DOI: 10.15514/ISPRAS-2019-31(5)-4

Компиляция модели памяти OCaml в Power

[1,3] *Е.С. Намаконов, ORCID: 0000-0002-7517-9389 <st070466@student.spbu.ru>*
[2,3,4] *А.В. Подкопаев, ORCID: 0000-0002-9448-6587 <apodkopaev@hse.ru>*

[1] *Санкт-Петербургский государственный университет,*
199034, Россия, Санкт-Петербург, Университетская набережная, д. 7–9
[2] *Национальный исследовательский университет «Высшая школа экономики»,*
194100, Россия, Санкт-Петербург, Кантемировская ул., 3А корпус 1
[3] *JetBrains Research,*
197342, Россия, Санкт-Петербург, Кантемировская ул., 2
[4] *Институт им. Макса Планка: Программные Системы,*
67663, Германия, Кайзерслаутерн, ул. Пауль-Эрлих G26.

Аннотация. В настоящее время для языков программирования и процессоров активно разрабатываются модели памяти, направленные на решение различных проблем многопоточного программирования. Так, модель памяти языка OCaml (OCamlMM) позволяет избежать неопределённого поведения, вызванного гонками по данным. Для применения этой модели на практике необходимо доказать корректность её компиляции в распространённые архитектуры процессоров. На данный момент это выполнено для моделей x86 и ARM, но не для Power. Для того, чтобы упростить доказательство корректности компиляции OCamlMM в модель Power, предлагается построить схему компиляции OCamlMM в промежуточную модель памяти (IMM). Для этой модели уже доказана корректность компиляции в модель Power и другие архитектуры, поэтому доказательство корректности компиляции OCamlMM в модель Power сводится к доказательству корректности компиляции OCamlMM в IMM. В данной работе предложена схема компиляции OCamlMM в IMM и доказана её корректность. В этой схеме используются барьеры памяти и инструкции compare-and-swap, которые позволяют исключить поведение, допустимое IMM и запрещённое моделью OCaml. Полученная схема компиляции даёт корректную схему компиляции OCamlMM в модель Power. Кроме того, при таком подходе доказать корректность компиляции OCamlMM в другую архитектуру можно, доказав корректность компиляции IMM в эту архитектуру.

Ключевые слова: слабые модели памяти; корректность компиляции; многопоточность; OCaml memory model; IMM.

Для цитирования: Намаконов Е.С., Подкопаев А.В. Компиляция модели памяти OCaml в Power. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 63-78. DOI: 10.15514/ISPRAS-2019-31(5)-4

Благодарности: Авторы благодарят коллег по научной группе за участие в обсуждении ранних версий статьи.

Compilation of OCaml memory model into Power

[1,3] *E.S. Namakonov, ORCID: 0000-0002-7517-9389 <st070466@student.spbu.ru>*
[2,3,4] *A.V. Podkopaev, ORCID: 0000-0002-9448-6587 <apodkopaev@hse.ru>*

[1] *St Petersburg University,*
7–9, Universitetskaya embankment, St Petersburg, Russia, 199034.
[2] *National Research University «Higher School of Economics»,*
3A bld. 1, Kantemirovskaya st., St Petersburg, Russia, 194100.
[3] *JetBrains Research,*
2, Kantemirovskaya st., St Petersburg, Russia, 197342.
[4] *Max Planck Institute for Software Systems,*
G26, Paul-Ehrlich st., Kaiserslautern, Germany, 67663.

Abstract. The development of memory models aimed at solving various concurrency problems is an active research topic. One such model is the OCaml memory model (OCamlMM), which allows to mitigate undefined behavior caused by data races. To use this model in practice one has to prove the correctness of its compilation into mainstream CPU architectures. At the moment, it is done for x86 and ARM but not for Power.One way to prove the correctness of compilation of OCamlMM into the Power model is to develop a compilation scheme from OCamlMM into the Intermediate Memory Model (IMM). It would be sufficient since there already exists a compilation scheme from IMM to the Power model. In this paper, the compilation scheme from OCamlMM into IMM is presented and proved to be correct. Memory fences and compare-and-swap instructions are used to permit only behavior allowed by OCamlMM. The resulting compilation scheme gives a correct compilation scheme from OCamlMM to the Power model. Moreover, when a compilation scheme from IMM into another CPU architecture will be developed, such an approach would immediately give a correct scheme of compilation of OCamlMM into that architecture.

Keywords: weak memory models; compilation correctness; concurrency; OCaml memory model; IMM.

For citation: Namakonov E. S., Podkopaev A. V. Compilation of OCaml memory model into Power. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019, pp. 63-78 (in Russian). DOI: 10.15514/ISPRAS-2019-31(5)-4

Acknowledgements. Authors are grateful to the research group colleagues for participating in early paper versions' discussions.

1. Введение

Семантика языка программирования, поддерживающего параллельные вычисления, определяет множество возможных состояний системы (главным образом – оперативной памяти) после выполнения программы посредством *модели памяти*. Наиболее известная модель памяти – модель *последовательной согласованности* (sequential consistency, SC [1]), в которой любой результат исполнения программы может быть получен путём попеременного исполнения инструкций отдельных потоков согласно программному порядку в них. Однако из-за оптимизаций, выполняемых компилятором и процессором, могут наблюдаться поведения, невозможные в такой модели. Например, при выполнении программы[1] на рис. 1, написанной на C++ и скомпилированной с помощью компилятора GCC, на архитектуре x86 оба потока могут прочитать значение 0, что объясняется буферизацией записи (store buffering, [2] [3]).

Так как отказ от подобных оптимизаций нежелателен, современные модели памяти допускают некоторые сценарии поведения, невозможные в модели SC. Такие модели памяти называются *слабыми*. Например, слабыми являются модели памяти языков C++

[1] Здесь и далее используется упрощённый синтаксис программ: x и y обозначают адреса в памяти, a и b — локальные переменные (регистры), rlx – режим доступа. В комментариях указаны наблюдаемые при чтении значения.

[4], Java [5] и JavaScript [6] [7], а также архитектур x86 [2] [8], Power [9] [10]и ARM [11] [12] [13].

$x = y = 0$	
$[x]^{rlx} := 1;$	$[y]^{rlx} := 1;$
$a := [y]^{rlx};$ //0	$b := [x]^{rlx};$ //0

Рис. 1. Буферизация записи
Fig. 1. Store buffering

Так как отказ от подобных оптимизаций нежелателен, современные модели памяти допускают некоторые сценарии поведения, невозможные в модели SC. Такие модели памяти называются *слабыми*. Например, слабыми являются модели памяти языков C++ [4], Java [5] и JavaScript [6] [7], а также архитектур x86 [2] [8], Power [9] [10]и ARM [11] [12] [13]. Для усиления гарантий на поведение программы в слабой модели памяти необходимо использовать инструкции с более строгим *режимом доступа*: например, модель памяти C++ гарантирует, что если в программе на рис. 1 заменить режим доступа всех инструкций с rlx на sc, то поведение полученной программы будет согласовано с моделью SC, поскольку такие инструкции будут скомпилированы с использованием т.н. барьеров памяти, запрещающих перестановку инструкций программы.

Одной из проблем таких моделей памяти является то, что они предоставляют слабые гарантии на поведение программ, содержащих гонки по данным. В частности, в модели C++ поведение таких программ не определено (см. [4], раздел 2). А модель памяти Java допускает чтение произвольных значений по отдельному адресу, если раньше по нему произошла гонка (см. [14]).

Для решения этой проблемы была предложена модель памяти OCaml (далее – `OCamlMM`) [15], обладающая свойством локальной свободы от гонок (Local Data Race Freedom property): результат обращения по данному адресу в памяти не зависит от гонок по другим адресам, а также от предыдущих гонок по этому же адресу. Это свойство гарантирует, что результат исполнения всех участков программы, не содержащих гонок по данным, будет согласовано с моделью SC.

Для того, чтобы использовать `OCamlMM` на практике, необходимо доказать её реализуемость на распространённых архитектурах процессоров. Для этого нужно доказать корректность компиляции в каждую из них – показать, что для любой программы при замене инструкций языка на процессорные инструкции согласно схеме компиляции получается программа, все сценарии поведения которой разрешены `OCamlMM` для исходной программы. В [15] приведены схемы компиляции `OCamlMM` в модели x86 и ARMv8 и доказана их корректность. При этом отсутствует схема компиляции в модель Power – архитектуру, часто используемую в серверном оборудовании [16]. Задача построения такой схемы осложнена тем, что модель Power, в отличие от моделей x86, ARMv8 и `OCamlMM`, не обладает т.н. свойством *multicopy atomicity*, т.е. не гарантирует, что записанные в память значения становятся доступны всем потокам в одном и том же порядке [11].

Для доказательства корректности компиляции `OCamlMM` в модель Power достаточно построить корректную схему компиляции `OCamlMM` в промежуточную модель памяти (Intermediate Memory Model, далее – `IMM`) [17], для которой уже доказана корректность компиляции в модель Power. Использование `IMM` как промежуточного этапа компиляции позволяет разбить доказательство корректности компиляции модели языка в модель архитектуры на два, которые впоследствии можно использовать в других доказательствах для этих моделей.

В данной работе предлагается схема компиляции `OCamlMM` в `IMM` и доказывается её корректность. Так как для `IMM` корректность компиляции в модель Power уже доказана, полученная схема даёт корректную схему компиляции `OCamlMM` в модель Power.

Статья имеет следующую структуру. Разд. 2 описывает проблему корректности компиляции `OCamlMM` в `IMM` на примерах. В разд. 3 приводится определение графа исполнения – способа представить исполнение программы в декларативных моделях памяти, к которым относятся `OCamlMM` и `IMM`. Затем, в разд. 4 описываются формальные модели `OCamlMM` и `IMM`. В разд. 5 и 6 предлагается схема компиляции `OCamlMM` в `IMM` и доказывается её корректность. В разд. 7 приводится обзор связанных работ. Наконец, в разд. 8 подводятся итоги работы и описываются направления дальнейших исследований.

2. Корректность компиляции OCamlMM в IMM на примерах

`OCamlMM` и `IMM` определены декларативно. Это означает, что исполнение программы представляется в виде т.н. графа исполнения. Пример программы и одного из графов её исполнения приведён на рис. 2.

$x = y = 0$		
$[x]^{at} := 1;$	$a := [x]^{at};$//1	$b := [y]^{na};$//1
	$[y]^{na} := 1;$	$c := [x]^{at};$//0

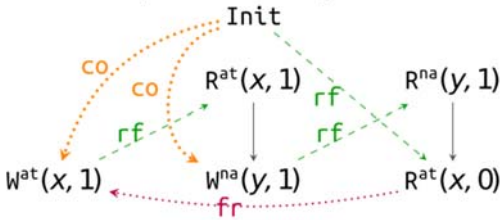

Рис. 2. Программа и пример её исполнения, не согласованного в `OCamlMM`
Fig. 2. A program and its execution inconsistent under `OCamlMM`.

Вершины графа соответствуют событиям – операциям над памятью, которые производятся при выполнении инструкций программы. Так, событие $W^{at}(x, 1)$ соответствует записи по адресу x значения 1 в режиме at. Кроме того, в графе выделяются инициализирующие события, которые соответствуют инициализирующей записи нулей в память. На рис. 2 они для краткости объединены в множество $Init$; далее в графах мы будем опускать эти события, если это не будет важно для рассуждений.

Рёбра графа задают бинарные отношения между событиями. В данном графе есть четыре различных отношения: рёбра po соответствуют программному порядку инструкций, rf – чтению записанного ранее значения, co – порядку записей по одному адресу, fr – чтению до указанного события записи. Отношения po и co являются транзитивными, поэтому для их задания достаточно указывать только непосредственные рёбра. Кроме того, для краткости будем опускать подпись "po" рядом с соответствующими рёбрами.

Согласованными (допустимыми моделью) называются те исполнения, графы которых удовлетворяют некоторому предикату, заданному моделью. В частности, предикат согласованности `OCamlMM` требует, чтобы в графе не было циклов, состоящих только из рёбер co и fr, проходящих между вершинами с меткой at, а также рёбер po и rf. Это условие формализует свойство multicopy atomicity, описанное выше.

Граф исполнения на рис. 2 не является OCamlMM-согласованным. Действительно, это исполнение нарушает свойство multicopy atomicity: второй поток читает записанное в x значение 1 до записи 1 в y, однако третий поток читает старое значение 0 из x после чтения 1 из y. Соответствующий граф исполнения не удовлетворяет предикату согласованности OCamlMM, так как между вершинами есть цикл, подходящий под описание выше. Таким образом, в OCamlMM после исполнения программы на рис. 1 переменные a, b и c не могут содержать значения 1, 1 и 0 соответственно

$x = y = 0$		
$[x]^{sc} := 1;$	$a := [x]^{sc};//1$ $[y]^{rlx} := 1;$	$b := [y]^{rlx};//1$ $c := [x]^{sc};//0$

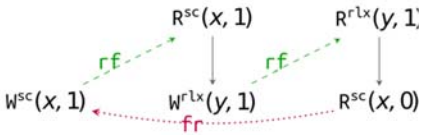

Рис. 3. Результат компиляции программы на рис. 2 с использованием тривиальной схемы компиляции и его IMM-согласованный граф исполнения
Fig. 3. The result of compilation of the program from fig. 2 using the trivial compilation scheme and its IMM-consistent execution graph

Скомпилированная программа не должна демонстрировать поведение, запрещённое исходной моделью памяти. Поэтому граф исполнения скомпилированной программы не должен быть согласованным в целевой модели памяти.

На рис. 3 показана программа, полученная в результате компиляции программы на рис. 2 согласно тривиальной схеме компиляции, и граф её исполнения. Такая схема лишь заменяет режимы инструкций на их аналоги в IMM: na заменяется на rlx, а at – на sc; дополнительных инструкций не вводится. Соответственно, граф на рис. 3 отличается от графа на рис. 2 только метками вершин, и в нём сохраняется цикл того же вида. IMM не гарантирует свойство multicopy atomicity, и потому предикат её согласованности не требует отсутствия таких циклов, что делает граф на рис. 3 IMM-согласованным, а соответствующее ему поведение – разрешённым. Поэтому тривиальная схема компиляции не является корректной.

На рис. 4 приведена программа, полученная в результате компиляции программы на рис. 2 согласно схеме компиляции, приведённой в разделе 5. В результате компиляции в ней появляются инструкции барьеров памяти, запрещающие некоторые оптимизации процессора и компилятора. С ними граф исполнения перестаёт быть согласованным: из рёбер fr, po, окружённого барьерами rf, а также rf между событиями с меткой sc образуется цикл, запрещённый в IMM.

$x = y = 0$		

$fence^{acq};$
$exchg(x, 1);$
	$fence^{acq};$
	$a := [x]^{sc};//1$
	$fence^{acqrel};$
	$[y]^{rlx} := 1;$
		$b := [y]^{rlx};//1$
		$fence^{acq};$
		$c := [x]^{sc};//0$

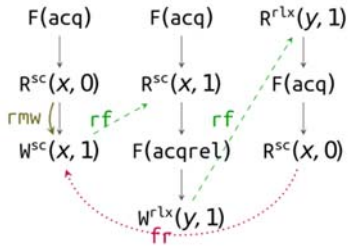

Рис. 4. Результат компиляции программы на рис. 2 с использованием тривиальной схемы компиляции и его IMM-согласованный граф исполнения
Fig. 4. The result of compilation of the program from fig. 2 using the trivial compilation scheme and its IMM-consistent execution graph.

Таким образом, для доказательства корректности компиляции необходимо доказать следующую теорему.

Теорема 2.1. *Пусть G_I – IMM-согласованный граф исполнения, соответствующий графу исполнения G_O в OCamlMM. Тогда G_O является OCamlMM-согласованным.*

Для доказательства теоремы достаточно доказать OCamlMM-согласованность графа G_I. Из этого следует OCamlMM-согласованность графа G_O, так как он фактически является подграфом G_I, а условия OCamlMM-согласованности графа таковы, что они выполняются для любого его подграфа. Условие OCamlMM-согласованности состоит в иррефлексивности одного отношения и ацикличности другого. Для каждого из этих отношений доказывается включение в такое отношение, для которого в IMM-согласованном графе соответствующее условие выполняется.

3. Понятие графов исполнения

В описаниях декларативных моделей памяти часто встречаются следующие обозначения отношений между вершинами. Для бинарного отношения R обозначения $R^?$, R^+, R^* соответствуют его рефлексивному, транзитивному и транзитивно-рефлексивному замыканиям соответственно. Обратное отношение записывается как R^{-1}, а области определения и значений – $dom(R)$ и $codom(R)$. Тождественное отношение на множестве A обозначается как $[A]$, что часто используется в обозначениях вида $[A]; R; [B] \triangleq R \cap (A \times B)$. Левая композиция отношений записывается как $R1; R2 \triangleq \{x, y \mid \exists z.(x, z) \in R1 \wedge (z, y) \in R2\}$. Непосредственные рёбра R обозначаются как $R|_{imm} \triangleq R \backslash (R; R)$.

В данном разделе описываются графы исполнения в наиболее общем виде, без привязки к конкретным моделям памяти или языкам.

Считаем, что анализируемая программа P состоит из последовательных подпрограмм отдельных потоков P_i : $P = ||_{i \in Tid} P_i$, где $||$ – оператор параллельной композиции программ, Tid – конечное множество идентификаторов потоков.

Определение 1. *Граф исполнения G* задаётся множеством вершин $G.E$, отображением $G.Lab$, задающим параметры операций над памятью, и бинарными отношениями на вершинах.

Множество $G.E$ делится на инициализирующие события вида $Init\ loc$ и неинициализирующие события вида $ThreadEvent\ tid\ n$, где:

- $loc \in Loc$ – адрес инициализации, где Loc – конечное множество адресов;
- $i \in Tid$ – номер потока;
- $n \in Q$ – порядковый номер внутри потока (нумерация плотным порядком упрощает формальное определение соответствия графов, см. раздел 5).

Обозначения $e.tid$ и $e.n$ для $e = ThreadEvent\ tid\ n$ соответствуют tid и n соответственно.

Функция $G.Lab$ сопоставляет событиям метки вида $type^{mode}(loc, val)$, где:

- $type \in \{R, W, F\}$ – тип операции (чтение, запись, барьер);
- $mode$ – один из режимов доступа, частично упорядоченных отношением “строже чем” ($\sqsupset$); конкретное множество режимов и их порядок определяется моделью памяти;
- $loc \in Loc$ – адрес памяти (для барьера не определено);
- $val \in Val$ – прочитанное/записанное значение (в случае барьера не определено), где Val – множество значений, которые могут храниться в памяти.

При этом инициализирующие события обрабатываются особым образом: $G.Lab(Init\ loc) = W^{mode_{init}}(loc, val_{init})$, где $mode_{init}$– режим доступа для

инициализирующих записей (например, в IMM $-rlx$), а val_{init}– начальное значение в памяти (как правило, 0).

Введём обозначения для множеств событий с определёнными метками: например, события с меткой чтения в режиме acq или более строгим будем обозначать как $G.R^{acq}$ (или просто R^{acq}, если граф очевиден из контекста).

Рёбра графа представляют собой отношения между событиями:

- программный порядок (program order): $G.po(x,y) \Leftrightarrow (x \in Init \wedge y \notin Init) \vee (x.tid = y.tid \wedge x.n < y.n)$;
- порядок согласованности (coherence order): $G.co = \cup_{l \in Loc} co_l$, где co_l – тотальный порядок на событиях записи по адресу l;
- наблюдение записанного значения (reads from, «читает-из»): $G.rf \subseteq \cup_{l \in Loc} G.W_l \times G.R_l$, где $G.rf(w,r) \Rightarrow G.Lab(w).val = G.Lab(r).val$, $codom(G.rf) = G.R$ и $G.rf$ является функциональным отношением;
- чтение до указанной записи: $G.fr = G.rf^{-1}; G.co$ (from-read, «читает-перед»).

В различных моделях памяти в граф исполнения могут добавляться другие отношения. Например, в IMM также есть отношение $rmw \subseteq \cup_{l \in Loc} [G.R_l]; po|_{imm}; [G.W_l]$, соответствующее паре событий чтения и записи в операции read-modify-write.

Введём понятие сужения графа на поток i: $G_i.E = \{e \in G.E | e.tid = i\}, G_i.Lab = G.Lab$.

Определение 2. *Графом исполнения программы* P называется такой граф исполнения G, что его сужение на любой поток i является *однопоточным графом исполнения* программы P_i. Соответствие подпрограммы потока и однопоточного графа исполнения определяется средствами операционной семантики, специфичной для языка ([15], [17], [18]). Мы не приводим подробностей здесь, скажем лишь, что такая семантика задаёт соответствие между выполнением инструкций языка и изменением графа исполнения. Так, для IMM выполнение инструкции $[x]^{rel} := 1$ соответствует добавлению в текущий граф вершины с очередным порядковым номером и меткой вида $W^{rel}(x, 1)$ и рёбер, отражающих синтаксические зависимости данной записи.

Определение 3. *Результатом работы графа* G называется функция $f : Loc \to Val$, отображающая адрес в последнее (согласно порядку co) записанное по нему значение. Декларативная модель памяти задаётся предикатом согласованности, которому должны удовлетворять графы исполнения программ.

Определение 4. *Результатом работы программы* P в модели памяти M является результат работы некоторого графа её исполнения, удовлетворяющего предикату согласованности M.

4. Описание используемых моделей памяти

В данном разделе описываются рассматриваемые модели памяти и их предикаты согласованности.

4.1. OCaml Memory Model

OCamlMM задана в [15] эквивалентными операционным и декларативным описаниями. Для доказательства корректности компиляции будет использоваться декларативное описание.

OCamlMM поддерживает два режима доступа: неатомарный na и атомарный at (схожи с pln и sc в C++). При этом память также разделена на неатомарные и атомарные адреса, и к конкретному адресу можно обратиться только операцией соответствующего режима.

В графе исполнения OCamlMM есть только операции чтения и записи, барьеры отсутствуют.

Перед рассмотрением предиката согласованности введём ещё несколько обозначений. Для отношения R будем обозначать Ri рёбра R, проходящие между вершинами одного потока, а Re –между вершинами разных потоков.

Определение 5. Исполнение называется OCamlMM-*согласованным*, если в соответствующем графе исполнения выполняются следующие аксиомы:

1) отношение hbo; $(co \cup fr)$ иррефлексивно, где
   $hbo \triangleq po \cup [E^{at}]; (co \cup rf); [E^{at}]$;
2) отношение $po \cup rfe \cup [E^{at}]; (coe \cup fre); [E^{at}]$ ациклично.

4.2. Intermediate Memory Model

Для построения схемы компиляции мы пользуемся IMM_{SC} [19] – расширением IMM, которое дополняет оригинальную модель [17] SC-операциями.

IMM определена декларативно. Полный предикат согласованности IMM достаточно сложен, поэтому мы рассмотрим лишь часть модели, которая будет необходима для построения схемы компиляции.

Синтаксис программ в IMM напоминает таковой в C++ – помимо инструкций атомарного чтения и записи есть инструкции барьеров памяти, а также операций read-modify-write. Пары событий чтения и записи, порождаемых инструкциями read-modify-write, связаны отношением $rmw \subseteq ([G.R]; po|_{imm}; [G.W])_{loc}$. Доступно несколько режимов доступа, упорядоченных следующим образом: $\sqsubset \triangleq \{ (rlx, acq), (rlx, rel), (acq, acqrel), (rel, acqrel), (acqrel, sc)\}$.

Введём ещё несколько обозначений. R_{loc} будем обозначать рёбра R, проходящие между вершинами с метками одного и того же адреса, $R_{\neq loc}$ – между вершинами с метками разных адресов.

Определение 6. Исполнение называется IMM-*согласованным*, если в соответствующем графе исполнения выполняются следующие аксиомы:

1) отношение hb; $(rf \cup co \cup fr)^+$ иррефлексивно, где
   $hb \triangleq (po \cup sw)^+$
   $sw \triangleq release; (rfi \cup po^?_{loc}; rfe); ([R^{acq}] \cup po; [F^{acq}])$
   $release \triangleq ([W^{rel}] \cup [F^{rel}]; po); rs$
   $rs \triangleq [W]; po_{loc}; [W] \cup [W]; (po^?_{loc}; rfe; rmw)^*$
2) операции read-modify-write являются атомарными: $rmw \cap (fre; coe) = \emptyset$
3) отношение ar ациклично, где
   $ar \supset rfe \cup bob$
   $bob \supset [R^{acq}]; po \cup po; [F] \cup [F]; po$
4) отношение psc_{base} ациклично, где
   $psc_{base} \triangleq ([E^{sc}] \cup [F^{sc}]; hb^?); scb; ([E^{sc}] \cup hb^?; [F^{sc}])$
   $scb \triangleq po \cup po_{\neq loc}; hb; po_{\neq loc} \cup hb_{loc} \cup co \cup fr$.

5. Схема компиляции

Предлагаемая схема компиляции OCamlMM в IMM описана в табл. 1. За основу взята схема компиляции OCamlMM в модель ARM из [15].

Табл. 1. Схема компиляции модели OCaml в промежуточную модель.
Table 1. Scheme of compilation of OCaml model into IMM

OCamlMM	IMM
$r := [x]^{na}$	$r := [x]^{rlx}$
$[x]^{na} := v$	$fence^{acqrel}; [x]^{rlx} := v$
$r := [x]^{at}$	$fence^{acq}; r := [x]^{sc}$
$[x]^{at} := v$	$fence^{acq}; exchg(x, v)$

Напомним, что схема компиляции корректна, если все возможные сценарии поведения скомпилированной программы являются допустимыми сценариями поведения исходной программы согласно исходной модели памяти. В случае декларативных моделей памяти это можно переформулировать так: если граф исполнения скомпилированной программы согласован с целевой моделью памяти, то соответствующий ему граф исполнения исходной программы согласован с исходной моделью памяти.

Чтобы формализовать понятие соответствия графов, опишем, чем отличаются графы исполнения скомпилированной программы и исходной. Во-первых, все вершины исходного графа сохраняются, а новые вершины добавляются согласно схеме компиляции. Во-вторых, порядок согласованности сохраняется, так как события записи в новом графе те же, что и в исходном. В-третьих, при компиляции используются инструкции CAS, поэтому в графе появляется отношение rmw. Наконец, так как инструкции CAS в графе выражаются с помощью пар чтения и записи, в отношении "читает-перед" появляются новые рёбра, которые указывают на значения, наблюдаемые при исполнении CAS. Формально эти условия описываются следующим образом.

Для краткости события вида $ThreadEvent\ i\ n$ будем обозначать как $\langle i, n\rangle$.

Определение 7. Граф исполнения по OCamlMM G_O, события в потоках которого пронумерованы натуральными числами, *соответствует* графу исполнения по IMM G_I, если выполняются следующие условия:

1) $G_I.E = G_O.E \cup \{\langle i, n-0.5\rangle | \langle i,n\rangle \in G_O.(E \backslash R^{na})\} \cup \{\langle i, n-0.25\rangle \mid \langle i,n\rangle \in G_O.W^{at}\}$
2) $G_I.Lab = \{e \to (t,l,rename(m),v)) \mid G_O.Lab(e) = (t,l,m,v)\}$
   $\cup \{\langle i,n-0.5\rangle \to (F,-,acq,-) \mid \langle i,n\rangle \in G_O.E^{at}\}$
   $\cup \{\langle i,n-0.5\rangle \to (F,-,acqrel,-) \mid \langle i,n\rangle \in G_O.W^{na}\}$
   $\cup \{\langle i,n-0.25\rangle \to (R,l,sc,v) \mid \langle i,n\rangle \in G_O.W^{at} \wedge G_O.Lab(\langle i,n\rangle) = (W,l,at,-) \wedge v \in Val\}$,
   где $rename(na) = rlx, rename(at) = sc$
3) $G_I.rmw = \{(r,w) | w \in G_I.W^{sc} \wedge r \in G_I.R^{sc} \wedge r.tid = w.tid \wedge r.n = w.n - 0.25\}$
4) $G_I.co = G_O.co$
5) $G_I.rf \supseteq G_O.rf$.

Видно, что нумерация событий в потоках рациональными числами позволяет добавлять в граф новые события, по программному порядку находящиеся между существующими.

Будем рассматривать граф исполнения скомпилированной IMM-программы как граф исполнения исходной OCamlMM-программы. В самом деле, при компиляции лишь добавляются новые вершины и изменяются метки у существующих. Выполнив обратное преобразование, можно получить исходный граф исполнения в OCamlMM. Поэтому для

графа исполнения в IMM можно анализировать также и его согласованность по OCamlMM, заменяя в предикате согласованности OCamlMM режимы с na на rlx и с at на sc.

Теперь можно объяснить выбор данной схемы компиляции: использование инструкций compare-and-swap и барьеров накладывает на исполнение программы в IMM условия, достаточно строгие для выполнения в согласованном по IMM графе первого и второго условий согласованности по OCamlMM соответственно.

Рассмотрим несколько примеров, демонстрирующих необходимость расположения барьеров.

Рис. 5 демонстрирует, что при компиляции инструкции неатомарной записи необходимо использовать именно барьер в режиме $acqrel$, а не rel. Такая оптимизация может показаться разумной, т.к. sw (и, следовательно, hb) в IMM может начинаться с F^{rel}. Описанное поведение должно быть запрещено, так как в графе есть запрещённый в OCamlMM цикл $po; rf; po; rf; po; [E^{sc}]; fr; [E^{sc}]$. Однако IMM разрешает такое поведение, так как после первого ребра rf нет ни R^{acq}, ни F^{acq}. Если бы вместо F^{rel} во втором потоке располагался F^{acqrel}, как это предполагает схема компиляции, то через первое ребро rf прошёл бы hb, и результирующий граф был бы запрещён IMM.

$x = y = 0$		
$[x]^{at} := 1;$	$a := [y]^{na};//1$	$b := [z]^{na};//1$
$[y]^{na} := 1;$	$[z]^{na} := 1;$	$c := [x]^{at};//0$

$x = y = 0$		
$fence^{acq};$	$a := [y]^{rlx};//1$	$b := [z]^{rlx};//1$
$exchg(x,1);$	$fence^{rel};$	$fence^{acq};$
$fence^{rel};$	$[z]^{rlx} := 1;$	$c := [x]^{sc};//0$
$[y]^{rlx} := 1;$		

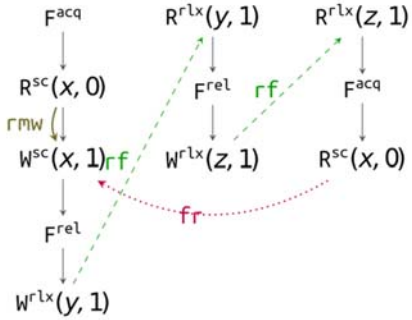

Рис. 5. Пример программы, результата её компиляции схемой с rel-барьером перед инструкциями rlx-записи и IMM–согласованного графа её исполнения
Fig. 5. An example of a program, the result of its compilation with a scheme using rel fence before rlx write instructions and its IMM–consistent execution graph.

Рис.6 демонстрирует, что барьер перед CAS удалить нельзя. В нём есть цикл $po; rf; po; rf; po; [E^{sc}]; fr$, запрещённый в OCamlMM, но разрешённый в IMM: sw должен оканчиваться либо R^{acq}, либо R с последующим барьером. Ни того, ни другого во втором потоке нет, поэтому между первым и вторым потоком нельзя проложить hb. Если перед R^{sc} во втором потоке расположить acq-барьер, то результирующий граф будет также запрещён и IMM.

6. Доказательство корректности компиляции

Напомним формулировку теоремы о корректности компиляции.

Теорема 2.1. Пусть G_I – IMM-согласованный граф исполнения, соответствующий графу исполнения G_O в OCamlMM. Тогда G_O является OCamlMM-согласованным.

Как было показано выше, OCamlMM-согласованность G_O следует из OCamlMM-согласованности G_I. Поэтому далее будем доказывать два условия OCamlMM-согласованности для G_I.

$x = y = 0$		
$[x]^{at} := 1;$ $[y]^{na} := 1;$	$a := [y]^{na};//$ $[z]^{at} := 1;$	$b := [z]^{at};//1$ $c := [x]^{at};//0$

$x = y = 0$		

$exchg(x, 1);$
$fence^{acqrel};$
$[y]^{rlx} := 1;$
	$a := [y]^{rlx}; //1$
	$exchg(z, 1);$
		$fence^{acq};$
		$b := [z]^{sc}; //1$
		$fence^{acq};$
		$c := [x]^{sc};//0$

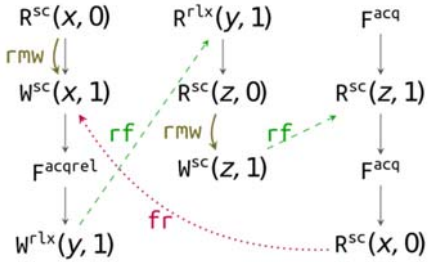

Рис. 6. Пример программы, результата её компиляции схемой, не использующей барьер перед инструкциями $compare - and - swap$, и IMM–согласованного графа его исполнения
Fig. 6. An example of a program, the result of its compilation with a scheme not using a fence before $compare - and - swap$ instructions and its IMM–consistent execution graph.

6.1. Первое условие OCamlIMM-согласованности

Теорема 6.1. Отношение $hbo; (co \cup fr)$ иррефлексивно.

Докажем, что с выбранной схемой компиляции $hbo \subseteq hb$. С учётом этого доказательство теоремы тривиально: согласно IMM-согласованности, отношение $hb; (rf \cup co \cup fr)$ иррефлексивно.

Для этого сначала покажем, что последовательность co по sc-событиям порождает hb.

Лемма 6.2. Порядок sc-записей согласуется с отношением happens-before:

$[E^{sc}]; co; [E^{sc}] \subseteq hb$.

Доказательство. Заметим, что $[E^{sc}]; co; [E^{sc}]$ транзитивно. Тогда можно перейти к рассмотрению непосредственных co-соседей.

Предположим, что $[E^{sc}]; co|_{imm}; [E^{sc}] \subseteq [E^{sc}]; rf; [E^{sc}]; po$. Тогда доказательство тривиально: rf по sc событиям порождает hb, как и следующий за ним po. Значит, остаётся доказать утверждение о включении в $[E^{sc}]; rf; [E^{sc}]; po$.

Рассмотрим два события $w1, w2 \in W^{sc}$ – непосредственных co-соседей. По схеме компиляции перед $w2$ следуют $f \in F^{acq}$ и $r \in R^{sc}$, причём $rmw(r, w2)$.

Покажем, что $rf(w1, r)$. В самом деле, рассмотрим событие записи w', из которого читает r. Так как обращения по одному и тому же адресу имеют один и тот же режим, то $w' \in W^{sc}$.

Пусть $w1 \neq w'$. Тогда либо $co(w', w1)$, либо, наоборот, $co(w1, w')$. В первом случае нарушается атомарность rmw между $w2$ и r. Во втором случае получается, что между co-соседями $w1$ и $w2$ расположен w', что невозможно. □

Лемма 6.3. Отношение happens-before в OCamlMM является подмножеством happens-before в IMM: $hbo \subseteq hb$.

Доказательство. $hbo \triangleq po \cup [E^{sc}]; (co \cup rf); [E^{sc}]$. По предыдущей лемме co, ограниченный на sc, входит в hb.

rf по sc событиями порождает sw, и, следовательно, hb. □

Таким образом, $hbo \subseteq hb$, из чего, согласно IMM-согласованности, следует, что $hbo; (co \cup fr)$ иррефлексивно. □

6.2. Второе условие OCamlIMM-согласованности

Сначала докажем утверждения, которые позволят нам находить барьеры в программном порядке между событиями.

Лемма 6.4. В программном порядке между произвольным событием и записью располагается барьер: $[E \backslash F]; po; [W] \subseteq po; ([F^{acqrel}]; po; [E^{rlx}] \cup [F^{acq}]; po; [E^{sc}]); [W] \cup rmw$.

Доказательство. Согласно схеме компиляции, инструкция барьера располагается либо непосредственно перед инструкцией неатомарной записи, либо перед инструкцией `compare-and-swap`. Таким образом, барьера между событием и po-следующей записью может не быть, только если это событие чтения в rmw. □

Лемма 6.5. В программном порядке между rlx и sc событиями располагается барьер: $[E^{rlx}]; po; [E^{sc}] \subseteq po; [F^{acq}]; po$.

Доказательство. Согласно схеме компиляции, все инструкции в режиме sc предваряются acq барьерами. □

Кроме того, понадобятся следующие факты из алгебры:

Лемма 6.6. Цикл из рёбер двух типов можно представить в виде чередующихся участков рёбер каждого типа: $(x \cup y)^+ = y^+ \cup y^*; (x; y^*)^+$, где x, y – произвольные отношения.

Лемма 6.7. Отношение $x \cup y$ ациклично, если ацикличны отношения x, y и $x^+; y^+$.

Наконец, мы можем перейти ко второму условию согласованности по OCamlMM.

Теорема 6.8. Отношение $po \cup rfe \cup [E^{sc}]; (coe \cup fre); [E^{sc}]$ ациклично.

Сгруппируем первые два отношения в объединении. Тогда по лемме 6.7 нужно показать ацикличность следующих отношений:

- $po \cup rfe$;
- $[E^{sc}]; (coe \cup fre); [E^{sc}]$;
- $(po \cup rfe)^+; ([E^{sc}]; (coe \cup fre); [E^{sc}])^+$, что эквивалентно ацикличности $[E^{sc}]; (po \cup rfe)^+; [E^{sc}]; ([E^{sc}]; (coe \cup fre); [E^{sc}])^+$.

Второе и третье утверждение докажем, показав, что соответствующие отношения лежат в $([E^{sc}]; scb; [E^{sc}])^+ \subseteq psc_{base}{}^+$, где, напомним, $scb \triangleq po \cup po_{\neq loc}; hb; po_{\neq loc} \cup hb_{loc} \cup co \cup fr$ и $psc_{base} \triangleq ([E^{sc}] \cup [F^{sc}]; hb^?); scb; ([E^{sc}] \cup hb^?; [F^{sc}])$. В свою очередь, ацикличность psc_{base} следует из IMM-согласованности графа.

Теперь видно, что второе утверждение верно по определению scb. По этой же причине для доказательства третьего утверждения будет достаточно показать, что $[E^{sc}]; (po \cup rfe)^+; [E^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Теорема 6.9. Отношение $po \cup rfe$ ациклично.

Доказательство. Вновь воспользуемся леммой 6.7 и разложим условие ацикличности объединения на ацикличность отношений po (следует из IMM-согласованности) и rfe (двух и более таких рёбер подряд идти не может, т.к. их концы имеют разные типы), а также $po^+; rfe^+$, что эквивалентно ацикличности $po; rfe$.

Пусть такой цикл существует. Покажем, что это противоречит условию ацикличности ar (что следует из IMM-согласованности). Напомним, что $ar \supset rfe \cup bob$ и $bob \supset [R^{acq}]; po \cup po; [F] \cup [F]; po$.

По лемме 6.4 перед событием записи, которой начинается ребро rfe, есть барьер $F^{\supseteq acq}$, либо весь po является rmw. В первом случае внутри po есть барьер, а такое отношение лежит в $bob \subseteq ar$. Во втором случае rmw начинается с R^{sc}, и такое ребро $po \supseteq rmw$ также содержится в bob. Наконец, $rfe \subseteq ar$. □

Теперь для доказательства второго условия согласованности по OcamlMM осталось доказать утверждение $[E^{sc}]; (po \cup rfe)^+; [E^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Теорема 6.10. Последовательность из рёбер po и rfe между вершинами sc состоит из рёбер scb между вершинами sc: $[E^{sc}]; (po \cup rfe)^+; [E^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Доказательство. Сначала введём утверждение, которое позволит отбрасывать rfe-рёбра.

Лемма 6.11. Рёбра rfe, у которых один из концов - sc, входят в scb: $[E^{sc}]; rfe \cup [W \backslash Init]; rfe; [E^{sc}] \subseteq [E^{sc}]; scb; [E^{sc}]$.

Доказательство. Если один из концов ребра rf является sc, таким же является и второй (исключение–чтение из инициализирующих записей, которые в IMM являются rlx). Тогда $[E^{sc}]; rf; [E^{sc}] \subseteq [E^{sc}]; hb_{loc}; [E^{sc}] \subseteq [E^{sc}]; scb; [E^{sc}]$. □

По лемме 6.6 имеем

$[E^{sc}]; (po \cup rfe)^+; [E^{sc}] = [E^{sc}]; (rfe^+ \cup rfe^*; (po; rfe^*)^+); [E^{sc}] = [E^{sc}]; (rfe \cup rfe^?; (po; rfe^?)^+); [E^{sc}]$.

Начальные участки rfe, если они есть, можно отбросить по лемме 6.11. Рассмотрим оставшееся транзитивное замыкание:

$(po; rfe^?)^+ = po; (po; rfe)^*; po^? = po; po^? \cup po; (po; rfe)^+; po^? = po \cup (po; rfe)^+; po^?$.

В первом случае отношение сводится к $po \subseteq scb$. Во втором, если последним ребром является rfe, то его можно отбросить по лемме 6.11. Остаётся случай $(po; rfe)^+; po$. В каждой паре $po; rfe$ можно применить лемму 6.4. В результате нужно доказать такое утверждение:

$[(W \cup R)^{sc}]; (po; ([F^{acqrel}]; po; [E^{rlx}]; rfe \cup [F^{acq}]; po; [E^{sc}]; rfe) \cup rmw; rfe)^+; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Воспользуемся леммой 6.6: либо транзитивное замыкание состоит только из пар $rmw; rfe$, либо такие пары рёбер могут следовать после $po; ([F^{acqrel}]; po; [E^{rlx}]; rfe \cup [F^{acq}]; po; [E^{sc}]; rfe)$. В первом случае замыкание имеет вид $hb_{loc} \subseteq scb$, а так как оно заканчивается sc-событием, оставшееся ребро po также пройдёт по sc и образует scb. Во втором случае рассмотрим, что именно находится под транзитивным замыканием:

$(po; ([F^{acqrel}]; po; [E^{rlx}]; rfe \cup [F^{acq}]; po; [E^{sc}]; rfe); (rmw; rfe)^*)^+ = ((po; [F^{acq}]; ([F^{acqrel}]; po; [E^{rlx}] \cup [F^{acq}]; po; [E^{sc}]); (rfe; rmw)^*; rfe)^+) = ((po; [F^{acq}]; C; rfe))^+$,

где $C = C1 \cup C2 = [F^{acqrel}]; po; [E^{rlx}]; (rfe; rmw)^* \cup [F^{acq}]; po; [E^{sc}]; (rfe; rmw)^*$.

Заметим, что $(po; [F^{acq}]; C; rfe)^+ = po; [F^{acq}]; (C; rfe; po; [F^{acq}])^*; C; rfe$. В результате необходимо доказать следующее:

$[(W \cup R)^{sc}]; po; [F^{acq}]; (C; rfe; po; [F^{acq}])^*; C; rfe; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Заметим, что

$(C; rfe; po; [F^{acq}])^* = (([F^{acqrel}]; po; [E^{rlx}] \cup [F^{acq}]; po; [E^{sc}]); (rfe; rmw)^*; rfe; po; [F^{acq}])^* \subseteq hb^?$, так как

$hb \triangleq (po \cup sw)^+$,

$sw \supseteq release; rfe; po; [F^{acq}]$,

$release \triangleq ([Wrel] \cup [F^{rel}]; po); rs$,

$rs \supseteq (rfe; rmw)^*$.

Вспомним, что $po_{\neq loc}; hb; po_{\neq loc} \subseteq scb$. Воспользуемся тем, что po между между событием чтения/записи и барьером образует именно $po_{\neq loc}$. Тогда видно, что $[(W \cup R)^{sc}]; po; [F^{acq}] \subseteq [E^{sc}]; po_{\neq loc}$.

Остаётся показать, что $[E^{sc}]; po_{\neq loc}; hb; C; rfe; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$. Для этого перепишем $C = C1 \cup C2$ и докажем утверждение для $C1$ и $C2$ по отдельности.

- Покажем, что $[E^{sc}]; po_{\neq loc}; hb^?; C1; rfe; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Заметим, что по лемме 6.5 в последнем ребре po найдётся acq-барьер, с помощью которого можно будет построить ребро hb:

$C1; rfe; po; [(W \cup R)^{sc}] = [F^{acqrel}]; po; [E^{rlx}]; (rfe; rmw)^*; rfe; po; [(W \cup R)^{sc}] = [F^{acqrel}]; po; [E^{rlx}]; (rfe; rmw)^*; rfe; [E^{rlx}]; po; [F^{acq}]; po; [(W \cup R)^{sc}] \subseteq hb; po_{\neq loc}; [E^{sc}]$.

Тогда

$[E^{sc}]; po_{\neq loc}; hb^?; C1; rfe; po; [(W \cup R)^{sc}] \subseteq [E^{sc}]; po_{\neq loc}; hb^?; hb; po_{\neq loc}; [E^{sc}] \subseteq [E^{sc}]; scb; [E^{sc}]$.

- Покажем, что $[E^{sc}]; po_{\neq loc}; hb^?; C2; rfe; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}])^*$.

Заметим, что последовательность пар рёбер $rfe; rmw$ входит в scb:

$C2 = [F^{acq}]; po; [E^{sc}]; (rfe; rmw)^* \subseteq po_{\neq loc}; [E^{sc}]; ([E^{sc}]; rfe; [E^{sc}]; rmw; [E^{sc}])^*; [E^{sc}] \subseteq po_{\neq loc}; [E^{sc}]; ([E^{sc}]; scb; [E^{sc}])^*; [E^{sc}]$.

В этом случае

$[E^{sc}]; po_{\neq loc}; hb^?; C2; rfe; po; [(W \cup R)^{sc}] \subseteq [E^{sc}]; po_{\neq loc}; hb^?; po_{\neq loc}$. Тогда

$[E^{sc}]; ([E^{sc}]; scb; [E^{sc}])^*; [E^{sc}]; rfe; po; [(W \cup R)^{sc}] \subseteq ([E^{sc}]; scb; [E^{sc}]); ([E^{sc}]; scb; [E^{sc}])^*; ([E^{sc}]; scb; [E^{sc}])^2$. □

7. Связанные работы

Проблема корректности схем компиляции из OCamlMM и в IMM рассматривается и в других работах. Так, в [15] приводится схема компиляции OCamlMM в модель архитектуры ARMv8 [11]. В ней, в отличие от предложенной нами схемы, при компиляции неатомарной записи используется барьер F^{acq}, а не F^{acqrel}. Это объясняется тем, что в модели ARMv8 отношение ob (аналог ar в IMM) включает в себя $rfe \cup fre \cup coe$ по неатомарным операциям и po с acq-барьером перед событием записи, поэтому в последовательности рёбер вида $(po; rfe)^+$ не требуется rel-барьер.

В [17] приведена схема компиляции моделей RC11 [18] в IMM. Так как предикат согласованности IMM схож с таковым в RC11, то эта схема компиляции лишь незначительно отличается от тривиальной.

В [20], [21] и [22] разработана схема компиляции модели Promising [23] в модель ARMv8 [11].

Доказательство корректности данной схемы значительно сложнее, так как модель Promising, в отличие от модели ARMv8, задана с помощью операционной семантики, что требует при доказательстве корректности компиляции задавать соответствие между графами исполнения и последовательностями шагов операционной семантики. В [17] идея этого доказательства была обобщена для построения корректной схемы компиляции IMM в ARMv8.

8. Заключение

В данной работе представлена корректная схема компиляции `OCamlMM` в `IMM`, дающая корректную схему компиляции `OCamlMM` в модель Power. Для доказательства корректности было доказано, что в графах исполнения скомпилированных программ `IMM`-согласованность влечёт `OCamlMM`-согласованность. Так как `IMM` является более слабой моделью, чем `OCamlMM`, в предложенной схеме компиляции задействуются барьеры памяти и инструкции compare-and-swap, которые накладывают более строгие условия на поведение скомпилированной программы.

Свойство локальной свободы от гонок может быть реализовано и в других моделях памяти– например, изучается возможность включить его в модель памяти C++ [24]. Данная работа может быть использована для построения схем компиляции таких моделей.

В некоторых опубликованных доказательствах корректности компиляции впоследствии были найдены неточности. Например, [18] демонстрирует ошибку в схеме компиляции модели C++ в модель Power, [17] – в схеме компиляции модели Promising в модель Power. Чтобы избежать этого, доказательство в данной статье в дальнейшем планируется формализовать в Coq с использованием имеющейся формальной модели `IMM` [25].

Список литературы / References

[1]. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, vol. C-28, isssue 9, 1979, pp. 690–691.

[2]. Owens S., Sarkar S., and Sewell P. A better x86 memory model: x86-TSO. Lecture Notes in Computer Science, vol. 5674, 2009, pp. 391–407.

[3]. Alglave J., Maranget L., Sarkar S., and Sewell P. Litmus: Running Tests Against Hardware. Lecture Notes in Computer Science, vol. 6605, 2011, pp. 41–44.

[4]. Batty M., Owens S., Sarkar S., Sewell P., and Weber T. Mathematizing C++ concurrency. In Proc. of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2011, pp. 55–66.

[5]. Manson J., Pugh W., and Adve S.V. The Java memory model. In Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2005, pp. 378–391.

[6]. ECMA International. 2018b. ECMAScript 2018 Language Specification – Memory Model. Available at: https://www.ecma-international.

[7]. Watt C., Rossberg A., Pichon-Pharabod J. Weakening WebAssembly. Proceedings of the ACM on Programming Languages, vol. 3, issue OOPSLA, 2019, 28 p.

[8]. Sewell P., Sarkar S., Owens S., Nardelli F.Z., and Myreen M. O. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Communications of the. ACM, vol. 53, issue 7, 2010, pp. 89-97.

[9]. Alglave J., Maranget L., and Tautschnig M. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Transactions on Programming Languages and Systems, vol, 36, issue 2, 2014, pp. 7:1–7:74.

[10]. Sarkar S., Sewell P., Alglave J., Maranget L., and Williams D. Understanding POWER Multiprocessors. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 175-186.

[11]. Pulte C., Flur S., Deacon W., French J., Sarkar S., and Sewell P. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for armv8. Proceedings of the ACM on Programming Languages, vol. 2, issue POPL, 2017, pp. 19:1–19:29.

[12]. ARM Limited. ARM architecture reference manual: ARMv7-A and ARMv7-R edition, 2014. Available at: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf.

[13]. Flur S., Gray K., Pulte C., Sarkar S., Sezgin A., Maranget L, Deacon W., and Sewell P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 608–621.

[14]. Dolan S., Sivaramakrishnan K., and Madhavapeddy A. Bounding data races in space and time. Extended version. Available at: http://kcsrk.info/papers/pldi18-memory.pdf.

[15]. Dolan S., Sivaramakrishnan K., and Madhavapeddy A. Bounding data races in space and time. In Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, pp. 242-255.

[16]. IBM Power Systems. IBM power systems facts and features: enterprise and scale-out systems with POWER8 processor technology. Available at: https://www.ibm.com/downloads/cas/JDRZDG0A.

[17]. Podkopaev A., Lahav O., and Vafeiadis V. Bridging the gap between programming languages and hardware weak memory models. Proceedings of the ACM on Programming Languages, vol. 3, issue POPL, 2019, pp. 69:1–69:31.

[18]. Lahav O., Vafeiadis V., Kang J., Hur C.-K., and Dreyer D. Repairing sequential consistency in C/C++11. Programming Language Design and Implementation, vol. 52, issue 6, 2017, pp. 618–632.

[19]. Podkopaev A., Lahav O., Melkonian O., and Vafeiadis V. Extending Intermediate Memory Model with SC accesses. Technical report. Available at: http://plv.mpi-sws.org/imm/immsctr.pdf. 2019.

[20]. Подкопаев А.В. Операционные методы в приложениях к слабым моделям памяти. Диссертация на соискание учёной степени кандидата физико-математических наук. Санкт-Петербург, 2018, 190 стр. / Podkopaev A.V. Operational methods in applications to weak memory models. The dissertation for the degree of candidate of physical and mathematical sciences. St. Petersburg, 2018,190 p. (in Russian).

[21]. Подкопаев А.В., Лахав О., Вафеядис В. О корректности компиляции подмножества обещающей модели памяти в аксиоматическую модель ARMv8.3. Научно-технические ведомости СПбГПУ, том 10, № 4, 2017, стр. 51–69 / Podkopaev A.V., Lahav O., Vafeyadis V. On the correct compilation of a subset of a promising memory model into an axiomatic model ARMv8.3. St. Petersburg Polytechnic University Journal of Engineering Science and Technology, vol. 10, № 4, pp. 51-69 (in Russian).

[22]. Подкопаев А.В., Лахав О., Вафеядис В. Обещающая компиляция в ARMv8.3. Труды ИСП РАН, том 29, вып. 5, 2017, стр. 149-164 / 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.

[23]. Kang J., Hur C.-K., Lahav O., Vafeiadis V., and Dreyer D. A promising semantics for relaxed-memory concurrency. In Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, 2017, pp. 175–189.

[24]. Doherty S. Local data-race freedom and the C11 memory model. Surrey Concurrency Workshop Absracts, 2019. Available at: https://cw-srepls-19.github.io/abstracts.html#doherty/

[25]. Coq formalization of the Intermediate Memory Model. Available at: https://github.com/weakmemory/imm. 2019.

Информация об авторах / Information about authors

Егор Сергеевич НАМАКОНОВ – студент магистратуры Санкт-Петербургского государственного университета, член группы исследования слабых моделей памяти в JetBrains Research. Сферы научных интересов: слабые модели памяти, верификация ПО.

Egor Sergeevich NAMAKONOV – a Master's student in St Petersburg University, a member of weak memory model research group in JetBrains Research. Research interests: weak memory models, software verification.

Антон Викторович ПОДКОПАЕВ – доцент НИУ ВШЭ (СПб), руководитель группы исследования слабых моделей памяти в JetBrains Research, постдок Институт им. Макса Планка: Программные Системы. Сферы научных интересов: слабые модели памяти, верификация ПО, функциональное программирование.

Anton Viktorovich PODKOPAEV – an associate professor in NRU HSE (SPb), the weak memory model group leader in JetBrains Research, a postdoc in MPI-SWS. Research interests: weak memory models, software verification, functional programming.