DOI: 10.15514/ISPRAS-2025-37(4)-5

HOREC: компилятор специализированных регулярных выражений для проектирования программируемой и ресурсоэффективной аппаратной архитектуры

П.Н. Советов, ORCID: 0000-0002-1039-2429 <sovetov@mirea.ru>

Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА — Российский технологический университет», Москва.

Аннотация. Разработка программируемых и ресурсоэффективных аппаратных ускорителей для поиска по регулярным выражениям представляет собой актуальное направление в области сетевой безопасности, где критически важны высокая пропускная способность потоковой обработки данных и устойчивость к атакам типа ReDoS (Regular Expression Denial of Service). В настоящей работе представлен компилятор HOREC, применяемый в цикле высокоуровневого проектирования программируемого аппаратного ускорителя. В HOREC используется новое расширение детерминированного конечного автомата, позволяющее компактно описывать интервальные квантификаторы с большим числом повторений, типичные для правил в системах обнаружения вторжений. Рассмотрены алгоритмы, сокращающие число переходов в командах и уменьшающие общее число команд. Представлена программная модель ускорителя на основе интерпретатора сопоставления с скомпилированным образцом и задан набор параметров для поиска в пространстве архитектурных вариантов аппаратного ускорителя, в том числе: объем доступной памяти, формат команд и режимы сопоставления символов. Экспериментальное исследование проведено для 7234 регулярных выражений, извлеченных из правил ET OPEN. Результаты оценки демонстрируют высокую ресурсоэффективность предлагаемого решения: в память объемом 64K команд удалось разместить до 7000 выражений. При этом в 60% случаев число переходов на команду не превышает 4, а наличие глобальной таблицы множеств символов для 256 часто встречающихся элементов позволяет ограничить объем памяти локальной таблицы всего 10 множествами символов на программу. Представленные результаты подтверждают применимость HOREC для аппаратной реализации поиска по РВ в задачах сетевой безопасности и показывают перспективность предложенного подхода для высокоуровневого проектирования ускорителей с низкими аппаратными затратами.

Ключевые слова: регулярные выражения; DSL-компилятор; сетевая безопасность; аппаратный ускоритель; высокоуровневое проектирование; совместное программно-аппаратное проектирование.

Для цитирования: Советов П.Н. HOREC: компилятор специализированных регулярных выражений для проектирования программируемой и ресурсоэффективной аппаратной архитектуры. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 79–96. DOI: 10.15514/ISPRAS–2025–37(4)–5.

HOREC: a Specialized Regular Expression Compiler for Designing Programmable and Resource-Efficient Hardware Architecture

P.N. Sovietov, ORCID: 0000-0002-1039-2429 <sovetov@mirea.ru>

Federal State Budgetary Educational Institution of Higher Education "MIREA - Russian Technological University", Moscow.

Abstract. The development of programmable and resource-efficient hardware accelerators for regular expression matching is an important research direction in network security, where high throughput for streaming data processing and resilience against ReDoS (Regular Expression Denial of Service) attacks are critical. This paper presents the HOREC compiler, which is used in the high-level design loop of a programmable hardware accelerator. HOREC utilizes a novel extension of a deterministic finite automaton, enabling compact representation of interval quantifiers with a large number of repetitions, typical for rules in intrusion detection systems. Algorithms are described to reduce the number of transitions in instructions and decrease the total number of instructions. The paper presents a software model of the accelerator based on an interpreter for matching compiled patterns and defines a set of parameters for architectural design space exploration of the hardware accelerator, including available memory size, instruction format, and symbols matching modes. Experimental evaluation was performed on 7234 regular expressions extracted from ET OPEN rules. The results demonstrate the high resource efficiency of the proposed solution: up to 7000 expressions were accommodated in a 64K instruction memory. Furthermore, in 60% of cases, the number of transitions per instruction does not exceed 4, and the use of a global symbol set table for 256 frequently used elements allows each program's local table to be limited to just 10 symbol sets. The presented results confirm HOREC's applicability for hardware implementation of regular expression matching in network security tasks and show the potential of the proposed approach for high-level design of low-hardware-cost accelerators.

Keywords: regular expressions; DSL compiler; network security; hardware accelerator; high-level design; hardware-software co-design.

For citation: Sovietov P.N. HOREC: a specialized regular expression compiler for designing programmable and resource-efficient hardware architecture. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025, pp. 79-96 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-5.

1. Введение

Регулярные выражения (РВ) являются одним из самых популярных предметно-ориентированных языков. Поиск в строковых данных по задаваемым с помощью РВ шаблонам широко используется в таких областях, как биоинформатика, системы управления базами данных и сетевая безопасность.

Перспективным подходом для ускорения поиска по РВ в больших объемах данных и в условиях жестких временных ограничений является использование специализированных аппаратных решений.

В частности, в системах обнаружения вторжений (СОВ) к анализу сетевых пакетов предъявляется ряд требований, среди которых:

- Потоковая обработка за один просмотр и на скорости поступления данных.
- Защита от ReDoS-атак [1], направленных на значительное замедление работы системы путем подачи специально сформированных входных данных.
- Возможность оперативного обновления набора РВ.

Эти требования затрудняют использование исключительно CPU или GPU при поиске по РВ из-за непредсказуемых задержек доступа к памяти и недостаточного распараллеливания уровня задач [2]. В связи с этим, задача поиска по РВ в сетевых пакетах, зачастую, полностью или частично, делегируется специализированным аппаратным ускорителям на базе ПЛИС или СБИС.

Для представления РВ в СОВ используется автоматный подход, гарантирующий линейное время работы, в отличие от многих универсальных библиотек для РВ, использующих механизм отката. Подход на основе отката позволяет поддержать расширенные диалекты РВ, такие как PCRE (Perl Compatible Regular Expressions), более выразительные, чем регулярные грамматики. При этом механизм отката может приводить к экспоненциальной вычислительной сложности поиска по РВ.

Трансляция РВ в ДКА позволяет в процессе поиска хранить лишь одно текущее состояние. В случае НКА требуется уже отслеживание $O(n)$ состояний, в которых одновременно может пребывать НКА в текущий момент. Несмотря на преимущество ДКА в скорости обработки, их применение сопряжено с проблемой экспоненциального роста числа состояний (до $O(2^n)$ после детерминизации НКА) и сложностью поддержки расширенных конструкций РВ. По этим причинам многие аппаратные ускорители базируются на модели НКА.

В СОВ Snort и Suricata используется специальный набор РВ, для которого, в частности, характерно наличие интервальных квантификаторов с сотнями повторений, что приводит к существенному увеличению числа состояний даже в случае НКА.

В настоящей работе представлен компилятор HOREC (Hardware-Oriented Regular Expression Compiler), предназначенный для трансляции РВ в оптимизированный код для различных архитектурных вариантов программируемого аппаратного ускорителя. Входные параметры компилятора определяют конкретный формат команд для скомпилированного РВ, а также иные, актуальные для исследуемого варианта целевой архитектуры значения. В качестве программной модели ускорителя используется универсальный интерпретатор, поддерживающий выполнение различных вариантов форматов команд. Такой цикл проектирования архитектуры аппаратного ускорителя с использованием компилятора [3-4] предполагает обратную связь со стороны специалистов в области логического синтеза и физического проектирования.

В компиляторе HOREC используется представление программ на основе нового расширения ДКА, позволяющего компактно описывать выражения с интервальными квантификаторами с большим числом повторений. Используются новые оптимизации, позволяющие сократить число команд и переходов в программах.

Полученные оценки HOREC на основе набора РВ из области сетевой безопасности демонстрируют ресурсоэффективность на уровне программной модели ускорителя. В частности, разработанные модели и алгоритмы позволяют значительно сократить объем памяти, необходимый для хранения скомпилированных РВ по сравнению с традиционными автоматными представлениями, особенно для РВ с интервальными квантификаторами.

Дальнейшее изложение имеет следующую структуру. Дается обзор архитектуры компилятора HOREC. Описываются новые модели R-НКА и R-ДКА вместе с расширенным алгоритмом детерминизации для компактного представления интервальных квантификаторов. Рассматриваются вопросы трансляции из автоматного представления в список машинных команд с использованием оптимизации для сокращения числа состояний (в HOREC это оптимизация называется Path-opt) и для сокращения числа переходов (оптимизация под названием Trans-opt). Описывается программная модель аппаратного ускорителя и алгоритм Match для интерпретации скомпилированных программ. Приводятся экспериментальные оценки компилятора HOREC. Статья завершается выводами и обсуждением перспектив дальнейших исследований.

2. Архитектура компилятора HOREC

Компилятор HOREC, архитектура которого изображена на рис. 1, принимает на вход РВ на языке подмножества PCRE, в котором не поддерживаются, в частности, предпросмотры и захваты подвыражений. Задаются параметры, описывающие варианты целевой аппаратной архитектуры. Результатом компиляции является программа в промежуточном представлении

(IR, Intermediate Representation), дополненная таблицей множеств символов. Результат компиляции используется для моделирования работы сопоставления с РВ при помощи интерпретатора. Полученная программа также может быть преобразована в двоичный формат для загрузки в память аппаратного ускорителя.

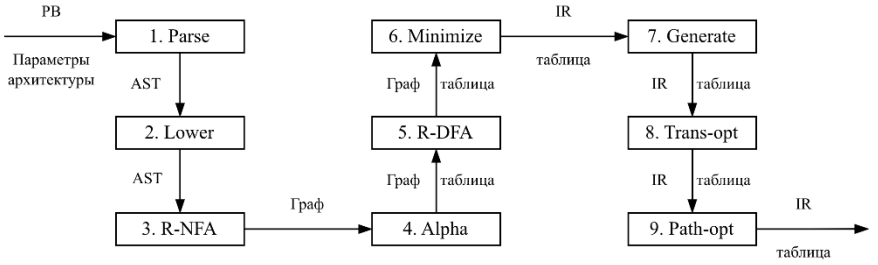

Рис. 1. Схема архитектуры компилятора HOREC.
Fig. 1. Scheme of the HOREC compiler architecture.

Компилятор написан на языке Python и имеет конвейерную архитектуру, состоящую из следующих стадий:

1) *Parse*. Осуществляется синтаксический разбор текста РВ с использованием библиотеки комбинаторов pecc [5]. Исходный текст РВ преобразуется в дерево абстрактного синтаксиса (AST, Abstract Syntax Tree).
2) *Lower*. Происходит понижение семантического уровня AST. Интервальные квантификаторы, повторяющие множество символов, транслируются в специальные R-узлы.
3) *R-NFA*. По входному AST строится граф НКА с поддержкой повторяемых множеств символов (R-НКА). При построении используется подход на основе частных производных Антимирова из [6], расширенный для поддержки R-узлов.
4) *Alpha*. На основе алгоритма Minterms из [7], осуществляется преобразование алфавита полученного графа автомата, сокращающее число переходов. Формируется таблица множеств символов.
5) *R-DFA*. Происходит детерминизация входного графа автомата, в процессе работы которой переходы, описывающие повторения множеств символов, пошагово раскручиваются в промежуточные состояния до достижения детерминизма.
6) *Minimize*. Полученный детерминированный автомат минимизируется с использованием алгоритма Бржозовского [8].
7) *Generate*. Граф автомата транслируется в программу в формате IR.
8) *Trans-opt*. Применяется алгоритм для сокращения числа переходов в командах программы с помощью модификации таблицы множеств символов.
9) *Path-opt*. Применяется алгоритм для сокращения числа команд программы за счет экономного кодирования последовательностей символов в команде.

3. Построение R-НКА

Интервальные квантификаторы являются одним из основных источников сложности порождаемых НКА. Например, РВ вида `[C]{N}` дает результат с не менее чем N состояниями. Для сокращения числа состояний, порождаемых интервальными квантификаторами, в компиляторе HOREC применена новая модель R-НКА.

R-НКА представляет собой частный случай обобщенного НКА, переходы которого помечены символами, являющимися РВ. Алфавит R-НКА расширен двумя дополнительными

символами, называемыми R-символами. Это символы `repset` и `repsetq` с аргументами. Аргумент C задает множество символов, а N – число повторений:

1) `repset(C, N)` для РВ вида `[C]{N}`.
2) `repsetq(C, N)` для РВ вида `[C]{0,N}`.

На стадии *Lower*, как показано на рис. 2, AST-узлы для различных типов квантификаторов сводятся к универсальному узлу `rep`, при этом частные случаи повторений множеств символов описываются с помощью R-узлов `repset` и `repsetq`, которые при дальнейшей трансляции в R-НКА становятся R-символами.

$$\begin{aligned}
\mathrm{star}(X) &= \mathrm{rep}(X, 0, \infty) \\
\mathrm{plus}(X) &= \mathrm{rep}(X, 1, \infty) \\
\mathrm{opt}(X) &= \mathrm{rep}(X, 0, 1) \\
\mathrm{rep}(\mathrm{cset}(C), N, \infty) &= \mathrm{seq}(\mathrm{repset}(C, N), \mathrm{rep}(\mathrm{cset}(C), 0, \infty)) \\
\mathrm{rep}(\mathrm{cset}(C), N, \mathrm{int}(M)) &= \mathrm{seq}(\mathrm{repset}(C, N), \mathrm{repsetq}(C, M - N))
\end{aligned}$$

Рис. 2. Правила переписывания для квантификаторов.
Fig. 2. Rewriting rules for quantifiers.

Для построения R-НКА используется подход на основе частных производных Антимирова. Этот подход, как показано в [9], позволяет получить НКА без ϵ-переходов, в отличие от НКА Томпсона, а также дает в среднем результат с меньшим числом состояний, чем НКА Глушкова.

В оригинальном алгоритме построения НКА из [6] ключевую роль играет функция `lf` (линейная форма):

$$\mathrm{lf}\colon \mathrm{t} \to \{(c_1, t_1), \dots\}.$$

Здесь t представляет собой некоторое РВ в представлении AST, а очередная пара (c_i, t_i) описывает недетерминированный переход в состояние t_i по символу c_i.

При построении R-НКА используется функция `after`, расширяющая функцию `lf` для поддержки узлов `rep`, `repset` и `repsetq`. На рис. 3 показано определение функции `after`, а также приведены правила для вспомогательного предиката `nullable`, возвращающего 1, если его аргумент допускает пустую строку. Бинарная операция $\odot$ определена в [6].

$$\begin{aligned}
\mathrm{after}(\epsilon) &= \varnothing \\
\mathrm{after}(\mathrm{rep}(X, 0, 0)) &= \varnothing \\
\mathrm{after}(\mathrm{repset}(C, 0)) &= \varnothing \\
\mathrm{after}(\mathrm{repsetq}(C, 0)) &= \varnothing \\
\mathrm{after}(\mathrm{repset}(C, N)) &= \{(t, \epsilon)\}, N \neq 0 \\
\mathrm{after}(\mathrm{repsetq}(C, N)) &= \{(t, \epsilon)\}, N \neq 0 \\
\mathrm{after}(\mathrm{cset}(C)) &= \{(c, \epsilon) \mid c \in C\} \\
\mathrm{after}(\mathrm{rep}(X, 0, \infty)) &= \mathrm{after}(X) \odot t \\
\mathrm{after}(\mathrm{rep}(X, 0, \mathrm{int}(M))) &= \mathrm{after}(\mathrm{rep}(X, 1, M)) \\
\mathrm{after}(\mathrm{alt}(X, Y)) &= \mathrm{after}(X) \cup \mathrm{after}(Y) \\
\mathrm{after}(\mathrm{seq}(X, Y)) &= \mathrm{after}(X) \odot Y, \neg\mathrm{nullable}(X) \\
\mathrm{after}(\mathrm{seq}(X, Y)) &= (\mathrm{after}(X) \odot Y) \cup \mathrm{after}(Y), \mathrm{nullable}(X) \\
\mathrm{after}(\mathrm{rep}(X, N, \mathrm{int}(M))) &= \mathrm{after}(X) \odot \mathrm{rep}(X, N - 1, M - 1), N \neq 0 \\
\mathrm{after}(\mathrm{rep}(X, N, \infty)) &= \mathrm{after}(X) \odot \mathrm{rep}(X, N - 1, \infty), N \neq 0
\end{aligned}$$

$$\begin{aligned}
\mathrm{nullable}(\epsilon) &= 1 \\
\mathrm{nullable}(\mathrm{cset}(C)) &= 0 \\
\mathrm{nullable}(\mathrm{rep}(X, 0, M)) &= 1 \\
\mathrm{nullable}(\mathrm{rep}(X, N, M)) &= \mathrm{nullable}(X), N \neq 0 \\
\mathrm{nullable}(\mathrm{repset}(X, 0)) &= 1 \\
\mathrm{nullable}(\mathrm{repset}(C, N)) &= \mathrm{nullable}(C), N \neq 0 \\
\mathrm{nullable}(\mathrm{repsetq}(C, N)) &= 1 \\
\mathrm{nullable}(\mathrm{seq}(X, Y)) &= \mathrm{nullable}(X) \wedge \mathrm{nullable}(Y) \\
\mathrm{nullable}(\mathrm{alt}(X, Y)) &= \mathrm{nullable}(X) \vee \mathrm{nullable}(Y)
\end{aligned}$$

Рис. 3. Функция `after` для построения переходов R-НКА.
Fig. 3. `After` function for constructing R-NFA transitions.

В полученном R-НКА R-узлы `repset` и `repsetq` становятся R-символами, по которым осуществляются переходы, как показано на рис. 4а.

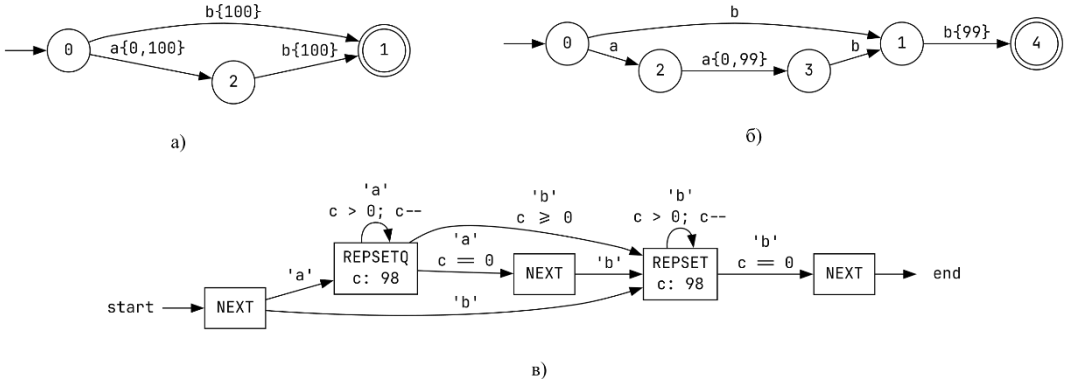

Рис. 4. Пример трансляции РВ `a{0,100}b{100}`*:*
а) R-НКА; б) R-ДКА; в) управляющий граф с выделенным счетчиком `c`*.*
Fig. 4. Example of translation of RE `a{0,100}b{100}`*:*
a) R-NFA; b) R-DFA; c) control flow graph with dedicated counter `c`*.*

Дополнительно выполняется стадия *Alpha* для сокращения числа переходов в R-НКА. Создается таблица множеств символов. Элементами нового алфавита становятся индексы множеств символов из этой таблицы. Для получения таблицы создается множество S, которое состоит из одноэлементных подмножеств, содержащих символы исходного алфавита, за исключением символов кратных переходов (для одной и той же пары состояний), объединяемых в общее подмножество. Задача создания таблицы сводится к задаче получения на основе S множества минимальной мощности S', обладающего следующими свойствами:

- Элементы S' попарно не пересекаются.
- Каждый элемент S представим как объединение одного или нескольких элементов из S'.

Эта задача решается с помощью алгоритма Minterms из [7].

В компиляторе HOREC поддерживается режим сопоставления (match), а не поиска (search) с неявной вставкой `*.` в начало РВ. Поэтому якорь `^` может явно размещаться только в начале РВ и наличие этого якоря в процессе компиляции игнорируется.

Для реализации якоря `$` используется AST-узел `eof` и соответствующий ему специальный символ расширенных алфавитов R-НКА и R-ДКА. Компиляция `eof` производится аналогичным R-узлам образом и поэтому в целях экономии объема статьи далее не рассматривается.

4. Построение R-ДКА

Для сокращения вычислительной сложности сопоставления с РВ необходимо перейти от модели R-НКА к R-ДКА. Одним из способов это сделать является предварительная полная раскрутка всех состояний, извлеченных из R-символов, за которой следует детерминизация получившегося ДКА. После этого осуществляется свертка состояний обратно в R-символы там, где это возможно. Этот способ отличается высокой ресурсоемкостью, особенно для РВ, в которых фигурируют интервальные квантификаторы с большим числом повторений.

В компиляторе HOREC для получения R-ДКА используется расширенный вариант алгоритма конструкции подмножеств, в котором раскрутка состояний для R-символов производится пошагово и по необходимости.

На рис. 5 показан псевдокод функции `unroll`, которая принимает на вход граф R-НКА (G) и функцию переходов (M), заданную для текущего состояния, рассматриваемого в процессе

детерминизации. Результатом `unroll` является новая функция переходов, с возможной частичной раскруткой для R-символов.

```
1:  function UNROLL(G, M)
2:      match dom(M) do
3:          case {repset(C, N)} ∧ N ≥ MIN_N
4:              return M
5:          case {r} ∧ r = repsetq(C, N) ∧ N ≥ MIN_N
6:              M' ← ∅
7:              for each e ∈ M(r) do
8:                  M' ← merge(M', unroll_moves(G, G(e)))
9:              if set(C) ∩ dom(M') = ∅ then
10:                 s ← new_state(G, M')
11:                 return {(r, {s})}
12:             else
13:                 return unroll_moves(G, M)
14:         case _
15:             return unroll_moves(G, M)

16: function UNROLL_MOVES(G, M)
17:     M' ← ∅
18:     for each (c, s) ∈ M do
19:         M' ← merge(M', unroll_char(G, c, s))
20:     return M'
```

Рис. 5. Псевдокод алгоритма раскрутки состояний для R-символов.
Fig. 5. Pseudocode of the state unrolling algorithm for R-symbols.

Детали реализации функции `unroll`:

- *Строка 2*. Сопоставление с областью определения M – множеством символов, по которым возможны переходы из текущего состояния.
- *Строки 3–4*. Случай, когда имеется единственный переход по символу `repset`. Этот переход является детерминированным и дальнейшей раскрутки повторяемых состояний не производится в случае, если число итераций не меньше значения архитектурного параметра `MIN_N`. Примером такого перехода является переход из состояния 1 на рис. 4б.
- *Строки 5–13*. Случай, когда имеется единственный переход по R-символу `repsetq`. Здесь возможен ранний выход по одному из символов, распознаваемых в множестве целевых состояний из $M(r)$.
- *Строки 6–8*. Строится функция M' для нового общего целевого состояния. При этом отдельные целевые состояния с помощью функции `unroll_moves` раскручиваются на один шаг. Функция `merge` объединяет две функции переходов, то есть два множества пар (символ, множество целевых состояний).
- *Строки 9–11*. Если множество символов из аргумента `repsetq` и множество символов переходов из M' не пересекаются, то переход по `repsetq`, без дальнейшей раскрутки состояний, осуществляется в созданное (функция `new_state`) общее целевое состояние, для которого заданы переходы из M'. Шаг раскрутки и создание нового состояния необходимы для того, чтобы не возникло ситуации раннего выхода по R-символу, требующей специальной обработки.
- *Строки 12–15*. Случаи, когда повторяемые состояния раскручиваются на один шаг.
- *Строки 16–20*. Функция `unroll_moves` принимает на вход граф R-НКА (G) и функцию переходов (M) для текущего состояния. Результатом является новая функция переходов M', где для каждого символа возможные повторяемые состояния раскручены на один шаг.

Псевдокод функции `unroll_char` приведен на рис. 6. Эта функция принимает на вход граф R-НКА (G), символ c и множество целевых состояний s. Результатом работы `unroll_char` является функция переходов для символа c, в которой повторяемые состояния раскручены на один шаг. Следует отметить использование рекурсии в `unroll_char` (строка 13).

Рекурсивный вызов нужен для обработки случаев, когда символы переходов целевых состояний тоже являются R-символами. Это причина появления перехода $3 \xrightarrow{b} 1$ на рис. 4б. Случаи бесконечной рекурсии на нетипичных РВ вида `(x{0,4})*` выявляются на этапе компиляции.

```
1:  function UNROLL_CHAR(G, c, s)
2:      match c do
3:          case repset(C, N)
4:              if N > 1 then
5:                  s ← {new_state(G, {(repset(C, N − 1), s)})}
6:              return {(c′, s) | c′ ∈ C}
7:          case repsetq(C, N)
8:              s′ ← s
9:              if N > 1 then
10:                 s ← {new_state(G, {(repsetq(C, N − 1), s′)})}
11:             M ← {(c′, s) | c′ ∈ C}
12:             for each e ∈ s′ do
13:                 M ← merge(M, unroll_moves(G, G(e)))
14:             return M
15:         case _
16:             return {(c, s)}
```

Рис. 6. Псевдокод функции `unroll_char`.
Fig. 6. Pseudocode of the `unroll_char` *function.*

Далее R-ДКА минимизируется с помощью алгоритма Бржозовского [8]. Этот простой алгоритм применяется без модификаций, в том числе и для переходов по R-символам.

В некоторых РВ R-символ в процессе детерминизации полностью раскручивается в отдельные состояния. Два типичных примера РВ, где это происходит:

1) `.*a{100}`.
2) `a{0,100}a`.

Для первого примера возможно введение нового типа R-символа. Тем не менее, в общем случае поиска с префиксом `.*` для ДКА затруднительно избежать комбинаторного взрыва состояний. Для второго примера возможно использование правил переписывания вида

$$C\{0, N\}C = C\{1, N + 1\}.$$

5. Генератор кода

На этапе генерации кода граф R-ДКА транслируется в управляющий граф, пример которого показан на рис. 4в. Осуществляется ряд оптимизаций управляющего графа и на заключительном этапе создается список команд.

Команда включает в себя следующие поля:

- `op`. Код операции.
- `chars`. Массив кодов – индексов таблицы множеств символов.
- `jumps`. Массив адресов переходов, той же длины, что и `chars`.
- `counter`. Значение счетчика цикла для R-символов.
- `breaks`. Булев массив для команды `REPSETQ`, в котором 1 по индексу i сигнализирует о раннем выходе из цикла, переходе по коду из `chars` с тем же индексом i.
- `is_end`. Флаг допускающего состояния автомата.

Пример результата трансляции для РВ `TO[^\n]{0,100}\n`:

#	op	chars	jumps	counter	breaks	is_end
0	NEXT	[3]	[1]	0	[]	0
1	NEXT	[2]	[2]	0	[]	0
2	NEXT	[0, 1, 2, 3]	[4, 3, 3, 3]	0	[]	0
3	REPSETQ	[1, 2, 3, 0]	[5, 5, 5, 4]	98	[0, 0, 0, 1]	0
4	NEXT	[]	[]	0	[]	1
5	NEXT	[0]	[4]	0	[]	0

Команда `NEXT` – переход в режиме обычного ДКА. Алфавит экономно закодирован, поэтому в строках 2 и 3 для множества символов `[^\n]` имеется только 4 перехода. Тем не менее, кодам {1,2,3} в строке 2 соответствует общий адрес 3, а в строке 3 те же коды задают переход по общему адресу 5, то есть число переходов в некоторых командах этой программы можно сократить.

Для устранения этой избыточности в компиляторе HOREC используется алгоритм, псевдокод которого показан на рис. 7. Ключевая идея алгоритма состоит в нахождении среди элементов `chars` такого подмножества C, элементы которого задают переход по общему адресу j. При этом замена множества C единственным кодом n должна приводить к сокращению максимальной длины поля `chars` всей программы. Таким образом создается возможность перехода на формат команды меньшей длины.

```
1: function REDUCE_TRANS(P, A)
2:     loop
3:         C, n ← group_cset(P, A)
4:         if |C| < 2 then
5:             return P, A
6:         P ← replace_cset(P, C, n)
7:         A[n] ← ⋃_{∀i∈C} A_i
8: function GROUP_CSET(P, A)
9:     W ← max({|p.chars| | p ∈ P})
10:    S ← ∅
11:    C_o ← ∅
12:    for each p ∈ P do
13:        if |p.chars| = W then
14:            j ← most_freq(p.jumps)
15:            S ← S ∪ {{p.chars[i] | i ∈ {0, ..., W − 1} ∧ p.jumps[i] = j}}
16:        else
17:            C_o ← C_o ∪ set(p.chars)
18:    C ← ⋂_{∀s∈S} s
19:    U ← C − C_o
20:    n ← |A|
21:    if |U| ≠ 0 then
22:        n ← min(U)
23:    return C, n
```

Рис. 7. Псевдокод алгоритма `Trans-opt`.
Fig. 7. Pseudocode of the `Trans-opt` *algorithm.*

Функция `reduce_trans` принимает на вход программу P в виде списка команд, а также таблицу множеств символов A, полученную на стадии компиляции *Alpha*. Результатом работы этой функции являются новая программа P' и новая таблица A'.

Детали работы функции `reduce_trans`:

- *Строки 3–5*. Функция `reduce_trans` является итеративной и выполняется до тех пор, пока очередное множество C, найденное функцией `group_cset`, целесообразно ($|C| > 1$) заменить кодом n.
- *Строка 6*. В функции `replace_cset` осуществляется замена множества C, найденного среди элементов поля `chars` команд программы, на код n.
- *Строка 7*. В таблицу A по индексу n заносится множество всех исходных символов, соответствующих кодам из C.

Функция `group_cset`:

- *Строка 9*. Значение W – максимальная длина `chars` среди всех команд программы,

эту длину требуется уменьшить.

- *Строки 10–11*. S и C – два множества, получаемые далее из элементов `chars` отдельных команд.
- *Строки 12–17*. Осуществляется просмотр всех команд программы.
- *Строки 13–15*. Если длина `chars` оказалась максимальной, то в S добавляется новое множество, которое состоит из тех кодов `chars`, которые указывают на один и тот же адрес перехода j. При этом j определяется, как наиболее часто встречающийся адрес в `jumps` (функция `most_freq`).
- *Строки 16–17*. В противном случае множество из `chars` добавляется в C_0.
- *Строка 18*. Множество C состоит из тех кодов, которые фигурируют во всех элементах S.
- *Строки 19-23*. Получив C, остается выбрать новый код для n. Это может быть первый свободный индекс в A. Желательно повторно использовать один из кодов в C, при условии, что этот код более нигде не фигурирует (строка 21).

Новый вариант программы для того же РВ, но с примененным преобразованием `Trans-opt`:

#	op	chars	jumps	counter	breaks	is_end
0	NEXT	[3]	[1]	0	[]	0
1	NEXT	[2]	[2]	0	[]	0
2	NEXT	[0, 1]	[4, 3]	0	[]	0
3	REPSETQ	[0, 1]	[4, 5]	98	[1, 0]	0
4	NEXT	[]	[]	0	[]	1
5	NEXT	[0]	[4]	0	[]	0

В результате максимальное число переходов в командах сократилось с 4 до 2.

В рассматриваемом примере команды с адресами 0 и 1 задают последовательность сопоставления со строкой `T0`. В этой связи полезно ввести дополнительное преобразование `Path-opt`, в результате которого подобные пути сворачиваются в новую команду `PATH`, работающую сходным образом с `REPSET`:

#	op	chars	jumps	counter	breaks	is_end
0	PATH	[2, 3]	[1]	1	[]	0
1	NEXT	[0, 1]	[4, 2]	0	[]	0
2	REPSETQ	[0, 1]	[4, 3]	98	[1, 0]	0
3	NEXT	[0]	[4]	0	[]	0
4	NEXT	[]	[]	0	[]	1

В `PATH` коды перечислены в обратном порядке для использования при индексации счетчика цикла. Программа теперь содержит на одну команду меньше.

Для сокращения числа переходов в командах возможна также традиционная оптимизация ветвлений, при которой переход по адресу следующей команды (`pc + 1`) явно не указывается.

6. Программная модель аппаратного ускорителя

Программное моделирование сопоставления с РВ осуществляется с помощью интерпретатора `match`, псевдокод которого показан на рис. 8. На вход функции `match` подается скомпилированная программа P в формате IR, таблица множеств символов A и текст T. Функция возвращает булево значение результата сопоставления. В реализации `match` используются легковесные операции, допускающие непосредственную аппаратную реализацию.

```
1:  function MATCH(P, A, T)
2:      pos, c, pc, in_loop ← 0, 0, 0, false
3:      loop
4:          p ← P[pc]
5:          if p.is_end then
6:              return true
7:          if pos ≥ |T| then
8:              return p.is_eof
9:          if p.op ∈ {PATH, REPSET, REPSETQ} ∧
    ¬in_loop then
10:             c ← p.counter
11:             in_loop ← true
12:         C ← p.chars
13:         if p.op = PATH then
14:             C ← C[c : c + 1]
15:         if (i := get_index(A, C, T_pos)) = ⊥ then
16:             return false
17:         match p.op do
18:             case NEXT
19:                 pc ← p.jumps[i]
20:             case PATH | REPSET
21:                 if c = 0 then
22:                     in_loop ← false
23:                     pc ← p.jumps[0]
24:                 else
25:                     c ← c − 1
26:             case REPSETQ
27:                 if c = 0 ∨ p.breaks[i] then
28:                     in_loop ← false
29:                     pc ← p.jumps[i]
30:                 else
31:                     c ← c − 1
32:         pos ← pos + 1
```

Рис. 8. Псевдокод функции `match`.
Fig. 8. Pseudocode of the `match` *function.*

Подробности работы функции `match`:

- *Строка 1*. Инициализация состояния интерпретатора: позиция `pos` в `T`, счетчик цикла `c`, счетчик команд `pc` и флаг режима выполнения цикла `in_loop`.
- *Строки 3–5*. Выборка очередной команды `p` и успешный выход при достижении допускающего состояния (`is_end`).
- *Строки 7–8*. Выход по завершению просмотра `T`. Этот выход является успешным, если текущая команда содержит переход по якорю `$` (`is_eof`).
- *Строки 9–11*. Подготовка нового цикла.
- *Строки 12–16*. Сопоставление входного символа T_{pos} с элементами `chars`. В случае команды `PATH` для сопоставления передается единственный элемент `chars` по индексу, указанному в счетчике цикла. Функция `get_index` возвращает индекс i в массиве `chars`, если $T_{\text{pos}} \in A_n$ и $n = \text{chars}[i]$.
- *Строки 18–19*. Команда `NEXT`. Безусловный переход.
- *Строки 20–25*. Команды `PATH` и `REPSET` функционируют одинаково, поскольку коды `chars` в случае `PATH` размещены в обратном порядке.
- *Строки 26–31*. Команда `REPSETQ` аналогична `REPSET`, но поддерживает также ранний выход из цикла при $\text{breaks}[i] = 1$.

Особенностью аппаратной реализации `get_index` является параллельная проверка входного символа на принадлежность одному из множеств, индексы которых указаны в `chars`. При этом для представления множеств в таблице A используются битовые вектора.

На рис. 9 представлены основные компоненты архитектуры аппаратного ускорителя с точки зрения разработчика компилятора. На вход ускорителя поступает поток данных, а сопоставление осуществляется с помощью параллельно работающих ядер Match, каждое из которых осуществляет сопоставление одного скомпилированного РВ. У ядра Match имеется локальная память для хранения скомпилированного РВ и хранения таблицы множеств символов. В составе ускорителя находится также глобальная память для часто используемых таблиц множеств символов. Со стороны CPU осуществляется настройка ускорителя и обновление набора скомпилированных программ, а сам ускоритель сигнализирует CPU о факте успешного сопоставления.

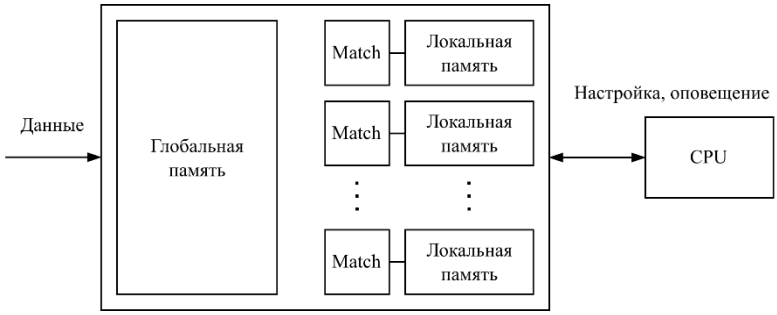

Рис. 9. Схема основных компонентов аппаратного ускорителя.
Fig. 9. Scheme of the main components of the hardware accelerator.

Представленной архитектуре аппаратного ускорителя соответствует множество вариантов его возможной микроархитектуры. При этом микроархитектурное проектирование оказывает обратное влияние на архитектуру. В этой связи важно производить с помощью компилятора поиск в пространстве архитектурных вариантов программной модели ускорителя.

Среди параметров, описывающих архитектурный вариант ускорителя, можно выделить следующие:

- Размеры локальной и глобальной памяти для программ и таблиц.
- Размеры полей команд.
- Максимальное значение счетчика циклов. Циклы, имеющие большее число итераций, могут быть разбиты на несколько команд.
- Формат команды. На уровне IR команда может иметь неограниченное число переходов, которые определяются в полях `chars`, `jumps` и `breaks`. При аппаратной реализации необходимо зафиксировать максимально допустимое число переходов. Возможен также выбор из нескольких форматов команды фиксированной длины. Например, для команды `NEXT` могут быть созданы ее аппаратные варианты в виде `NEXT4` (до 4 переходов в команде), `NEXT8` (до 8 переходов) и так далее.
- Режим последовательного сопоставления. При котором, например, две последовательные команды `NEXT4` могут быть использованы вместо одной команды `NEXT8`. Неудачное сопоставление в первой `NEXT4` при сброшенном специальном флаге `final_match` позволяет перейти к выполнению второй команды. Аналогичный подход может использоваться для команды `PATH`.

7. Экспериментальные оценки компилятора HOREC

В рамках эксперимента использован открытый набор правил для обнаружения угроз ET OPEN [10], из которого извлечены PB, начинающиеся с ^. Модификаторы PB игнорируются. Компилятор HOREC оценивался в тестовой конфигурации с на основе процессора Core i7-3770 3.40 ГГц, 32 ГБ ОЗУ и Python 3.13.1.

Общие сведения об извлеченном наборе PB и результатах компиляции приведены в табл. 1. Следует отметить, что интервальные квантификаторы в наборе PB, за исключением 13 сложных подвыражений, повторяют только множества символов, то есть выступают в качестве кандидатов в R-символы.

Как показано на рис. 10а, в варианте использования R-ДКА с оптимизацией Path-opt достаточно локальной памяти кода на 16 команд, чтобы поддержать 75% PB. При этом используется в 3 раза больше программ, чем для базового ДКА.

На рис. 10б выбраны PB, содержащие интервальные квантификаторы. Здесь более заметен эффект от применения R-ДКА, чем на предыдущем графике: при ограничении объема

локальной памяти кода в 32 команды использование R-ДКА позволяет поддержать в 2 раза больше программ, чем базовый ДКА.

Табл. 1. Результаты компиляции РВ из набора правил ET OPEN.
Table 1. Results of RE compilation from ET OPEN ruleset.

Параметр	Значение
Всего РВ	7647
Лимит на компиляцию одного РВ, с	10
Скомпилированных РВ	7234
Скомпилированных РВ с {}	1455
Скомпилированных РВ с []{}	1442
Всего множеств в таблицах	101473
Уникальных множеств в таблицах	4220
Макс. значение счетчика цикла	999

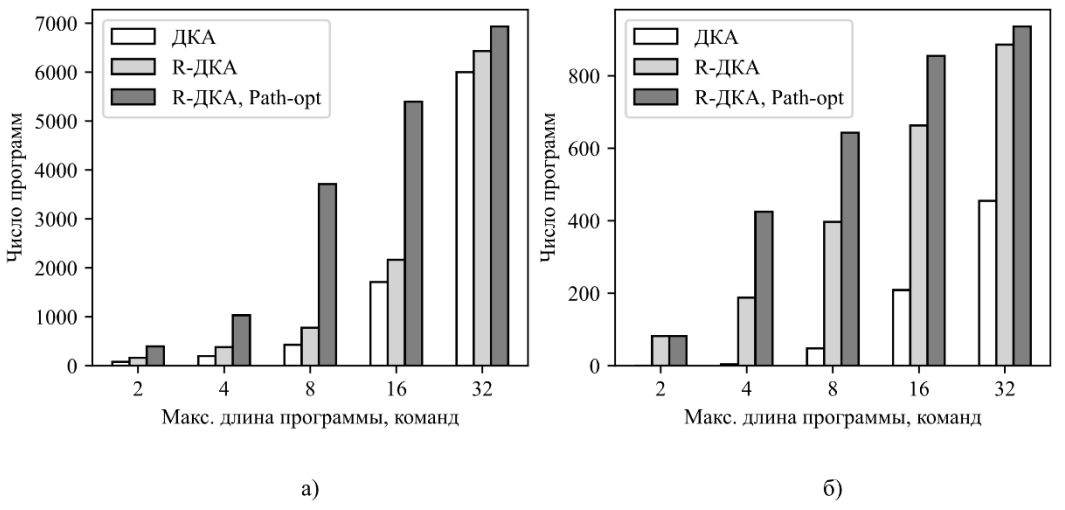

Рис. 10. Распределение длин программ:
а) для общего набора РВ; б) для РВ с интервальными квантификаторами.
Fig. 10. Distribution of program sizes:
a) for the whole set of REs; b) for REs with interval quantifiers.

Вариант R-ДКА с Path-opt, как показано на рис. 11а, позволяет разместить 6728 РВ в объеме памяти кода в 65536 команд. Это в 1,7 раз больше, чем для случая базового ДКА.

В оценках объема кода необходимо учитывать не только число команд, но и ширину команды, которая определяется, в первую очередь, числом полей для возможных переходов. Из рис. 11б видно, что оптимизации Minterms и Trans-opt существенно сокращают число переходов в командах. В частности, для ограничения в 4 перехода (целесообразное ограничение ширины команды с точки зрения аппаратуры) введение Minterms позволяет разместить на ускорителе в 7 раз больше РВ, чем в базовом варианте, а совместное применение Minterms и Trans-opt — в 21 раз. Применение этих оптимизаций для формата команды с 4 полями переходов дает возможность разместить 60% РВ. Для поддержки большего числа правил может быть использован режим последовательного сопоставления.

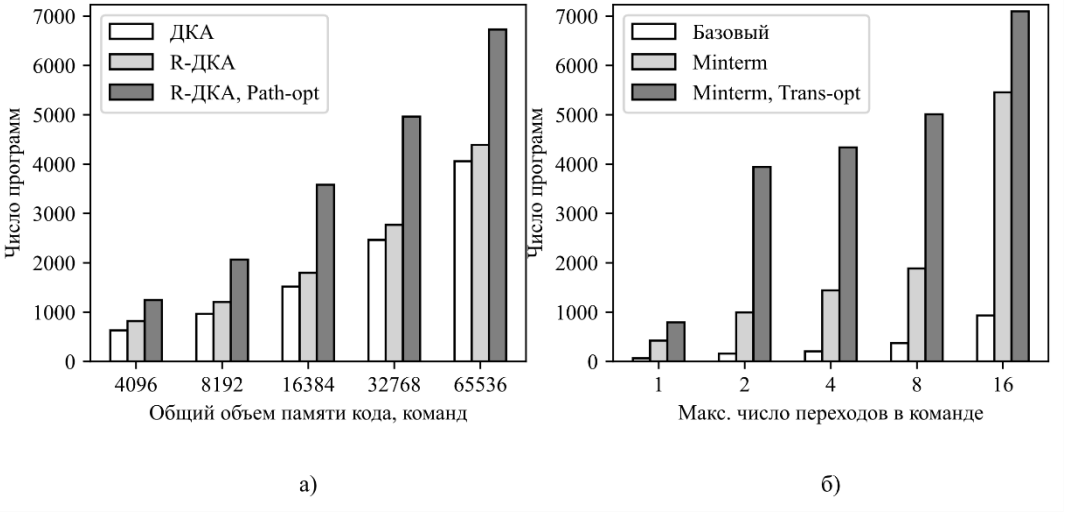

Рис. 11. Влияние оптимизаций:
а) на общий объем кода, б) на распределение числа переходов в командах.
Fig. 11. Effect of optimizations:
a) on the total code size, b) on the distribution of the number of transitions in instructions.

Распределение числа переходов (без оптимизации Path-opt) в командах на рис. 12а показывает, что наиболее часто в программах находятся команды с 1 и 2 переходами. Для этих популярных случаев могут быть спроектированы специальные варианты формата команды.

Распределение размера таблицы в программах, приведенное на рис. 12б, показывает, что наиболее часто встречаются таблицы из 16 множеств, при этом таблицы редко содержат 30 и более множеств. На основе этой информации можно определить объем локальной памяти для хранения таблицы.

В глобальной памяти возможно размещение таблицы с наиболее часто встречающимися в программах множествами. Если, как показано на рис. 13, глобальная таблица содержит 256 элементов, то размер локальной памяти таблицы может быть сокращен с 62 до 10 множеств.

В целом, такого рода статистика, полученная со стороны компилятора HOREC, может быть использована при проектировании архитектуры аппаратного ускорителя, специализированной для выбранного набора РВ.

8. Обзор существующих решений

В литературе предложен ряд моделей конечного автомата с добавленными счетчиками циклов для сокращения числа состояний, порожденных интервальными квантификаторами. В частности, в [11] предложена модель NBVA (Nondeterministic Bit Vector Automaton), которая, в отличие от настоящей работы, основана на НКА, требующем большего числа вычислительных ресурсов, чем ДКА. В [12] рассматривается модель XFA (Extended Finite Automaton) на основе ДКА, недостатком которой является низкопроизводительный алгоритм детерминизации, в процессе работы которого, в отличие от детерминизации R-НКА, производится полная раскрутка состояний с циклами. В [13] предложена модель CsA (Counting-set Automaton) с использованием сложного алгоритма детерминизации, который при получении ДКА не всегда сохраняет семантику исходного НКА, что не имеет места в случае детерминизации R-НКА. В [14] описан упрощенный вариант CsA под названием MCA (Monadic Counting Automaton), в котором, как и в R-ДКА, повторение возможно только для

классов символов. Во всех этих моделях предполагается работа со множеством счетчиков, а не с одним глобальным счетчиком, как в случае R-ДКА.

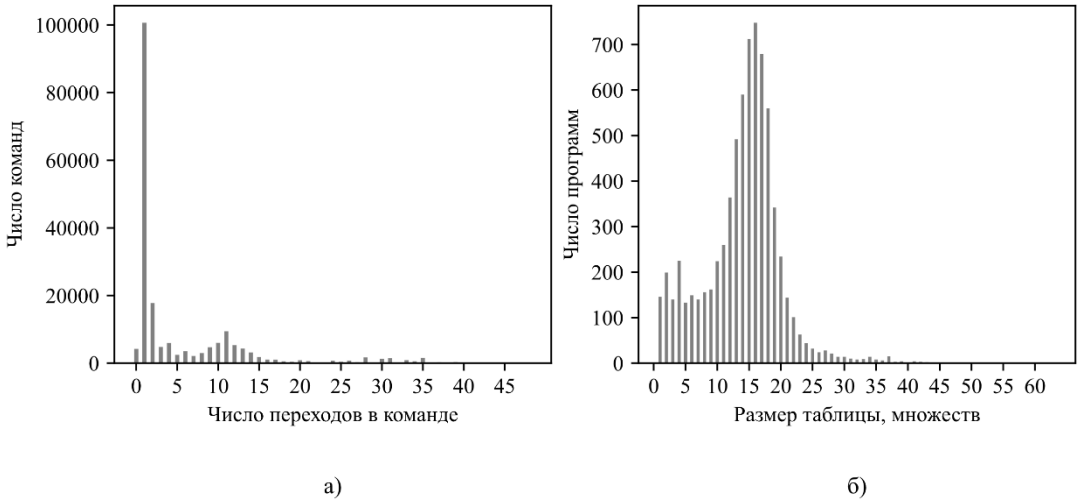

Рис. 12. Распределение:
а) числа переходов в командах без использования Path-opt, б) размеров таблиц в программах.
Fig. 12. Distribution of:
a) number of transitions in instruction without using Path-opt, b) table sizes in programs.

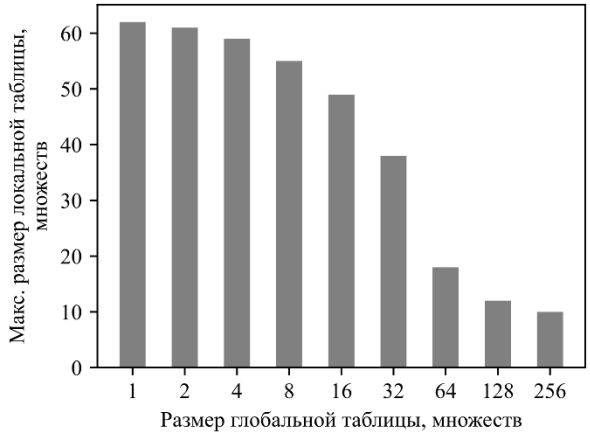

Рис. 13. Варианты размеров глобальной и локальной памяти таблиц.
Fig. 13. Variants of global and local table memory sizes.

Многие современные аппаратные ускорители для регулярных выражений поддерживают только варианты модели НКА. При этом для решений на FPGA, таких как [15], характерен подход на основе конфигурируемости, а не программируемости, то есть использование прямой трансляции графа НКА в представление уровня RTL. Недостатком этого подхода является длительное время синтеза конфигурации FPGA, что препятствует оперативному обновлению набора регулярных выражений в задачах информационной безопасности. Существуют также автоматные процессоры, широко использующие параллелизм НКА на основе вычислений в памяти (in-memory computing). В частности, архитектура автоматного процессора BVAP (Bit Vector Automata Processor) из [16] использует модель НКА с счетчиками циклов.

Программируемым является спецпроцессор Cicero из [17], в котором для машинного представления регулярных выражений используется оперативно обновляемый программный код. Однако, в отличие от HOREC, Cicero использует набор команд для аппаратного выполнения НКА, а не ДКА.

Примером аппаратной архитектуры, использующей ДКА, является работа [18], в которой описан процессор LAP (Lightweight Automata Processor), основанный на модели A-DFA (Amortized time/bandwidth overhead DFA). Недостатком LAP является отсутствие аппаратной поддержки циклов для интервальных квантификаторов. В [19] описана архитектура XAV (Xor filter, Anchor DFA, Verification). В рамках этой архитектуры сложные РВ разбиваются на множество якорных ДКА (Anchor DFA), сформированных из регулярных подвыражений с добавленным в начало символом ^. В HOREC такие РВ также являются предпочтительными для трансляции, но они преобразуются в компактные программы, а не в таблицы ДКА.

9. Заключение

В настоящей статье рассмотрен компилятор HOREC, используемый в цикле проектирования архитектуры программируемого аппаратного ускорителя поиска по РВ. Описана архитектура компилятора и новые модели R-НКА и R-ДКА, позволяющие существенно сократить число состояний автоматов при наличии интервальных квантификаторов. Рассмотрены вопросы оптимизирующей генерации кода с использованием, в частности, нового алгоритма для сокращения числа переходов в командах. Описан алгоритм интерпретации скомпилированных программ и представлен набор архитектурных параметров для проектирования аппаратного ускорителя. Приведены результаты экспериментальной оценки разработанного компилятора, подтверждающие ресурсоэффективность на уровне программной модели ускорителя.

В дальнейшем планируется расширение модели R-ДКА для обработки более сложных случаев сворачивания повторяемых состояний в R-символы, а также для поддержки дополнительных конструкций РВ. Интерес представляет возможность объединения нескольких автоматов и поддержка вызовов процедур в скомпилированных программах. Наконец, перспективы имеет гибридная архитектура аппаратного ускорителя, часть процессорных элементов которого поддерживает выполнение на основе НКА, для сложных случаев РВ.

Список литературы / References

[1]. Turoňová L., Holík L., Homoliak I., Lengál O., Veanes M., Vojnar T. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers / 31st USENIX Security Symposium (USENIX Security 22). — Boston, MA: USENIX Association, 2022. — С. 4165–4182.

[2]. Carloni F., Conficconi D., Moschetto I., Santambrogio M. D. YARB: a Methodology to Characterize Regular Expression Matching on Heterogeneous Systems / 2023 IEEE International Symposium on Circuits and Systems (ISCAS). — IEEE, 2023. — С. 1–5. — DOI: 10.1109/iscas46773.2023.10181547.

[3]. Sovetov P. N. Development of DSL Compilers for Specialized Processors // Programming and Computer Software. — Pleiades Publishing Ltd, 2021. — Т. 47, вып. 7. — С. 541–554. — DOI: 10.1134/s0361768821070082.

[4]. Sovietov P. N. Algorithms for Improving the Automatically Synthesized Instruction Set of an Extensible Processor // Programmnaya Ingeneria. — New Technologies Publishing House, 2023. — Т. 14, вып. 5. — С. 225–231. — DOI: 10.17587/prin.14.225-231.

[5]. Tiny parser combinators library written in Python, Available at: https://github.com/true-grue/peco, accessed 02.08.2025.

[6]. Antimirov V. Partial derivatives of regular expressions and finite automaton constructions // Theoretical Computer Science. — Elsevier BV, 1996. — Т. 155, вып. 2. — С. 291–319. — DOI: 10.1016/0304-3975(95)00182-4.

[7]. D'Antoni L., Veanes M. Minimization of symbolic automata / Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — ACM, 2014. — DOI: 10.1145/2535838.2535849.
[8]. Holzer M., Jakobi S. Brzozowski's Minimization Algorithm—More Robust than Expected // Implementation and Application of Automata. — Springer Berlin Heidelberg, 2013. — С. 181–192. — DOI: 10.1007/978-3-642-39274-0_17.
[9]. Sulzmann M., Lu K. Z. M. Regular expression sub-matching using partial derivatives / Proceedings of the 14th symposium on Principles and practice of declarative programming. — ACM, 2012. — С. 79–90. — DOI: 10.1145/2370776.2370788.
[10]. Proofpoint Emerging Threats Rules, Available at: https://rules.emergingthreats.net/, accessed 02.08.2025.
[11]. Le Glaunec A., Kong L., Mamouras K. Regular Expression Matching using Bit Vector Automata // Proceedings of the ACM on Programming Languages. — Association for Computing Machinery (ACM), 2023. — Т. 7, вып. OOPSLA1. — С. 492–521. — DOI: 10.1145/3586044.
[12]. Smith R., Estan C., Jha S. XFA: Faster Signature Matching with Extended Automata / 2008 IEEE Symposium on Security and Privacy (sp 2008). — IEEE, 2008. — С. 187–201. — DOI: 10.1109/sp.2008.14.
[13]. Turoňová L., Holík L., Lengál O., Saarikivi O., Veanes M., Vojnar T. Regex matching with counting-set automata // Proceedings of the ACM on Programming Languages. — Association for Computing Machinery (ACM), 2020. — Т. 4, вып. OOPSLA. — С. 1–30. — DOI: 10.1145/3428286.
[14]. Holík L., Lengál O., Saarikivi O., Turoňová L., Veanes M., Vojnar T. Succinct Determinisation of Counting Automata via Sphere Construction // Programming Languages and Systems. — Springer International Publishing, 2019. — С. 468–489. — DOI: 10.1007/978-3-030-34175-6_24.
[15]. Rahimi R., Sadredini E., Stan M., Skadron K. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs / 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). — IEEE, 2020. — С. 138–147. — DOI: 10.1109/fccm48280.2020.00027.
[16]. Wen Z., Kong L., Le Glaunec A., Mamouras K., Yang K. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions / Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. — ACM, 2024. — С. 151–166. — DOI: 10.1145/3620665.3640412.
[17]. Somaini A., Carloni F., Agosta G., Santambrogio M. D., Conficconi D. Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching / Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. — ACM, 2025. — С. 255–270. — DOI: 10.1145/3696443.3708916.
[18]. Gong L., Wang C., Xia H., Chen X., Li X., Zhou X. Enabling Fast and Memory-Efficient Acceleration for Pattern Matching Workloads: The Lightweight Automata Processing Engine // IEEE Transactions on Computers. — Institute of Electrical; Electronics Engineers (IEEE), 2023. — Т. 72, вып. 4. — С. 1011–1025. — DOI: 10.1109/tc.2022.3187338.
[19]. Zhong J., Chen S., Yu C. Xav: A High-Performance Regular Expression Matching Engine for Packet Processing. — Elsevier BV, 2024. — DOI: 10.2139/ssrn.4760269.

Информация об авторах / Information about authors

Петр Николаевич СОВЕТОВ – кандидат технических наук, доцент кафедры корпоративных информационных систем Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет». Сфера научных интересов: синтез программ, автоматизация проектирования спецпроцессоров, разработка компиляторов для спецпроцессоров.

Petr Nikolayevich SOVIETOV – Cand. Sci. (Tech.), Associate Professor of the Corporate Information Systems Department of the Institute of Information Technologies of the Institute of Information Technologies, Federal State Budgetary Educational Institution of Higher Education "MIREA - Russian Technological University". Research interests: program synthesis, design automation of specialized processors, compiler writing for specialized processors.