DOI: 10.25559/SITITO.020.202401.149-156
УДК 004.4'422

Оригинальная статья

# Эффективное построение переупорядочиваний множества операций с памятью в многопоточной программе

**И. В. Андреев, К. И. Владимиров***
ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация
Адрес: 141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9
* konstantin.vladimirov@gmail.com

**Аннотация**
Данная работа посвящена различным методам построения всевозможных линейных порядков для частично упорядоченных множеств. Сложность задачи заключается в том, что даже для множества небольшого размера, построение и проверка на какой-либо признак его всевозможных линейных порядков может приводить к необходимости перебора огромного количества вариантов. Частным случаем этой общей задачи является построение всевозможных переупорядочений для операций с памятью в многопоточных программах с учётом модели памяти. Цель этой работы – это сокращение пространства перебора в этом случае. В качестве важного практического примера рассматриваются модели памяти, позволяющие буферизацию загрузок. При этом оказывается возможным установить классы эквивалентности над перестановками операций. Это в свою очередь имеет большое значение для верификации памяти в многопоточных системах. На сегодняшний день для такого рода верификации применяются литмус-тесты, которые имеют свои ограничения. В данной работе предложен алгоритм, который учитывает классы эквивалентности при переупорядочиваниях операций с памятью, что позволяет существенно сократить как количество необходимых перестановок, так и время, затраченное на их перебор. Результаты исследования позволяют эффективно обобщить концепцию литмус-тестов на произвольные частично упорядоченные множества и существенно расширить их применимость.

**Ключевые слова:** многопоточность, переупорядочение операций, модели памяти, линейный порядок, литмус-тесты

**Конфликт интересов:** авторы заявляют об отсутствии конфликта интересов.

**Для цитирования:** Андреев И. В., Владимиров К. И. Эффективное построение переупорядочиваний множества операций с памятью в многопоточной программе // Современные информационные технологии и ИТ-образование. 2024. Т. 20, № 1. С. 149-156. https://doi.org/10.25559/SITITO.020.202401.149-156

Original article

# Efficient Reordering of Multiple Memory Operations in a Multithreaded Program

**I. V. Andreev, K. I. Vladimirov***
Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation
Address: 9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation
* konstantin.vladimirov@gmail.com

## Abstract
This paper is devoted to various methods of constructing all kinds of linear orders for partially ordered sets. The complexity of the problem lies in the fact that even for a set of small size, constructing and checking for any feature of its possible linear orders can lead to the need of enumeration of huge number of options. A special case of this general problem is the construction of all possible reorderings for memory operations in multithreaded programs, taking into account the memory model. The goal of this work is to reduce the search space in this case. An important practical example is the consideration of memory models that allow buffering of downloads. In this case, it becomes possible to establish equivalence classes over permutations of operations. This in turn is of great importance for memory verification in multithreaded systems. Today, litmus tests are used for this type of verification, which have their limitations. In this paper, an algorithm is proposed that takes into account equivalence classes when reordering memory operations, which can significantly reduce both the number of necessary permutations and the time spent on their enumeration. The results of the study allow us to effectively generalize the concept of litmus tests to arbitrary partially ordered sets and significantly expand their applicability.

**Keywords:** multithreading, reordering of operations, memory models, linear order, litmus tests

**Conflict of interests:** The authors declare no conflict of interest.

**For citation:** Andreev I.V., Vladimirov K.I. Efficient Reordering of Multiple Memory Operations in a Multithreaded Program. *Modern Information Technologies and IT-Education.* 2024;20(1):149-156. https://doi.org/10.25559/SITITO.020.202401.149-156

## Введение

Актуальной проблемой в анализе многопоточных программ является генерация различных порядков выполнения операций, возможных при их исполнении [1, 2]. Это важно для доказательства различных свойств многопоточных алгоритмов и для проверки корректности компьютерных систем, выполняющих эти алгоритмы [3-6]. В данной статье мы сосредоточимся на эффективном решении задачи переупорядочения множеств операций с памятью. Это прикладное направление находит своё применение в системах верификации когерентности памяти в многопоточных системах, таких как литмус-тесты [7, 8]. Однако, литмус-тесты ориентированы на исследование ограниченных наборов с жёсткими рамками. Ниже мы продемонстрируем, как ослабить эти ограничения и обобщить концепцию литмус-тестов на произвольные частично упорядоченные множества операций с учётом модели памяти.

Частично упорядоченные множества

**Частично упорядоченное множество** – множество из элементов, упорядоченных транзитивным, антисимметричным отношением . При этом для любых выполняется одно из трёх условий.

1. $x <_p y$
2. $y <_p x$
3. $x, y$ – не сравнимы

**Отношение покрытия** $\prec_p$ – отношение, возникающее между элементами $x$ и $y$ в том случае, если не существует такого элемента $z$, что $x \prec_p z$ и при этом $z \prec_p y$. Можно назвать его отношением ближайшего соседства в частичном порядке.

**Линейный порядок элементов** – биекция $\sigma$: $P \to [n]$, *такая, что* $x \prec_p y \to \sigma_x < \sigma_y$. *По сути, это топологическая сортировка графа с рёбрами в виде отношений частичного порядка.*

Элемент $x$, такой что $\nexists\, y <_p x$ называется **минимальным** в частично упорядоченном множестве. Количество перестановок $l(P)$ задаётся формулой 1 (см. например [9]).

$$l(P) = \sum_{x \in \min(P)} l(P \setminus x) \qquad (1)$$

Задача перебора всевозможных линейных порядков относится к классу #P-полных [9]. Для сколько-нибудь больших частично упорядоченных множеств (от 30 элементов и больше) количество всевозможных порядков может превышать $10^{18}$ [10, 11]. Разумеется, полное построение, анализ и хранение такого объема частичных порядков затруднительны.

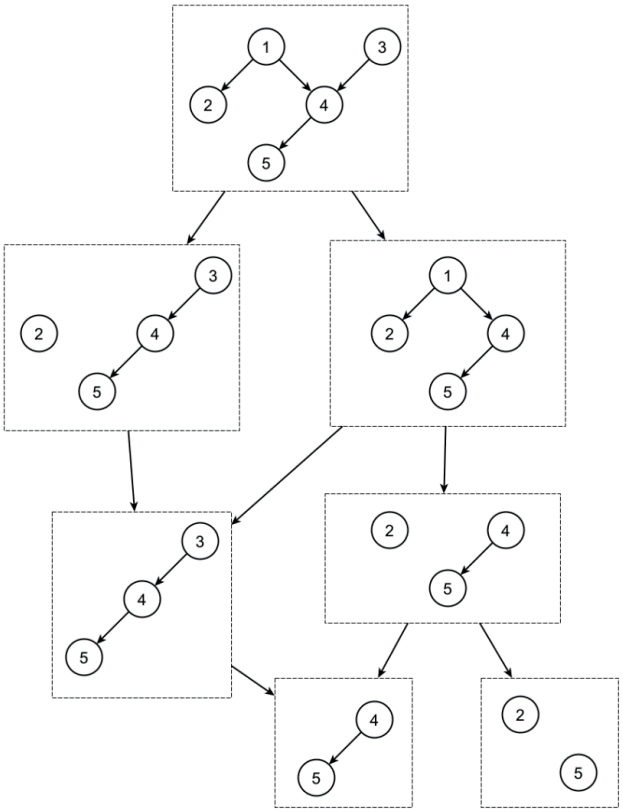
Для решения задачи полного построения линейных порядков можно использовать представление всех возможных линейных порядков в виде дерева, где вершинами являются элементы множества, а рёбра представляют отношение $<_p$ в исследуемом линейном порядке. Это дерево может быть явно построено таким образом, что все линейные порядки представлены путями от корня дерева до каждого из его листьев.

## Алгоритм прямого построения всевозможных линейных порядков

Обход множества и построение дерева происходит рекурсивно, по формуле (2), аналогично формуле (1).

$$tree(P) = \bigcup_{x \in \min(P)} x \to tree(P \setminus x) \qquad (2)$$

В начале алгоритма выбираются минимальные элементы из частично упорядоченного множества. Для каждого из этих элементов производится добавление его в текущую генерируемую перестановку, а затем он удаляется из множества. После этого алгоритм повторяется для нового частично упорядоченного множества.



Р и с. 1. Пример обхода частично упорядоченного множества

F i g. 1. An example of traversing a partially ordered set

*Источник:* здесь и далее в статье все таблицы и рисунки составлены авторами.

*Source:* Hereinafter in this article all tables and figures were made by the authors.

Обход частично упорядоченного множества, изображённого в самом верхнем блоке показан на рисунке 1.

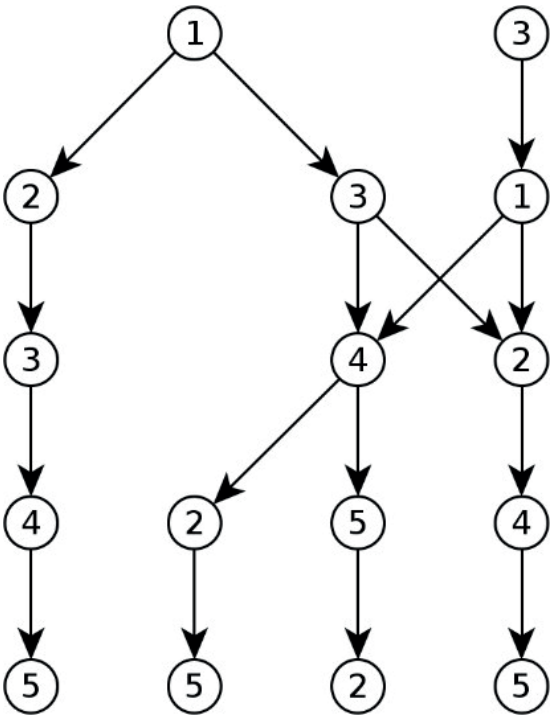
Алгоритм рекурсивно генерирует перестановку для всех элементов множества. Псевдокод алгоритма приведён на рисунке 2.

```
def  traversal(P, Path ):
  Xs = getMin(P);
  for X in Xs:
    Path.append(X)
    traversal(P \ X, Path)
```

Р и с. 2. Псевдокод для обхода частично упорядоченного множества

F i g. 2. Pseudocode for traversing a partially ordered set

Полное дерево показано на рисунке 3. Его представление упаковано в ориентированный ациклический граф для компактности представления (объединены одинаковые ветки).




Р и с. 3. Пример построенного дерева (сжатого в ОАГ) линейных порядков
F i g. 3. An example of a tree (compressed in OAG) of linear orders

При построении дерева алгоритмом с рис. 2, в памяти хранятся только текущая генерируемая ветвь, минимальные элементы с недостроенных ветвей и само частично упорядоченное множество. Таким образом, расход памяти можно асимптотически оценить как $O(\|P\|^2)$.

Перебор всех возможных линейных порядков может оказаться излишним, особенно когда в частично упорядоченном множестве присутствуют элементы, принадлежащие одному классу эквивалентности, перестановки внутри которого не требуют рассмотрения. Эти классы эквивалентности могут содержать значительное количество элементов, что позволяет существенно сократить время анализа.

## Введение классов эквивалентности

Рассмотрим всевозможные линейные перестановки $S_n$. Две перестановки в $S_n$ считаются эквивалентными, если одна может быть преобразована в другую путём замены подпоследовательности элементов теми же элементами, переставленными определённым образом. Обобщённая теория представлена в работах [12-14].

Например, перестановка 123456 может быть преобразована в 125436 путём замены подпоследовательности 345 на подпоследовательность 543. Можем сказать, что 123456 эквивалентна 125436 относительно замены 123 ⇔ 321. Или, используя нотацию множеств: перестановки эквивалентны относительно {123, 321}.

Пусть $\pi \in S_n$ и пусть $B = \{B_1, \ldots, B_t\}$ – заданное разбиение $S_k$, где $k < n$. Каждый $B_j$ является шаблоном длиной $k$, относительно которого можно переставлять элементы. Назовём две перестановки – эквивалентными, если одна может быть получена из другой путём последовательных перестановок, где каждая перестановка $\sigma_i$ шаблона производится с теми же элементами $\sigma_j$ шаблона, а $\sigma_i$ и $\sigma_j$ принадлежат одному $B_l$. Тогда $Eq(\pi, B)$ – множество перестановок эквивалентных относительно $B$.

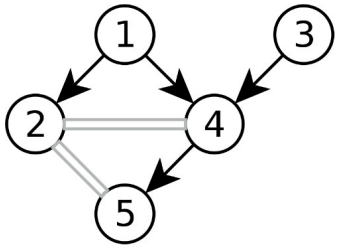
Таким образом, $125436 \in Eq(123456,\{123,321\})$.

Ограничим дальнейшее исследование перестановками только соседних элементов. Обозначим $B^|$ – отношение эквивалентности и $Eq^|(\pi, B)$ – класс эквивалентности $\pi$ с учётом замены $B$ только соседних элементов. То есть: $2134 \in Eq^|(1234,\{123,213\})$.

Далее будем рассматривать не шаблоны, а множества конкретных элементов, которые можно переставлять.

## Алгоритм построения всевозможных линейных порядков с учётом эквивалентности

Рассмотрим частично упорядоченное множество, аналогичное представленному на рисунке 1, но с дополнительными блоками эквивалентных перестановок: . Это множество изображено на рисунке 4, при этом блоки эквивалентных перестановок обозначены двойной чертой.




Р и с. 4. Частично упорядоченное множество с рёбрами эквивалентности
F i g. 4. Partially ordered set with equivalence edges

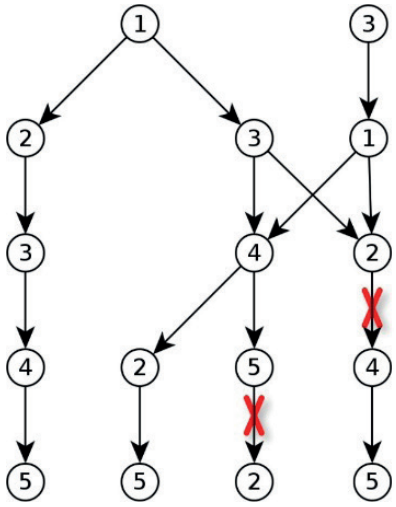
Такие перестановки симметричны и не транзитивны. Если $\{24,42\} \in B$ и $\{25,52\} \in B$, то $\{45,54\} \notin B$. Из-за отсутствия транзитивности при построении линейных порядков необходимо проверять элементы частично упорядоченного множества только попарно.

Псевдокод построения дерева с учётом эквивалентности перестановок представлен на рисунке 5.

```
def  traversal(P, Path):
  PrevX = Path.back()
  Processed = PrevX.parent.processed
  Xs = X in getMin(P)
  if Processed.notContains(X, PrevX)
    for X in Xs:
      Path.append(X)
      Processed.insert(X, PrevX)
      traversal(P \ X, Path)
```

Р и с. 5. Псевдокод для обхода частично упорядоченного множества с учётом эквивалентности
F i g. 5. Pseudocode for traversing a partially ordered set taking equivalence into account

В отличие от рисунка 2, прежде чем добавлять минимальный элемент *X* в дерево, проверяется, что пара *Prev X, X* уже была построена. Для этого в родительскую вершину дополняется информация о построенных парах дочерних вершин. Дерево перестановок с учётом эквивалентности представлено на рисунке 6.



Р и с. 6. Дерево (сжатое в ОАГ) перестановок с учётом эквивалентности
F i g. 6. Tree (compressed in OAG) of permutations with consideration of account equivalence

Для оценки асимптотики рассмотрим полный граф с $n = \|P\|$ вершинами, в котором каждая вершина эквивалентна каждой. В таком графе общее количество рёбер составляет $E = \frac{n(n-1)}{2}$. Пусть средняя степень вершины равна $\rho$, тогда количество эквивалентных вершин составляет $\rho n$. Временная сложность построения всех возможных перестановок оценивается как $O(2^n n)$ [9]. Если две вершины эквивалентны, то одна из них не участвует в построении перестановки. С учётом эквивалентности сложность составляет $O\left(2^{n\left(1-\frac{2\rho}{n-1}\right)}n\right)$.

Алгоритм, учитывающий классы эквивалентности, имеет асимптотически такую же сложность, что и алгоритм перебора всех возможных линейных порядков (см. рисунок 2), однако он гораздо более эффективен во многих практически важных случаях. Сравнение количества перестановок с учётом эквивалентности (нижняя строка) и без неё (верхняя строка) представлено в таблице 1 [15].

При построении таблицы 1 были выбраны такие множества, в которых количество рёбер эквивалентности близко к количеству элементов в частично упорядоченном множестве: $\|B\| \sim \|P\|$.

## Построение графа исполнения многопоточной программы

В многопоточной программе некоторые операции с памятью устанавливают строгий порядок выполнения через отношение «должно случиться до», в то время как другие могут быть переупорядочены [16-19]. Таким образом, существует строгий порядок видимости операций в памяти другими потоками и существуют «несравнимые» операции, порядок которых не определён до момента выполнения. Это приводит к формированию частично упорядоченного множества операций с памятью. Например, в однопоточной программе операции *store x*, 0 и *x = load* 0 образуют упорядоченное множество *store* → *load*. Однако, если добавить дополнительную операцию *amoadd* 0, в другом потоке, то частично упорядоченное множество будет зависеть от модели памяти. Например, для моделей, допускающих использование буфера сохранение (*store buffering*), таких как TSO (*total store order*) [20], множество будет содержать три несравнимые вершины: *store, load, amoadd*. Отсутствие отношений порядка между ними означает, что эффект этих операций в памяти может быть наблюдаем в любом порядке.

Т а б л и ц а 1. **Сравнение количества перестановок**
T a b l e 1. **Comparison of the number of permutations**

| Кол-во операций | Кол-во перестановок |
|---|---|
| 4 | 12 |
| | 3 |
| 9 | 105 |
| | 20 |
| 16 | 2041200 |
| | 196 |
| 20 | 1036062752256 |
| | 38433 |
| 28 | 6587142659584819200 |
| | 61881 |

## Пакет litmus-test

Наиболее распространённым открытым решением для верификации подсистемы памяти является пакет litmus-tests [21]. Этот пакет содержит как тесты, сгенерированные с помощью инструмента DIY, так и тесты, написанные вручную. Название пакета происходит от названия лакмусовой (litmus) бумаги. Он ориентирован на очень короткие направленные тесты.

```
X86 SB { x = 0; y = 0; }
    P0        |    P1
MOV [y], $1   |  MOV [x], $1
MOV EAX, [x]  |  MOV EAX ,[y]
exists (0: EAX = 0, 1: EAX = 0)
```

Р и с. 7. Пример litmus теста
F i g. 7. Example of litmus test

На рисунке 7, в двух потоках P0 и P1 в начале происходит запись в память единицы, затем чтение из памяти, изменяемой другим потоком.

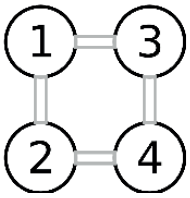
```
(1) 0: EAX = 1, 1: EAX = 0
(2) 0: EAX = 0, 1: EAX = 1
(3) 0: EAX = 1, 1: EAX = 1
(4) 0: EAX = 0, 1: EAX = 0
```

Р и с. 8. Возможные выходные состояния регистров
F i g. 8. Possible output states of registers

В последовательно согласованной модели памяти, у такой программы возможно только три возможных выходных состояния регистров, показанных на строчках 1-3 рис. 8. В моделях памяти с буферизацией загрузок (TSO, PSO, Weak) возможен ещё один сценарий работы, представленный на строчке 4.

## Использование частично упорядоченного множества для построения графа исполнения

Как было описано в начале главы, операции с памятью формируют частично упорядоченное множество. На рисунке 9 приведено построение всех переупорядочиваний на примере литмус-теста рис. 7 для модели памяти с буферизацией загрузок.




Р и с. 9. Частично упорядоченное множество для литмус теста с Рис. 7
F i g. 9. Partially ordered set for the litmus test from Fig. 7

Соответствия между вершинами и операциями следующие:
(1) mov [y], $1
(2) mov eax, [x]
(3) mov [x], $1
(4) mov eax, [y]

Также на рисунке 9 обозначены **независимые операции** — это операции, переупорядочивание которых не изменяет состояние системы (памяти, регистров). Порядок операций *AB* приведёт систему в то же состояние, что и порядок операций *BA*. Такие операции образуют множество эквивалентных перестановок. Дополнительно, можно ввести **независимые последовательности** *AB* относительно операции *C*, когда порядок *ABC* приводит систему в то же состояние, что и *CAB*.

Для рисунка 9 множество эквивалентных перестановок приведено ниже.

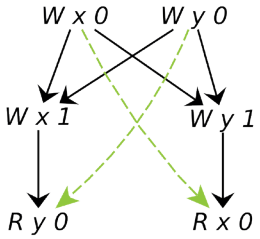
*B* = {{12,21}, {13,31}, {42,24}, {34,43}, {142,214}, {143,314}, {231,123}, {234,423}}

Если применить алгоритм с рис. 5 с учётом независимых операций и последовательностей, представленных в *B*, то всего будет 4 линейных порядка частично упорядоченного множества, показанных на рис. 8. Причём на каждой строке линейный порядок соответствует состоянию регистров из рис. 7.

```
(1) 1 - 3 - 2 - 4

(2) 1 - 4 - 2 - 3

(3) 4 - 1 - 3 - 2

(4) 4 - 2 - 1 - 3
```

Р и с. 10. Возможные линейные порядки
F i g. 10. Possible linear orders

Линейный порядок операций в памяти позволяет восстановить граф исполнения (execution graph) и проверить его соответствие модели памяти [22-25]. Например, для линейного порядка, представленного на строке 4 рис. 10, может быть построен граф исполнения, изображенный на рисунке 11. Данный граф соответствует модели памяти с использованием буфера сохранений, однако не соответствует последовательно согласованной модели памяти [20].




Р и с. 11. Возможный граф исполнения литмус теста с рис. 7
F i g. 11. Possible execution graph of the litmus test from Fig. 7

Таким образом, построение линейных переупорядочиваний частично упорядоченного множества операций с памятью в многопоточной программе может быть использовано для сравнения с графом исполнения программы в контексте конкретной модели памяти. Это помогает выявлять и устранять ошибки, возникающие в случае расхождения линейного порядка с графом исполнения.

## Заключение

В данной работе был представлен алгоритм построения линейных порядков частично упорядоченного множества с учетом эквивалентности операций. Показана расширяемость этого подхода на задачи построения переупорядочиваний множества операций с памятью в многопоточной программе для проверки на соответствие с различными моделями памяти. Предложенный алгоритм является эффективным инструментом для расширения концепции литмус тестов на любое частично упорядоченное множество операций с памятью.

Основываясь на проведенном исследовании, можно сделать вывод, что использование предложенного алгоритма позволяет более полно и точно анализировать поведение многопоточных программ в различных моделях памяти, выявлять потенциальные ошибки и улучшать производительность систем. Дальнейшие исследования могут быть направлены на улучшение алгоритма, его оптимизацию для работы с более сложными моделями памяти, а также на разработку практических инструментов на основе данного подхода для анализа и верификации многопоточных программ.

## References

[1] De Loof K., De Baets B., De Meyer H. Counting linear extension majority cycles in partially ordered sets on up to 13 elements. *Computers & Mathematics with Applications*. 2010;59(4):1541-1547. https://doi.org/10.1016/j.camwa.2009.12.021

[2] Banks J., et al. Using TPA to count linear extensions. *Journal of Discrete Algorithms*. 2018;51:1-11. https://doi.org/10.1016/j.jda.2018.04.001

[3] Bennett A.J., Field T., Harrison P. Modelling and validation of shared memory coherency protocols. *Performance Evaluation.* 1996;27-28:541-563. https://doi.org/10.1016/S0166-5316(96)90045-0

[4] Petrot F., Greiner A., Gomez P. On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures. In: 9th EUROMICRO Conference on Digital System Design (DSD'06). Cavtat, Croatia: IEEE Press; 2006. p. 53-60. https://doi.org/10.1109/DSD.2006.73

[5] Borrmann L., Istavrinos P. Store coherency in a parallel distributed-memory machine. In: Bode A. (ed.) Distributed Memory Computing. EDMCC 1991. *Lecture Notes in Computer Science*. Vol. 487. Berlin, Heidelberg: Springer; 1991. p. 32-41. https://doi.org/10.1007/BFb0032920

[6] Acquaviva J.-T., Jalby W. Experimental analysis of coherency behavior of shared memory scientific applications. In: Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728). San Francisco, CA, USA: IEEE Press; 2000. p. 142-151. https://doi.org/10.1109/MASCOT.2000.876439

[7] Szőllősi Á., et al. Litmus test of rich episodic representations: Context-induced false recognition. *Cognition*. 2023;230:105287. https://doi.org/10.1016/j.cognition.2022.105287

[8] Pautet L., Robert T., Tardieu S. Litmus-RT plugins for global static scheduling of mixed criticality systems. *Journal of Systems Architecture*. 2021;118:102221. https://doi.org/10.1016/j.sysarc.2021.102221

[9] Brightwell G., Winkler P. Counting linear extensions is #P-complete. In: Proceedings of the twenty-third annual ACM symposium on Theory of Computing (STOC '91). New York, NY, USA: Association for Computing Machinery; 1991. p. 175-181. https://doi.org/10.1145/103418.103441

[10] Farley J.D. Linear extensions of ranked posets, enumerated by descents. A problem of Stanley from the 1981 Banff Conference on Ordered Sets. *Advances in Applied Mathematics*. 2005;34(2):295-312. https://doi.org/10.1016/j.aam.2004.05.007

[11] De Loof K., De Baets B., De Meyer H. On the random generation and counting of weak order extensions of a poset with given class cardinalities. *Information Sciences.* 2007;177(1):220-230. https://doi.org/10.1016/j.ins.2006.04.003

[12] Knuth D. Permutations, matrices, and generalized Young tableaux. *Pacific journal of mathematics*. 1970;34(3):709-727. https://doi.org/10.2140/pjm.1970.34.709

[13] Haiman M.D. Dual equivalence with applications, including a conjecture of Proctor. *Discrete Mathematics*. 1992;99(1-3):79-113. https://doi.org/10.1016/0012-365x(92)90368-p

[14] Robinson G. de B. On the representations of the symmetric group. *American Journal of Mathematics*. 1938;745-760. https://doi.org/10.2307/2371609

[15] Dittmer S., Pak I. Counting Linear Extensions of Restricted Posets. *The Electronic Journal of Combinatorics*. 2020;27(4):4.48. https://doi.org/10.37236/8552

[16] Fava D.S., Steffen M., Stolz V. Operational semantics of a weak memory model with channel synchronization. *Journal of Logical and Algebraic Methods in Programming*. 2019;103:1-30. https://doi.org/10.1016/j.jlamp.2018.10.004

[17] Dan A., et al. Effective abstractions for verification under relaxed memory models. *Computer Languages, Systems & Structures*. 2017;47:62-76. https://doi.org/10.1016/j.cl.2016.02.003

[18] Marvik O.A. A method for IC layout verification. In: Proceedings of the 21st Design Automation Conference (DAC '84). Albuquerque, NM, USA: IEEE Press; 1984. p. 708-709. https://doi.org/10.1109/dac.1984.1585890

[19] Winter K., Smith G., Derrick J. Modelling concurrent objects running on the TSO and ARMv8 memory models. *Science of Computer Programming.* 2019;184:102308. https://doi.org/10.1016/j.scico.2019.102308

[20] Cohen E., Schirmer B. From Total Store Order to Sequential Consistency: A Practical Reduction Theorem. In: Kaufmann M., Paulson L.C. (eds.) Interactive Theorem Proving. ITP 2010. *Lecture Notes in Computer Science.* Vol. 6172. Berlin, Heidelberg: Springer; 2010. p. 403-418. https://doi.org/10.1007/978-3-642-14052-5_28

[21] Mador-Haim S., Alur R., Martin M.M.K. Litmus tests for comparing memory consistency models: how long do they need to be? In: Proceedings of the 48th Design Automation Conference (DAC '11). New York, NY, USA: Association for Computing Machinery; 2011. p. 504-509. https://doi.org/10.1145/2024724.2024842

[22] Jiménez E., Fernández A., Cholvi V. A parametrized algorithm that implements sequential, causal, and cache memory consistencies. *Journal of Systems and Software*. 2008;81(1):120-131. https://doi.org/10.1016/j.jss.2007.03.012

[23] Wrenger L., Töllner D., Lohmann D. Analyzing the memory ordering models of the Apple M1. *Journal of Systems Architecture.* 2024;149:103102. https://doi.org/10.1016/j.sysarc.2024.103102

[24] Tabbakh A., Annavaram M. An efficient sequential consistency implementation with dynamic race detection for GPUs. *Journal of Parallel and Distributed Computing*. 2024;187:104836. https://doi.org/10.1016/j.jpdc.2023.104836

[25] Gopalakrishnan K., Ravi D. Anvil: Best in Class Multiprocessor Coherency Verification Tool. In: 2017 18th International Workshop on Microprocessor and SOC Test and Verification (MTV). Austin, TX, USA: IEEE Press; 2017. p. 21-25. https://doi.org/10.1109/mtv.2017.18

*Поступила 13.11.2023; одобрена после рецензирования 23.01.2024; принята к публикации 15.02.2024.*
*Submitted 13.11.2023; approved after reviewing 23.01.2024; accepted for publication 15.02.2024.*

Об авторах:

**Андреев Илья Витальевич**, магистрант кафедры микропроцессорных технологий в интеллектуальных системах, факультет радиотехники и кибернетики, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0000-0001-6450-7917,** andreev.iv@phystech.edu

**Владимиров Константин Игоревич**, старший преподаватель кафедры микропроцессорных технологий в интеллектуальных системах, факультет радиотехники и кибернетики, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0000-0003-0925-1300,** konstantin.vladimirov@gmail.com

*Все авторы прочитали и одобрили окончательный вариант рукописи.*

About the authors:

**Ilya V. Andreev,** Master degree student of the Chair of Microprocessor Technologies in Intelligent Systems, Department of Radio Engineering and Cybernetics, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0000-0001-6450-7917,** andreev.iv@phystech.edu

**Konstantin I. Vladimirov,** Senior Lecturer of the Chair of Microprocessor Technologies in Intelligent Systems, Department of Radio Engineering and Cybernetics, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0000-0003-0925-1300,** konstantin.vladimirov@gmail.com

*All authors have read and approved the final manuscript.*