, $\mathrm{coi} = \{\langle w, w' \rangle \in \mathrm{co} \mid \mathrm{tid}(w) = \mathrm{tid}(w')\}$ и $\mathrm{rfe} = \{\langle w, r \rangle \in \mathrm{rf} \mid \mathrm{tid}(w) \neq \mathrm{tid}(r)\}$.

$$
\begin{array}{lll}
\mathrm{obs} \triangleq & \mathrm{rfe} \cup \mathrm{rbe} \cup \mathrm{coe} & \text{(observed-by)} \\
& (\mathrm{addr} \cup \mathrm{data}); \mathrm{rfi}^? \cup & \\
\mathrm{dob} \triangleq & (\mathrm{ctrl} \cup \mathrm{data}); [\mathrm{W}]; \mathrm{coi}^? \cup & \text{(dependency-ordered-before)} \\
& \mathrm{addr}; \mathrm{po}; [\mathrm{W}] & \\
\mathrm{bob} \triangleq & \begin{array}{l} \mathrm{po}; [\mathrm{F(sy)}] \cup [\mathrm{F(sy)}]; \mathrm{po} \cup \\ [\mathrm{R}]; \mathrm{po}; [\mathrm{F(ld)}] \cup [\mathrm{F(ld)}]; \mathrm{po} \end{array} & \text{(barrier-ordered-before)}
\end{array}
$$

Определение. Исполнение G является ARM-согласованным, если выполняется следующее:

• $\mathrm{R} = \mathrm{codom(rf)}$; (COMPLETENESS)

• отношение $\mathrm{po}|_{\mathrm{loc}} \cup \mathrm{rb} \cup \mathrm{co} \cup \mathrm{rf}$ не имеет циклов; (INTERNAL)

• отношение $\mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob}$ не имеет циклов. (EXTERNAL)

7. Обход ARM-согласованных исполнений

В разделе 4 мы обсудили, что наше доказательство корректности компиляции представляет собой симуляцию обещающей машиной некоторого обхода ARM-согласованных исполнений. В этом разделе мы введем данный обход. Для этого мы сначала определим несколько вспомогательных понятий, потом формально введем шаг обхода с требуемыми ограничениями, а потом докажем существование последовательности шагов, которые полностью обходят любое ARM-согласованное исполнение.

Определение. Конфигурацией обхода для ARM-согласованного исполнения G называется пара множеств $\langle C, I \rangle$, таких что

• $C \subseteq G.E$;

• $dom(G.\mathrm{po}; [C]) \subseteq C$;

• $C \cap G.\mathrm{W} \subseteq I \subseteq \mathrm{W}$.

Элементы C мы будем называть покрытыми (covered), а I — выпущенными (issued).

Интуитивно, в рамках симуляции покрытые события будут соответствовать префиксу программы, который исполнен обещающей семантикой, тогда как выпущенные — записями, находящимся в памяти обещающей машины в соответствующий момент исполнения.

Определение. *Множеством следующих событий* $\mathrm{Next}(G, C)$ для исполнения G и его покрытия $C \subseteq G.E$ называется множество, состоящие из событий, все po-предшествующие события которых уже покрыты:

$$\mathrm{Next}(G, C) \triangleq$$
$$\triangleq \{b \in G.E \mid \mathrm{dom}(G.\mathrm{po};[b]) \subseteq C\} \setminus C.$$

В рамках симуляции данное множество будет содержать по одному событию для каждого еще не завершенного потока в обещающей семантике, причем события будут соответствовать следующему переходу в потоке, который не является обещанием. Так, если бы обещающая семантика была определена поверх некоторого синтаксиса, то множество следующих событий соответствовало инструкциям, на которые указывают счетчики команд (program counter) потоков в обещающей модели.

Определение. Событие w является *покрываемым* для исполнения G в конфигурации обхода $\langle C, I\rangle$, $a \in \mathrm{Coverable}(G, C, I)$, если a является:

- событием чтения и связанное событие записи является выпущенным ($a \in G.\mathrm{R}$ и $\mathrm{rf}^{-1}(a) \in I$),
- выпущенным событием ($a \in I$) или
- барьером памяти ($a \in G.\mathrm{F}$).

В нашем обходе, заданном как операционная семантика, есть шаг, который «покрывает» событие, т. е. добавляет его во множество покрытых. Событие, покрываемое данным правилом, должно соответствовать приведенным выше ограничениям, которые, в свою очередь, соответствуют ограничениям обещающей семантики. Например, обещающая семантика может исполнить инструкцию чтения и прочитать из некоторого конкретного сообщения только в том случае, если данное сообщение есть в памяти, т. е. соответствующее ему событие w уже выпущено, $w \in I$.

Определение. Событие w называется выпускаемым для исполнения G в конфигурации обхода $\langle C, I\rangle$, $w \in \mathrm{Issuable}(G, C, I)$, если выполняется следующее:

- w является событием записи ($w \in G.\mathrm{W}$);
- все po-предшествующие барьеры покрыты ($\mathrm{dom}([\mathrm{F}]; G.\mathrm{po};[w] \subseteq C)$); (WRITE-BOB)
- все записи других потоков, от которых зависит w, выпущены ($\mathrm{dom}(G.\mathrm{rfe}; G.\mathrm{dob}^{+};[w]) \subseteq I$). (WRITE-DOB)

Шаг обхода, который «выпускает» событие записи, соответствует в симуляции обещанию, которое делает обещающая семантика. Ограничение WRITE-BOB является более строгим, чем ограничение из обещающей семантики: обещание может быть сделано «через» приобретающий барьер, который соответствует fence(ld) в ARMv8.3, однако более строгое ограничение позволяет упростить симуляцию. Ограничение WRITE-DOB нужно для того, чтобы обещающая семантика могла сертифицировать обещание, которое она делает.

Шаги обхода задаются следующим образом:

$$\frac{a \in \mathrm{Next}(G, C) \cap \mathrm{Coverable}(G, C, I)}{G \vdash \langle C, I\rangle \to_{\mathrm{TC}} \langle C \cup \{a\}, I\rangle} \qquad \frac{w \in \mathrm{Issuable}(G, C, I) \setminus I}{G \vdash \langle C, I\rangle \to_{\mathrm{TC}} \langle C, I \cup \{w\}\rangle}$$

Здесь левое правило покрывает событие, если событие принадлежит множеству следующих событий и при этом покрываемо в данной конфигурации. Правое правило выпускает событие записи, если событие выпускаемо и при этом еще не выпущено.

Поскольку каждое конкретное ARM-согласованное исполнение является некоторым конечным графом, а приведенные выше шаги обхода наращивают конфигурации, то для доказательства наличия полного обхода графа достаточно показать, что для каждой неконечной конфигурации существует шаг к новой конфигурации.

Теорема 7.1. Пусть $\langle C, I\rangle$ является конфигурацией ARM-согласованного исполнения G, при этом конфигурация достижима из начальной ($G \vdash \langle \varnothing, \varnothing\rangle \to^{*}_{\mathrm{TC}} \langle C, I\rangle$)

и $C \neq G.E$. Тогда существуют C' и I', т. ч. $G \vdash \langle C, I\rangle \to_{TC} \langle C', I'\rangle$.

Доказательство теоремы основано на следующих вспомогательных леммах:

Лемма 7.2. Пусть $G \vdash \langle \varnothing, \varnothing\rangle \to^{*}_{TC} \langle C, I\rangle$. Тогда $C \subseteq \mathrm{Coverable}(G, C, I)$ и $I \subseteq \mathrm{Issuable}(G, C, I)$.

Доказательство. Следует из определений Coverable и Issuable и того, что множества покрытых и выпущенных событий растут в течение обхода G.

Лемма 7.3. Пусть $G \vdash \langle \varnothing, \varnothing\rangle \to^{*}_{TC} \langle C, I\rangle$. Тогда $\mathrm{W} \cap \mathrm{Next}(G, C) \subseteq \mathrm{Issuable}(G, C, I)$.

Доказательство. Зафиксируем $w \in \mathrm{W} \cap \mathrm{Next}(G, C)$ и покажем, что $w \in \mathrm{Issuable(G, C, I)}$. Так как $w \in \mathrm{Next}(G, C)$, то все po-предшествующие события являются покрытыми (т. е. находятся в C), из чего напрямую следует, что выполняется WRITE-BOB. Кроме того, из леммы 7.1 следует, что все покрытые события являются покрываемыми, из чего следует WRITE-DOB.

Доказательство теоремы 7.1. Нам известно, что $\mathrm{Next}(G, C) \neq \varnothing$, так как $C \neq G.E$. В доказательстве мы рассматриваем два варианта. Первым мы рассматриваем вариант, когда существует элемент $\mathrm{Next}(G, C)$, который покрываем, или сначала выпускаем, а потом покрываем. Вторым мы рассматриваем вариант, когда все элементы в $\mathrm{Next}(G, C)$ не покрываемы и не выпускаемы. Для этой ситуации мы показываем, что в G существует событие записи, которое выпускаемо в данной конфигурации.

Первый вариант. Зафиксируем элемент $a \in \mathrm{Next}(G, C)$. Если $a \in \mathrm{R}$ и $\mathrm{rf}^{-1}(a) \in I$, или $a \in \mathrm{F}$, или $a \in \mathrm{W} \cap I$, тогда a покрываем по определению. Если $a \in \mathrm{W} \setminus I$, тогда a – выпускаемо по лемме 7.3.

Второй вариант. В этом случае мы предполагаем, что нет события из $\mathrm{Next}(G, C)$, которое покрываемо или выпускаемо. Таким образом, $\mathrm{Next}(G, C) \subseteq \mathrm{R}$ является следствием леммы 7.2 и определения покрываемого события. Кроме того, для любого чтения события r из $\mathrm{Next}(G, C)$ мы знаем, что $\mathrm{rf}^{-1}(r) \notin I$. Далее мы покажем, что существует событие записи, которое выпускаемо в данной конфигурации. Для этого введем вспомогательное отношение $\mathrm{eord} \triangleq (\mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob})^{+}$, которое антирефлексивно по определению ARM-согласованности EXTERNAL.

Мы знаем, что есть как минимум одно событие (чтения) из $\mathrm{Next}(G, C)$, которое при этом не покрываемо. Это означает, что существует событие записи, которое еще не выпущено. Выберем событие записи $w \in \mathrm{W} \setminus I$, которое является минимальным по отношению eord среди записей, которые еще не выпущены, то есть $\forall w' \in \mathrm{W} \setminus I. \mathrm{eord}(w', w)$. Осталось показать, что событие w выпускаемо.

Так как $w \notin \mathrm{Next}(G, C)$, то существует событие чтения $r \in \mathrm{Next}(G, C)$, такое что $\mathrm{po}(r, w)$ и $\mathrm{rf}^{-1}(r) \notin I$. При этом $\mathrm{rf}^{-1}(r) = \mathrm{rfe}^{-1}(r)$, так как $C \cap \mathrm{W} \subseteq I$ и $\forall e \in C. \mathrm{dom}(\mathrm{po}; [e]) \in C$. Для доказательства того, что w выпускаемо, нужно показать, что два утверждения выполняются.

WRITE-BOB: Пусть $f \in \mathrm{F}$, такое что $\mathrm{po}(f, w)$. Предположим, что $f \notin C$. Тогда $\mathrm{po}(r, f)$ и $\langle \mathrm{rfe}^{-1}(r), w\rangle \in \mathrm{obs}; \mathrm{bob}^{+} \subseteq \mathrm{eord}$. Так как $\mathrm{rfe}^{-1}(r) \notin I$, то существует eord-предшествующее w событие записи, которое не выпущено. Это противоречит выбору w.

WRITE-DOB: Пусть существует событие чтения r', такое что $\mathrm{dob}^{+}(r', w)$. Если $\mathrm{rfe}^{-1}(r') \neq \perp$, то $\langle \mathrm{rfe}^{-1}(r', w) \in \mathrm{rfe}; \mathrm{dob}^{+} \subseteq \mathrm{obs}; \mathrm{dob}^{+} \subseteq \mathrm{eord}$. По определению w это означает, что $\mathrm{rfe}^{-1}(r') \in I$. ■

8. Симуляция обхода обещающей машиной

В этом разделе мы приводим доказательство основной теоремы 4.1.

Теорема 4.1. Для любых программы *Prog*, результата ее компиляции $Prog_{\mathrm{ARM}}$ и ARM-согласованного исполнения G программы $Prog_{\mathrm{ARM}}$ существует исполнение *Prog* обещающей машиной, т. ч. финальное состояние памяти машины совпадает с состоянием памяти G.

Доказательство. Зафиксируем ARM-согласованное исполнение G. Далее мы покажем, что обещающая семантика может симулировать некоторый обход исполнения G, который существует согласно теореме 7.1.

Почему нам достаточно рассмотреть некоторый обход исполнения G? Это так, по-

скольку сам обход задает нам лишь схему индукции по исполнению G и не влияет на финальное состояние памяти G, поскольку оно полностью определяется отношением G.co.

Для того чтобы показать симуляцию между обещающей семантикой и обходом, мы введем отношение симуляции $\mathcal{I}$, которое будет связывать конфигурацию обхода G с состоянием обещающей машины. Формально утверждение о наличии симуляции выглядит следующим образом.

Лемма 8.1. Существуют такие $\mathcal{TS}$ и M, что $\mathcal{I}(G.E, G.\mathrm{W}, \mathcal{TS}, \mathrm{M})$ и $\langle \mathcal{TS}_{\mathrm{init}}, \mathrm{M}_{\mathrm{init}} \rangle \to^{*}_{\mathrm{Promise}} \langle \mathcal{TS}, \mathrm{M} \rangle$.

При этом мы определим отношение симуляции $\mathcal{I}$ так, чтобы из него было очевидно, что финальное состояние памяти M совпадает с состоянием памяти исполнения G, а также, что $\mathcal{I}(\varnothing, \varnothing, \mathcal{TS}_{\mathrm{init}}, \mathrm{M}_{\mathrm{init}})$ выполняется. Тогда доказательство теоремы будет непосредственно следовать из доказательства леммы 8.1. Сама же лемма доказывается индукцией с использованием леммы 8.2, которая показывает, что для любой неполной конфигурации обхода существует шаг, который обещающая машина может симулировать.

Лемма 8.2. Пусть для некоторых конфигураций обхода $\langle C, I \rangle$ и $\langle C', I' \rangle$ исполнения G, а также некоторого состояния обещающей машины $\langle \mathcal{TS}, \mathrm{M} \rangle$ выполняется $G \vdash \langle C, I \rangle \to_{\mathrm{TC}} \langle C', I' \rangle$ и $\mathcal{I}(C, I, \mathcal{TS}, \mathrm{M})$. Тогда существуют $\mathcal{TS}'$, M', такие что $\langle \mathcal{TS}, \mathrm{M} \rangle \to^{+}_{\mathrm{Promise}} \langle \mathcal{TS}', \mathrm{M}' \rangle$ и $\mathcal{I}(C', I', \mathcal{TS}', \mathrm{M}')$.

Далее в этом разделе мы опишем вспомогательные отношения на графах исполнения, которые будут нужны для выражения связи с фронтами обещающей модели в отношении симуляции, и само отношение симуляции. Как и в доказательстве теоремы 4.1, мы зафиксируем ARM-согласованное исполнение G и будем вводить все определения в контексте G.

8.1 Аналоги фронтов для ARM-согласованных исполнений. Введем вспомогательную функцию $T : \mathrm{TimeMap} = \mathrm{W} \to \mathbb{N}$, сопоставляющую событиям записи в G их порядковые номера в отношении co так, чтобы $\text{correct-tmap}(G, T)$ выполнялось:

$$\text{correct-tmap}(G, T) \triangleq \forall w, w'.$$
$$(\langle w, w' \rangle \in \mathrm{co} \Rightarrow T(w) < T(w')) \wedge$$
$$(w \notin \mathrm{codom}(\mathrm{co}) \Rightarrow T(w) = 0).$$

Эта функция фактически вводит метки времени на событиях записи G. Далее введем вспомогательные отношения sw (synchronized-with) и hb (happens-before):

$$\mathrm{sw} \triangleq [\mathrm{F(sy)}]; \mathrm{po}; \mathrm{rf}; \mathrm{po}; [\mathrm{F(ld)}]$$
$$\mathrm{hb} \triangleq (\mathrm{sw} \cup \mathrm{po})^{+}.$$

Здесь отношение sw представляет пути в графе, соответствующие передачи информации о событиях записи с помощью ARM-аналогов высвобождающего (F(sy)) и приобретающего (F(ld)) барьеров памяти. Отношение hb мы далее используем, чтобы определить отношения cur-rel, acq-rel и rel-rel. Они в паре с функцией T будут использованы в симуляции как ограничения для базового, приобретающего и высвобождающего фронтов потока в обещающей модели памяти:

$$\text{cur-rel} \triangleq \mathrm{rf}^{?}; \mathrm{hb}$$
$$\text{acq-rel} \triangleq \mathrm{rf}^{?}; \mathrm{hb}; ([\mathrm{F(sy)}]; \mathrm{po}; \mathrm{rf}; \mathrm{po})^{?}$$
$$\text{rel-rel} \triangleq \mathrm{rf}^{?}; \mathrm{hb}; [\mathrm{F(sy)}]; \mathrm{po}.$$

Аналогичным образом отношение $\text{msg-rel} \subseteq \mathrm{W} \times \mathrm{W}$ представляет ограничение на фронт сообщения из обещающей модели. Так, если $\langle w, w' \rangle \in \text{msg-rel}$ и события w и w' соответствуют сообщениям m и m' в памяти обещающей семантики, то $m.\tau = T(w)$, $m'.\tau = T(w')$ и $m'.view(\mathrm{loc}(w)) \geq m.\tau$

$$\text{msg-rel} \triangleq \mathrm{rf}^{?}; \mathrm{hb}; [\mathrm{F(sy)}]; \mathrm{po} \cup [\mathrm{W}].$$

8.2 Отношение симуляции $\mathcal{I}$. Отношение симуляции мы определяем следующим образом:

$$\mathcal{I}(C, I, \mathcal{TS}, \mathrm{M}) \triangleq \mathcal{I}_{\mathrm{mem1}}(C, I, \mathcal{TS}, \mathrm{M}) \wedge$$
$$\wedge\, \mathcal{I}_{\mathrm{mem2}}(C, I, \mathrm{M}) \wedge \mathcal{I}_{\mathrm{view}}(C, \mathcal{TS}) \wedge$$
$$\wedge\, \mathcal{I}_{\text{view-rel}}(\mathcal{TS}) \wedge \mathcal{I}_{\mathrm{state}}(C, \mathcal{TS}).$$

$\mathcal{I}_{\mathrm{mem1}}(C, I, \mathcal{TS}, \mathrm{M})$ утверждает, что все выпущенные события I имеют аналоги в памяти обещающей машины, причем выполненные обещания соответствуют по-

крытым событиям записи:

$$\mathcal{I}_{\text{mem1}}(C, I, \mathcal{TS}, \text{M}) \triangleq \forall w \in I.\exists view \leq$$
$$\leq \text{dom-view}(\text{msg-rel};[w]).$$
$$\text{let } msg \triangleq \langle \text{loc}(w) : \text{val}_w(w) @ T(w), view\rangle \text{ in}$$
$$\text{let P} \triangleq \mathcal{TS}(\text{tid(w)}).\text{P in}$$
$$(w \in C \Rightarrow msg \in \text{M} \setminus \text{P}) \wedge (w \notin C \Rightarrow msg \in \text{P}).$$

Здесь dom-view является вспомогательной функцией, которая по множеству записей строит соответствующий множеству фронт:

$$\text{set-view}(S) \triangleq \lambda\ell.\max\{T(w) \mid w \in S, \text{loc}(w) = \ell\};$$
$$\text{dom-view}(R) \triangleq \text{set-view}(\text{dom}(R)).$$

$\mathcal{I}_{\text{mem2}}(C, I, \text{M})$ обозначает обратную связь: для каждого сообщения из M существует соответствующее ему выпущенное событие в $\mathcal{I}$:

$$\mathcal{I}_{\text{mem2}}(C, I, \text{M}) \triangleq \forall\langle \ell : v @ \tau, view\rangle \in$$
$$\in \text{M}.\tau \neq 0 \Rightarrow \exists w \in \mathcal{I}.$$
$$\ell = \text{loc}(w) \wedge v = \text{val}_w(w) \wedge \tau = T(w) \wedge view \leq$$
$$\leq \text{dom-view}(\text{msg-rel};[w]).$$

$\mathcal{I}_{\text{view}}(C, \mathcal{TS})$ утверждает, что для любого элемента множества следующих событий ($\text{Next}(G, C)$) фронты, представляющие связанные с ним отношениями cur-rel, acq-rel и rel-rel события записи, больше, чем базовый, приобретающий и высвобождающие фронты соответствующего потока:

$$\mathcal{I}_{\text{view}}(C, \mathcal{TS}) \triangleq \forall e \in \text{Next}(G, C).$$
$$\text{let } \langle \text{cur}, \text{acq}, \text{rel}\rangle \triangleq \mathcal{TS}(\text{tid}(e)).V \text{ in}$$
$$\text{cur} \leq \text{dom-view}(\text{cur-rel};[e]) \wedge$$
$$\text{acq} \leq \text{dom-view}(\text{acq-rel};[e]) \wedge$$
$$\text{rel} \leq \text{dom-view}(\text{rel-rel};[e]).$$

$\mathcal{I}_{\text{view-rel}}(\mathcal{TS})$ показывает, что все еще не выполненные обещания имеют специальную форму их фронтов – значение высвобождающего фронта соответствующего потока плюс метка времени самого обещания:

$$\mathcal{I}_{\text{view}-rel}(\mathcal{TS}) \triangleq \forall tid, \langle \ell : _ @ \tau, view\rangle \in \mathcal{TS}(tid).\text{P}.$$
$$view = [\ell @ \tau] \sqcup \mathcal{TS}(tid).V.\text{rel}.$$

$\mathcal{I}_{\text{state}}(C, \mathcal{TS})$ утверждает, что для каждого потока tid существует список переходов $\{t_i\}_{i \in [1..k]}$, который соответствует событиям в tid-подграфе исполнения G, и текущее состояние потока $\mathcal{TS}(tid).\sigma$ получено с помощью p переходов из списка, где p – количество покрытых событий, $|E_{tid} \cap C|$.

$$\mathcal{I}_{\text{state}}(C, \mathcal{TS}) \triangleq \forall tid,$$
$$k = |E_{tid}|.\exists\{\sigma_i\}_{i \in [0..k]}, \{t_i\}_{i \in [1..k]}.$$
$$\text{let } p \triangleq |E_{tid} \cap C| \text{ in}$$
$$\mathcal{TS}(tid).\sigma =$$
$$= \sigma_p \wedge (\forall j \in [0..k-1].\sigma_j \xrightarrow{\varepsilon}{}^{*} \to \sigma_{j+1}) \wedge$$
$$\forall n \in [1..k], a \in \text{nth}([E_{tid}];\text{po};$$
$$[E_{tid}])(n-1).t_{n+1} \approx \text{lab}(a).$$

Здесь функция nth $porder$ n возвращает элементы с номером n из отношения частичного порядка $porder$, а предикат $t \approx \text{lab}(a)$ выполняется тогда и только тогда, когда метка перехода t соответствует метке события a.

9. Связанные работы

Корректность компиляции в общем смысле является большой, проработанной областью, которая продолжает развиваться. Так, недавно были представлены верифицированные компиляторы для языков C, CompCert [15] и ML, CakeML [11]. Эти компиляторы для однопоточных программ, но существуют и компиляторы для многопоточных программ в архитектуры со слабыми моделями памяти, в частности из языка C в архитектуру x86-TSO [22]. В работах [5, 6] приведены доказательства корректности компиляции из аксиоматической модели C/C++11 [7] в модели архитектур x86-TSO и Power.

Наиболее близкими работами к нашей являются [10] и [19]. В [10] приведены доказательства корректности компиляции из обещающей модели в модели памяти x86-TSO [23] и Power [3], которые, так же как и [20], являются аксиоматическими. Доказательства для моделей x86-TSO и Power имеют следующую структуру: (1) обе модели памяти, x86-TSO и Power, представляются как некоторые более строгие модели и набор трансформаций над программами; (2) приводится доказательство того, что трансформации корректны в рамках

обещающей модели памяти, т. е. поведения трансформированной программы являются поведениями изначальной программы в обещающей модели памяти; (3) показывается корректность компиляции для более строгих версий моделей.

Первый пункт этого доказательства приведен в [12]. Так, каждое поведение некоторой программы A в рамках модели x86-TSO является поведением программы B в рамках модели последовательной консистентности, где B может быть получена из A путем (многократного) применения двух трансформаций — переупорядочивание инструкции записи с непосредственно следующей операцией чтения из другой локации и удаления повторного чтения:

$$\begin{array}{lcl} [x] := 1; & \rightarrow & a := [y]; \\ a := [y]; & & [x] := 1; \end{array}$$

$$\begin{array}{lcl} a := [y]; & \rightarrow & a := [y]; \\ b := [y]; & & b := a \end{array}$$

Для модели Power приведен аналогичный результат: каждое поведение программы A в рамках модели Power является поведением программы B в рамках более строгой модели StrongPower, запрещающей часть поведений, наблюдаемых в модели Power. Здесь B может быть получена из A путем (многократного) применения трансформации, которая переупорядочивает произвольные независимые инструкции чтения или записи, следующие друг за другом.

Мы пытались применить аналогичный подход для доказательства корректности компиляции для модели ARMv8.3 [20], но отказались от него ввиду того, что существует программа, одно из поведений которой разрешено моделью ARMv8.3 и не выразимо с помощью ранее использованных трансформаций кода над более строгой моделью памяти. Упомянутая программа выглядит следующим образом:

$$\begin{array}{l||l} a := [x]; \ //1 & [y] := 1; \\ \text{if a} = 1 \text{ then} & \text{fence(sy)}; \\ b := [y]; \ //0 & [x] := 1 \end{array}$$

В левом потоке между операциями чтения есть зависимость по управлению, а в правом потоке используется барьер памяти, который запрещает исполнение записей не по порядку. Модель памяти ARMv8.3 разрешает программе завершиться с $a = 1$ и $b = 0$, при этом инструкции в обоих потоках не могут быть переставлены ввиду наличия зависимости и барьера.

В работе [19] приведено доказательство корректности компиляции из обещающей модели в модель ARM POP [9]. Несмотря на то, что этот результат выглядит очень близким к тому, что предлагается в данной статье, существует важное различие: модель ARM POP представлена в операционном стиле. Последнее позволило авторам [19] провести доказательство через непосредственную симуляцию модели ARM POP обещающей моделью. При этом модель ARM POP обладает более широкими возможностями по исполнению инструкций не по порядку по сравнению с обещающей моделью. Для решения этой проблемы в [19] используется т. н. *запаздывающая симуляция*, которая позволяет обещающей семантике в рамках симуляции повторять действия модели ARM POP в возможном для обещающей семантики порядке. Наша операционная семантика обхода графа исполнения использует похожую идею, но в приложении к аксиоматической модели ARMv8.3.

10. Заключение

В данной статье приведено доказательство корректности компиляции для подмножества обещающей модели памяти [10], состоящего из ослабленных операций чтения и записи, приобретающих и высвобождающих барьеров памяти, в модель памяти ARMv8.3 [20]. Доказательство базируется на новой идее обхода исполнений в аксиоматических семантиках, который может симулировать обещающая модель памяти. Мы убеждены в том, что данная идея может использоваться для прямых доказательств корректности компиляции в другие модели памяти, а подобные задачи регулярно появляются ввиду бурного развития области.

Несмотря на то, что подмножество обещающей модели, рассмотренное в статье, достаточно ограничено, доказательство кор-

ректности компиляции для него достаточно сложно и объемно. В наши дальнейшие планы входит расширение доказательства до полной обещающей модели памяти. Для этого нужно будет поддержать операции приобретающего чтения (read acquire) и высвобождающей записи (write release), атомарные инструкции чтения и записи (read-modify-write), частным случаем которых является сравнение с обменом (compare-and-set – CAS), а также полные барьеры памяти (SC fences). Поддержка данных инструкций потребует небольшого усложнения обхода ARM-согласованных исполнений, а также существенного усложнения отношения симуляции ввиду большого количества мелких деталей, связанных с упомянутыми инструкциями в рамках обещающей модели памяти.

Работа выполнена при поддержке компании JetBrains (http://jetbrains.com).

СПИСОК ЛИТЕРАТУРЫ

1. ARM architcture reference manual: ARMv8, for ARMv8-A architecture profile // URL: https://developer.arm.com/docs/ddi0487/latest/ arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile (Дата обращения: 16.05.2017).

2. C/C++11 mappings to processors // URL: https://www.cl.cam.ac.uk/~pes20/cpp/ cpp0xmappings.html. (Дата обращения: 16.05.2017).

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

4. **Batty M., Memarian K., Nienhuis K., Pichon-Pharabod J., Sewell P.** The problem of programming language concurrency semantics // ESOP. Springer, 2015. Vol. 9032 of LNCS. Pp. 283–307. DOI: 10.1007/978-3-662-46669-8_12

5. **Batty M., Memarian K., Owens Sc., Sarkar S., Sewell P.** Clarifying and compiling C/C++ concurrency: From C++11 to POWER // POPL 2012. ACM, 2012. DOI: 10.1145/2103621.2103717.

6. **Batty M., Owens Sc., Sarkar S., Sewell P., 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.

7. **Batty M., Owens Sc., Sarkar S., Sewell P., Weber T.** Mathematizing C++ concurrency // POPL 2011. ACM, 2011. Pp. 55–66. DOI: 10.1145/1925844.1926394

8. **Boehm H.-J., Demsky B.** Outlawing ghosts: Avoiding out-of-thin-air results // MSPC 2014. ACM, 2014. Pp. 7:1–7:6. DOI: 10.1145/2618128.2618134

9. **Flur Sh., Gray K.E., Pulte Ch., Sarkar S., Sezgin A., Maranget L., Deacon W., Sewell P.** Modelling the ARMv8 architecture, operationally: Concurrency and ISA // POPL 2016. ACM, 2016. Pp. 608–621. DOI: 10.1145/2837614.2837615

10. **Jeehoon Kang, Chung-Kil Hur, Lahav O., Vafeiadis V., Dreyer D.** A promising semantics for relaxed-memory concurrency // POPL 2017. ACM, 2017. DOI: 10.1145/3009837. 3009850

11. **Kumar R., Myreen M.O., Norrish M., Owens Sc.** CakeML: A verified implementation of ML // POPL 2014: Proc. of the 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM Press, 2014. Pp. 179–191. DOI: 10.1145/2535838.2535841

12. **Lahav O., Vafeiadis V.** Explaining relaxed memory models with program transformations // FM 2016. Springer, 2016. DOI: 10.1007/978-3-319-48989-6_29

13. **Lahav O., Vafeiadis V., Jeehoon Kang, Chung-Kil Hur, Dreyer D.** Repairing sequential consistency in C/C++11 // In PLDI 2017. ACM, 2017.

14. **Lamport L.** How to make a multiprocessor computer that correctly executes multiprocess programs // IEEE Trans. Computers. 1979. Vol. 28(9). Pp. 690–691. DOI: 10.1109/TC.1979.1675439

15. **Leroy X.** A formally verified compiler back-end // J. Autom. Reasoning. 2009. Vol. 43(4). Pp. 363–446. URL: https://doi.org/10.1007/s10817-009-9155-4. DOI: 10.1007/s10817-009-9155-4

16. **Manerkar Ya.A., Trippel C., Lustig D., Pellauer M., Martonosi M.** Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. CoRR, abs/1611.01507, 2016 // URL: http://arxiv.org/abs/1611.01507.

17. **Manson J., Pugh W., Adve S.V.** The Java memory model // POPL 2005. ACM, 2005. Pp. 378–391. DOI: 10.1145/1040305.1040336.

18. **Milner R.** Communication and concurrency // PHI Series in Computer Science. Prentice Hall, 1989.

19. **Podkopaev A., Lahav O., Vafeiadis V.** Promising compilation to ARMv8 POP // ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

20. **Pulte Ch., Flur Sh., Deacon W., French J., Sarkar S., Sewell P.** Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8 // Accepted to POPL'18.

21. **Sarkar S., Sewell P., Alglave J., Maranget L., Williams D.** Understanding POWER multiprocessors // PLDI 2011. ACM, 2011. Pp. 175–186. DOI: 10.1145/1993498. 1993520

22. **Sevcík J., Vafeiadis V., Nardelli F.Z., Jagannathan S., Sewell P.** Relaxed-memory concurrency and verified compilation // Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. POPL 2011. Austin, TX, USA, 2011. Pp. 43–54. URL:http://doi.acm.org/10.1145/1926385.1926393. DOI: 10.1145/1926385.1926393

23. **Sewell P., Sarkar S., Owens Sc., Nardelli F.Z., Myreen M.O**. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors // Commun. ACM, 2010. Vol. 53(7). Pp. 89–97. DOI: 10.1145/1785414.1785443

24. **Vafeiadis V., Balabonski Th., Chakraborty S., Morisset R., Nardelli F.Z.** Common compiler optimisations are invalid in the C11 memory model and what we can do about it // POPL 2015. ACM, 2015. Pp. 209–220.

Статья поступила в редакцию 27.09.2017.

REFERENCES

1. *ARM architcture reference manual: ARMv8, for ARMv8-A architecture profile*. Available: https://developer.arm.com/docs/ddi0487/latest/ arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile (Accessed: 16.05.2017).

2. *C/C++11 mappings to processors*. Available: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. (Accessed: 16.05.2017).

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

4. **Batty M., Memarian K., Nienhuis K., Pichon-Pharabod J., Sewell P.** The problem of programming language concurrency semantics. *In ESOP*. Springer, 2015, Vol. 9032 of LNCS, Pp. 283–307. DOI: 10.1007/978-3-662-46669-8_12

5. **Batty M., Memarian K., Owens Sc., Sarkar S., Sewell P.** Clarifying and compiling C/C++ concurrency: From C++11 to POWER. *In POPL 2012*, ACM, 2012. DOI: 10.1145/2103621.2103717

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

7. **Batty M., Owens Sc., Sarkar S., Sewell P., Weber T.** Mathematizing C++ concurrency. *In POPL 2011*, ACM, 2011, Pp. 55–66. DOI: 10.1145/1925844.1926394

8. **Boehm H.-J., Demsky B.** Outlawing ghosts: Avoiding out-of-thin-air results. *In MSPC 2014*, ACM, 2014, Pp. 7:1–7:6. DOI: 10.1145/2618128.2618134

9. **Flur Sh., Gray K.E., Pulte Ch., Sarkar S., Sezgin A., Maranget L., Deacon W., Sewell P.** Modelling the ARMv8 architecture, operationally: Concurrency and ISA. *In POPL 2016*, ACM, 2016, Pp. 608–621. DOI: 10.1145/2837614.2837615

10. **Jeehoon Kang, Chung-Kil Hur, Lahav O., Vafeiadis V., Dreyer D.** A promising semantics for relaxed-memory concurrency. In POPL 2017. ACM, 2017. DOI: 10.1145/3009837.3009850

11. **Kumar R., Myreen M.O., Norrish M., Owens Sc.** CakeML: A verified implementation of ML. *In POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 2014, Pp. 179–191. DOI: 10.1145/2535838.2535841

12. **Lahav O., Vafeiadis V.** Explaining relaxed memory models with program transformations. *In FM 2016*. Springer, 2016. DOI: 10.1007/978-3-319-48989-6_29

13. **Lahav O., Vafeiadis V., Jeehoon Kang, Chung-Kil Hur, Dreyer D.** Repairing sequential consistency in C/C++11. *In PLDI 2017*. ACM, 2017.

14. **Lamport L.** How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 1979, Vol. 28(9), Pp. 690–691. DOI: 10.1109/TC.1979.1675439

15. **Leroy X.** A formally verified compiler back-end. *J. Autom. Reasoning*, 2009, Vol. 43(4), Pp. 363–446. Available: https://doi.org/10.1007/s10817-009-9155-4. DOI: 10.1007/s10817-009-9155-4

16. **Manerkar Ya.A., Trippel C., Lustig D., Pellauer M., Martonosi M.** *Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. CoRR, abs/1611.01507, 2016*. Available: http://arxiv.org/abs/1611.01507.

17. **Manson J., Pugh W., Adve S.V.** The Java memory model. *In POPL 2005*, ACM, 2005, Pp. 378–391, DOI: 10.1145/1040305.1040336.

18. **Milner R.** Communication and concurrency. *PHI Series in Computer Science*. Prentice Hall, 1989.

19. **Podkopaev A., Lahav O., Vafeiadis V.** Promising compilation to ARMv8 POP. *In ECOOP 2017.* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

20. **Pulte Ch., Flur Sh., Deacon W., French J., Sarkar S., Sewell P.** Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Accepted to POPL'18.*

21. **Sarkar S., Sewell P., Alglave J., Maranget L., Williams D.** Understanding POWER multiprocessors. *In PLDI 2011*, ACM, 2011, Pp. 175–186. DOI: 10.1145/1993498. 1993520

22. **Sevcík J., Vafeiadis V., Nardelli F.Z., Jagannathan S., Sewell P.** Relaxed-memory concurrency and verified compilation. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, Austin, TX, USA, 2011, Pp. 43–54. Available: http://doi.acm.org/10.1145/1926385.1926393. DOI: 10.1145/1926385.1926393

23. **Sewell P., Sarkar S., Owens Sc., Nardelli F.Z., Myreen M.O.** x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 2010, Vol. 53(7), Pp. 89–97. DOI: 10.1145/1785414.1785443

24. **Vafeiadis V., Balabonski Th., 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, Pp. 209–220.

Received 27.09.2017.

СВЕДЕНИЯ ОБ АВТОРАХ / THE AUTHORS

ПОДКОПАЕВ Антон Викторович
PODKOPAEV Anton V.
E-mail: a.podkopaev@2009.spbu.ru

ЛАХАВ Ори
LAHAV Ori
E-mail: orilahav@tau.ac.il

ВАФЕЯДИС Виктор
VAFEIADIS Viktor
E-mail: viktor@mpi-sws.org

© Санкт-Петербургский политехнический университет Петра Великого, 2017