СОДЕРЖАНИЕ

Номер 6, 2021

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

Контроль и восстановление целостности многомерных массивов данных посредством криптокодовых конструкций
С. А. Диченко, О. А. Финько 3

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ

Быстрый алгоритм поиска распределенных рассеивателей SqueeSAR в задаче построения скоростей смещений земной поверхности
С. Е. Попов, В. П. Потапов 16

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Модели памяти языков программирования: обзор и тенденции
Е. А. Моисеенко, А. В. Подкопаев, Д. В. Кознов 30

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

Многооконная визуализация авиационного дисплея с использованием аппаратного ускорения
Б. Х. Барладян, Н. Б. Дерябин, Л. З. Шапиро, Ю. А. Солоделов, А. Г. Волобой, В. А. Галактионов 51

АНАЛИЗ ДАННЫХ

Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace
А. Е. Бородин, А. В. Горемыкин, С. П. Вартанов, А. А. Белеванцев 62

CONTENTS

No. 6, 2021

INFORMATION SECURITY

Control and Restoring the Integrity of Multi-Dimensional Data Arrays Through Cryptocode Constructs
S. A. Dichenko, O. A. Finko 3

PARALLEL AND DISTRIBUTED SOFTWARE

A Fast Search Algorithm for SqueeSAR Distributed Scatterers in the Problem of Calculating Displacement Velocities
S. E. Popov, V. P. Potapov 16

PROGRAMMING LANGUAGES, COMPILERS, AND INTEGRATED DEVELOPMENT ENVIRONMENT

A Survey of Programming Language Memory Models
E. A. Moiseenko, A. V. Podkopaev, D. V. Koznov 30

COMPUTER GRAPHICS AND VISUALIZATION

Multi-window visualization of aircraft display using hardware acceleration
B. Kh. Barladian, N. B. Deryabin, L. Z. Shapiro, Yu. A. Solodelov, A. G. Voloboy, V. A. Galaktionov 51

DATA ANALYSIS

Searching for tainted vulnerabilities in static analysis tool Svace
A. E. Borodin, A. V. Goremykin, S. P. Vartanov, A. A. Belevantsev 62

ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 3–15

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

УДК 519.718

КОНТРОЛЬ И ВОССТАНОВЛЕНИЕ ЦЕЛОСТНОСТИ МНОГОМЕРНЫХ МАССИВОВ ДАННЫХ ПОСРЕДСТВОМ КРИПТОКОДОВЫХ КОНСТРУКЦИЙ

© 2021 г. С. А. Диченко[a],*, О. А. Финько[a],**

[a] *Краснодарское высшее военное орденов Жукова и Октябрьской Революции Краснознаменное училище имени генерала армии С. М. Штеменко 350063 Краснодар, ул. Красина, д. 4, Россия*

**E-mail: dichenko.sa@yandex.ru*

***E-mail: ofinko@yandex.ru*

Поступила в редакцию 09.03.2021 г.
После доработки 31.05.2021 г.
Принята к публикации 23.06.2021 г.

Разработан метод контроля и восстановления целостности данных в многомерных системах хранения. Предложенные конструкции контроля и восстановления целостности многомерных массивов данных основаны на агрегировании методов криптографии и помехоустойчивого кодирования. На основе представленных криптокодовых конструкций показана особенность комплексирования существующих методов, заключающаяся в повышении вероятности обеспечения целостности информации в условиях разрушающих воздействий злоумышленника и среды для многомерных систем хранения данных. Получены расчетные данные вероятности обнаружения и исправления возникающих в многомерных массивах данных ошибок, приводящих к нарушению их целостности, посредством разработанного метода.

DOI: 10.31857/S0132347421060042

1. ВВЕДЕНИЕ

Эффективность любой информационной системы (ИС) независимо от условий ее функционирования, в первую очередь, характеризуется степенью достижения целей, поставленных при ее создании [1, 2]. Поэтому при разрушающих воздействиях злоумышленника и среды для ИС, обеспечивающих своевременность и правильность принимаемого пользователем системы решения, необходимым условием достижения требуемого качества их функционирования является обеспечение безопасности обрабатываемой информации.

Безопасность информации определяется состоянием защищенности ИС, а также систем хранения данных (СХД), применяемых в их интересах, от различных угроз и в итоге – способностью ИС обеспечить конкретному пользователю доступность, целостность и конфиденциальность требуемой информации в системе [3].

Для обеспечения целостности информации используются методы из области теоретических положений защиты информации и теории надежности, при которых для обнаружения и исправления ошибок, приводящих к искажению (или уничтожению) информации, требуется введение избыточности. При этом объем вводимой избыточности определяется, к примеру, необходимостью хранения эталонных хэш-кодов, используемых при контроле целостности данных посредством применения хэш-функции, и резервных копий данных для возможности восстановления их целостности на основе технологии резервного копирования.

Вместе с тем, в современных условиях постоянного роста объема и ценности непрерывно накапливаемой в СХД информации, введение требуемой избыточности и, как следствие, увеличение длительности загрузки новых данных в СХД, перерасход используемой памяти и снижение производительности системы в целом, приводит к необходимости разработки новых методов обеспечения целостности информации.

Формируемые в таких условиях требования к качеству функционирования ИС должны быть направлены на достижение цели применения системы за счет повышения уровня безопасности информации без увеличения при этом вводимой избыточности.

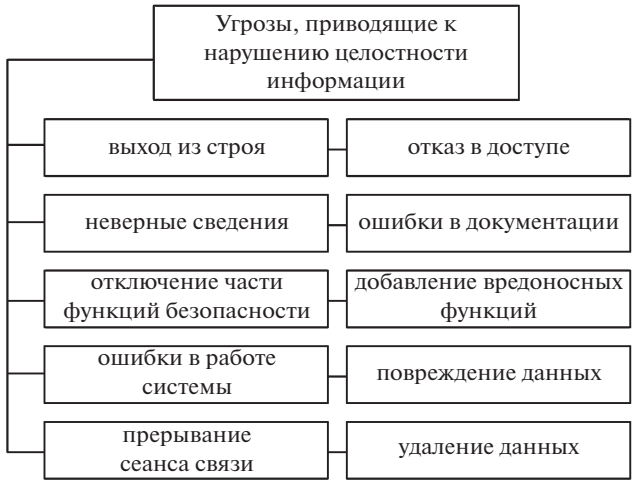


Рис. 1. Классификация угроз безопасности информации, приводящих к нарушению целостности информации.

2. УГРОЗЫ БЕЗОПАСНОСТИ ИНФОРМАЦИИ И ПОСЛЕДСТВИЯ ИХ РЕАЛИЗАЦИИ

Классификация угроз безопасности информации по признаку “результат реализации”, приводящие к нарушению ее целостности, представлена на рис. 1.

В то же время, целостность данных, хранящихся в СХД, чаще всего нарушается в результате случайных ошибок и преднамеренного несанкционированного изменения данных (например, посредством действия вредоносного кода) или выхода из строя части носителя (например, отдельных ячеек, секторов).

До настоящего времени задачу обеспечения целостности данных в СХД принято было решать путем повышения надежности средств хранения. Для этих целей широкое развитие получили системы резервирования и архивирования данных на дополнительные носители, построение каналов для их тиражирования и т.д.

3. АНАЛИЗ СУЩЕСТВУЮЩИХ МЕТОДОВ ЗАЩИТЫ ЦЕЛОСТНОСТИ ДАННЫХ

Известны методы из области теоретических положений защиты информации, позволяющие осуществить контроль целостности данных за счет вычисления значений контрольных сумм (CRC-кодов) и их сравнения со значениями эталонных [4, 5] (рис. 2).

Достоинством данных методов является простота реализации, недостатком – отсутствие обеспечения криптографической стойкости выполняемых при контроле целостности данных преобразований.

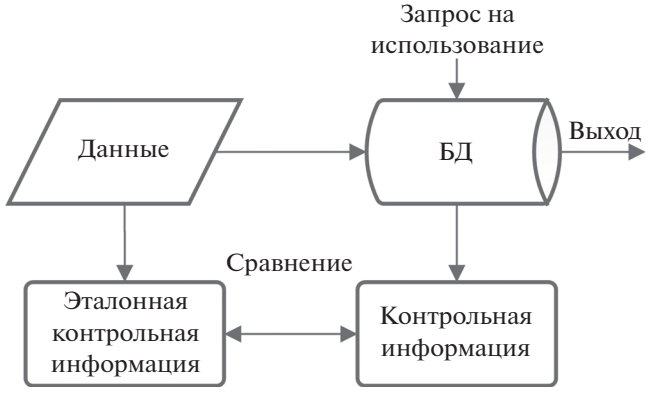


Рис. 2. Пояснение принципа контроля целостности данных на основе применения алгоритма вычисления контрольных сумм (CRC-кодов).

Известны решения, основанные на применении криптографических методов: ключевое и бесключевое хэширование, средства электронной подписи [6–9] (рис. 3).

Однако контроль целостности данных, разбитых на блоки небольшой размерности, где от каждого блока данных, подлежащего защите, вычисляется хэш-код, характеризуется вводом высокой избыточности контрольной информации. При попытке устранения данного недостатка за счет вычисления одного хэш-кода от нескольких блоков данных приводит к отсутствию возможности локализации блока данных с признаками нарушения целостности (рис. 4).

Наиболее популярные из существующих методов теории надежности, позволяющие осуществить контроль и восстановление целостности данных, основаны на применении различных видов резервирования (рис. 5), в том числе, с использованием программно-аппаратной или программной реализации технологии RAID (Redundant Array of Independent Disks) (RAID-массивы) [10] (рис. 6), методы дублирования и избыточного кодирования [11–15] (рис. 7).

Однако они предназначены для борьбы со случайными ошибками, поэтому в условиях разрушающих воздействий злоумышленника они не всегда являются эффективными. Данное утверждение складывается из того, что современная теория надежности, помимо множества других разделов математики, в основном, базируется на математическом аппарате теории вероятностей. При этом для повышения вероятности обнаружения и исправления возникающих ошибок, приводящих к нарушению целостности данных, требуется увеличение количества вводимой избыточности. К примеру, избыточность, вводимая для восстановления целостности данных посредством применения технологии резервного копи-

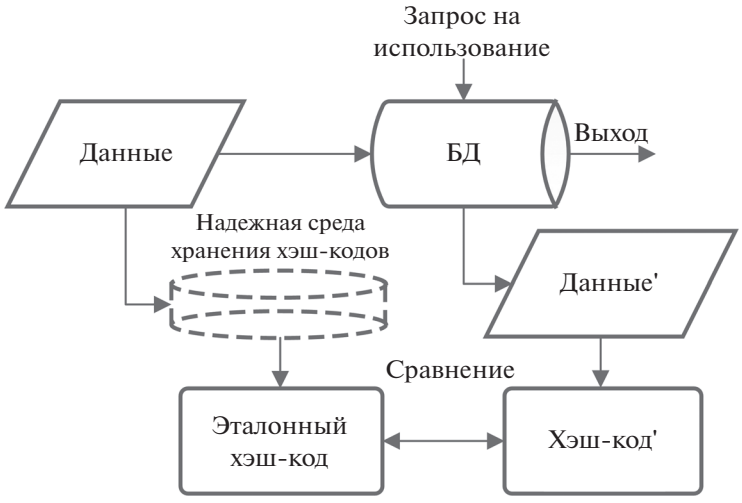


Рис. 3. Пояснение принципа контроля целостности данных на основе применения хэш-функции.

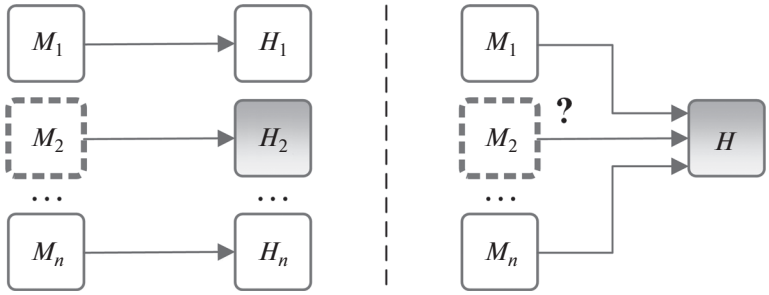


Рис. 4. Пояснение различий по выявлению блока данных M_2 при раздельном (схема слева) и общем (схема справа) применении криптографических методов (линией "---" обозначен блок данных с нарушением целостности).

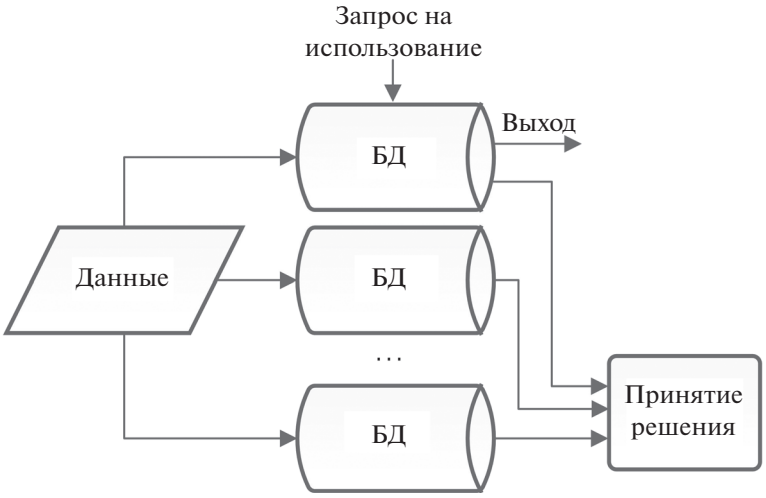


Рис. 5. Пояснение принципа резервирования БД.

Рис. 6. Пояснение принципа применения в БД технологии RAID (тип – 1).

рования, равна 100% от общего объема защищаемых данных (рис. 8).

Таким образом, ввод высокой избыточности в существующих методах объясняется потребностью в повышении обнаруживающей и исправляющей способностей при контроле и восстановлении целостности данных в условиях разрушающих воздействий.

4. МЕТОД КОНТРОЛЯ И ВОССТАНОВЛЕНИЯ ЦЕЛОСТНОСТИ МНОГОМЕРНЫХ МАССИВОВ ДАННЫХ

Предложенный метод предназначен для контроля и восстановления целостности данных, представленных в виде упорядоченных многомерных массивов. Он основан на совместном использовании методов криптографической защиты информации (функции хэширования) и теории кодов, контролирующих ошибки (избыточных модулярных полиномиальных кодов (МПК)), агрегирование которых позволяет повысить обнаруживающую и исправляющую способности при контроле и восстановлении целостности данных в условиях разрушающих воздействий без увеличения для этого вводимой избыточности.

Формализованное представление многомерных массивов данных. Блоки данных M, хранящиеся в СХД в виде упорядоченных многомерных массивов, по аналогии с теорией многомерных матриц [16] могут быть представлены p-мерным массивом данных, под которым будем понимать совокупность элементов $M_{\psi_1 \ldots \psi_p}$, где индексы $\psi_1 \ldots \psi_p$ принимают значения от 1 до k_a ($a = 1, \ldots, p$) соответственно.

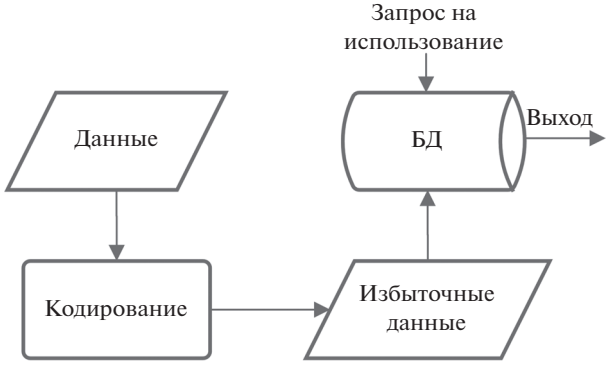


Рис. 7. Пояснение принципа применения метода избыточного кодирования в БД.

При этом p-мерный массив данных содержит $k_1 \times k_2 \times \ldots \times k_p$ элементов и обозначается как

$$\mathbf{M}[k_1, k_2, \ldots, k_p] = \{M_{\psi_1 \ldots \psi_p}\}.$$

На примере 3-мерного куба данных (рис. 9), содержащего систему координат с осями: x, y, z, по которым откладываются блоки данных M_{ijr} ($i, j, r = 0, \ldots, k-1$), получим 3-мерный массив данных:

$$\mathbf{M}[k, k, k] = \{M_{ijr}\},$$

который может быть представлен в следующем виде:

$$\mathbf{M}[k,k,k] = \left\{ \begin{vmatrix} M_{000}, & \ldots, & M_{k-1,0,0} \\ \vdots & \ddots & \vdots \\ M_{0,k-1,0}, & \ldots, & M_{k-1,k-1,0} \end{vmatrix} \ldots \right.$$
$$\left. \ldots \begin{vmatrix} M_{0,0,k-1}, & \ldots, & M_{k-1,0,k-1} \\ \vdots & \ddots & \vdots \\ M_{0,k-1,k-1}, & \ldots, & M_{k-1,k-1,k-1} \end{vmatrix} \right\}^{\mapsto(z)} \begin{matrix} \to (x) \\ \downarrow \\ (y), \end{matrix}$$

где стрелка "$\mapsto$" с индексом (z) показывает порядок представления массива посредством сечений, ориентированных по оси z; стрелки "$\to$" и "$\downarrow$" с индексами (x) и (y) указывают направления, в которых возрастают соответствующие индексы у элементов массива по осям x и y.

Построение модели контроля целостности многомерных массивов данных. Для осуществления контроля целостности полученного массива данных $\mathbf{M}[k, k, k]$ разместим его в массиве большего размера, который с помощью сечений ориентации (z) может быть представлен в следующем виде:

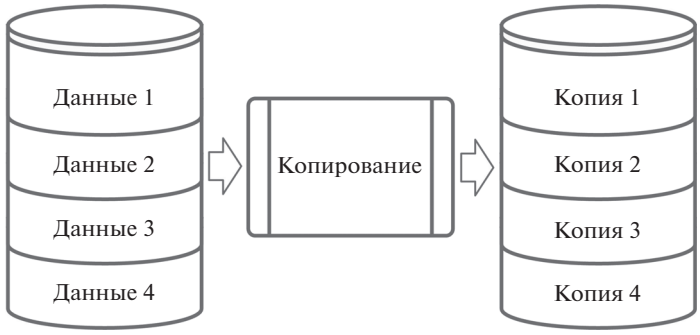


Рис. 8. Пояснение принципа применения технологии резервного копирования в БД.

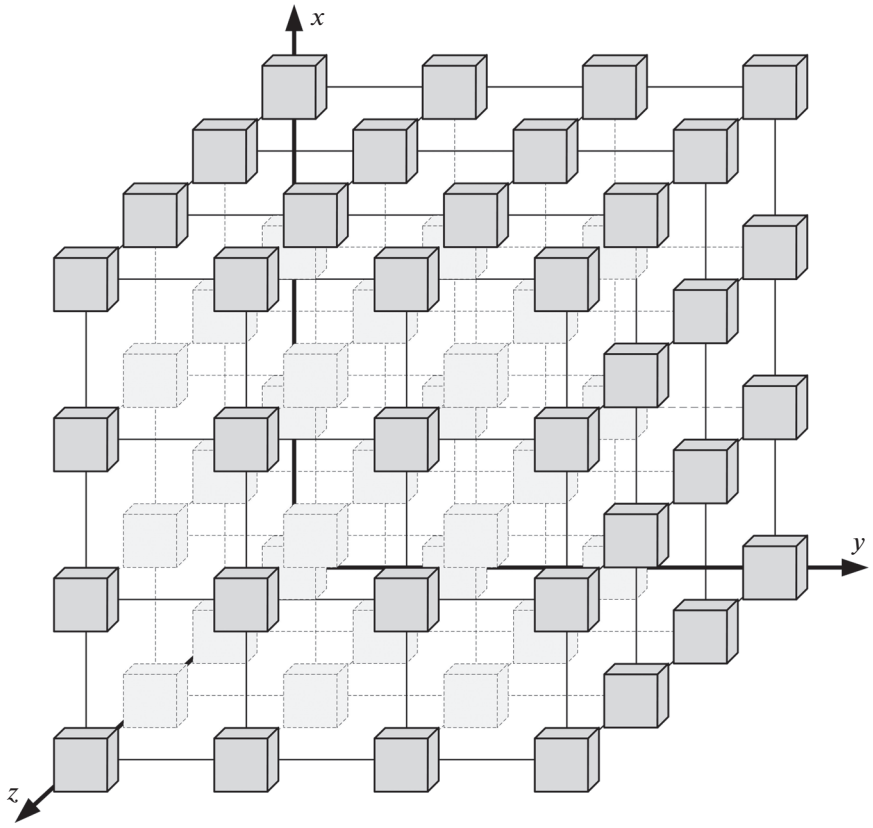


Рис. 9. Схема 3-мерного куба.

$$\mathbf{W}'[k+1,k+1,k+1] =$$
$$= \left\{ \begin{vmatrix} M_{000}, & \ldots, & M_{k-1,0,0}, & 0 \\ \vdots & \ddots & \vdots & \vdots \\ M_{0,k-1,0}, & \ldots, & M_{k-1,k-1,0}, & 0 \\ 0, & \ldots, & 0, & 0 \end{vmatrix} \ldots \right.$$
$$\left. \ldots \begin{vmatrix} M_{0,0,k-1}, & \ldots, & M_{k-1,0,k-1}, & 0 \\ \vdots & \ddots & \vdots & \vdots \\ M_{0,k-1,k-1}, & \ldots, & M_{k-1,k-1,k-1}, & 0 \\ 0, & \ldots, & 0, & 0 \end{vmatrix} \ldots \right\}^{\mapsto(z)}$$
$$\ldots \begin{vmatrix} 0, & 0, & \ldots, & 0 \\ 0, & 0, & \ldots, & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0, & 0, & 0, & 0 \end{vmatrix} \begin{matrix} \rightarrow (x) \\ \downarrow \\ (y), \end{matrix}$$

где “0” обозначает свободную для записи ячейку массива.

Контроль целостности 3-мерного массива данных $\mathbf{M}[k,k,k]$ осуществляется посредством применения к его элементам M_{ijr} хэш-функции [17].

Определение 4.1. Под хэш-функцией h понимается функция, отображающая строку бит $M \in \{0,1\}^*$ в строку бит $H \in \{0,1\}^q$:

$$h(M) = H,$$

где "$\{\cdot\}^*$" – произвольный размер строки бит, "$\{\cdot\}^q$" – фиксированный размер, $q \in N$.

Определение 4.2. Строка бит $M \in \{0,1\}^*$, которую хэш-функция h отображает в строку бит $H \in \{0,1\}^q$, называется блоком данных.

Определение 4.3. Под хэш-кодом понимается строка бит $H \in \{0,1\}^q$, являющаяся выходным результатом хэш-функции h.

На основе ранее разработанных схем хэширования [18–23] вычислим для блоков данных M_{ijr} $(i, j, r = 0, \ldots, k-1)$, подлежащих защите, хэш-коды H_{ijr} $(i, j, r = 0, \ldots, k)$.

В соответствии с правилами построения кубических кодов [15] разместим полученные хэш-коды в свободные для записи ячейки массива $\mathbf{W}'[k+1, k+1, k+1]$.

Получим массив, представленный с помощью сечений ориентации (z):

$$\mathbf{W}[k+1, k+1, k+1] =$$
$$= \begin{Vmatrix} M_{000}, & \ldots, & M_{k-1,0,0}, & H_{k00} \\ \vdots & \ddots & \vdots & \vdots \\ M_{0,k-1,0}, & \ldots, & M_{k-1,k-1,0}, & H_{k,k-1,0} \\ H_{0k0}, & \ldots, & H_{k-1,k,0}, & 0 \end{Vmatrix} \ldots$$
$$\ldots \begin{Vmatrix} M_{0,0,k-1}, & \ldots, & M_{k-1,0,k-1}, & H_{k,0,k-1} \\ \vdots & \ddots & \vdots & \vdots \\ M_{0,k-1,k-1}, & \ldots, & M_{k-1,k-1,k-1}, & H_{k,k-1,k-1} \\ H_{0,k,k-1}, & \ldots, & H_{k-1,k,k-1}, & 0 \end{Vmatrix} \ldots$$
$$\ldots \begin{Vmatrix} H_{00k}, & \ldots, & H_{k-1,0,k}, & 0 \\ \vdots & \ddots & \vdots & \vdots \\ H_{0,k-1,k}, & \ldots, & H_{k-1,k-1,k}, & 0 \\ 0, & \ldots, & 0, & 0 \end{Vmatrix}^{\mapsto(z)} \begin{matrix} \to (x) \\ \downarrow \\ (y). \end{matrix}$$

Хэш-коды H_{ijr}, вычисленные посредством применения к блокам данных M_{ijr} хэш-функции $h(M_{ijr})$ (или $h_k(M_{ijr})$, k – секретный ключ), будут являться эталонными.

Таким образом, полученный массив, содержащий $(k+1)^3$ ячеек, будет состоять из:

– k^3 блоков данных M_{ijr}, подлежащих защите целостности $(i, j, r = 0, \ldots, k-1)$;

– $3k^2$ блоков с вычисленными эталонными хэш-кодами H_{ijr} $(i, j, r = 0, \ldots, k)$;

– $3k+1$ оставшихся свободных ячеек.

В соответствии с полученной моделью к каждому блоку данных, подлежащему защите, по трем осям добавляются блоки с вычисленными от них эталонными хэш-кодами, используемыми для обнаружения данных с признаками нарушения целостности.

Определение 4.4. Под нарушением целостности одного блока данных будем понимать возникновение в нем ошибки, соответственно нарушение целостности q блоков данных определяется возникновением q-кратной ошибки.

Обнаружение блока данных с признаками нарушения целостности выполняется путем сравнения значений предварительно вычисленных от него эталонных хэш-кодов и хэш-кодов, вычисленных при запросе на его использование. В случае несоответствия сравниваемых значений хэш-кодов делается вывод о возникновении ошибки и определяется ее синдром.

Определение 4.5. Под синдромом ошибки будем понимать двоичное число, полученное при написании символа "0" для каждой выполненной проверки на соответствие значений вычисленного и эталонного хэш-кода и символа "1" при несоответствии сравниваемых значений.

Порядок контроля целостности многомерных массивов данных. В соответствии с разработанной моделью разместим в 3-мерном кубе (рис. 10) блоки данных, подлежащие защите, и соответствующие им хэш-коды.

При этом блоки данных $\mathbf{M}_{ijr}$, подлежащие защите, и вычисленные от них эталонные хэш-коды $\mathbf{H}_{ijr}$ интерпретируются как двоичные векторы:

$$\mathbf{M}_{ijr} = [\mu_1^{(ijr)}, \mu_2^{(ijr)}, \ldots, \mu_t^{(ijr)}];$$
$$\mathbf{H}_{ijr} = [\xi_1^{(ijr)}, \xi_2^{(ijr)}, \ldots, \xi_t^{(ijr)}],$$

где $\mu_g^{(ijr)}, \xi_g^{(ijr)} \in \{0,1\}$; $g = 1, 2, \ldots, t$.

Получим массив, представленный посредством сечений ориентации (x):

$$\mathbf{W}[3,3,3] = \begin{Vmatrix} \mathbf{M}_{000}, & \mathbf{M}_{010}, & \mathbf{H}_{020} \\ \mathbf{M}_{001}, & \mathbf{M}_{011}, & \mathbf{H}_{021} \\ \mathbf{H}_{002}, & \mathbf{H}_{012}, & 0 \end{Vmatrix}$$
$$\begin{Vmatrix} \mathbf{M}_{100}, & \mathbf{M}_{110}, & \mathbf{H}_{120} \\ \mathbf{M}_{101}, & \mathbf{M}_{111}, & \mathbf{H}_{121} \\ \mathbf{H}_{102}, & \mathbf{H}_{112}, & 0 \end{Vmatrix}$$
$$\begin{Vmatrix} \mathbf{H}_{200}, & \mathbf{H}_{210}, & 0 \\ \mathbf{H}_{101}, & \mathbf{H}_{111}, & 0 \\ 0, & 0, & 0, \end{Vmatrix}^{\mapsto(x)} \begin{matrix} \to (y) \\ \downarrow \\ (z), \end{matrix}$$

при этом каждый хэш-код вычисляется от двух блоков данных, расположенных с ним в одной строке или одном столбце массива.

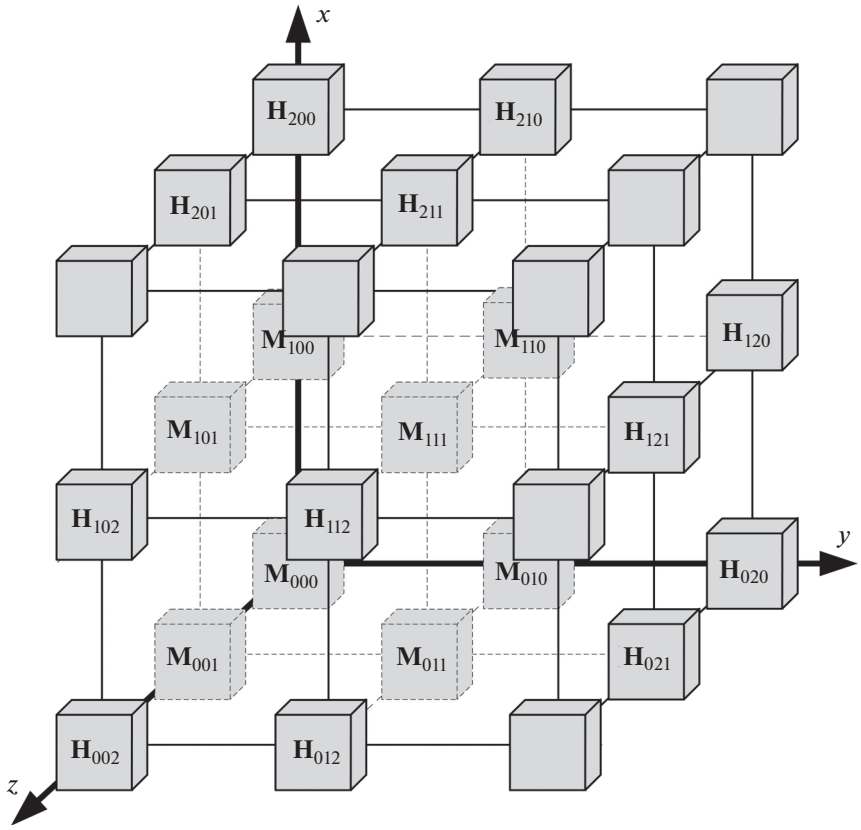


Рис. 10. Принцип организации 3-мерного куба (при $k = 2$).

Пример 4.1. Хэш-коды $\mathbf{H}_{002}$, $\mathbf{H}_{012}$, $\mathbf{H}_{020}$, $\mathbf{H}_{021}$ вычисляются следующим образом:

$$\mathbf{H}_{002} = h(\mathbf{M}_{000} \| \mathbf{M}_{001});$$
$$\mathbf{H}_{012} = h(\mathbf{M}_{010} \| \mathbf{M}_{011});$$
$$\mathbf{H}_{020} = h(\mathbf{M}_{000} \| \mathbf{M}_{010});$$
$$\mathbf{H}_{021} = h(\mathbf{M}_{001} \| \mathbf{M}_{011}),$$

где "$\|$" – операция конкатенации (объединения).

Построим сеть хэширования (рис. 11) для однократной ошибки, в которой каждому блоку данных $\mathbf{M}_{ijr}$ будет соответствовать неповторяющаяся совокупность из трех хэш-кодов $\mathbf{H}_{ijr}$.

Пример 4.2. Блокам данных $\mathbf{M}_{100}$, $\mathbf{M}_{011}$ соответствуют следующие неповторяющиеся совокупности из трех хэш-кодов:

– для $\mathbf{M}_{100}$: $\mathbf{H}_{120}, \mathbf{H}_{102}, \mathbf{H}_{200}$;

– для $\mathbf{M}_{011}$: $\mathbf{H}_{021}, \mathbf{H}_{012}, \mathbf{H}_{211}$.

Построим таблицу синдромов ошибок, приводящих к нарушению целостности блока данных $\mathbf{M}_{ijr}$, в которой место ошибки определяется наличием символова "1" в соответствующих столбцах и строках.

Таблица 1. Таблица с синдромами ошибок

	Синдромы ошибок			
Хэш-коды	–	$[\mathbf{M}_{100}]$	...	$[\mathbf{M}_{011}]$
$\mathbf{H}_{020}$	0	0	...	0
$\mathbf{H}_{021}$	0	0	...	1
$\mathbf{H}_{120}$	0	1	...	0
$\mathbf{H}_{121}$	0	0	...	0
$\mathbf{H}_{012}$	0	0	...	1
$\mathbf{H}_{002}$	0	0	...	0
$\mathbf{H}_{112}$	0	0	...	0
$\mathbf{H}_{102}$	0	1	...	0
$\mathbf{H}_{211}$	0	0	...	1
$\mathbf{H}_{201}$	0	0	...	0
$\mathbf{H}_{210}$	0	0	...	0
$\mathbf{H}_{200}$	0	1	...	0

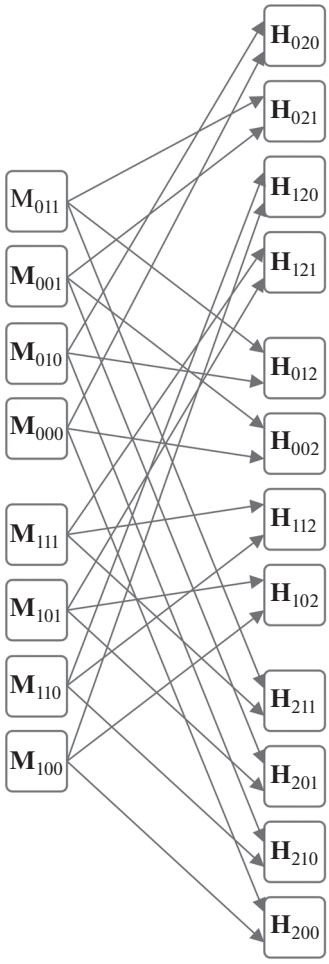


Рис. 11. Сеть хэширования.

Пример 4.3. Таблица синдромов ошибок, приводящих к нарушению целостности блоков данных $[\mathbf{M}_{100}]$ и $[\mathbf{M}_{011}]$, примет вид (табл. 1).

Сеть хэширования и соответствующая ей таблица синдромов ошибок для обнаружения и локализации q блоков данных с признаками нарушения целостности строятся по аналогичным правилам.

Построение модели восстановления целостности многомерных массивов данных. Блоки данных M_{ijr}, подлежащие защите, интерпретируются как элементы $\mathrm{GF}(2^t)$:

$$\Omega_{ijr}(\varpi) = \sum_{g=0}^{t-1} \mu_g^{(ijr)} \varpi^g = \mu_{t-1}^{(ijr)} \varpi^{t-1} + \\ + \mu_{t-2}^{(ijr)} \varpi^{t-2} + \ldots + \mu_0^{(ijr)}, \tag{4.1}$$

где ϖ – фиктивная переменная; $\mu_g^{(ijr)} \in \{0,1\}$; $g = t-1,\ t-2,\ldots,0$. Вычисленные от блоков данных M_{ijr} эталонные хэш-коды H_{ijr} также интерпретируются как элементы $\mathrm{GF}(2^t)$.

При этом представленные согласно выражению (4.1) блоки данных и эталонные хэш-коды являются наименьшими полиномиальными вычетами по основаниям $p_{ijr}(\varpi)$ таким, что

$$\gcd(p_{i_1jr}^{\mapsto(x)}(\varpi), p_{i_2jr}^{\mapsto(x)}(\varpi)) = 1, \tag{4.2}$$

где $p_{i_1jr}^{\mapsto(x)}(\varpi)$, $p_{i_2jr}^{\mapsto(x)}(\varpi)$ представлены при помощи сечений ориентации (x); $i_1, i_2 = 0,1,\ldots,k$; $i_1 \neq i_2$. При этом

$$\deg \Omega_{ijr}(\varpi) < \deg p_{ijr}(\varpi), \tag{4.3}$$

где $\deg \Omega_{ijr}(\varpi)$ – степень полинома $\Omega_{ijr}(\varpi)$.

Полученный массив, представленный посредством сечений ориентации (x):

$$\Omega[k+1,k+1,k+1] = \tag{4.4}$$

$$= \begin{Vmatrix} \Omega_{000}(\varpi), & \ldots, & \Omega_{0k0}(\varpi) \\ \vdots & \ddots & \vdots \\ \Omega_{0,0,k-1}(\varpi), & \ldots, & \Omega_{0,k,k-1}(\varpi) \\ \Omega_{00k}(\varpi), & \ldots, & 0 \end{Vmatrix} \ldots$$

$$\ldots \begin{Vmatrix} \Omega_{k-1,0,0}(\varpi), & \ldots, & \Omega_{k-1,k,0}(\varpi) \\ \vdots & \ddots & \vdots \\ \Omega_{k-1,0,k-1}(\varpi), & \ldots, & \Omega_{k-1,k,k-1}(\varpi) \\ \Omega_{k-1,0,k}(\varpi), & \ldots, & 0 \end{Vmatrix}$$

$$\begin{Vmatrix} \Omega_{k00}(\varpi), & \ldots, & 0 \\ \vdots & \ddots & \vdots \\ \Omega_{k,0,k-1}(\varpi), & \ldots, & 0 \\ 0, & \ldots, & 0 \end{Vmatrix}^{\mapsto(x)} \begin{matrix} \to (y) \\ \downarrow \\ (z), \end{matrix}$$

будем рассматривать как единый суперблок МПК по системе оснований:

$$\mathbf{P}[k+1,k+1,k+1] = \tag{4.5}$$

$$= \begin{Vmatrix} p_{000}(\varpi), & \ldots, & p_{0k0}(\varpi) \\ \vdots & \ddots & \vdots \\ p_{0,0,k-1}(\varpi), & \ldots, & p_{0,k,k-1}(\varpi) \\ p_{00k}(\varpi), & \ldots, & 0 \end{Vmatrix} \ldots$$

$$\ldots \begin{Vmatrix} p_{k-1,0,0}(\varpi), & \ldots, & p_{k-1,k,0}(\varpi) \\ \vdots & \ddots & \vdots \\ p_{k-1,0,k-1}(\varpi), & \ldots, & p_{k-1,k,k-1}(\varpi) \\ p_{k-1,0,k}(\varpi), & \ldots, & 0 \end{Vmatrix}$$

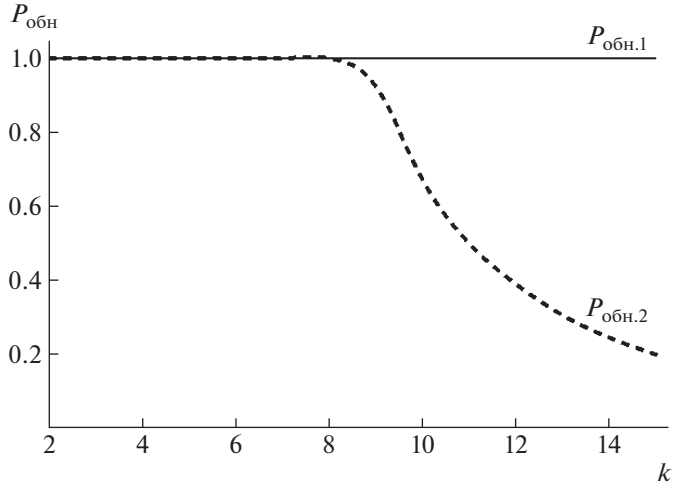


Рис. 12. Зависимости вероятностей $P_{\text{обн.1}}$, $P_{\text{обн.2}}$.

$$\left|\begin{matrix} p_{k00}(\varpi), & \ldots, & 0 \\ \vdots & \ddots & \vdots \\ p_{k,0,k-1}(\varpi), & \ldots, & 0 \\ 0, & \ldots, & 0 \end{matrix}\right|^{\mapsto(x)} \begin{matrix} \to (y) \\ \downarrow \\ (z). \end{matrix}$$

В соответствии с Китайской теоремой об остатках для многочленов, представленных в виде (4.5) и удовлетворяющих условию (4.2), и многочленов, представленных в виде (4.4) таких, что выполняется условие (4.3), система сравнений:

$$\begin{cases} \Xi(\varpi) \equiv \Omega_{000}(\varpi) \bmod p_{000}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{0k0}(\varpi) \bmod p_{0k0}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{00k}(\varpi) \bmod p_{00k}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{0,k-1,k}(\varpi) \bmod p_{0,k-1,k}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{k00}(\varpi) \bmod p_{k00}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{k,k-1,0}(\varpi) \bmod p_{k,k-1,0}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{k,0,k-1}(\varpi) \bmod p_{k,0,k-1}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Xi(\varpi) \equiv \Omega_{k,k-1,k-1}(\varpi) \bmod p_{k,k-1,k-1}(\varpi) \end{cases} \tag{4.6}$$

имеет единственное решение $\Xi(\varpi)$ [24].

Выполним операцию расширения МПК путем введения $n-k$ избыточных оснований: $p_{0,k+1,0}(\varpi)$, ..., $p_{0,k-1,n}(\varpi)$, ..., $p_{k-1,k+1,0}(\varpi)$, ..., $p_{k-1,k-1,n}(\varpi)$, $p_{k+1,0,0}(\varpi)$, ..., $p_{n,k-1,k-1}(\varpi)$ и получим соответствующие им избыточные вычеты:

$$\begin{cases} \Omega_{0,k+1,0}(\varpi) \equiv \Xi(\varpi) \bmod p_{0,k+1,0}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Omega_{0,k-1,n}(\varpi) \equiv \Xi(\varpi) \bmod p_{0,k-1,n}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Omega_{k-1,k+1,0}(\varpi) \equiv \Xi(\varpi) \bmod p_{k-1,k+1,0}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Omega_{k-1,k-1,n}(\varpi) \equiv \Xi(\varpi) \bmod p_{k-1,k-1,n}(\varpi), \\ \Omega_{k+1,0,0}(\varpi) \equiv \Xi(\varpi) \bmod p_{k+1,0,0}(\varpi), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Omega_{n,k-1,k-1}(\varpi) \equiv \Xi(\varpi) \bmod p_{n,k-1,k-1}(\varpi). \end{cases}$$

Причем выполняется условие (4.2), а условие (4.3) примет вид $\deg p_{000}(\varpi), \ldots, \deg p_{k,k-1,k-1}(\varpi) < \ldots < \deg p_{n,k-1,k-1}(\varpi)$.

Получим расширенный МПК в кольце многочленов:

$$\{\Omega_{000}(\varpi), \ldots, \Omega_{k,k-1,k-1}(\varpi), \Omega_{0,k+1,0}(\varpi), \ldots, \ldots, \Omega_{n,k-1,k-1}(\varpi)\}_{\text{МПК}}. \tag{4.7}$$

В соответствии с правилами построения кубических кодов [15] разместим избыточные блоки данных в свободных для записи ячейках массива.

Таким образом, полученный массив, содержащий n^3 ячеек, будет состоять из:

– k^3 блоков данных M_{ijr}, подлежащих защите целостности;

– $3k^2$ блоков с вычисленными эталонными хэш-кодами H_{ijr};

– $3k^2(n-k)$ избыточных блоков данных.

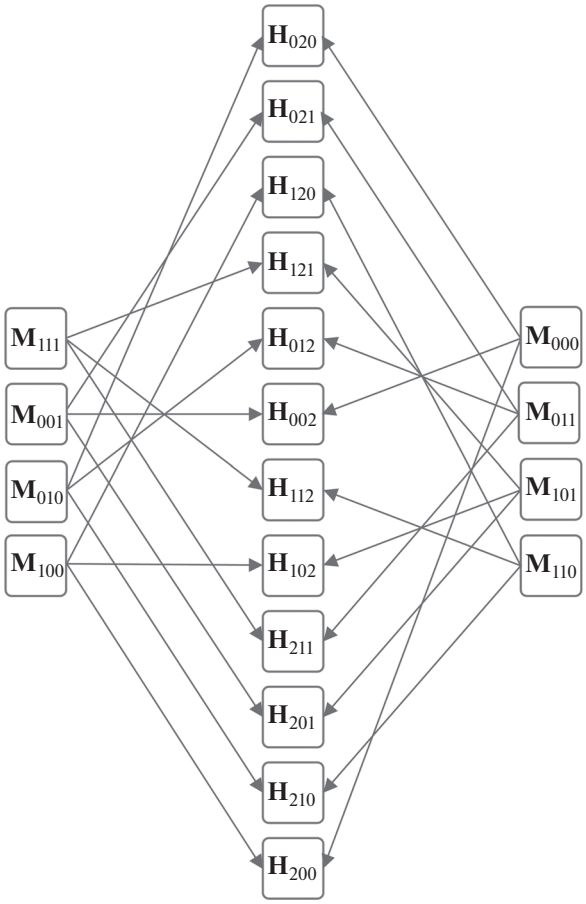


Рис. 13. Сеть хэширования для двух сочетаний блоков данных с совпадающими синдромами ошибок.

Данные, подлежащие защите, представленные в виде (4.7), отправляются на хранение.

Порядок восстановления целостности многомерных массивов данных. После осуществления контроля целостности данных в случае обнаружения ошибки выполняется процедура восстановления целостности данных.

В соответствии с правилами декодирования модулярных кодов [25–32] критерием отсутствия обнаруживаемых ошибок в расширенном МПК (4.7) является выполнение неравенства:

$$\deg \Xi'(\varpi) < \deg P_{nnn}(\varpi),$$

где $\Xi'(\varpi)$ является решением системы (4.6) для $\Omega'_{000}(\varpi), \ldots, \Omega'_{k,k-1,k-1}(\varpi)$, $\Omega'_{0,k+1,0}(\varpi), \ldots, \Omega'_{n,k-1,k-1}(\varpi)$; $P_{nnn}(\varpi) = \prod_{i,j,r=0}^{n} p_{ijr}(\varpi)$, символ "$(\cdot)'$" указывает на возможное нарушение целостности данных.

Критерий обнаруживаемой ошибки – выполнение неравенства:

$$\deg \Xi'(\varpi) \geq \deg P_{nnn}(\varpi).$$

Таблица 2. Таблица с совпадающими синдромами ошибок для двух сочетаний блоков данных

	Синдромы ошибок	
Хэш-коды	1 сочетание	2 сочетание
$\mathbf{H}_{020}$	1	1
$\mathbf{H}_{021}$	1	1
$\mathbf{H}_{120}$	1	1
$\mathbf{H}_{121}$	1	1
$\mathbf{H}_{012}$	1	1
$\mathbf{H}_{002}$	1	1
$\mathbf{H}_{112}$	1	1
$\mathbf{H}_{102}$	1	1
$\mathbf{H}_{211}$	1	1
$\mathbf{H}_{201}$	1	1
$\mathbf{H}_{210}$	1	1
$\mathbf{H}_{200}$	1	1

Восстановление целостности блока $[\Omega_{ijr}(\varpi)]$ выполняется путем вычисления наименьшего вычета:

$$\Omega_{ijr}(\varpi) \equiv \Xi(\varpi) \bmod p_{ijr}(\varpi),$$

где $\Xi(\varpi)$ повторно вычислено с учетом исключения искаженного блока данных $[\Omega_{ijr}(\varpi)]$.

Проверка достоверности и полноты данных после восстановления их целостности выполняется путем повторного сравнения значений эталонных хэш-кодов и вычисленных хэш-кодов от блоков данных.

5. ОЦЕНИВАНИЕ РЕЗУЛЬТАТОВ

Оценивание выполним в сравнении с существующим решением, в котором контроль целостности данных осуществляется посредством классического хэширования блоков данных, подлежащих защите, а восстановление целостности посредством кодов, корректирующих ошибки (МПК).

Оценивание результатов при контроле целостности данных. Разработанный метод в сравнении с существующим решением при фиксированном объеме вводимой избыточности позволяет гарантированно обнаружить бóльшее количество блоков данных с признаками нарушения целостности.

Так, например, СХД объемом 432 Кбайт позволяет хранить 675 хэш-кодов (при размере 1-го хэш-кода – 512 бит [33]).

Такое количество хэш-кодов при применении разработанного метода обеспечивает гарантированное обнаружение и локализацию (без учета коллизий) всех блоков данных с признаками на-

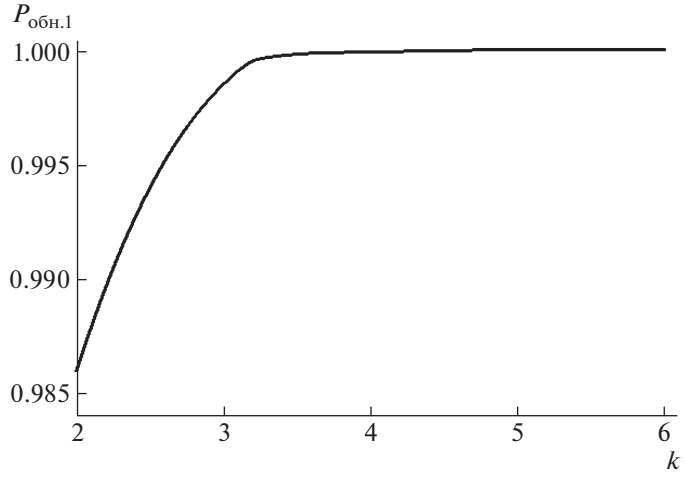


Рис. 14. Зависимость вероятности $P_{обн.1}$.

рушения целостности, размещенных в 3-мерном массиве с размером от $k = 2$ до $k = 15$.

Зависимости вероятностей $P_{обн}$ обнаружения и локализации блоков данных с признаками нарушения целостности представлены на рисунке 12, где $P_{обн.1}$ – вероятность, при применении разработанного метода, $P_{обн.2}$ – вероятность, при применении существующего решения.

Разработанный метод позволяет повысить вероятность обнаружения и локализации q-кратных ошибок. Его применение (без учета коллизий) позволяет обнаруживать и локализовывать все 1-, 2-, 3-кратные ошибки, а также до ≈98.6% всех 4-кратных ошибок без увеличения для этого количества вычисляемых хэш-кодов.

В частности, при $k = 2$ в 3-мерном массиве из всех 70 возможных различных сочетаний по четыре блока данных обнаруживаются и локализуются (без учета коллизий) до ≈98.6% всех ошибок, за исключением одной (≈1.4%), возникающей в одном из следующих сочетаний блоков данных:

- первое сочетание: $\mathbf{M}_{001}$, $\mathbf{M}_{010}$, $\mathbf{M}_{100}$, $\mathbf{M}_{111}$;
- второе сочетание: $\mathbf{M}_{000}$, $\mathbf{M}_{011}$, $\mathbf{M}_{101}$, $\mathbf{M}_{110}$.

Для рассматриваемых сочетаний построим сеть хэширования (рис. 13), на основе которой составим таблицу синдромов ошибок (табл. 2), из которой видно, что для представленных сочетаний блоков данных, подлежащих защите, синдромы ошибок совпадают.

При $k = 3$ в 3-мерном массиве существует 17550 различных сочетаний по четыре блока данных. В этом случае обнаруживаются и локализуются (без учета коллизий) до ≈99.85% всех ошибок, за исключением 27 (≈0.15%), которые не обнаружи-

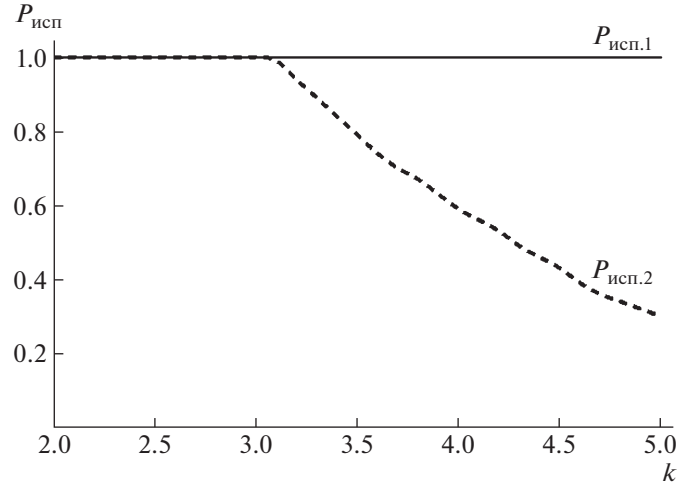


Рис. 15. Зависимости вероятностей $P_{исп.1}$, $P_{исп.2}$.

ваются в результате совпадения синдромов ошибок у 27 различных пар сочетаний блоков данных.

При последующем возрастании размерности k количество всех обнаруживаемых и локализуемых 4-кратных ошибок при применении разработанного метода стремится к 100% (без учета коллизий).

Зависимость вероятности $P_{\text{обн.1}}$ обнаружения и локализации 4-кратных ошибок при возрастании размерности k представлена на рис. 14.

Оценивание результатов при восстановлении целостности данных. В разработанном методе, учитывающем структуру многомерного представления данных, в соответствии с полученной моделью избыточные блоки данных располагаются по осям гиперкуба, что в сравнении с классическим применением избыточного МПК позволяет исправлять бóльшее количество возникающих ошибок.

В зависимости от размерности k максимальное количество возможных q блоков данных с нарушением целостности в кубе определяется в соответствии с: $q = k^3$. При этом для восстановления целостности всех искаженных блоков данных в разработанном методе потребуется $3k^2$ избыточных блоков данных. В то же время, классическое применение избыточного МПК, при котором, как известно [25, 26] исправляется q или менее ошибок, если $n - k \geq 2q$, требуется $2k^3$ избыточных блоков данных.

Зависимости вероятностей $P_{\text{исп}}$ исправления возникающих ошибок представлены на рисунке 15, где $P_{\text{исп.1}}$ – вероятность, при применении разработанного метода, $P_{\text{исп.2}}$ – вероятность, при применении существующего решения.

Повышение вероятности исправления возникающих ошибок за счет использования разработанного метода объясняется тем, что $n - k$ избыточных блоков данных используется для восстановления целостности не только $\frac{n-k}{2}$ искаженных блоков данных, как при классическом применении избыточного МПК, но и других блоков данных, подлежащих защите, в соответствии с введенными связями в 3-мерном кубе при построении модели восстановления целостности многомерных массивов данных.

6. ЗАКЛЮЧЕНИЕ

Представленное решение основано на агрегировании методов криптографической защиты информации (область теоретических положений защиты информации) и теории кодов контролирующих ошибки (область теории надёжности).

Объединение существующих методов контроля и восстановления целостности данных с учетом особенностей многомерной модели представления данных в современных СХД открывает новые возможности для повышения вероятности обнаружения и исправления возникающих ошибок без требуемого для этого увеличения объема вводимой избыточности, что, как следствие, приводит к сокращению времени загрузки новых данных в СХД и повышению эффективности ИС в целом.

СПИСОК ЛИТЕРАТУРЫ

1. Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Термины и определения. ГОСТ 34.003-90. М.: Стандартинформ, 2009. 48 с.
2. *Петухов Г.Б., Якунин В.И.* Методологические основы внешнего проектирования целенаправленных процессов и целеустремленных систем. М.: АСТ, 2006. 504 с.
3. Федеральный закон “Об информации, информационных технологиях и о защите информации” от 27.07.2006 № 149-ФЗ. 99 с.
4. *Schneier B.* Applied Cryptography Second Edition: protocols, algorithms and source code in C. John Wiley & Sons, Inc. 2016. 653 p.
5. *Menezes A.J., Oorschot P., Vanstone S.* Handbook of Applied Cryptography. CRC Press, Inc. 2015. 770 p.
6. *Knuth D.E.* The Art of Computer Programming: Volume 3: Sorting and Searching. 2nd edn. 2018. 803 p.
7. *Biham E., Dunkelman O.* A framework for iterative hash functions. – HAIFA. ePrint Archive, Report 2007/278. 2007. P. 1–20.
8. *Wang X., Yu H.* How to break MD5 and Other Hash Function. EUROCRYPT. LNCS 3494. Springer-Verlag. 2005. P. 19–35.
9. *Bellare M.* New Proofs for NMAC and HMAC: Security without Collision-Resistance.CRYPTO. ePrint Archive, Report 2006/043. 2006. P. 1–16.
10. *Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson.* RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, 26, 2014. P. 14–35.
11. *Сапожников В.В.* Основы теории надежности и технической диагностики. СПб.: Лань, 2019. 588 с.
12. *Хетагуров Я.А.* Повышение надежности цифровых устройств методами избыточного кодирования. М.: Энергия, 2015. 376 с.
13. *Зубарев Ю.М.* Основы надежности машин и сложных систем. СПб.: Лань, 2017. 180 с.
14. *Morelos-Zaragoza R.H.* The Art of Error Correcting Coding. 2nd edn John Wiley & Sons, Inc. 2006. 263 p.
15. *Hamming R.* Coding and Information Theory. Prentice-Hall, 2008. 259 p.
16. *Соколов Н.П.* Введение в теорию многомерных матриц. М.: Просвещение, 2012. 175 с.
17. *Tilborg H.* Fundamentals of cryptology. A professional reference and interactive tutorial. Kluwer Academic Publishers, 2014. 414 p.
18. *Dichenko S.A., Finko O.A.* Two-dimensional control and assurance of data integrity in information systems based on residue number system codes and cryptographic hash functions // Integrating Research Agen-

das and Devising Joint Challenges International Multidisciplinary Symposium ICT Research in Russian Federation and Europe. 2018. P. 139–146.

19. *Диченко С.А., Финько О.А.* Гибридный крипто-кодовый метод контроля и восстановления целостности данных для защищенных информационно-аналитических систем // Вопросы кибербезопасности. 2019. № 6 (34). С. 17–36.
20. *Диченко С.А.* Контроль и обеспечение целостности информации в системах хранения данных // Наукоемкие технологии в космических исследованиях Земли. 2019. Т. 11. № 1. С. 49–57.
21. *Диченко С.А., Финько О.А.* Обобщенный способ применения хэш-функции для контроля целостности данных // Наукоемкие технологии в космических исследованиях Земли. 2020. Т. 12. № 6. С. 48–59.
22. Способ многоуровневого контроля и обеспечения целостности данных. Пат. на изобретение RU 2707940. Заявка № 2019103723 от 11.02.2019. Опуб. 02.12.2019. Бюл. № 34. 17 с.
23. Способ двумерного контроля и обеспечения целостности данных. Пат. на изобретение RU 2696425. Заявка № 2018118919 от 22.05.2018. Опуб. 02.08.2019. Бюл. № 22. 26 с.
24. *Ananda Mohan P.V.* Residue Number Systems. Springer International Publishing, 2016. 351 p.
25. *Акушский И.Я., Юдицкий Д.М.* Машинная арифметика в остаточных классах. М.: Советское радио, 1968. 604 с.
26. *Бояринов И.М.* Помехоустойчивое кодирование числовой информации. М.: Наука, 1983. 386 с.
27. *Szabo N.S., Tanaka R.I.* Residue Arithmetic and its Applications Computer Technology. New York: McGraw-Hill, 1967. 476 p.
28. *Амербаев В.М.* Теоретические основы машинной арифметики. Алма-Ата: Наука, 1976. 498 с.
29. *Торгашов В.А.* Система остаточных классов и надежность ЦВМ. М.: Советское радио, 1973. 329 с.
30. *Mandelbaum D.M.* Error correction in residue arithmetic // IEEE Trans. Comput. 1972. № 21. P. 538–545.
31. *Omondi A., Premkumar B.* Residue Number Systems: Theory and Implementation. Imperial College Press. London, 2007. 296 p.
32. *Mandelbaum D.M.* A method of coding for multiple errors // IEEE Trans. OnInformation Theory. 1968. № 14. P. 518–521.
33. Информационная технология. Криптографическая защита информации. Функция хэширования. ГОСТ Р 34.11-2012. М.: Стандартинформ, 2013. 30 с.

ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 16–29

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ

УДК 004.021,004.042

БЫСТРЫЙ АЛГОРИТМ ПОИСКА РАСПРЕДЕЛЕННЫХ РАССЕИВАТЕЛЕЙ SQUEESAR В ЗАДАЧЕ ПОСТРОЕНИЯ СКОРОСТЕЙ СМЕЩЕНИЙ ЗЕМНОЙ ПОВЕРХНОСТИ

© 2021 г. С. Е. Попов[a],*, В. П. Потапов[a],**

[a] *Федеральное государственное бюджетное научное учреждение "Федеральный исследовательский центр информационных и вычислительных технологий" 630090 Новосибирск, пр. Академика Лаврентьева, 6, Россия*

**E-mail: popov@ict.sbras.ru,*

***E-mail: vadimptpv@gmail.com*

Поступила в редакцию 23.12.2020 г.
После доработки 27.04.2021 г.
Принята к публикации 15.06.2021 г.

В статье описывается программная реализация быстрого алгоритма поиска распределенных рассеивателей для задачи построения скоростей смещений земной поверхности на базе платформы Apache Spark. Рассматривается полная схема расчета скоростей смещений методом постоянных рассеивателей (PS). Предложенный алгоритм интегрируется в схему после этапа совмещения с субпиксельной точностью стека изображений временной серии радарных снимков космического аппарата Sentinel-1. Поиск распределенных рассеивателей происходит независимо в окнах сдвига по всей площади снимка. Наличие последних определяется путем предположения о гомогенности пар выборок в окне, составленных из векторов комплексных значений пикселей в каждом из N изображений. Данное предположение вытекает из выполнимости критерия Колмогорова–Смирнова для каждой из пар. Для оценки значений фаз гомогенных пикселей решается задача максимизации. Показано, что предложенный алгоритм не является итерационным и может быть реализован в парадигме параллельных вычислений. Применяемая платформа Apache Spark позволила распределенно обрабатывать массивы стека радарных данных (от 60 изображений) в памяти на большом количестве физических узлов в сетевой среде. При этом, время поиска распределенных рассеивателей удалось снизить в среднем в 10 раз по сравнению с однопроцессорной реализацией алгоритма. Приведены сравнительные результаты тестирования вычислительной системы на демонстрационном кластере. Алгоритм реализован на языке программирования Python с подробным описанием объектов и методов алгоритма.

DOI: 10.31857/S0132347421060066

1. ВВЕДЕНИЕ

Основная ценность космической информации, поступающей при мониторинге земной поверхности, зачастую, заключается в возможности ее оперативного анализа. Поэтому, активное развитие методов дифференциальной интерферометрии и средств дистанционного зондирования кроме совершенствования аппаратной части спутников, также требует создания проблемно-ориентированных алгоритмов для автоматизированной и оперативной обработки большого объема радарных данных. Для стандартного в радарной интерферометрии аналитического аппарата DInSAR [1] для оценки скоростей смещений земной поверхности широкое распространение получил метод постоянных отражателей (Persistent Scatterer, PS) [2]. PS направлен на идентификацию когерентных радиолокационных целей, демонстрирующих высокую фазовую стабильность в течение всего периода наблюдения. Эти цели (пиксели изображения), на которые лишь незначительно влияет временная и геометрическая декорреляция, часто соответствуют точечным рассеивателям и обычно характеризуются высокими значениями отражательной способности [3]. В работах [4, 5 и др.] показано, что эти пиксели также соответствуют пикселям изображения, принадлежащим областям умеренной когерентности в некоторых интерферометрических парах доступного набора данных. Здесь многие соседние пиксели имеют одинаковые значения отражательной способности, поскольку они принадлежат к одному объекту. Эти цели, называемые распределенными рассеивателями (Distributed Scatterers (DS)), обычно соответствуют участкам небольших строений, необрабатываемых земель с короткой растительно-

стью, промышленному мусору, металлическим завалам или пустынным территориям. Хотя средняя временная когерентность этих естественных радиолокационных целей обычно низкая из-за явлений как временной, так и геометрической декорреляции, количество пикселей с одинаковым статистическим поведением может быть достаточно большим, чтобы некоторые из них могли превысить порог когерентности и стать постоянными рассеивателями.

В работах [4–6] описывается решение актуальной задачи слияния данных для правильного объединения рассеивателей PS и DS для увеличения плотности точек измерения. Это позволяет увеличить пространственную плотность точек областей, характеризующимися DS при сохранении высококачественной информации, полученной с помощью метода PS по детерминированным целям. Данные пространственно усредняются по статистически однородным областям, увеличивая отношение сигнал/шум (SNR), без ущерба для идентификации когерентных точечных рассеивателей. Данный алгоритм назван SqueeSAR [7, 8]. Он позволяет проводить поиск и обработку точек распределённых рассеивателей, интегрируя их в общую схему расчета скоростей смещений методом PS.

Тщательный анализ алгоритма SqueeSAR выявил места, критически влияющие на его производительность. Весь алгоритм строится на переборе начальных данных. На каждом шаге выполняются нетривиальные преобразования данных. Так, например, ощутимо сложными в плане вычислительных затрат оказались этапы поиска смежных точек в окне сдвига и решение задачи максимизации при оценке реальных значений интерферометрических фаз [9]. Кроме того, при уменьшении размеров окна, увеличивалось общее количество шагов проходки по всей площади снимка.

Для устранения выявленных проблем вычислительного характера было предложено использование платформы массово-параллельных вычислений Apache Spark [10]. Apache Spark – это фреймворк с открытым исходным кодом для распределённой пакетной и потоковой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop. В отличие от классического обработчика ядра Apache Hadoop с двухуровневой концепцией MapReduce на базе дискового хранилища, Spark использует специализированные примитивы (Resilient Distributed Data) для рекуррентной обработки в оперативной памяти. Благодаря этому появляется возможность многократного доступа к загруженным в память радарным данным с каждого узла кластера. Это позволяет логически разделять стек снимков на подобласти и проводить вычисления независимо.

Целью данной работы является разработка высокопроизводительного алгоритма поиска распределенных рассеивателей для математической модели SqueeSAR.

Для выполнения поставленной цели предлагается решение следующих задач:

1. Разработка программного алгоритма на основе математической модели SqueeSAR на базе открытых библиотек с возможностью его интеграции в схему расчета скоростей смещений методом постоянных отражателей.

2. Адаптация представленного алгоритма для запуска его в среде массово-параллельного исполнения заданий.

2. МАТЕМАТИЧЕСКАЯ МОДЕЛЬ АЛГОРИТМА SqueeSAR

Алгоритм SqueeSAR базируется на идее поиска статистически однородных пикселей, в результате чего выявляются подмножества точечных, распределенных рассеивателей, и выполняется пространственно-адаптивная фильтрация последних. Вследствие чего резко возрастает суммарный набор постоянных отражателей, что позволяет повысить эффективность решения задач дифференциальной радарной интерферометрии [9]. Алгоритм состоит из следующих этапов [11]:

1. Определение статистически однородных точек (Statistically Homogeneous Pixels – SHP) по всей площади стека интерферометрических изображений.

2. Формирование наборов распределенных рассеивателей путем фильтрации по количественному порогу *SHP*-пикселей.

3. Поиск для каждого DS-набора уточненных (оценка) значений интерферометрической фазы *SHP*-пикселей, используя специальную матрицу когерентности, и решение на ее основе задачи максимизации.

4. Фильтрация полученных уточненных интерферометрических фаз пикселей DS-набора на основе оценки параметра γ_{PTA}.

5. Замена значений первоначальных фаз в интерферометрическом стеке их уточненными значениями. Модифицированные изображения далее участвуют в процессе обработки методом постоянных рассеивателей [2].

Будем рассматривать стек из N совмещенных с субпиксельной точностью радарных изображений длинной временной серии (от 60 снимков). Здесь субпиксельная точность означает, что любая произвольно взятая точка на мастер-изображении будет иметь абсолютно одинаковые географические координаты и на N–1 подчиненных

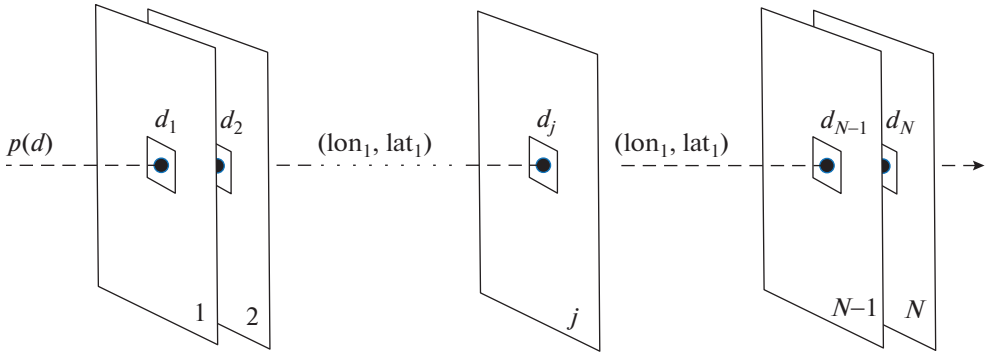


Рис. 1. Стек из N совмещенных с субпиксельной точностью изображений.

изображениях (рис. 1). Обозначим W и H ширину и высоту изображений в пикселях, соответственно.

Делается предположение, что данные радарных изображений и рассчитываемые на их основе геофизические параметры являются общими для каждой статистически однородной выборки близлежащих пикселей. В соответствии с этим предположением однородность (гомогенность) пикселей оценивается в пределах заданного окна размером *mxn* (обычно 21 × 15 пикселей). Сдвиг окна происходит по ширине и высоте изображения с шагом равным ширине и высоте окна.

На каждом шаге выбирается центральный пиксель p_0 в окне сдвига. Так как на изображениях каждый пиксель характеризуется комплексным представлением (каналами I и Q, для данных Sentinel-1), то можно сформировать вектор комплексных чисел d в стеке из N изображений (1)

$$d_k = [d_{k,1}, d_{k,2} \ldots d_{k,N}]^T, \qquad (1)$$

где $d_{k,j} = a + ib$, $k = 0 \ldots mxn$, $j = 1 \ldots N$, $a \in I$, $b \in Q$ – комплексное представление значения пикселя в соответствующем j-м изображении в стеке для k-го пикселя, T – оператор транспонирования.

Два пикселя p_0 и p_k в окне будем считать однородными если для них выполняется тест Колмогорова–Смирнова. Каждая пара пикселей (p_0, p_k) рассматривается как две независимые выборки $|p_0(d_j)|$ и $|p_k(d_j)|$, где $k = 1 \ldots mxn$, $|p_k(d_j)| = [|d_{k,1}|, |d_{k,2}|, \ldots |d_{k,N}|]^T$, $|*|$ – модуль комплексного числа. Здесь тест Колмогорова–Смирнова выполняется следующим образом:

1. Выдвигаются две гипотезы: H_0 – различия между двумя выборками недостоверны, H_1 – противоположная (различия между выборками достоверны), соответственно для заданного уровня значимости α.

2. Строится частотное распределение каждой выборки.

3. Вычисляются относительные частоты, равные частному от деления частот на объём выборки, для каждой из имеющихся выборок.

4. Определяется модуль разности соответствующих относительных частот.

5. Определяется наибольший модуль D_{max}.

6. Вычисляется эмпирическое значение критерия λ_{emp}.

$$\lambda_{emp} = D_{max}\sqrt{\frac{N}{2}}.$$

7. Определяется критическое значение критерия для выбранного уровня значимости $\alpha = 0.05$.

8. Если эмпирическое значение критерия меньше критического, то нулевая гипотеза принимается, и выборки по рассмотренному признаку не отличаются существенно, т.е. пара пикселей считается однородной.

В получившемся наборе пикселей, прошедших тест, отбрасываются те пиксели, которые не являются смежными с центральным (p_0) либо напрямую, либо через соседние (рис. 2).

Оставшиеся пиксели (обозначим их количество параметром N_{ds}) образуют DS-набор. Если параметр N_{ds} меньше порогового значения ($N_p = 20$), то такой набор не принимается. Таким образом, при проходе всего снимка образуются наборы распределенных рассеивателей. В каждом корректном DS-наборе (2) для центрального пикселя p_0 выполняется процедура уточнения значения его “свернутой” интерферометрической фазы в каждом из N изображений стека.

$$DS = \{p_i(d)\}, \quad i = 1 \ldots N_{ds}, \quad N_{ds} > N_p \qquad (2)$$

Значения интерферометрических фаз пикселей в DS-наборе могут быть статистически охарактеризованы с использованием матрицы когерентности, которая определяется следующим образом (3):

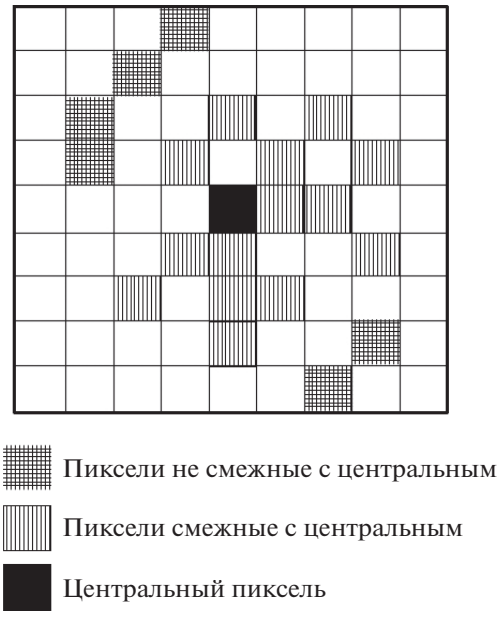


Рис. 2. Вариант расположения пикселей в наборе DS. Белые пиксели не являются гомогенными с центральным, заштрихованные пиксели прошли тест Колмогорова–Смирного и являются гомогенными с центральным.

$$T = \frac{1}{N_{ds}} \sum_{i=1}^{N_{ds}} \hat{p}_i \hat{p}_i^{+}, \tag{3}$$

где

$+$ – операция эрмитового сопряжения вектора $\hat{p}_i$,

$$\hat{p}_i = p_i(d) / \sqrt{E[|p_i(d)|^2]},$$

$$E[|p_i(d)|^2] = \frac{1}{N} \sum_{j=1}^{N} |d_{ij}|^2,$$

$|d_{ij}|$ – модуль комплексного числа, T – матрица комплексных чисел размерностью $N \times N$.

Ключевой проблемой является нахождение вектора $\theta = [\theta_1, \theta_2 \ldots \theta_N]^T$, который бы служил оценкой полученных значений фаз φ для недиагональных элементов в матрице T. Для этого представим матрицу T в параметрическом виде, заменив диагональные элементы на 1 (4):

$$T = \begin{bmatrix} 1 & \gamma_{1,2} e^{j\varphi_{1,2}} & \ldots & \gamma_{1,N} e^{j\varphi_{1,N}} \\ \gamma_{2,1} e^{j\varphi_{2,1}} & 1 & \ldots & \gamma_{2,N} e^{j\varphi_{2,N}} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{N,1} e^{j\varphi_{N,1}} & \gamma_{N,2} e^{j\varphi_{N,2}} & \ldots & 1 \end{bmatrix} \tag{4}$$

При этом значения фаз $\varphi_{m,n} = -\varphi_{n,m}$, $n.m = 1 \ldots N$, а матрица $|T|$ будет представлена элементами $\gamma_{m,n}$.

Рассматривается метод максимального правдоподобия (ML estimator (MLE)). Построение корректной формы MLE проводится путем анализа статистических свойств матрицы когерентности (4), используя комплексное распределение Уишарта. На основе центральной предельной теоремы, делается предположение, что нормализованный вектор радарных данных $\hat{p}_i$ соответствует комплексному многомерному нормальному распределению с нулевым средним и дисперсионной матрицей Σ [12–14]. Предполагается, что матрица T логически может быть представлена комплексным распределением Уишарта с N_{ds} – степенями свободы в виде $T \sim W_c(N, N_{ds}, \Sigma)$, где Σ для пикселя $p_0(d)$ может быть определена с использованием уточненных значений когерентности и фазы как (5)

$$\Sigma = \Theta \Upsilon \Theta^H, \tag{5}$$

где $\Theta = \begin{bmatrix} e^{j\theta_1} & 0 & \ldots & 0 \\ 0 & e^{j\theta_2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & e^{j\theta_N} \end{bmatrix}$, $\Upsilon = \begin{bmatrix} 1 & \gamma_{1,2} & \ldots & \gamma_{1,N} \\ \gamma_{2,1} & 1 & \ldots & \gamma_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{N,1} & \gamma_{N,2} & \ldots & 1 \end{bmatrix}$,

$\theta = [\theta_1, \theta_2 \ldots \theta_N]^T$ – вектор оптимальных (уточненных) фаз для вектора $\varphi = [\varphi_1, \varphi_2 \ldots \varphi_N]$. При вычисленной матрице T корректная форма MLE для Σ может быть получена путем максимизации абсолютного значения логарифма следующей функции плотности вероятности [12]:

$$\begin{aligned} \hat{\Sigma} &= \arg\max_{\Sigma}\{\ln[p(T \mid \Sigma)]\} = \\ &= \arg\max\{-\mathrm{trace}(N_{dc}\Sigma^{-1}T) - N_{ds}\ln(\mathrm{Det}(\Sigma))\} = \\ &= \arg\max_{\Upsilon,\Theta}\{-\mathrm{trace}(\Theta\Upsilon^{-1}\Theta^H T) - \ln(\mathrm{Det}(\Sigma))\} \end{aligned} \tag{6}$$

Для оценки Θ в (6) требуется значения γ матрицы Υ. Не теряя общности, их можно заменить значениями γ из $|T|$ [12]. Следовательно, (6) примет следующий вид:

$$\begin{aligned} \Theta_{ML} &= \arg\max_{\Theta}\{-\mathrm{trace}(\Theta|T|^{-1}\Theta^H T) - \ln(\mathrm{Det}(|T|))\} \\ &= \arg\max_{\Theta}\{-\mathrm{trace}(\Theta|T|^{-1}\Theta^H T)\} = \\ &= \arg\max_{\Theta}\{\Lambda^H(-|T|^{-1} \circ T)\Lambda\}, \end{aligned} \tag{7}$$

где $\Lambda = [e^{j\theta_1}, e^{j\theta_2} \ldots e^{j\theta_N}]^T$, знак $\circ$ – произведение Адамара.

Для удобства программной реализации преобразуем (7) в следующий вид, используя тригонометрическую форму комплексного числа:

Таблица 1. Начальные переменные

Переменная	Тип	Размерность	Описание
img_width, img_height, N	int	1	Ширина, высота изображений, их количество в серии
i, q	Array, float32	width*height, N	Массивы значений пикселей для полос Q и I для каждого изображения в серии, соответственно
Np	int		Пороговое значение для количества пикселей в DS-наборе
shift_win_widt, shift_win_height	int	1	Ширина и высота окна сдвига
num_slices	int	1	Количество разделов (partitions) во вновь создаваемом RDD [10]

$$\Theta_{ML} = \arg\max_{\Theta}\{\mathrm{sum}((-|T|^{-1}\circ|T|)\circ\Phi\circ(\Lambda\Lambda^{H})^{T})\} = \\ = \arg\max_{\Theta}\{F_{\max}\}, \quad (8)$$

$$F_{\max} = \sum_{m=1}^{N}\sum_{n>m}^{N}\hat{\gamma}_{m,n}\cos(\varphi_{m,n} - \theta_m - \theta_n),$$

где $\hat{\gamma}_{m,n}$ – элемент матрицы $-|T|^{-1}\circ|T|$ в строке m и столбце n, соответственно. sum(*) – оператор суммирования.

Решение (8) можно проводить различными методами. В данной работе будем использовать итерационный метод численной оптимизации BFGS (Broyden, Fletcher, Goldfarb, Shanno). После того, как будет найден вектор оптимальных решений $\theta = [\theta_1, \theta_2 \ldots \theta_N]^T$, необходимо оценить качество полученной оптимизации. Для этого используется следующий фильтр:

$$\gamma_{PTA} = \frac{2}{N^2 - N}\mathrm{Re}\sum_{n=1}^{N}\sum_{k=n+1}^{N} e^{i\varphi_{n,k}} e^{-i(\theta_n - \theta_k)}, \quad (9)$$

где оператор Re(*) – взятие вещественной части комплексного числа, $\varphi_{n,k}$ – значения фаз комплексных чисел матрицы T. Вектор оптимальных решений θ принимается если значение $\gamma_{PTA} \geq 0.5$ [15]. Значения начальных фаз пикселей каждого из изображений в стеке заменяются соответствующими значениями $\theta = [\theta_1, \theta_2 \ldots \theta_N]^T$ пикселей из DS-наборов. Далее, продолжается стандартная пре- и постобработка для метода постоянных отражателей [2] с измененными интерферометрическими изображениями.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Программная реализация алгоритма построена на базе языка Python, параллельные вычисления – на базе Apache Spark API For Python [10]. Параллелизация строится на том факте, что каждое окно является независимой сущностью. Вычисления над входными данными не используют результаты аналогичных вычислений. Значения переменных и объектов внутри окна не влияют на выполнение вычислений в других окнах сдвига. Количество окон определяется только размером исходных изображений в стеке. Совмещение всех изображений в стеке с субпиксельной точностью гарантирует равный размер и количество окон. Таким образом, можно утверждать, что представленный алгоритм не является итерационным, и может быть реализован в парадигме параллельных вычислений.

Исходный программный код находится в открытом доступе по адресу https://bitbucket.org/ogidog/pysqueesar/src/master/.

Для удобства работы координаты пикселей изображений будем представлять в одномерном виде, т.е. пиксель с координатами (x, y) ↔ xy = yW + x. x ∈ [0,W−1], y ∈ [0, H−1], W и H – ширина и высота изображений в пикселях, соответственно (переменные: width и height).

Создаются следующие программные переменные (табл. 1):

Формируется массив, содержащий координаты центральных пикселей central_pixels, тип int, заполняем массивы i, q. Данная процедура называется **get_input_data()** (см. исходный код). На основе данного массива происходит полный расчет распределенных отражателей и их уточненных значений фаз согласно алгоритму, представленному в разделе “**Математическая модель алгоритма SqueeSAR**”.

Для подключения, управления и инициализации вычислений на кластере применяется технология контекстного запуска независимых разделяемых исполнителей (Executors) Spark Context (объект SparkContext из библиотеки pyspark). SparkContext предоставляет методы соединение с кластером Spark и может использоваться для создания RDD и широковещательных переменных в кластере (рис. 3). Настройка контекста происходит непосредственно в коде и в командной строке запуска программы-драйвера (см. исходный код, файл **main_parallel.py**)

```
#######################
# Spark config and init
#######################

conf = SparkConf()
conf.setMaster("yarn-client")
conf.setAppName("PySqueeSar")
sc = SparkContext(conf=conf)
num_slices = 30

i_broadcast = sc.broadcast(i)
q_broadcast = sc.broadcast(q)
img_height_broadcast = sc.broadcast(lines)
img_width_broadcast = sc.broadcast(samples)
N_broadcast = sc.broadcast(N)
Np_broadcast = sc.broadcast(Np)

############################
# Partitioning central_pixels
############################

central_pixels_slices = np.array_split(central_pixels, num_slices)
central_pixels_slices_broadcast = sc.broadcast(central_pixels_slices)

#################
# Start calc proc
#################

theta_entire = sc.parallelize([*np.arange(num_slices)], numSlices=num_slices)\
     .map(processing_ds).reduce(lambda x, y: x + y)
```

Рис. 3. Фрагмент кода инициализации и запуска расчетного задания на кластере Apache Spark/Yarn.

Для эффективного предоставления каждому узлу копии большого входного набора данных применяются широковещательные переменные (broadcast). Они позволяют хранить данные только для чтения в кэше на каждой машине, а не отправлять их копию вместе с задачами. Spark автоматически передает данные, необходимые для задач на каждом этапе, если эти данные являются глобальными для функции, которая исполняется на map-стадии (рис. 3). Общие данные кэшируются в сериализованной форме и десериализуются перед запуском каждого расчетного задания.

Для создания broadcast-переменных используется метод объекта SparkContext.broadcast() (рис. 3, переменные i_broadcast, q_broadcast, Np_broadcast, N_broadcast, img_width_broadcast, img_height _broadcast для соответствующих переменных из табл. 1). Данные, хранящиеся в перечисленных переменных, являются общими и неизменяемыми на протяжении всех преобразований в алгоритме.

Массив координат центральных пикселей central_pixels разбивается на разделы (slices). Для этого используется функция numpy.array_split() (рис. 3), и создается broadcast-переменная central_pixels_ slices_broadcast размерностью (num_slices, len(central_pixels)// num_slices).

Для работы с данными в среде Apache Spark в параллельном режиме создаются специальные объекты RDD (Resilient Distributed Dataset). RDD – это отказоустойчивый набор элементов, с которыми можно работать параллельно [10]. Формирование RDD происходит путем логического распараллеливания входной коллекции/массива данных в управляющей программе-драйвере при помощи метода SparkContext.parallelize() (рис. 3). При этом на получившихся RDD-наборах возможно использование классических функций map-трансформаций, которые имплементируют алгоритм построения DS-наборов и поиска оптимальных решений (метод processing_ds(), рис. 3,4).

В качестве входного параметра в метод parallelize() передается массив, содержащий индексы разделов (slices) переменной central_pixels_slices_broadcast. Все разделы обрабатываются расчетными заданиями на map-стадии параллельно. В то время как отдельно взятое расчетное задание обрабатывает свой конкретный набор центральных пикселей в разделе последовательно на выделенном заданию ядре, используя метод processing_ds().

Ниже представлена последовательность действий и их описание для процедуры processing_ds.

```
    def processing_ds(slice_number):
1       central_pixels_slice = central_pixels_slices_broadcast
2                           .value[slice_number]
3       theta_slice = []
4       for central_pixel in central_pixels_slice:
5           ds_pixels = np.array(get_ds_within_window(central_pixel))
6           if len(ds_pixels) >= Np_broadcast.value:
7               T = np.zeros((N_broadcast.value, N_broadcast.value))
8               for ds_pixel in ds_pixels:
9                   norm_complex_ds_pixel, norm_hermitian_complex_ds_pixel \
10                      = get_norm_complex_sn(ds_pixel)
11                  T = T + norm_complex_ds_pixel * norm_hermitian_complex_ds_pixel
12              T = T / len(ds_pixels)
13              theta_initial = np.arctan2(q[central_pixel],
14                                         i[central_pixel]).astype("float32")
15              gamma = -linalg.pinv(np.abs(T)) * np.abs(T)
16              sol2 = optimize.minimize(fun=objective, x0=theta_initial,
17                                       args=(np.angle(T), gamma, N),
18                                          method='L-BFGS-B',
19                                          jac=jacobian,
20                                          bounds=optimize.Bounds(-np.pi, np.pi),
21                                          options={'ftol': 1e-3})
22              theta = sol2.x
23              test_result, test_value = pta_test(theta, np.angle(T), N)
24              if test_result:
25                  theta_slice.append([theta, ds_pixels])
26              else:
27                  theta_slice.append([])
28          else:
29              theta_slice.append([])
30      return (theta_slice)
```

Рис. 4. Фрагмент кода метода processing_ds.

1. По индексу раздела из broadcast-переменной central_pixels_slices_broadcast получаем список central_pixels_slice (тип **List**), содержащий координаты центральных пикселей окон сдвига (mxn) для данного раздела (рис. 4, строка 1). Координаты записаны в формате xy = yW + x, как упоминалось выше.

2. Для каждого пикселя в текущем разделе получаем массив (DS-набор, тип List, рис. 4, строка 5), в котором хранятся индексы тех точек серии изображений, которые могут считаться кандидатами в распределенные рассеиватели (2), метод get_ds_within_window() (см. исходный код, файл **main_parallel.py**). В данном методе для каждого пикселя-кандидата выполняется тест Колмогорова–Смирнова, описанный в разделе “**Математическая модель алгоритма SqueeSAR**”.

Используется вспомогательный метод get_sn_vec() (рис. 5) для получения относительных частот по каждому пикселю в окне. В данном методе происходит обращение к broadcast-переменным i_broadcast и q_broadcast (рис. 5, строки 1,2), расчет амплитуд, их частотного распределения и кумулятивного накопления (рис. 5, строки 5–9).

Далее, в методе get_ds_within_window() для каждого пикселя в окне рассчитывается модуль разности соответствующих относительных частот и его наибольшее значение Dmax (рис. 6, строка 13), вычисляется эмпирическое значение критерия lambda_emp (рис. 6, строка 14). Если эмпирическое значение критерия lambda_emp меньше критического lambda_crit, то индекс текущего пикселя добавляется в массив window_ds_pixels (рис. 6, строка 15, 16). После того как найдены все пиксели-кандидаты в распределенные отражатели в окне сдвига необходимо отсеять те, которые не являются смежными с центральным.

3. Алгоритм поиска смежных пикселей строится на базе k-мерного дерева, используя объект scipy.spatial.cKDTree, которому передается массив window_ds_pixels (рис. 7, строка 21). Метод query_ball_point() объекта cKDTree (рис. 7, строка 32), позволяет находить ближайшие соседние точки к заданной с координатами (x,y), находящиеся на определенном расстоянии в декартовой системе координат. Данный метод возвращает список координат (x,y) всех соседних элементов, принадлежащих k-мерному дереву. Примем расстояние (радиус) D равным $\sqrt{2}$, т.е. значения координат (x, y) смежных пикселей отличаются ровно на 1.

Создается массив window_neighbors, тип List, с начальным элементов – индексом центральной

```
    def get_sn_vec(pixel):
1       pixel_i = i_broadcast.value[pixel]
2       pixel_q = q_broadcast.value[pixel]
3       amplitude_vec = []
4       for i in [*range(N_broadcast.value)]:
5           amplitude_vec.append(np.sqrt(pixel_i[i] ** 2 + pixel_q[i] ** 2))
6
7       sort_ampl = np.sort(amplitude_vec)
8       hist, bin_edges = np.histogram(sort_ampl, bins=8)
9       sn_vec_p = np.cumsum(hist / N_broadcast.value)
10
11      return sn_vec_p
```

Рис. 5. Метод get_sn_vec().

```
   def get_ds_within_window(central_pixel):
1       lambda_crit = 0.75
2
3       sn_vec_p0 = get_sn_vec(central_pixel)
4       center_y, center_x = central_pixel // img_width_broadcast.value, \
5                            central_pixel % img_width_broadcast.value
6
7       window_ds_pixels = []
8       for i in range(-6, 7):
9           for j in range(-8, 9):
10              pixel_pk = (center_y + i) * img_width_broadcast.value
11                       + (center_x + j)
12              sn_vec_pk = get_sn_vec(pixel_pk)
13              Dmax = np.max(np.abs(np.array(sn_vec_p0) - np.array(sn_vec_pk)))
14              lambda_emp = Dmax * np.sqrt(N_broadcast.value / 2)
15              if lambda_emp < lambda_crit:
16                  window_ds_pixels.append([center_y + i, center_x + j])
17
18      ......
19      ......
```

Рис. 6. Фрагмент метода get_ds_within_window().

точки. Он будет содержат все индексы смежных пикселей, после завершения работы метода. Так как метод query_ball_point() принимает координаты точек в формате (x,y), то на каждом шаге итерации будем преобразовывать индексы элементов массива window_ds_pixels в такой формат (рис. 7, строки 28, 29).

Создается временный массив tmp_point_neighbors (рис. 7, строка 26), который на каждой итерации для каждой точки из window_ds_pixels будет заполняться смежными с ней. Создается массив point_neighbors. Он будет содержать индексы точек, которых еще нет в основном массиве window_neighbors, и, соответственно, дополнять основной массив ими.

На первой итерации выполняется метод query_ball_point() для центрального пикселя в массиве point_neighbors. Получается модифицированный массив tmp_point_neighbors, содержащий не только координаты центральной точки, но и координаты смежных пикселей, находящихся на расстоянии $\sqrt{2}$ от нее (рис. 7, строки 30–33). Ищется различие массива tmp_point_neighbors с основным window_neighbors и записывается в массив point_neighbors (рис. 7, строки 35–38). Соответственно, расширяется массив window_neighbors (рис. 7, строки 39).

Следующая итерация начинается с проверки длины массива point_neighbors. Если длина массива больше 0, т.е. на предыдущей итерации были найдены смежные пиксели, отличные от уже содержащихся в массиве window_neighbors по значениям индексов, то выше описанная процедура повторяется. Другими словами, область поиска в окне сдвига расширяется на расстояние $\sqrt{2}$ от центральной (рис. 8).

В конечном итоге данный итерационный метод позволит перебрать каждый пиксель-кандидат из массива window_ds_pixels, достигнув самых отдаленных пикселей от центрального. То есть, метод numpy.setdiff1d() (рис. 7, строка 35) вернет пустой массив point_neighbors, так как вновь найденные смежные точки уже будут содержаться в

```
    def get_ds_within_window(central_pixel):
18      ......
19      ......
20
21      window_neighbors = [center_y * img_width_broadcast.value + center_x]
22
23      tree = cKDTree(np.array(window_ds_pixels))
24      point_neighbors = [center_y * img_width_broadcast.value + center_x]
25      while len(point_neighbors) > 0:
26          tmp_point_neighbors = np.array([[center_y, center_x]])
27          for point in point_neighbors:
28              point = [point // img_width_broadcast.value,
29                       point % img_width_broadcast.value]
30              tmp_point_neighbors = np.concatenate(
31                  (tmp_point_neighbors, np.array(tree.data[
32                      tree.query_ball_point(point, r=np.sqrt(2))
33                  ]))).astype('int32')
34
35          point_neighbors = np.setdiff1d(
36              [tmp_point_neighbor[0] * img_width_broadcast.value
37                                     + tmp_point_neighbor[1]
38              for tmp_point_neighbor in tmp_point_neighbors], window_neighbors)
39          window_neighbors.extend(point_neighbors)
40
41      return window_neighbors
```

Рис. 7. Фрагмент метода get_ds_within_window().

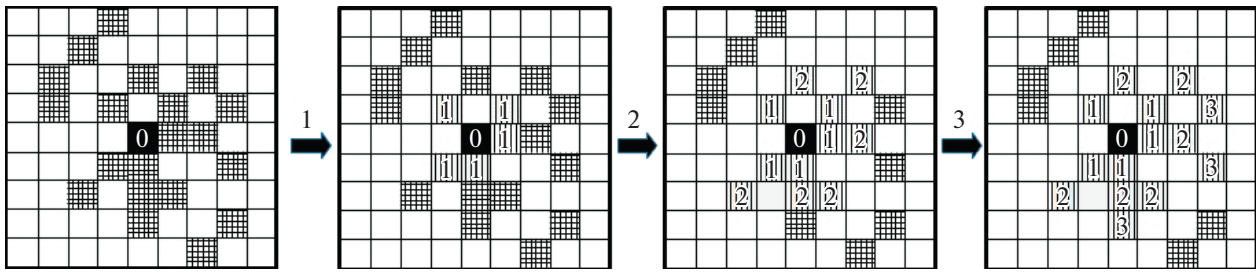


Рис. 8. Схем поиска смежных пикселей. Цифрой обозначен номер итерации и, соответственно, пикселы, найденные как смежные с центральным на текущей итерации.

основном массиве window_neighbors. Итерации прекращаются.

4. Если количество смежных пикселей в сформированном массиве ds_pixels (рис. 4, строка 5) больше заданной переменной Np_broadcast (не менее 20), выполняем расчет уточненных значений вектора фаз theta.

Рассчитываются нормированные комплексные значения (3) ($\widehat{p_i}$ и $\hat{p}_i^+$) каждого пикселя из массива ds_pixels, метод get_norm_complex_sn() (см. исходный код, файл main_parallel.py). Рассчитывается матрица когеренции T (3) (рис. 4, строки 11–12, тип complex). Согласно математической модели алгоритма задается начальное значение для theta, переменная theta_initial (рис. 4, строки 13).

5. Для оценки значений интерферометрических фаз каждого центрального пикселя из central_pixels_slices_broadcast для соответствующего DS-набора ds_pixels используется объект **scipy.optimize**. Объект **optimize** не содержит методов, позволяющих решать задачу максимизации. Поэтому применяется метод **minimize()** (рис. 4, строки 16–21) для решения задач минимизации. Другими словами, мы минимизируем функцию $-F_{max}$ из (8) (параметр fun) для всех θ (theta), при заданном ограничении $-\pi \le \theta \le \pi$ (параметр **bounds**). Параметр **method** задает тип решателя для проблемы минимизации. В данном случае модификация метода BFGS с возможностью задания нижней и верхней границ поиска решения. Параметр x_0 – вектор начального приближения, равен theta_initial. В предположении того, что изначальные фазы уже являются оптимальными для достаточно

```
def pta_test(theta, phase, N):
    s = 0
    for n in range(N):
        for k in range(n + 1, N):
            s = s + np.exp(1j * phase[n, k]) * \
                    np.exp(-1j * (theta[n] - theta[k]))
    test_value = 2 * np.real(s) / (N ** 2 - N)
    if test_value > 0.5:
        return True, test_value
    else:
        return False, test_value
```

Рис. 9. Метод pta_test().

большого значения отношения сигнал/шум в стеке интерферометрических изображении [1, 15]. Параметром jac_fun задается якобиан (метод jacobian(), см. исходный код) для функции параметра fun (метод objective(), см. исходный код). Данные методы принимают на вход искомое значение (приближение на каждой итерации метода minimize()), а на выходе возвращают вычисленное значение согласно выражению внутри этих методов. То есть, это значение функции $-F_{max}$ (8) и ее якобиана (см. исходный код), вычисленные при заданных параметрах (args) равных numpy.angle(T) и gamma (рис. 4, строки 15, 16) и векторе искомых значений theta на каждой итерации.

6. Качество полученной оптимизации theta оценивается на основе специального теста (метод pta_test()) (рис. 9).

7. Если массив theta прошел тест, то все его значения в текущем окне сдвига добавляются в специально созданный массив theta_slice (тип List) вместе с индексами смежных пикселей (массив ds_pixels). Если нет, то в массив theta_slice добавляется пустой массив, ровно как и в случае длины массивы ds_pixel меньше чем заданное пороговое значение переменной Np_broadcast (рис. 4, строки 24–29).

Массив theta_slice содержит все результаты работы одного задания в среде Apache Spark, соответствующие текущему разделу.

Для объединения результатов в единый массив theta_entire используется метод объекта pyspark.RDD reduce() (рис. 3, строки 29, 30). Массив theta_entire сохраняется в бинарный файл в программе-драйвере (см. исходный код, файл main_parallel.py). В дальнейшем из этого файла считываются значения theta всех центральных пикселей и индексы смежных точек ds_pixels. Значения в каналах I и Q исходных снимков заменяются соответствующими расчетными значениями. При этом вещественная часть (полоса I) будет равна cos(theta), а мнимая (полоса Q) – sin(theta) (см. исходный код, файл main.py, метод modify_images()).

Ниже приведена диаграмма потоков данных (DFD) при выполнении программного кода алгоритма поиска распределенных рассеивателей (рис. 10).

4. ТЕСТ ПРОИЗВОДИТЕЛЬНОСТИ

Запуск программы-драйвера (см. исходный код, файл main_parallel.py), содержащего код (рис. 3) выполняется при помощи скрипты **spark-submit** [10] (рис. 11).

Параметр spark.driver.memory задает количество оперативной памяти, выделяемое для программы драйвера на управляющем узле Spark/Yarn (Master Node). Данный параметр влияет на объем предварительно подготавливаемых данных перед непосредственно их копированием на исполняющие узлы кластера. Устанавливать значение параметра необходимо с учетом суммарного объема всех переменных определяемых вне метода map(). Это переменные i, q, lines, samples, N, Np, central_pixels, central_pixels_slices (рис. 3).

Параметр spark.executor.memory задает количество оперативной памяти, выделяемое для каждого контейнера-исполнителя (Executor). При этом на одном узле-исполнителе (Worker Node) менеджером ресурсов Yarn может быть создано несколько контейнеров-исполнителей. Следовательно, общее количество выделяемой оперативной памяти на узле может быть увеличено на величину равной spark.executor.memory* (кол-во контейнеров-исполнителей). Также, данный параметр должен коррелировать с количеством создаваемых broadcast-переменных, т.к. каждая переменная копируется для каждого контейнера-исполнителя [10]. Это переменные i_broadcast, q_broadcast, img_height_broadcast, img_width_broadcast, N_broadcast, Np_broadcast, central_pixels_slices_broadcast (рис. 3).

Параметр spark.executor.instances задает общее количество контейнеров-исполнителей. При выборе значения данного параметра необходимо учитывать общий объем оперативной памяти, необходимой для всех broadcast-переменных. Непосредственно каждый физический узел должен об-

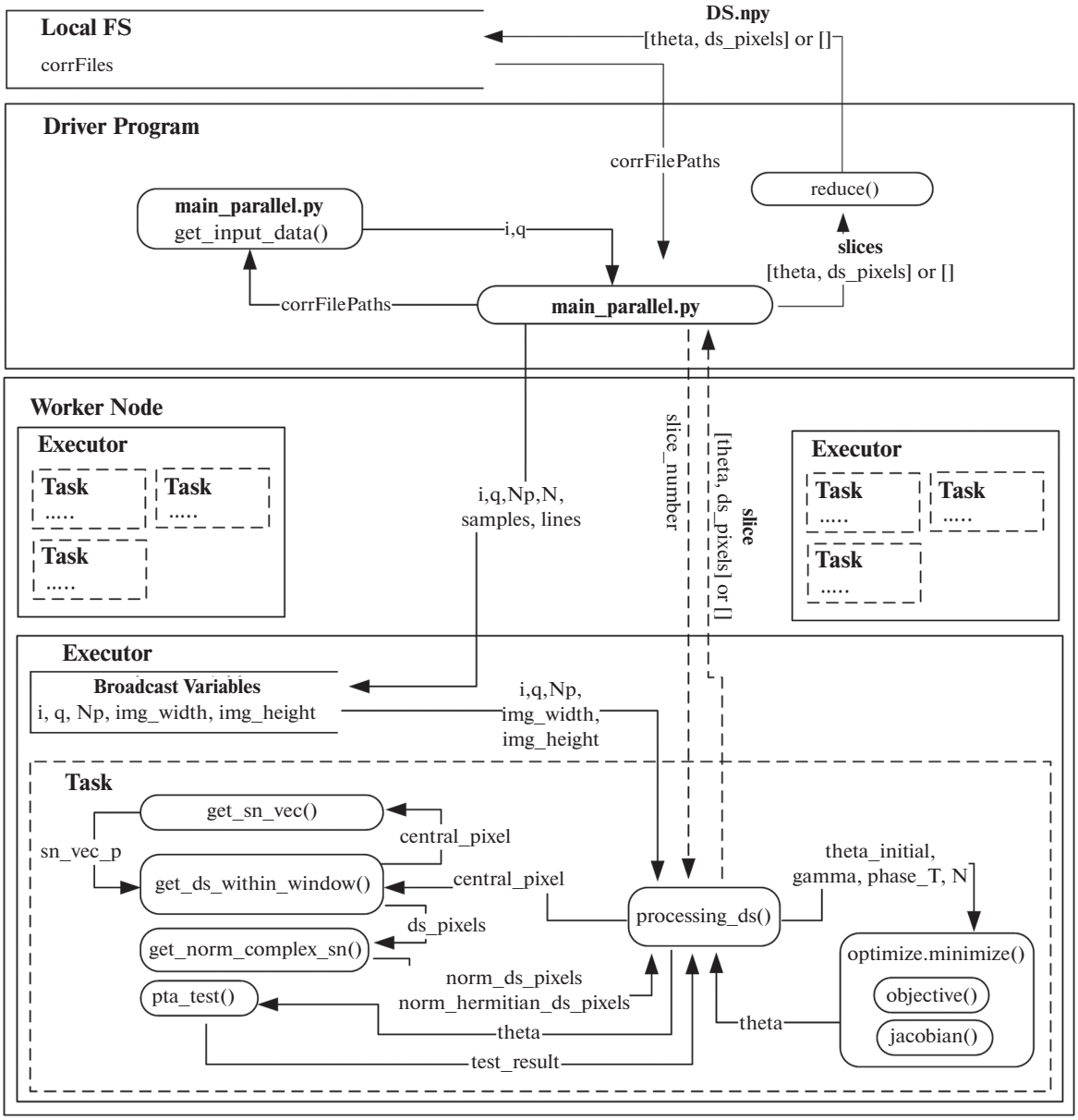


Рис. 10. DFD-диаграмма выполнения программного кода алгоритма.

```
spark-submit
    --conf spark.driver.memory=8g
    --conf spark.executor.memory=8g
    --conf spark.executor.instances=6
    --conf spark.executor.cores=5
    --master yarn-client
        main_parallel.py /path_to_input_dir /path_to_output_dir
```

Рис. 11. Команда запуска расчетного задания на кластере Apache Spark/Yarn.

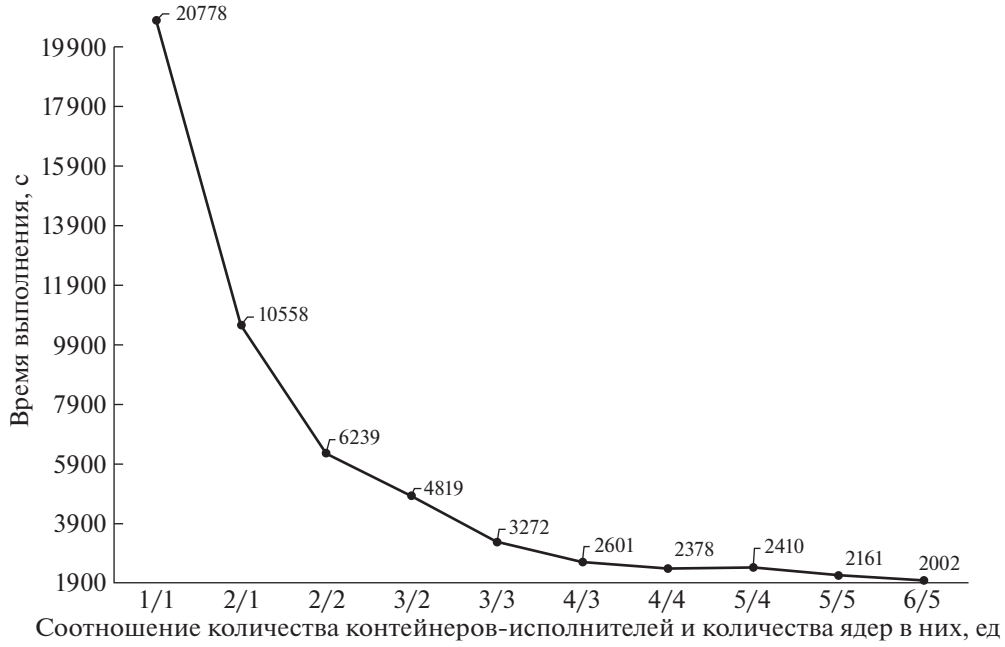


Рис. 12. График зависимости времени выполнения расчета от количества контейнеров-исполнителей и количества ядер процессора на один исполнитель для всей области снимка.

ладать памятью способной разместить объем равный spark.executor.instances *(суммарный размер в байтах broadcast-переменных).

Параметр spark.executor.cores задает количество ядер процессора на один исполнитель. При этом параметр неявно задает количество заданий Task, которое сможет выполнить исполнитель за один запуск. То есть в контексте алгоритма – это будет количество разделов (slices), для которых ведется расчет всех центральных точек в текущем разделе (рис. 10).

Тест производительности проводился для области снимка размером 6000 × 4000 пикселей, стек из 50 снимков (N = 60), на кластере, состоящем из 3 узлов со следующими аппаратными характеристиками: 3 сервера (AMD Ryzen 1700 (8 + 8 cores (Simultaneous Multi-Threading)) 3.2 GHz, 32 Gb RAM, 1 Gb/s скорость передачи данных между серверами). Один узел выступал в роли управляющего, два других – в роли узлов-исполнителей. Именно на них проводился расчет. В скрипте запуска spark-submit менялись значение параметров spark.executor.instances и spark.executor.cores. Остальные параметры были зафиксированы со значениями: spark.driver.memory=8g и spark.executor.memory=8g (рис. 11).

Ниже праведен график времени выполнения алгоритма с учетом вариативности вышеописанных параметров (рис. 12).

5. ПРИМЕНЕНИЕ АЛГОРИТМА SqueeSAR

Расчет скоростей смещений для тестовых радарных данных проводился в открытом пакете библиотек StamPS [16]. В стандартную схему пред- и постобработки был добавлен дополнительный шаг “SqueeSAR” (рис. 13), проводящий расчет DS-наборов и заменяющий исходные значения интерферометрических фаз их оптимальными значениями. Исходными данными для полного цикла расчета скоростей смещений методом постоянных рассеивателей (PS) послужили спутниковые радарные снимки аппарата Sentinel-1B. Для дифференциальной интерферометрии использовался канал типа IW (Interferometric Wide swath) с вертикальной поляризацией VV. Территория охвата – г. Норильск, и прилегающие территории (ТЭЦ-3), Россия. Данные были получены с открытого ресурса в сети Интернет “Copernicus Open Access Hub” (URL: https://scihub.copernicus.eu/dhus/#/home) Европейского космического агентства (URL: https://sentinel.esa.int/web/sentinel/home). Всего получено 60 снимков временной серии с февраля 2019 г. до апреля 2020 г. Модифицированная схема полного цикла расчета скоростей смещений представлена на рис. 13. Полное описание методов схемы приводится в [17, 18].

Рассматривалась географическая область, где 29 мая на территории ТЭЦ-3 в Норильске произошла утечка из резервуара, в котором было около 21 тысячи тонн дизельного топлива. На рис. 14 приведены расчеты смещений для аварийных ба-

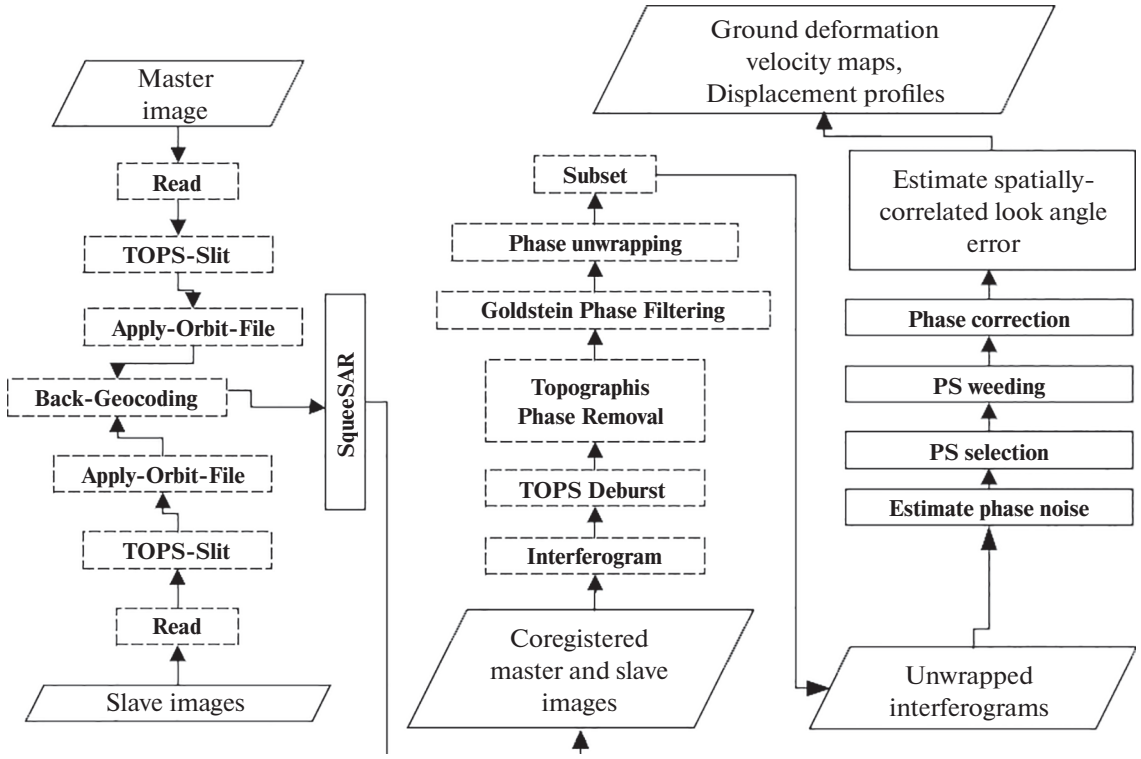


Рис. 13. Модифицированная схема полного цикла расчета скоростей смещений методом постоянных рассеивателей.

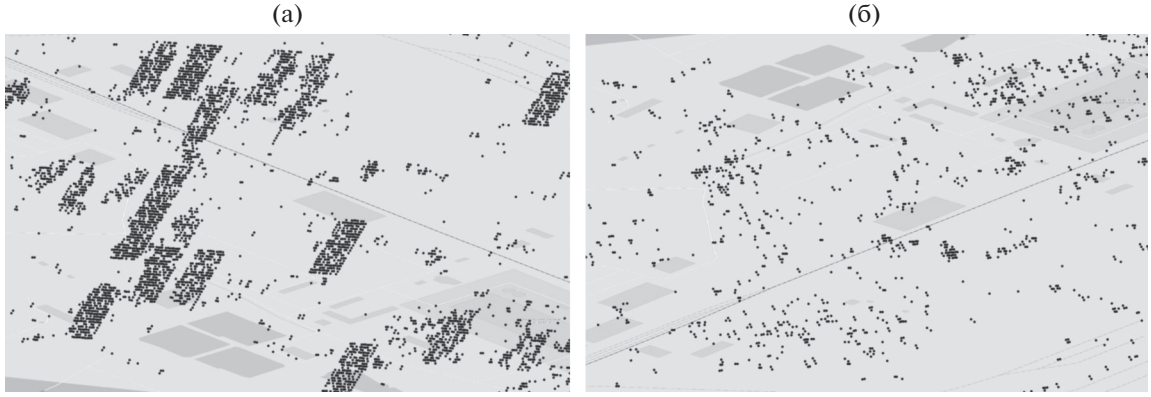


Рис. 14. Результат расчета полной схемы скоростей смещений с интегрированным алгоритмом SqueeSAR а) и без б).

ков. Показано облако точек (постоянных рассеивателей) с применением метода SqueeSAR и без него. Общее количество точек для расчетной области возросло в среднем до 130000 точек с интегрированным методом SqueeSAR, против 53000 в стандартном выполнении схемы.

Ниже приведен фрагмент области снимка для визуального сравнения облака постоянных рассеивателей с представленным методом и без него (рис. 14).

6. ЗАКЛЮЧЕНИЕ

Разработан быстрый алгоритм поиска распределенных отражателей для задачи построения скоростей смещений методом постоянных рассеивателей. Предложенный алгоритм позволяет увеличить количество искомых рассеивателей в два-три раза по сравнению с традиционным алгоритмом. Примеры расчетов на спутниковых радарных данных Sentinel-1A/B показывают увеличенную плотность и компактность точек расположения в местах возвышенностей и средних

построек (от 10 м). Это позволяет точнее строить карты цифровых моделей рельефа на базе временных серий снимков.

Показано, что предложенный алгоритм не является итерационным, и может быть реализован в парадигме параллельных вычислений. Применяемая платформа параллельной обработки Apache Spark позволила распределено обрабатывать массивы стека радарных данных (от 60 изображений) в памяти, на большом количестве физических узлов в сетевой среде. При этом время поиска распределенных рассеивателей удалось снизить в среднем в 10 раз по сравнению с однопроцессорной реализацией алгоритма.

Предложенный алгоритм и его параллельная имплементация позволят применять разработанные подходы в других задачах и типах спутниковых данных дистанционного зондирования земли из космоса.

7. БЛАГОДАРНОСТИ

Исследование выполнено при финансовой поддержке РФФИ и Кемеровской области в рамках научного проекта № 20-47-420002 р_а.

СПИСОК ЛИТЕРАТУРЫ

1. *Ferretti A., Prati C., Rocca F., Wasowski J.* Satellite interferometry for monitoring ground deformations in the urban environment // Proc. of the 10th Congress of the International Association for Engineering Geology and the Environment (IAEG). 2006. P. 1–4.

2. *Crosetto M., Monserrat O., Cuevas-González M., Devanthéry N., Crippa B.* Persistent Scatterer Interferometry: A review // ISPRS Journal of Photogrammetry and Remote Sensing. 2016. V. 115. P. 78–89.

3. *Perissin D., Ferretti A.* Urban target recognition by means of repeated spaceborne SAR images. IEEE Trans. Geosci. Remote Sens. 2007. V. 45. № 12. P. 4043–4058.

4. *Hooper A.*A multi-temporal InSAR method incorporating both persistent scatterer and small baseline approaches // Geophysics Research Letter. 2008. V. 35. P. 302

5. *Rocca F.* Modeling interferogram stacks // IEEE Transactions on Geoscience and Remote Sensing. 2007. V. 45. № 10. P. 3289–3299.

6. *Zebker H.A., Shanker A.P.* Geodetic imaging with time series persistent scatterer InSAR // American Geophysical Union all Meeting Abstracts. San Francisco. CA. 2008.

7. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* Exploitation of distributed scatterers in interferometric data stacks // International Geoscience Remote Sensing Symposium (IGARSS). Cape Town. South Africa. 2009.

8. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* The second generation PSInSAR approach: SqueeSAR. International Workshop ERS SAR Interferometry (FRINGE). Frascati. Italy. 2009.

9. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* A New Algorithm for Processing Interferometric Data-Stacks: SqueeSAR // IEEE Transactions on Geoscience and Remote Sensing. 2011. V. 49. P. 3460–3470.

10. Apache Spark Documentation // Apache Spark. https://spark.apache.org/docs/latest/index.html (дата обращения: 12.09.2020)

11. *Феоктистов А.А., Захаров А.И., Денисов П.В., Гусев М.А.* Исследование зависимости результатов обработки радиолокационных данных ДЗЗ от параметров обработки. Часть 4. Основные направления развития метода постоянных рассеивателей; ключевые моменты методов SQUEESAR И STAMPS // Журнал радиоэлектроники [электронный журнал]. 2017. № 7. http://jre.cplire.ru/jre/jul17/5/text.pdf

12. *Cao N., Lee H., Chul Jung H.* Mathematical Framework for Phase-Triangulation Algorithms in Distributed-Scatterer Interferometry // IEEE Geoscience and remote sensing letters. V. 12. № 9. P. 1838–1842.

13. *Guarnieri A.M., Tebaldini S.* On the exploitation of target statistics for SAR interferometry applications // IEEE Transactions on Geoscience and Remote Sensing. 2008. V. 46. № 11. P. 3436–3443.

14. *Bamlery R., Hartlz P.* Synthetic aperture radar interferometry // Inverse Problems. 1998. V. 14. № 4. P. R1–R54.

15. *Shamshiri R., Nahavandchi H., Motagh M., Hooper A.* Efficient Ground Surface Displacement Monitoring Using Sentinel-1 Data: Integrating Distributed Scatterers (DS) Identified Using Two-Sample t-Test with Persistent Scatterers (PS) // Remote Sensing. 2018. V. 10. № 5. P. 794–808.

16. STAMPS // A software package to extract ground displacements from time series of synthetic aperture radar (SAR) acquisitions. https://homepages.see.leeds.ac.uk/~earahoo/stamps/ (дата обращения: 12.09.2020)

17. *Потапов В.П., Попов С.Е., Костылев М.А.* Информационно-вычислительная система массивно-параллельной обработки радарных данных в среде Apache Spark // Вычислительные технологии. 2018. Т. 23. № 4. С. 110–123.

18. *Попов С.Е., Замараев Р.Ю., Миков Л.С.* Массово-параллельный подход к обработке радарных данных // Современные проблемы дистанционного зондирования Земли из космоса. 2020. Т. 17. № 2. С. 49–61.

ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 30–50

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.421.6

МОДЕЛИ ПАМЯТИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ: ОБЗОР И ТЕНДЕНЦИИ

© 2021 г. Е. А. Моисеенко[a,c,*], А. В. Подкопаев[b,c,], Д. В. Кознов[a,***]**

[a] *Санкт-Петербургский государственный университет*
198504 Санкт-Петербург, Петергоф, Университетский пр. 28, Россия
[b] *Национальный исследовательский университет "Высшая школа экономики"*
194100 Санкт-Петербург, Кантемировская ул. 3b. 1, Россия
[c] *JetBrains Research 197342 Санкт-Петербург, Кантемировская ул. 2, Россия*

**E-mail: e.moiseenko@2012.spbu.ru*

***E-mail: apodkopaev@hse.ru*

****E-mail: d.koznov@spbu.ru*

Поступила в редакцию 17.05.2021 г.
После доработки 13.06.2021 г.
Принята к публикации 13.07.2021 г.

Модель памяти языка программирования определяет семантику многопоточных программ, создаваемых на этом языке и оперирующих с разделяемой памятью. Наиболее известна модель последовательной согласованности, которая является слишком строгой, запрещая многие сценарии поведения, наблюдаемые при исполнении программ на современных процессорах. Попытки формально описать эти сценарии привели к возникновению так называемых слабых моделей памяти. В последние годы было предложено значительное количество слабых моделей памяти для различных языков программирования. Эти модели предлагают различные компромиссы относительно простоты/сложности рассуждений о поведении многопоточных программ и возможностей их оптимизации. Цель данной статьи заключается в обзоре существующих моделей памяти языков программирования и выработке общих рекомендаций по выбору/созданию модели памяти при создании/стандартизации языков программирования, а также при разработке компиляторов. Для данного обзора мы рассмотрели более 2000 статей, найденных по ключевым словам "Relaxed Memory Models", "Weak Memory Models", и "Weak Memory Consistency" поисковой системой Google Scholar. Используя разные критерии, мы сузили это множество до 40 статей, предлагающих и описывающих модели памяти языков программирования. Мы разделили эти модели на шесть классов и проанализировали их свойства и ограничения. В заключение мы показали, как дизайн языка программирования влияет на выбор модели памяти и обсудили возможные направления дальнейших исследований в этой области.

DOI: 10.31857/S0132347421060054

1. ВВЕДЕНИЕ

Многопоточное программирование активно используется в современном программировании, позволяя достичь существенного выигрыша в производительности системного программного обеспечения – ядер операционных систем, СУБД, высоконагруженных клиент-серверных приложений и пр. Главная трудность многопоточного программирования заключается в необходимости обеспечить корректную синхронизацию между различными потоками программы. Обычно это достигается при помощи специальных примитивов синхронизации, таких как блокировки, барьеры, каналы и т.д. Однако часто использование этих примитивов не позволяет достичь нужной производительности. Распространенным примером являются различные неблокирующие (lock-free) структуры данных. В подобных случаях необходимо обращаться к средствам низкоуровневого программирования и взаимодействовать с разделяемой потоками памятью напрямую. Здесь начинаются сложности.

Рассмотрим конкретный пример. Ниже представлена упрощенная версия алгоритма блокировки Деккера [1]:

$x := 1$	$y := 1$
$r_1 := y$	$r_2 := x$
if $r_1 := 0$ {	if $r_2 := 0$ {
//critical section	*//critical section*
}	}

(Dekker's Lock)

В этой программе два потока соревнуются за доступ к критической секции. Чтобы обозначить свое намерение войти в критическую секцию, потоки устанавливают значение переменных x и y соответственно[1]. Поток, который первым установит значение переменной и прочитает значение другой переменной до его установки, получает право войти в критическую секцию. Алгоритм полагается на тот факт, что оба потока не могут одновременно прочитать значение 0[2]. В противном случае два потока способны одновременно войти в критическую секцию, таким образом нарушая корректность алгоритма.

Естественно ожидать, что эта программа завершится с одним из следующих результатов: $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$ или $[r_1 = 1, r_2 = 1]$. Соответствующие сценарии поведения называются *последовательно согласованными* (*sequentially consistent*) [2], и это означает, что они могут быть получены в результате поочередного последовательного исполнения инструкций потоков.

Тем не менее не все сценарии поведения, наблюдаемые в многопоточных системах, являются последовательно согласованными. Например, если перевести псевдокод алгоритма Деккера на язык C, выполнить компиляцию полученной программы с помощью компилятора GCC и запустить получившийся код на процессорах семейства x86/x64, то можно наблюдать, помимо перечисленных выше результатов, еще и результат $[r_1 = 0, r_2 = 0]$. Сценарий поведения, который приводит к этому результату, является примером слабого (*weak*) поведения.

Слабые сценарии поведения появляются в результате оптимизаций программ компиляторами и процессорами. Например, рассматривая программу Dekker's Lock, оптимизатор может заметить, что запись в переменную x и чтение из y в левом потоке являются независимыми инструкциями и, следовательно, могут быть переупорядочены (заметим, что эта оптимизация является корректной для случая однопоточных программ). Для оптимизированной программы поведение с результатом $[r_1 = 0, r_2 = 0]$ является последовательно согласованным.

Множество допустимых сценариев поведения программы определяется семантикой многопоточной системы или *моделью памяти*. Модель *последовательной согласованности* (*sequential consistency*, SC) допускает только последовательно согласованные сценарии поведения. Модели памяти, которые допускают слабые сценарии поведения, называются, соответственно, *слабыми моделями памяти* (*weak memory models*).

Современные процессоры и языки программирования не ограничиваются моделью последовательной согласованности, так как она не позволяет многие важные оптимизации. Таким образом, важно понимать, на сколько слабой должна быть модель памяти. Более строгая модель допускает меньше сценариев поведения и предоставляет больше гарантии программисту. С другой стороны, более слабая модель позволяет выполнять большее количество оптимизаций, что повышает производительность программы.

Оказывается, что этот вопрос весьма сложен. Это привело к тому, что за последние 20 лет было предложено множество моделей для различных языков программирования, например, для Java [3, 4], C/C++ [5], LLVM [6], JavaScript [7], OCaml [3], Haskell [8] и т.д. Эти модели преследуют разные цели, делают различные компромиссы и имеют разнообразные ограничения. Более того, в данной области продолжают появляться новые исследования. По нашим подсчетам, за последние 10 лет в течение каждого года было опубликовано не менее 50 статей по этой тематике[3]. Тем не менее, несмотря на долгую историю и существенный прогресс, нет единого источника, который бы суммировал известную информацию и сравнивал существующие модели памяти различных языков программирования. Целью данной статьи является создание такого обзора.

Мы рассматриваем существующие модели памяти языков программирования, обсуждаем их дизайн, компромиссы и ограничения. Также мы сравниваем эти модели на предмет того, какие оптимизации они поддерживают и какие гарантии предоставляют программистам.

Мы надеемся, что наша работа будет полезна для исследователей в области языков программирования, желающих ознакомиться с темой слабых моделей памяти, а также для разработчиков компиляторов и виртуальных машин, которым необходимо выбрать модель памяти для их системы.

Данная статья организована следующим образом. В § 2 представлен обзор литературы. Далее мы описываем методологию нашего исследования § 3. Затем мы вводим критерии сравнения моделей памяти § 4, в частности, оптимальность схем компиляции, корректность преобразований программ и предоставляемые гарантии для рассуждения о поведении программ. Далее мы описываем способ, которым мы сравниваем модели § 5. В § 6 мы классифицируем модели на основе

[1] В этой статье переменные, разделяемые разными потоками программы, мы обозначаем следующим образом – x, y, z ..., а локальные переменные потока – r_1, r_2, r_3...

[2] Здесь и далее мы подразумеваем, что исходно переменные инициализированы значением 0, если иное не указано явно.

[3] Эти подсчеты подкреплены данными, полученными с помощью поисковой системы Google Scholar, подробности представлены 3.

их свойств и обсуждаем каждый класс в отдельности. В § 7 мы представляем набор рекомендации по выбору модели памяти на основе требований к языку программирования и рассматриваем их на примере языка Kotlin[4]. Наконец, в § 8 мы подводим итоги и обсуждаем возможные направления дальнейших исследований.

2. ОБЗОР ЛИТЕРАТУРЫ

Слабые модели памяти могут быть разделены на два класса: для архитектур процессоров и для языков программирования. Основная разница между ними заключается в том, что модели языков программирования должны поддерживать широкий класс оптимизаций, выполняемых компиляторами.

На сегодняшний день модели архитектур процессоров относительно хорошо изучены. Все основные архитектуры имеют формально определенные модели: x86 [9], IBM **POWER** [10–12], **Arm** [10, 12–15] и **RISC-V** [14]. Процессоры семейств x86 и **POWER** обладают стабильными моделями, которые не претерпевали изменений в течение последних нескольких лет, в то время как модель памяти процессоров семейства **Arm** существенно изменилась при переходе от **Armv7** [12] к **Armv8** [14] и была дополнена поддержкой новых инструкций для обращения к разделяемой памяти. Архитектура **RISC-V**[5] была представлена в 2010 году и недавно к ней была адаптирована модель [14], почти идентичная модели **Armv8**.

Все вышеупомянутые модели формализованы в *декларативном* (или *аксиоматическом*) стиле, ставшем стандартом для спецификации слабых моделей памяти и поддерживающим различные инструменты для тестирования и верификации моделей [12]. Модели некоторых языков программирования также используют декларативный стиль, например, C/C++ [5], JavaScript [7], Java [3] и другие. Тем не менее, декларативные модели не позволяют решить некоторые проблемы, характерные для моделей языков программирования, таких как C/C++, призванных обеспечить эффективную компиляцию в код целевой архитектуры и поддержку оптимизаций, которые потенциально могут удалять *синтаксические зависимости* между инструкциями (например, распространение констант).

Проблема заключается в том, что декларативные модели не способны отличить истинные *семантические зависимости* между инструкциями от ложных. Например, в примере ниже существует зависимость между инструкциями в левом потоке, в то время как зависимость между инструкциями в правом потоке является нет:

$$\begin{array}{l|l} r := x & r := x \\ y := r & y := r * 0 \end{array}$$

Для моделей памяти архитектур процессоров нет необходимости в таком различении, так как они сохраняют все синтаксические зависимости инструкций, в то время как оптимизирующий компилятор может, например, заменить выражение $r * 0$ на 0 и таким образом удалить соотвествующую зависимость. Таким образом, необходимо либо различать истинные и ложные зависимости, либо полностью игнорировать информацию о зависимостях (текущая формализация модели C/C++ использует второй подход). Но если игнорировать информацию о зависимостях в комбинации с поддержкой *буферизации операций чтения*, допустимых спецификацией процессоров **Arm** и **POWER**, то возникают так называемые *значения из воздуха* [16], нарушающие базовые гарантии поведения программ (эта проблема подробно обсуждается в § 4.3.4).

Для решения данной проблемы с сохранением поддержки различных оптимизаций было предложено множество моделей, использующих различные подходы: Java Memory Model (JMM) [3], Promising semantics [17, 18], Weakestmo [19], Modular Relaxed Dependencies (MRD) [20]. Некоторые другие модели, например, **RC11** [21] и модель памяти языка OCaml [22] избегают проблемы путем запрещения некоторых оптимизаций и предоставления больших гарантий [23].

Несмотря на то, что на сегодняшний день было предложено множество моделей языков программирования, которые делают различные компромиссы и поддерживают различные возможности, насколько нам известно, не существует детального обзора этих моделей. Этот факт мотивировал нашу работу над данной статьей.

3. МЕТОДОЛОГИЯ

Основной задачей нашей работы было изучение компромиссов в моделях памяти языков программирования. Мы хотим ответить на следующий вопрос.

• Как гарантии корректного поведения программ, предоставляемые моделью памяти, ограничивают возможности по оптимизации этих программ?

Для сравнения моделей в свете данного вопроса мы использовали стандартные критерии, встречающиеся в литературе.

C.1 *Оптимальность схемы компиляции.* Язык с моделью памяти, поддерживающей оптимальные схемы компиляции, может быть эффективно реализован на современных процессорах. Напротив, использование неоптимальных схем компиляции приводит к замедлению при исполнении програм-

[4] https://kotlinlang.org/
[5] https://riscv.org/

мы, но в то же время может предотвратить появление слабых сценариев поведения, допустимых спецификацией данной архитектуры.

C.2 *Корректность трансформаций над программным кодом*. При оптимизации компилятор применяет различные трансформации к исходному коду. Чем больше транформаций допускается моделью памяти языка программирования, тем больше оптимизаций потенциально может применить компилятор к программам на данном языке.

C.3 Наличие различных *гарантией для рассуждения о поведении программ* позволяет упростить выполнение доказательств корректности многопоточных программ на данном языке.

Чтобы отобрать модели памяти для нашего исследования, мы провели следующую процедуру поиска. На *первом этапе* мы вручную отобрали 10 рецензированных статей предлагающих новые модели памяти [3–5, 7, 17, 19–22, 24, 25], которые были представлены на высоко рейтинговых конференциях в области языков программирования, таких как “Symposium on Principles of Programming Languages” (POPL), “Conference on Programming Language Design and Implementation” (PLDI) и др. Затем мы взяли список ключевых слов из этих статей. Мы исключили ключевые слова, которые были слишком общими или, наоборот, слишком специфичными. В результате мы получили три ключевые фразы: Relaxed Memory Models, Weak Memory Models, Weak Memory Consistency.

На *втором этапе* мы использовали эти фразы в качестве поисковых запросов в Google Scholar[6]. По каждому запросу мы взяли первые 1000 результатов.[7] В итоге мы получили список из 2493 статей. Мы удостоверились, что каждая из 10 изначально выбранных статей попала в эту выборку.

На *третьем этапе* мы удалили из выборки дубликаты и нерецензированные статьи. Также мы удалили технические отчеты, диссертации, публикации не на английском языке и короткие статьи (меньше 4 страниц). В результате осталось 1077 статей.

На *четвертом этапе* мы продолжили отбор статей, изучив заголовки и аннотации. Мы оставили только те статьи, которые напрямую относятся к теме моделей памяти языков программирования, и, напротив, исключили статьи, которые только используют существующие результаты о моделях или статьи, относящиеся к смежным темам, таким как модели памяти архитектур процессоров, гетерогенных и распределенных систем; семантика транзакций и персистентности; методы верификации программ в контексте слабых моделей памяти. В результате осталось 105 статей.

На заключительном *пятом этапе* мы изучили содержание оставшихся статей. В итоговом списке мы оставили только те статьи, в которых основным результатом была либо новая модель памяти ЯП, либо изучение/уточнение существующей модели памяти ЯП. В итоге осталось 40 статей.

4. КРИТЕРИИ СРАВНЕНИЯ МОДЕЛЕЙ ПАМЯТИ

В этом разделе мы более подробно рассмотрим выбранные критерии сравнения моделей памяти языков программирования – оптимальность схем компиляции **C.1**, корректность трансформаций **C.2** и предоставляемые гарантии **C.3**. Эти критерии непосредственно связаны с примитивами, предоставляемыми абстракцией разделяемой памяти. Таким образом в начале нам необходимо рассмотерть эти примитивы.

Программные примитивы. Модель памяти определяет семантику разделяемой памяти программы при наличии параллельно исполняемых потоков. Разделяемая память состоит из переменных, каждая из которых имеет уникальный адрес.[8] Потоки могут обращаться к этим переменным, выполняя операции чтения и записи.

В большинстве языков программирования различатся следующие виды переменных: *неатомарные* (*non-atomic*), также именуемые как *обычные* (*plain*), и *атомарные* (*atomic*). Первые не должны использоваться для обращений из различных параллельно исполняемых потоков программы. В зависимости от конкретного языка параллельные обращения к неатомарным переменным либо полностью запрещены (например, в Haskell [8, 26] и Rust [27]), либо имеют неопределенное поведение (например, в C/C++ [5, 28]), либо обладают очень слабой семантикой, не предоставляющей гарантий о порядке, в котором потоки могут наблюдать эти обращения (например, в Java [3]).

В свою очередь, атомарные переменные как раз предназначены для параллельных обращений. Некоторые модели памяти вводят несколько типов обращений к атомарным переменным, аннотируя их *режимом доступа* (*access mode*). Например, языки C/C++ и Java (начиная с версии 9 [4]) имеют следующие режима доступа: ослабленный режим (*relaxed* или *opaque* в терминологии Java), режимы захвата и освобождения (*acquire/release*), последовательно согласованный режим (*sequentially consistent* или *volatile* в Java). Эти режи-

[6] https://scholar.google.com/

[7] Все поисквые запросы были выполнены 24 сентября 2020 года.

[8] В этой статье мы будем использовать термины “адрес” и “локация” как взаимозаменяемые.

мы обозначаются как **rlx**, **acq**, **rel** и **sc** соответственно. Заметим, что режим **acq** может быть применен только к операциям чтения, а режим **rel** – только к операциям записи. Неатомарные обращения иногда рассматриваются как дополнительный режим **na**, однако одновременное использование атомарных и неатомарных обращений к одной и той же переменной приводит к неопределенному поведению в программах на языке C/C++.

Режимы обращения упорядочены по гарантиям, которые они предоставляют, как показано на следующей диаграмме.

На одном конце спектра находятся последовательно согласованные обращения. При правильном использовании они гарантируют семантику последовательной согласованности (детали этого обсуждаются в разделе § 4.3.1). На другом конце спектра находятся неатомарные обращения, которые либо не дают никаких гарантий, либо предоставляют минимальные гарантии. Ослабленные обращения также имеют слабую семантику, тем не менее обычно они предоставляют свойство *когерентности* (см. раздел § 4.3.2). Наконец, в середине спектра находятся обращения, имеющие режимы захвата/освобождения. Они используются для поддержки идиомы передачи сообщений [29]. Поток, которому необходимо отправить сообщение, должен выполнить операцию освобождающей записи, а другой поток, ожидающий это сообщение, должен выполнить операцию захватывающего чтения. Если операция чтения наблюдает, что операция освобождающей записи выполнена, то два потока синхронизируются.

Модель памяти также может предоставлять атомарные операции *чтения-модификации-записи* (*read-modify-write*). Они включают в себя операции сравнения и замены (*compare-and-swap*), атомарного обмена (*exchange*) и разные вариации атомарного инкремента, например, *fetch-and-add*, *fetch-and-sub* и т.д. Операция сравнения и замены принимает на вход адрес разделяемой переменной, а также ожидаемое и желаемое значение. Она выполняет чтение переменной и сравнивает полученное значение с ожидаемым. Если они равны, то выполняется замена значения переменной на желаемое, а прочитанное значение возвращается как результат, вне зависимости от успеха проверки. Заметим, что описанные выше действия выполняются атомарно, т.е. ни одна другая операция записи не может выполниться между операциями чтения и записи. Операция обмена (*exchange*) атомарно заменит значение переменной и возвратит ее прежнее значение. Операция атомарного инкремента (*fetch-and-add* и др.) инкрементирует значение переменной и возвращает ее значение до модификации.

Некоторые модели памяти рассматривают блокировки (*locks*) как самостоятельный примитив [3]. Еще одним примитивом являются барьеры (*fence*) [5], которые соответствуют инструкциям барьеров памяти, выполняемых процессорами (см. раздел § 4.1).

Наконец, модель может рассматривать разделяемую память не как множество независимых типизированных переменных, а как последовательность байт, и допускать так называемые *смешанные* (*mixed-size*) параллельные обращения [30]. Например, в подобной модели операция чтения восьми байт может выполниться как два параллельных чтения по четыре байта.

4.1. Схема компиляции

Будем понимать под *схемой компиляции* отображение примитивов языка программирования в инструкции конкретного семейства процессоров. Мы будем рассматривать примитивы, представленные в разделе § 4. Процессоры обычно предоставляют инструкции для выполнения обычных операций чтения и записи[9], операции чтения-модификации-записи, а также различные типы барьеров памяти.

Схема компиляции должна быть корректной, т.е. обеспечивать, что множество сценариев поведения, допустимых моделью памяти процессора для скомпилированной программы, будет являться подмножеством сценариев поведения программы, допустимых моделью языка программирования.

Рассмотрим пример. Программа SB, представленная ниже, является фрагментом алгоритма Деккера, обсуждавшегося в § 1.

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB)}$$

Предположим, что язык программирования предоставляет модель последовательной согласованности, а программа должна быть скомпилирована для x86. Если компилировать операции чтения и записи обычным образом x86 [10], тогда, следуя спецификации модели памяти x86, допустимым будет следующий результат работы программы (он также будет наблюдаться и на практике):

[9] Некоторые архитектуры предоставляют дополнительные инструкции чтения и записи с определенным режим доступа, например, **Armv8** lda – захватывающее чтение (load acquire), stl – освобождающая запись (store release).

[10] В архитектуре x86 инструкция MOV используется для чтения и записи в память.

$[r_1 = 0, r_2 = 0]$. Данный результат может появиться вследствие *буферизации операций записи* – операция записи $x := 1$ может быть исполнена после выполнения всех остальных инструкций программы.

Отметим, что результат $[r_1 = 0, r_2 = 0]$ не является результатом последовательно согласованного сценария, и, следовательно, рассмотренная схема компиляции не является корректной. Как было продемонстрировано в разделе § 1, некорректность схемы компиляции может иметь негативные последствия и нарушать корректность программы.

Корректная схема компиляции для модели последовательной согласованности под архитектуру x86 может компилировать операцию записи как обычную инструкцию записи, за которой следует инструкция mfence [5, 9], как продемонстрировано ниже:

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ \texttt{mfence} & \texttt{mfence} \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB+MFENCE)}$$

Инструкция `mfence` является специальным барьером памяти в системе команд процессоров x86, который выполняет сброс буфера записей в основную память. Для программы SB+MFENCE результат $[r_1 = 0, r_2 = 0]$ запрещен моделью памяти x86.

Несмотря на то, что модифицированная схема компиляции является корректной, она не *оптимальна* [31] в том смысле, что она использует барьеры памяти, которые обычно являются причиной замедления программы на 10–30% на процессорах семейства x86 [32, 33]. К сожалению, современные процессоры не могут иметь одновременно *корректную и оптимальную* схему компиляции в рамках модели последовательной согласованности. Этот факт затрудняет использование модели **SC** для высокопроизводительных языков программирования и служит одним из стимулов к ослаблению моделей памяти.

В рамках этой статьи при обсуждении схем компиляции мы будем рассматривать процессоры семейств **x86**, **Armv7**, **Armv8** и **POWER** по следующим причинам. Во-первых, эти архитектуры наиболее распространены на сегодняшний день. Во-вторых, модели памяти для этих процессоров всесторонне изучены исследовательским сообществом, что привело к созданию строгих формальных спецификаций [9, 11, 14, 15].

4.2. Трансформации кода

Следующим критерием является **C.2** – корректность трансформаций, то есть правил переписывания исходного кода, применяемых в компиляторных оптимизациях.

Корректная трансформация должна сохранять семантику программы. В нашем контексте, как и в случае корректности схемы компиляции, это означает, что множество допустимых сценариев поведения программы после применения трансформации должно быть подмножеством допустимых сценариев поведения оригинальной программы.

Возвращаясь к примеру SB, снова рассмотрим модель последовательной согласованности и возьмем трансформацию, которая переставляет местами операции записи и чтения в левом потоке, предполагая, что они оперируют различными локациями в памяти:

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \quad \rightsquigarrow \quad \begin{array}{l|\!|l} r_1 := y & y := 1 \\ x := 1 & r_2 := x \end{array}$$

Для преобразованной версии программы (справа), результат $[r_1 = 0, r_2 = 0]$ является последовательно согласованным. Тем не менее, для оригинальной версии программы (слева) это неверно. Следовательно, вышеупомянутая трансформация является некорректной для модели SC.

В следующих разделах мы рассмотрим некоторые трансформации, обсуждаемые в различных исследованиях. Заметим, что этот список далеко не полон и не включает многие трансформации, выполняемые существующими оптимизирующими компиляторами [34]. Например, он не включает трансформации над циклами, так как в теории моделей памяти еще недостаточно проработаны темы гарантий прогресса (*liveness properties*) [35], которые необходимы для формального изучения этих трансформаций.

Трансформации, которые мы рассматриваем, разделены на два подкласса: *локальные* и *глобальные*. Локальные трансформации модифицируют небольшой участок кода в пределах одного потока; глобальные трансформации задействуют всю программу или ее большую часть, захватывая несколько потоков.

4.2.1. Локальные трансформации

Переупорядочивание независимых инструкций (Reordering of Independent Instructions). Эта трансформация переставляет местами две смежные инструкции, выполняющие обращение к различным адресам памяти. Выделяют четыре типа переупорядочивания: запись/чтение, запись/запись, чтение/чтение и чтение/запись.

$x := v; \quad r := y \rightsquigarrow r := y; \quad x := v$ store/load, SL

$x := v; \quad y := u \rightsquigarrow y := u; \quad x := v$ store/store, SS

$r := x; \quad s := y \rightsquigarrow s := y; \quad r := x$ load/load, LL

$r := x; \quad s := v \rightsquigarrow y := v; \quad r := x$ load/store, LS

Элиминация избыточного обращения (Elimination of Redundant Access). В паре двух смежных обраще-

ний к памяти одно из них может быть удалено, если его эффект покрывается другим. Например, две операции записи в одну и ту же переменную одного и того же значения могут быть заменены на одну операцию записи. Аналогично переупорядочиванию инструкций, выделяют четыре класса трансформации этого вида.

$x := v; \quad r := x \rightsquigarrow x := v; \quad r := v$ store/load, SL

$r := x; \quad s := x \rightsquigarrow r := x; \quad s := r$ store/store, SS

$r := x; \quad x := r \rightsquigarrow r := x$ load/store, LS

$x := v; \quad x := u \rightsquigarrow x := u$ store/store, SS

Элиминация нерелевантной операции чтения (Irrelevant Load Elimination, ILE). Эта трансформация удаляет инструкцию чтения, если ее результат не используется в программе.

$$r := x \rightsquigarrow \epsilon \mid r \text{ is never used}$$

Введение спекулятивной операции чтения (Speculative Load Introduction, SLI). Эта трансформация является обратной к предыдущей и вставляет инструкцию чтения в произвольное место программы.

$$\epsilon \rightsquigarrow r := x \mid r \text{ is never used}$$

В комбинации с элиминацей типа чтение/чтение эта трансформация может быть использована для того, чтобы вынести чтение из ветки условного оператора:

$$\texttt{if}\ (e)\ \texttt{then}\ \{r := x\} \rightsquigarrow s := x;\ \texttt{if}\ (e)\ \texttt{then}\ \{r := s\} \mid s\ \texttt{is never used}$$

Переупорядочивание с увеличением синхронизации (Roach Motel Reordering, RM). Этот класс трансформаций позволяет вносить дополнительные инструкции в блоки синхронизации. Например, инструкция записи может быть перенесена в критическую секцию, то есть переставлена за следующую операцию захвата блокировки. Интуитивно, такие перестановки могут только увеличить степень синхронизации в программе, то есть преобразованная программа должна обладать меньшим недетерминизмом и иметь меньшее количество допустимых сценариев поведения.

Неатомарные обращения могут быть внесены в критическую секцию без дополнительных предусловий. Кроме того, инструкция записи может быть перемещена после операции захвата блокировки, а инструкция чтения может быть перемещена до операции освобождения блокировки. Похожие правила применяются к перестановке операций вокруг захватывающих (*acquire*) и освобождающих (*release*) обращений к памяти.

$r :=_{\text{na}} x; \quad \text{lock}(l) \rightsquigarrow \text{lock}(l); \quad r :=_{\text{na}} x$

$x :=_{\text{o}} v; \quad \text{lock}(l) \rightsquigarrow \text{lock}(l); \quad r :=_{\text{o}} v$

$\text{unlock}(l); \quad x :=_{\text{na}} v \rightsquigarrow x :=_{\text{na}} v; \quad \text{unlock}(l)$

$\text{unlock}(l); \quad x :=_{\text{o}} x \rightsquigarrow r :=_{\text{o}} x; \quad \text{unlock}(l)$

Усиление обращений к памяти (Strengthening, S). Данная трансформация, подобно предыдущей, увеличивает степень синхронизации в программе путем усиления режима обращения к памяти. Например, неатомарное обращение к переменной может быть заменено на последовательно согласованное:

$$r :=_{o} x \rightsquigarrow r :=_{o'} x \mid o \sqsubset o'$$

$$r :=_{o} v \rightsquigarrow x :=_{o'} v \mid o \sqsubset o'$$

Трансформации, сохраняющие трассы (Trace Preserving Transformations, TP). Этот широкий класс трансформаций, который включает трансформации, не меняющие множество трасс потока [36]. Трассой называется последовательность видимых побочных эффектов, возникающих во время исполнения кода потока, при этом операции чтения и записи в разделяемую память тоже считаются эффектами. Классическим примером подобной трансформации является *распространение констант* [34, 37]. Ниже приведен пример применения данной трансформации.

$$x := 0 + v \rightsquigarrow x := v$$

Удаление общих подвыражений (Common Subexpression Elimination, CSE). CSE является еще одной классической трансформацией [34], которая выполняет поиск и удаление идентичных подвыражений. Вот пример выполнения этой трансформации:

$$r_1 := x + y; \quad r_2 := x + y \rightsquigarrow r_1 := x + y; \quad r_2 := r_1$$

4.2.2. Глобальные трансформации

Продвижение регистров (Register Promotion, RP). Если компилятор может определить, что обращения к разделяемой переменной происходят только из одного потока, тогда он может заменить все обращения к этой переменной на обращения к регистру.

$$x := v; \quad r := x \rightsquigarrow s := v; \quad r := s \mid \texttt{x}\ \text{is not accessed from other threads} \mid \texttt{s}\ \text{is a fresh register}$$

Слияние потоков (Thread Inlining, TI). Эта трансформация объединяет два потока в один. Оказывается, что эта на первый взгляд простая и очевидная трансформация не является корректной в некоторых моделях памяти.

$$P \parallel Q \rightsquigarrow P\ ;\ Q$$

Трансформации, основанные на анализе диапазона значений (Value Range Based Transformations, VR). Трансформации этого класса могут быть применены в случае, если программа удовлетворяет некоторому инварианту, выведенному с помощью

глобального анализа диапазона возможных значений переменных. Например, в программе ниже условный оператор может быть удален, так как статический анализ может вывести инвариант $x \geq 0$.

$$\begin{array}{l} r_1 := x \\ \text{if } (r_1 \geq 0) \text{ then} \\ y := 1 \end{array} \left\| \begin{array}{l} \\ r_2 := x \\ y := r_2 \end{array} \right. \quad \rightsquigarrow \quad \begin{array}{l} r_1 := x \\ y := 1 \end{array} \left\| \begin{array}{l} \\ r_2 := x \\ y := r_2 \end{array} \right.$$

4.3. Гарантии

Далее мы обсуждаем третий критерий **C.3** – гарантии о поведении программ, предоставляемые моделью памяти.

4.3.1. DRF-свойство

При рассуждении о многопоточных программах большинство программистов подразумевают модель последовательной согласованности. Действительно, было бы неправильно ожидать от программистов знания всех деталей слабых моделей, так как это только усложнило бы и без того непростую задачу проектирования и разработки многопоточных программ. Для того чтобы решить эту проблему, было предложено свойство *свободы от гонок* (*data-race freedom*, **DRF**) [3]. Это свойство гарантирует, что при наличии достаточной степени синхронизации программа будет иметь только последовательно согласованные сценарии поведения в слабой модели памяти. Другими словами, если слабая модель памяти обладает **DRF**-свойством, то при условии правильного использования примитивов синхронизации программист может не задумываться о слабых сценариях поведения и подразумевать модель последовательной согласованности.

Рассмотрим пример. Вернемся к программе SB из § 4.1. Как было продемонстрировано ранее, в слабой модели эта программа может допускать результат $[r_1 = 0, r_2 = 0]$. Тем не менее, семантика последовательной согласованности может быть восстановлена, например, при помощи блокировок, как показано в примере ниже:

$$\begin{array}{c} \texttt{lock}(l) \\ x := 1 \\ r_1 := y \\ \texttt{unlock}(l) \end{array} \left\| \begin{array}{c} \texttt{lock}(l) \\ y := 1 \\ r_2 := x \\ \texttt{unlock}(l) \end{array} \right. \qquad \text{(SB+LOCK)}$$

Совместимая с **DRF**-свойством слабая модель памяти должна гарантировать, что для программы выше допустимы только последовательно согласованные сценарии поведения с результатами $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$ или $[r_1 = 1, r_2 = 1]$.

Если модель предоставляет последовательно согласованный режим доступа, тогда программист также может аннотировать все обращения к переменным как последовательно согласованные и таким образом восстановить семантику SC:

$$\begin{array}{l} x :=_{\text{sc}} 1 \\ r_1 :=_{\text{sc}} y \end{array} \left\| \begin{array}{l} y :=_{\text{sc}} 1 \\ r_2 :=_{\text{sc}} x \end{array} \right. \qquad \text{(SB+SC)}$$

Более формально, **DRF**-свойство для слабой модели M утверждает, что если программа не содержит гонок в модели последовательной согласованности, тогда модель M допускает только последовательно согласованные сценарии поведения для этой программы.

Итак, свойство **DRF** позволяет свести рассуждения о поведении программы в слабой модели к рассуждениям в модели последовательной согласованности. Достаточно лишь показать, что программа не имеет гонок в модели SC, чтобы иметь факт, что она будет иметь только **SC** сценарии поведения в слабой модели.

Свойство **DRF** в приведенной выше формулировке иногда также называется *внешней свободой от гонок* (**eDRF**), чтобы отличать его от *внутренней свободы от гонок* (**iDRF**). **iDRF**-свойство гарантирует для программы семантику **SC** в слабой модели M только в случае, если программа не имеет гонок в самой модели M. Это свойство предоставляет более слабую гарантию по сравнению с внешней свободой от гонок. Оно не позволяет полностью избежать рассуждений в терминах слабой модели, так как сначала необходимо показать, что программа не имеет гонок именно в слабой модели. Как будет продемонстрировано далее (*см.* § 6.3), внутренняя свобода от гонок является компромиссом для определенного класса моделей, которые не могут предоставить внешнюю свободу от гонок.

4.3.2. Когерентность (COH)

Как было показано раньше, современные процессоры не гарантируют выполнимость модели последовательной согласованности. Тем не менее, обычно они предоставляют более слабую гарантию *последовательной согласованности по каждой локации в памяти*, именуемую также *когерентностью* [12]. Соответственно, модели памяти для языков программирования также зачастую предоставляют эту гарантию.

Когерентность гарантирует, что все операции записи по определенному адресу памяти будут полностью упорядочены, и что получающийся в результате *порядок когерентности* (*coherence order*) отражает порядок, в котором эффекты от операций записи, выполненных некоторым потоком, отражаются в основной памяти, и их результаты становятся видимыми для других потоков. В частности, из свойства когерентности следует, что программа, состоящая из обращений только к од-

ной локации в памяти, должна обладать семантикой последовательной согласованности. Например, рассмотрим следующую программу:

$$\begin{matrix} x := 1 \\ r_1 := x \end{matrix} \Bigg\| \begin{matrix} x := 2 \\ r_2 := x \end{matrix} \qquad \text{(COH)}$$

Наличие свойства когерентности предписывает модели памяти допускать для этой программы только последовательно согласованные сценарии поведения с результатами $[r_1 = 1, r_2 = 2], [r_1 = 1, r_2 = 1]$ или $[r_1 = 2, r_2 = 2]$. Для модели, не удовлетворяющей свойству когерентности, допустимым также является результат $[r_1 = 2, r_2 = 1]$. Например, модель памяти Java допускает подобный сценарий поведения [3].

4.3.3. Неопределенное поведение (no-UB)

Как мы уже кратко упоминали, некоторые модели памяти, например, C/C++, рассматривают программы с гонками на неатомарных обращениях как имеющие *неопределенное поведение* [28]. Другими словами, для таких программ любой сценарий поведения считается допустимым. Это свойство также иногда называется *возгорающейся семантикой* (*catch-fire semantics*).

Практическая польза такого подхода заключается в том, что он допускает оптимальную схему компиляции для неатомарных обращений и позволяет применять к ним любые оптимизации, корректные для последовательных программ. Дело в том, что эффекты от оптимизаций, осуществляемых процессором или компилятором, могут наблюдаться только при обращении к переменным из параллельных потоков. Если таким обращениям предписывается неопределенное поведение и на них не распространяются никакие гарантии, то тогда эффекты этих оптимизаций становятся неразличимы с точки зрения семантики программы.

4.3.4. Спекулятивное исполнение (In-Order) и значения из воздуха (no-OOTA)

Чтобы представить последние два свойства, а именно, наличие спекулятивного исполнения и значений из воздуха, мы рассмотрим еще один пример:

$$\begin{matrix} r_1 := x \\ y := 1 \end{matrix} \Bigg\| \begin{matrix} r_2 := y \\ x := r_2 \end{matrix} \qquad \text{(LB)}$$

Предположим, что модель памяти допускает результат $[r_1 = 1, r_2 = 1]$ для этой программы. Например, модели семейств процессоров **Armv7**, **Armv8** и **POWER** допускают сценарий поведения, ведущий к такому результату, и этот сценарий может наблюдаться на некоторых процессорах семейства **Armv7** [38].

Результат $[r_1 = 1, r_2 = 1]$ не может быть получен путем исполнения инструкции согласно их порядку внутри потоков. Чтобы получить подобное поведение, модель памяти должна использовать некоторую форму *спекулятивного исполнения* [16, 39]. Это означает, что операция чтения $r_1 := x$ должна быть буферизована, а операция записи $y := 1$ должна выполниться вне очереди (отсюда и название программы выше – буферизация операции чтения *load buffering*).

Однако неограниченные спекуляции могут привести к нежелательным последствиям. Операция записи, исполненная вне очереди, может обернуться самоисполняющимся пророчеством (self-fulfilling prophecy) [16]. Рассмотрим следующий вариант программы с буферизацией операции чтения:

$$\begin{matrix} r_1 := x \\ y := r_1 \end{matrix} \Bigg\| \begin{matrix} r_2 := y \\ x := r_2 \end{matrix} \qquad \text{(LB+data)}$$

Здесь гипотетическая абстрактная машина может спекулятивно исполнить операцию записи в переменную `y` значения `1` в левом потоке, затем прочитать это значение в правом потоке, записать его в переменную `x` и прочитать обратно из первого потока, таким образом сформировав парадоксальный цикл причинно-следственных связей. Значение `1` в примере выше появляется *из воздуха* (*out of thin-air*) и приводит к неожиданному результату $[r_1 = 1, r_2 = 1]$.

Как будет показано в § 6, возможность спекулятивного исполнение необходима для того, чтобы поддержать в модели памяти некоторый класс трансформаций программ. Тем не менее, спекулятивное исполнения необходимо ограничить должным образом, чтобы избежать появления значений из воздуха. В § 6.4–§ 6.6 мы рассмотрим то, как эта проблема решается в различных моделях памяти.

5. СРАВНЕНИЕ

Мы провели сравнение моделей памяти, отобранных с помощью процедуры описанной в § 3, по критериям, представленным в § 4. В этом разделе представлены результаты этого сравнения.

Сравнение моделей памяти было осложнено тем фактом, что рассмотренные статьи, зачастую, используют различную терминологию, предоставляют неполную информацию о моделях памяти, а в некоторых случаях статьи противоречат друг другу. Мы подошли к решению этих трудностей следующим образом. Во-первых, для обозначения свойств моделей памяти мы использовали единую терминологию, описанную нами в § 4. Во-вторых, мы дополняли информацию о каждой модели из разных источников. Если после этого

Таблица 1. Классы моделей памяти и их свойства

Class	# Models	Compilation				Transformations																Reasoning				
						Local														Global							
						Reordering				Elimination																
		x86	POWER	Armv7	Armv8	SL	SS	LL	LS	SL	SS	LL	LS	ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
Sequential Consistency	2	–	–	–	–	–	–	–	–	+	+	+	+	+	+	+	+	+	–	+	+	–	+	+	+	+	+
Total/Partial Store Order	2	+	–	–	–	+	–	–	+	+	+	+	+	–	+	+	–	–	+	–	–	–	+	+	+	+	+
Program Order Preserving	3	+	–	–	–	+	+	–	+	+	+	+	+	–	–	–	+	+	+	±	+	–	+	+	+	+	+
Syntax. Dep. Preserving	2	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	–	–	–	+	+	+	–	+
Semantic Dep. Preserving	7	+	+	+	+	+	+	+	+	+	+	+	+	±	+	+	+	+	+	±	–	+	+	+	+	–	+
Out of Thin-Air	5	+	+	+	+	+	+	+	+	+	+	+	+	–	–	–	+	+	+	–	+	–	–	+	+	–	–

наличие или отсутствие определенного свойства у модели все еще оставалось неясным, мы явно помечали этот факт.

Мы идентифицировали шесть классов моделей памяти: последовательно согласованные (sequentially consistent); модели с линейным/частичным порядком на операциях записи (total or partial store order); модели, сохраняющие программный порядок (program order preserving models); модели, сохраняющие синтаксические зависимости (syntactic dependency preserving); модели, сохраняющие семантические зависимости (semantic dependency preserving); модели допускающие значения из воздуха (out of thin-air). Модели из одного класса имеют схожие схемы компиляции, множество корректных трансформаций и предоставляемые гарантии. Сначала мы представляем результат сравнения различных классов (табл. 1), а затем – сравнение отдельных моделей (табл. 2).

В табл. 1 и 2 мы упорядочиваем модели по степени их ослабленности. Более строгие модели расположены в верхних строках, а более слабые – в нижних.

Колонки обоих таблиц соответствуют свойствам моделей памяти. Для краткости мы выбрали бинарную классификацию свойств, т.е. считаем, что модель либо удовлетворяет данному свойству, либо нет. Мы также разделили все свойства на несколько групп.

Первая группа свойств посвящена оптимальности схем компиляции для различных семейств процессоров. Мы классифицируем схему компиляции как оптимальную или неоптимальную следующим образом. Мы выбираем наиболее слабый режим доступа, поддерживаемый моделью памяти, и рассматриваем схему компиляции для обращений к памяти, аннотированных данным режимом. Для моделей, которые трактуют гонки на неатомарных переменных как неопределенное поведение, мы рассматриваем наиболее слабый режим доступа к атомарным переменным. Дело в том, что семантика с неопределенным поведением для программ с гонками тривиальным образом допускает оптимальную схему компиляции (*см.* § 4.3.3). Мы считаем схему компиляции *оптимальной*, если обращения к памяти, аннотированные выбранным режимом доступа, могут быть скомпилированы в обычные инструкции чтения и записи целевой архитектуры (т.е. без использования барьеров памяти или какого-либо другого дополнительного кода).

Вторая группа свойств посвящена корректности различных трансформаций. Здесь классификация также бинарная: конкретная трансформация либо является корректной в данной модели, либо нет. Мы вновь не рассматриваем все возможные комбинации трансформаций и режимов доступа. Вместо этого мы останавливаемся только на наиболее слабом режиме доступа с полностью определенной семантикой (т.е. без неопределенного поведения). Также мы разделяем трансформации на локальные и глобальные.

Третья группа свойств соответствует предоставляемым гарантиям о поведении программ. В частности, для каждой модели мы указываем, предоставляет ли она свойство внешней свободы от гонок **eDRF** (*см.* § 4.3.1), свойство когерентности (*см.* § 4.3.2), трактует ли она гонки на неатомарных переменных как неопределенное поведение (см. § 4.3.3), используется ли последовательное исполнение инструкций (*in-order execution*) или применяет спе-

Таблица 2. Модели памяти и их свойства

Class	Model	Compilation				Transformations																	Reasoning				
						Local														Global							
						Reordering				Elimination																	
		x86	POWER	Armv7	Armv8	SL	SS	LL	LS	SL	SS	LL	LS	ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
Sequential Consistency	SC [8, 32, 33, 40, 41]	–	–	–	–	+	–	–	–	+	+	+	–	+	+	+	+	+	–	+	+		+	+	+	+	+
	DRFx [42]	–	–	–	–	+	+	–	–	+	+	+	+	+	–	+	+	+	–	+	+		+	+	–	+	+
Total/Partial Store Order	BMM [43]	+	–	–	–	+	+	+	–	+	+	+	–	+	+	–		+	–		–		+	+	+	+	+
	RMMOA [44]	+	–	–	–	+	+	+	–														+		+	+	+
Program Order Preserving	RC11 [21, 45–47]	+	–	–	–	+	+	+	–	+	+	+	–		–	+	+	+	–	+	+		+	+	–	+	+
	OCMM [22]	+	–	–	–	+	+	+	–	+	+	+	–					+	+				+	+	–	+	+
	JAM [4]	+	–	–	–												+		–				+	+	+	+	+
Syntax. Dep. Preserving	LKMM [48]	+	+	+	+	+	+	+	+									–	–					+	+	–	+
	OHMM [49]					+	+	+	+	+	+	+	+	+	+	+		–					+		+	–	+
Semantic Dep. Preserving	JMM [3, 36, 50]	+	–	+	+	+	+	+	+	+	+	–	–	+	–	–		+	–		–		+	+	–	–	+
	PRM [17, 18]	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	+	+	+	–	–	+
	WMO [6, 19]	+	+	+	+	+	+	+	+	+	+	+	–		+	–	+		–		–		+	–	+	–	+
	CSRA [25]	+	+	–	–	+	+	+	+	+	+	+	–	+	–	+	–		–		–	+		+	+	–	+
	WJES [24]		–	–	–			–				–	–										+	–	+	–	+
	MRD [20]	+	+	+	+																		+	+	+	–	+
	GOS [51]																						+		+	–	+
Out of Thin-Air	C11 [5, 29, 30, 52–57]	+	+	+	+	+	+	+	+	+	+	+	–		–	+	+	+	–		+		–	+	–	–	–
	JSMM [7]	+	+	+	+																		–	–	+	–	–
	RMC [58]	+	+	+	+																		–	+	+	–	–
	RAO [59]																							–	+	–	–
	TSC [39]																						–		+	–	–

кулятивное исполнение (*speculative execution*), а также допускает ли данная модель значения из воздуха (см. § 4.3.4).

В табл. 1 каждая строка соответствует целому классу моделей. Клетка помечается символом “+” если большинство моделей данного класса обладают соответствующим свойством. Если рассматриваемым свойством обладает только некоторое количество моделей данного класса, не составляющих большинство, то клетка помечается символом “±”. Наконец, если ни одна из моделей данного класса не обладает данным свойством, тогда клетка помечается символом “–”. При подсчете большинства мы опускаем те модели данного класса, про которые неизвестно, обладают они данным свойством или нет. Также, если какое-то свойство вообще не изучалось в контексте определенного класса моделей, мы также помечаем клетку символом “–”. Таким образом, в табл. 1 символы “+” и “±” обозначают положительную информацию, а символ “–” обозначает как негативную информацию, так и отсутствие информации.

В табл. 2 каждая строка соответствует конкретной модели памяти, обозначаемой аббревиатурой, а каждая клетка описывает наличие или отсутствие определенного свойства этой модели. Мы помечаем клетку символом “+”, если рассматриваемая модель обладает данным свойством и символом “–” в противном случае. Если нам не удалось получить информацию о данном свойстве рассматриваемой модели, то соответствующая клетка заливается серым цветом ■.

Помимо табл. 1 и 2, которые описывают свойства моделей, мы также представляем табл. 3, содержащую список примитивов, поддерживаемых каждой моделью. Каждая строка данной таблицы соответствует отдельной модели памяти. Колонки соответствуют поддерживаемым примитивам. В частности, мы указываем, какие режимы обращений к переменным модель поддерживает: неатомарные (NA), ослабленные (RLX), захвата/освобождения (RA), последовательно согласованные (SC); какие типы барьеров поддержаны: захвата/освобождения (RA) и последовательно согласованные (F-SC); поддерживаются ли атомарные операции чтения-модификации-записи (RMW), поддерживаются ли явно операции блокировки (LK), и поддерживаются ли смешанные обращения (MIX).

6. АНАЛИЗ

В этом разделе мы обсуждаем результаты сравнения, представленные в предыдущем разделе. На основе данных из табл. 1 и 2 мы выводим взаимосвязи между оптимальностью схемы компиляции, корректностью трансформаций и предоставляемыми гарантиями. В частности, мы демонстрируем, как поддержка некоторых гарантий конфликтует с некоторыми трансформациями и как она влияет на оптимальность схемы компиляции. Мы начинаем с рассмотрения класса последовательно согласованных моделей § 6.1, затем переходим к моделям с линейным/частичным порядком на операциях записи § 6.2. После этого мы рассматриваем класс наиболее слабых моделей, допускающих значения из воздуха § 6.3, а далее переходим к обсуждению различных подходов к решению проблемы значений из воздуха и рассматриваем модели, сохраняющие программный порядок § 6.4, синтаксические § 6.5 и семантические § 6.6 зависимости. В § 6.7 мы отдельно обсуждаем некоторые конкретные свойства моделей, в частности, когерентность и возгорающуюся семантику. Факт наличия или отсутствия этих свойств ортогонален разбиению на вышеупомянутые классы. Тем не менее, наличие этих свойств у модели также влияет на корректность определенных трансформаций.

6.1. Модель последовательной согласованности

Данная модель является наиболее интуитивной моделью многопоточности. В рамках этой модели состояние памяти может быть представлено как отображение из адресов переменных в хранящиеся значения. Тогда каждый допустимый сценарий поведения программы может быть получен в результате поочередного последовательного исполнения инструкций потоков.

Многие распространенные трансформации оказываются некорректными в модели **SC**, включая все типы переупорядочивания операций, а также удаление общих подвыражений [32, 32]. Тот факт что переупорядочивание инструкций запрещено делает эту модель очень дорогостоящей при реализации на современных процессорах, так как даже относительно строгая модель памяти процессоров **x86** допускает переупорядочивание типа запись/чтение. Таким образом, чтобы гарантировать последовательную согласованность, компилятор вынужден вставлять в код тяжеловесные барьеры памяти между инструкциями записи и чтения, что приводит к неоптимальной схеме компиляции.

Однако в терминах предоставляемых программисту гарантий модель **SC** является весьма привлекательной. В частности, тривиальным образом гарантируются свойства **eDRF** и когерентности, так как модель присваивает программе только последовательно согласованные сценарии поведения.

Концептуальная простота и привлекательность модели **SC** вдохновила многих исследователей на попытки адаптации этой модели и смягчения накладываемых штрафов на время испол-

Таблица 3. Поддержка примитивов моделями памяти

Class	Model	Features								
		NA	RLX	RA	SC	F-RA	F-SC	RMW	LK	MIX
Sequential Consistency	EtE-SC [32, 40]	–	–	–	+	–	–	–	–	–
	VbD [33, 41]	+	–	–	+	–	–	+	+	–
	SC-Hs [8]	+	–	–	+	–	–	+		–
	DRFx [42]	+	–	–	+	–	–	–	–	–
Total/Partial Store Order	BMM [43]	+	–	–	+	–	–	–	–	–
	RMMOA [44]	+	–	–	–	–	–	–	+	–
Program Order Preserving	RC11 [21]	+	+	+	+	+	+	+	–	–
	ORC11 [47]	+	+	+	–	–	–	+	–	–
	RAR [46]	–	+	+	–	–	–	+	–	–
	CRC [45]	+	–	+	–	–	+	+	–	–
	OCMM [22]	+	–	–	+	–	–	+	–	–
	JAM [4]	+	+	+	+	+	+	+	–	–
Syntax. Dep. Preserving	LKMM [48]	–	+	+	–	+	+	+	–	–
	OHMM [49]	+	–	–	+	–	–	–	+	–
Semantic Dep. Preserving	JMM [3]	+	–	–	+	–	–	+	+	–
	PRM [17]	+	+	+	–	+	+	+	+	–
	WMO [19]	+	+	+	+	+	+	+	–	–
	CSRA [25]	+	+	+	–	–	–	+	+	–
	WJES [24]	–	+	–	–	–	–	+	+	–
	MRD [20]	–	+	–	–	–	–	–	+	–
	GOS [51]	+	–	–	–	–	–	–	+	–
Out of Thin-Air	C11 [5]	+	+	+	+	+	+	+	+	+
	JSMM [7]	+	–	–	+	–	–	+	+	+
	RMC [58]	–	+	+	+	+	+	+	–	–
	RAO [59]	+	–	–	+	–	–	–	–	–
	TSC [39]	+	–	–	+	–	–	–	+	–

нения программы. Общей идеей данных работ была попытка отделения локальных (доступных только одному потоку) и разделяемых переменных. Обращения к локальным переменным могут быть скомпилированы без добавления барьеров памяти, также к ним применим широкий спектр оптимизаций корректных для случая однопоточных программ. Чтобы безопасным образом классифицировать локальные и разделяемые переменные исследователи использовали системы типов [8], статический [40] или динамический анализ [41], поддержку на аппаратном уровне [40, 42], или различные комбинации вышеупомянутых методов.

Несмотря на эти усилия, модель **SC** все равно имеет существенные накладные расходы. Например, при компиляции на процессоры семейства **Armv8** замедление времени работы программ может достигать 70% [41]. Более того, хотя вышеупомянутые техники обычно уменьшают накладные расходы на локальные обращения (которые, зачастую, чаще встречаются в программах), они оказывают меньшее влияние на специфичные приложения, которые активно используют многопоточность, например, такие как неблокирующие структуры данных. Наконец, требуется значительное количество усилий и технической работы, чтобы модифицировать современные компиляторы для поддержки модели **SC** [32, 41].

6.2. Линейный/частичный порядок на операциях записи

Данный класс моделей был создан на основе моделей с *линейным порядком на операциях записей* (*total store order*, **TSO**) и *частичным порядком на операциях записей* (*partial store order*, **PSO**) [60]. Модели **TSO** и **PSO** являются моделями семейств процессоров **x86** [9] и **SPARC** [60] соответственно. В этих моделях потоки оснащены *буферами запи-*

сей. Все операции записи вначале попадают в эти буферы, а потом переносятся в основную память.

Для моделей этого класса схема компиляции для архитектуры **x86** является оптимальной, так как **x86** предоставляет модель **TSO**. Однако при компиляции под архитектуры с более слабой моделью памяти, например **POWER**, необходимо использовать практически такое же количество барьеров, как и при компиляции из модели SC [61].

Модели этого класса допускают большее количество трансформаций кода, чем **SC**. Использование буферов для операций записи позволяет выполнять переупорядочивание типа запись/чтение в случае **TSO** и запись/запись – в случае **PSO**.

Хотя модели **TSO** и **PSO** слабее, чем модель **SC**, тем не менее они все еще предоставляют довольно сильные гарантии, в частности, свойство **eDRF** и когерентность.

Таким образом, модели этого класса не имеют значительных преимуществ перед моделью **SC**, и при этом влекут соизмеримые накладные расходы при компиляции для архитектуры с более слабой моделью памяти, чем у **x86**. Следовательно, выбор этих моделей в качестве моделей для языка программирования оправдан только если предполагается поддержка компиляции исключительно для процессоров архитектуры **x86**.

6.3. Значения из воздуха

Далее мы переместимся на другой конец спектра моделей памяти и рассмотрим класс, в который входят наиболее слабые модели. Эти модели предоставляют оптимальные схемы компиляции и допускают, практически, любые разумные трансформации программ, но достигают этого ценой введения значений из воздуха (4).

Снова рассмотрим пример программы буферизации операции чтения:

$$\begin{array}{ll|l|ll} r_1 := x & r_2 := y & & y := 1 & r_2 := y \\ y := 1 & x := r_2 & \rightsquigarrow & r_1 := x & x := r_2 \end{array}$$

(LB) (LBtr)

Версия программы справа LBtr может быть получена из программы слева LB путем применения переупорядочивания инструкций типа чтение/запись. Результат $[r_1 = 1, r_2 = 1]$ является допустимым для программы LBtr. Тогда модель памяти, в которой переупорядочивание типа чтение/запись является корректной трансформацией, также должна допускать этот результат для программы LB. Как было продемонстрировано в § 4.3.4, для того, чтобы получить такой результат, необходимо применить спекулятивное исполнение.

Мы также обсуждали, что неограниченное спекулятивное исполнение может привести к появлению так называемых значений из воздуха, которые нарушают фундаментальные гарантии о поведении программ [16, 55], в частности, гарантии типобезопасности (type-safety) и композициональности. Также не выполняется и гарантия внешней свободы от гонок (**eDRF**). Чтобы убедиться в этом, рассмотрим еще один пример:

$$\begin{array}{l|l} r_1 := x & r_2 := y \\ \text{if } (r_1)\ \{ & \text{if } (r_2)\ \{ \\ \quad y := 1 & \quad x := 1 \\ \} & \} \end{array} \qquad \text{(LB+ctrl)}$$

Для модели памяти, допускающей значения из воздуха, результат $[r_1 = 1, r_2 = 1]$ также является допустимым (обоснование этого результата такое же, как и для примера LB+data из § 4.3.4). Однако этот результат не только не интуитивен, но и противоречит гарантии внешней свободы от гонок. Действительно, в модели **SC** единственный допустимый сценарий поведения этой программы ведет к результату $[r_1 = 0, r_2 = 0]$ и не содержит гонок, следовательно, в модели, предоставляющей гарантию **eDRF**, эта программа также должна иметь только этот единственный сценарий поведения.

Контр-интуитивное поведение моделей, допускающих значения из воздуха, а также тот факт, что они нарушают множество важных гарантий о поведении программ, привело к следующему консенсусу в исследовательском сообществе: эти модели не подходят на роль моделей памяти для языков программирования [16, 55]. Множество усилий было направлено на то, чтобы сделать невозможными значения из воздуха, но в то же время сохранить оптимальность схем компиляции и корректность как можно большего количества трансформаций. В оставшейся части этого раздела мы опишем различные способы преодоления проблемы значений из воздуха.

6.4. Сохранение программного порядка

Наиболее простой способ запретить значения из воздуха был предложен в работе [16]. Основная идея этого подхода – полностью запретить спекулятивное исполнение, что может быть достигнуто путем запрета переупорядочивания инструкций типа чтение/запись. Это решение позволяет не только восстановить свойство внешней свободы от гонок (**eDRF**) и другие гарантии [21], но также ведет к более простой модели. Абстрактная машина, реализующая данную модель, не нуждается в использовании спекулятивного исполнения и может выполнять инструкции потоков согласно их *программному порядку*, т.е. в том порядке, в котором они указаны. Память такой машины может быть организована как монотонно растущая история сообщений, где каждый поток имеет свое представление фронта данной истории [22, 46].

Данный подход был формализован в работе [21]. Там же было показано, что многие трансформации над кодом программ, за исключением переупорядочивания инструкций типа чтение/запись, остаются корректными в рамках моделей данного класса (см. табл. 1).

Схема компиляции программ для процессоров семейства **x86** является оптимальной, так как модель памяти данной архитектуры гарантирует сохранение порядка между операциями чтения и последующими операциями записи. Однако архитектуры с более слабыми моделями (**Arm**, **POWER**) не гарантируют сохранения этого порядка, и таким образом требуют принятия дополнительных мер при компиляции кода. В [16] было предложено компилировать инструкции ослабленного (**rlx**) чтения как обычные инструкции чтения, за которыми следует ложная инструкция условного ветвления (conditional jump), которая добавляет зависимость между операцией чтения и последующими операциями записи. Гарантируется, что процессоры семейств **Arm** и **POWER** сохранят такую зависимость и, таким образом, сохранят порядок между операцией чтения и последующими операциями записи. В работе [23] изучались накладные расходы этой схемы компиляции, при ее применении только к ослабленным (**rlx**) атомарным обращениям. При компиляции кода для процессоров семейства **Armv8** замедление времени работы составило 0% в среднем и 6.3% максимум на наборе тестов, реализующих различные многопоточные структуры данных, например, блокировки, стеки, очереди, деки, ассоциативные массивы и т.д. Заметим, что следует ожидать более существенного замедления времени работы программ при применении данной схемы компиляции также к неатомарным обращениям к памяти.

6.5. Сохранение синтаксических зависимостей

Альтернативное простое решение проблемы значений из воздуха заключается в сохранении *синтаксических зависимостей* между операциями обращения к памяти [16, 48]. В рамках этого подхода переупорядочивание инструкций типа чтение/запись разрешено, если переставляемые инструкции являются независимыми. Переупорядочивание запрещено, если операция записи зависит от значения, прочитанного операцией чтения, (в этом случае мы говорим что существует зависимость по данным, *data dependency*), или если это значение было использовано при вычислении адреса операции записи (зависимость по адресу (*address dependency*), или если путь исполнения программы, ведущей к операции записи, зависит от прочитанного значения (зависимость по управлению, (*control dependency*). Например, в программе LB+data существует зависимость по данным, так как инструкция $y := r_1$ записывает значение, прочитанное инструкцией $x := r_1$.

Заметим, что эти зависимости вычисляются следуя синтаксису программы (отсюда происходит и название), в противоположность *семантическим зависимостям*. Например, в модифицированной версии программы LB+data, представленной ниже, операция записи в переменную y в левом потоке имеет синтаксическую зависимость от предыдущей операции чтения.

$$\begin{array}{l} r_1 := x \\ y := 1 + 0 * r_1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := y \\ x := r_2 \end{array} \qquad \text{(LB+fakedata)}$$

В этом примере синтаксическая зависимость может быть удалена с помощью оптимизации распространения констант – подвыражение $1 + 0 * r_1$ может быть преобразовано в значение 1. Однако если модель памяти гарантирует сохранение синтаксических зависимостей, компилятору запрещено применять эту оптимизацию, так как после удаления зависимости ничто не мешает компилятору или процессору переставить операцию записи до предшествующей операции чтения.

На этом примере можно видеть главный недостаток моделей, сохраняющих синтаксические зависимости: различные оптимизации, сохраняющие трассы (например, распространение констант), оказываются некорректными в этих моделях. Распространение констант является одной из классических оптимизаций, и тот факт, что оно некорректно, препятствует применению моделей этого класса. Заметим, что модели памяти процессоров используют сходный подход и тоже сохраняют синтаксические зависимости между обращениями к разделяемой памяти [11, 12, 14]. Однако в этом случае это не является проблемой, так как процессоры во время исполнения программы не выполняют такие сложные оптимизации, как распространение констант.

В работе [23] изучалось замедление времени работы программ, накладываемое моделью памяти, сохраняющей синтаксические зависимости. Авторы модифицировали оптимизирующие проходы компилятора так, чтобы они сохраняли зависимости между неатомарными и ослабленными (**rlx**) атомарными обращениями. Затем они измерили время работы программ из тестового набора SPEC CPU2006, скомпилированных модифицированной версией компилятора LLVM для процессоров семейства **Armv8**, и сообщили об умеренном замедлении на 3.1% в среднем и 17.6% максимум.

6.6. Сохранение семантических зависимостей

Последний рассматриваемый нами подход к решению проблемы значений из воздуха заключается в построении понятия *семантических зави-*

симостей, которые могли бы точно характеризовать то, какие пары операций чтения/записи являются независимыми, и отфильтровали бы ложные зависимости как в примере LB+fakedata. Практическая ценность данного подхода заключается в том, что он не требует модификаций существующих компиляторов и процессоров и не накладывает дополнительных расходов на время исполнения скомпилированных программ. Конечной целью подхода является предоставление оптимальных схем компиляции, сохранение корректности большинства существующих трансформаций кода, и в то же время поддержка основных гарантий, таких как внешняя свобода от гонок (**eDRF**).

Оказывается, что эта задача является весьма трудной и на сегодняшний день не решена. Чтобы дать удовлетворительное определение семантических зависимостей, исследователи были вынуждены обратиться к концептуально сложным моделям памяти [17, 19, 20, 24, 25, 51]. Основная сложность работ в данном направлении – необходимость предоставления формальных доказательств того, что эти модели удовлетворяют предъявляемым требованиям, а именно поддерживают оптимальные схемы компиляции и широкий набор трансформаций, и в то же время сохраняют все важные гарантии о поведении программ.

На сегодняшний день наиболее полное решение данной проблемы предоставляет модель “обещающей” семантики (**Promising** semantics) [17, 18]. Было доказано, что эта модель допускает оптимальные схемы компиляции [62], разрешает применять большинство локальных и глобальных трансформаций (за исключением слияния потоков), и в то же время предоставляет свойство внешней свободы от гонок и другие гарантии.

6.7. Вспомогательная классификация

Теперь представим альтернативное разделение моделей на группы на основе того, обладают ли они конкретным свойством, в частности, когерентностью и возгорающейся семантикой. Продемонстрируем, как наличие этих свойств сказывается на оптимальности схемы компиляции и корректности некоторых трансформаций.

6.7.1. Когерентные модели

Свойство когерентности, которое также иногда называется последовательной согласованностью по каждой локации (§ 4.3.2), неожиданным образом взаимодействует с трансформацией удаления общих подвыражений (**CSE**). Впервые эта связь была замечена в контексте ранней версии модели памяти Java [63]. Чтобы увидеть проблему, рассмотрим программу ниже (слева) и ее версию после применения трансформации **CSE** (справа). Заметим, что оптимизация заменила второе обращение к переменной x присваиванием значения из регистра.

$$\begin{array}{l} r_1 := x \\ r_2 := y \\ r_3 := x \end{array} \;\Big\|\; y := 1 \qquad \rightsquigarrow \qquad \begin{array}{l} r_1 := x \\ r_2 := y \\ r_3 := r_1 \end{array} \;\Big\|\; y := 1$$

Предположим, что переменные x и y на самом деле указывают на одну и ту же ячейку памяти. При таком предположении можно сделать вывод о том, что сценарий выполнения программы с результатом $[r_1 = 0, r_2 = 1, r_3 = 0]$ должен быть запрещен моделью памяти, гарантирующей когерентность. В самом деле, когерентная модель памяти для программы, содержащей обращения только к одной локации в памяти, допускает только последовательно согласованные сценарии исполнения. Результат $[r_1 = 0, r_2 = 1, r_3 = 0]$ не может быть получен как последовательное чередование инструкций потоков, и, значит, он не является последовательно согласованным и должен быть запрещен. Тем не менее, данный результат допустим для оптимизированной версии этой программы.

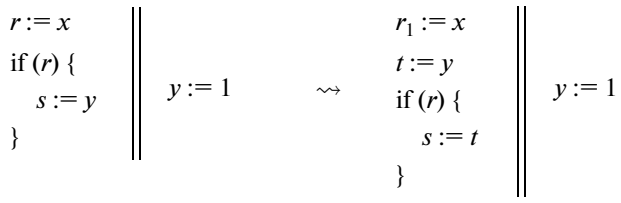
Заметим, что несмотря на вышесказанное, компилятор все равно может применить оптимизацию **CSE** к программе выше, но только в том случае, если он сможет вывести, что переменные x и y указывают на разные локации в памяти. Эта информация может быть получена при помощи анализа псевдонимов (alias analysis) [64]. По сути, в этом случае **CSE** может быть сведен к комбинации других трансформаций – переупорядочиванию и элиминации операций.

Таким образом, в общем случае свойство когерентность несовместимо с удалением общих подвыражений. Что касается оптимальности схем компиляции, то здесь когерентность не накладывает каких-то дополнительных ограничений, поскольку модели памяти процессоров гарантируют соблюдение когерентности [9, 11, 12, 21].

6.7.2. Модели с возгорающейся семантикой

Возгорающаяся семантика (*catch-fire semantics*) присваивает неопределенное поведение программам, содержащим гонки на неатомарных обращениях, и, таким образом, влияет на корректность трансформаций. Как было упомянуто ранее, наличие подобного неопределенного поведения позволяет использовать оптимальные схемы компиляции для неатомарных обращений и влечет корректность широкого класса трансформаций, корректных для случая однопоточных программ. Но помимо этого возгорающаяся семантика интересно взаимодействует с трансформацией введения спекулятивной операции чтения.

Рассмотрим следующий пример:



В § 4.2 мы упоминали, что спекулятивное введение операции чтения может быть использовано в комбинации с элиминацией типа чтение/чтение, чтобы вынести инструкцию чтения из ветки условного оператора. В частности, в примере выше спекулятивное введение операции чтения может добавить инструкцию $t := y$ перед условным оператором `if`, а затем элиминация типа чтение/чтение может заменить вторую операцию чтения присваиванием значения из регистра.

Тонкий момент тут заключается в том, что хотя программа слева не содержит гонок в модели последовательной согласованности, программа справа имеет гонку между операциями чтения и записи переменной y. В предположении, что все обращения в программе выше являются неатомарными, можно заключить, что возгорающаяся семантика должна рассматривать правую программу как имеющую неопределенное поведение. Другими словами, для программы справа допустим любой сценарий исполнения, в то время как для программы слева допустим только сценарий с результатом $[r = 0]$. Однако необходимость корректности трансформации требует, чтобы множество допустимых сценариев исполнения модифицированной программы было подмножеством сценариев поведения оригинальной программы. Можно видеть, что в этом примере условие корректности нарушается.

Говоря простыми словами, спекулятивное введение операции чтения в общем случае некорректно в моделях с возгорающейся семантикой, так как оно может привнести гонки в программы, которые их не содержали. Так как возгорающаяся семантика чувствительна к наличию или отсутствию гонок в программе, она не совместима с этой трансформацией.

Заметим, что эту проблему нельзя решить, запретив спекулятивное введение неатомарных операций чтения и разрешив вводить атомарные операции чтения. В самом деле, спекулятивно добавленная атомарная операция чтения может находиться в состоянии гонки с каким-то неатомарным обращением, расположенным в другом месте программы.

7. РЕКОМЕНДАЦИИ ПО ВЫБОРУ МОДЕЛИ

В этом разделе представлен краткий набор рекомендации для исследователей и системных разработчиков по выбору модели памяти для языка программирования.

Для языков, которые стремятся предоставить простую семантику и высокоуровневые абстракции ценой некоторых потерь в производительности (например, Haskell), целесообразно использовать простые модели памяти, такие как модель последовательной согласованности.

Языки программирования, фокусирующиеся на эффективности производимого кода, такие как C/C++, в свою очередь, вынуждены прибегать к наиболее слабым моделям, допускающим оптимальные схемы компиляции и широкий набор трансформаций кода. Для этих языков целесообразно использовать модели, сохраняющие семантические зависимости. Однако данные модели являются наиболее сложными, что существенно затрудняет рассуждения о корректности программ [65].

Между этими крайними вариантами находятся модели, сохраняющие программный порядок или синтаксические зависимости. Их целесообразно использовать для языков программирования, которые могут позволить умеренные накладные расходы на производительность в обмен на чуть более простое и предсказуемое поведение [23]. Примером такого языка может служить OCaml – высокоуровневый язык программирования, делающий упор на функциональной парадигме, который в то же время активно используется в областях, где критична производительность (разработка компиляторов и инструментов верификации программ).

Более того, модели, сохраняющие программный порядок, имеют дополнительное преимущество перед моделями, сохраняющими синтаксические или семантические зависимости: они не требуют использования спекулятивного исполнения. Этот факт еще больше упрощает рассуждение о корректности многопоточных программ. Ценой этой простоты является требование к моделям, сохраняющим программный порядок, использовать субоптимальные схемы компиляции для процессоров семейств **Arm** и **POWER**. С другой стороны, модели, сохраняющие синтаксические зависимости, позволяют эффективно компилировать код для процессоров **Arm** и **POWER**. Однако эти модели не поддерживают ряд трансформаций, сохраняющих трассы, например, распространение констант. Они также используют спекулятивное исполнение, что приводит к более сложной семантике.

Для языков программирования, использующих более строгие модели памяти, которые требуют использования неоптимальных схем компиляции и запрещают применение некоторых трансформаций, существует несколько общих

техник оптимизации, помогающих частично смягчить возникающие накладные расходы.

Система типов может существенно помочь в этом. Здесь преимущество имеют такие языки как Haskell, OCaml, Rust, которые статически различают и изолируют регионы памяти, к которым возможны обращения из параллельных потоков. Эти языки могут точно идентифицировать переменные, неизменяемые и локальные для одного потока, а затем компилировать обращения к таким переменным без использования барьеров. Более того, к этим обращениям можно применять широкий класс трансформаций кода, корректных для случая однопоточных программ.

Такие языки программирования как Java не могут использовать систему типов для предотвращения доступа к неатомарным переменным из параллельных потоков из-за обратной совместимости. Тем не менее подобные языки могут аппроксимировать множество локальных переменных потоков, используя консервативный статический анализ достижимости переменных (escape analysis) [66], или различные динамические техники [41], а затем оптимизировать обращения к этим переменным.

В функциональных языках программирования активно используются неизменяемые структуры данных. Этот стиль программирования минимизирует использование разделяемой памяти и помогает уменьшить накладные расходы, вызванные строгой моделью памяти [8].

Наконец, если язык допускает наличие неопределенного поведения как, например, C/C++, то альтернативой сложной модели, сохраняющей семантические зависимости, может служить более простая модель, сохраняющая программный порядок, и рассматривающая гонки на неатомарных обращениях как неопределенное поведение [16, 23]. В этом случае компилятор может использовать оптимальные схемы компиляции и широкий спектр трансформаций к неатомарным обращениям и в то же время предоставлять относительно простую семантику для атомарных обращений.

Выбор модели для языка Kotlin. Рассмотрим в качестве примера Kotlin[11], который является языком программирования общего назначения и еще не имеет стандартизированной модели памяти. На текущий момент Kotlin компилируется в байткод виртуальной машины Java, а также в код JavaScript в нативный код средствами LLVM (для платформ Linux, Windows, macOS, iOS и др.).

Kotlin не ориентирован на системное программирование, то есть он не обязан предоставлять абстракции с нулевой стоимостью для обращений к разделяемой памяти целевой архитектуры. Таким образом, модель памяти, сохраняющая программный порядок или синтаксические зависимости, подходит для Kotlin. Оба подхода ведут к умеренным накладным расходам на время работы программ. Однако модели памяти, сохраняющие программный порядок, лучше подходят для языков, допускающих неопределенное поведение для программ с гонками на неатомарных переменных [23], так как в этом случае можно компилировать неатомарные обращения как обычные инструкции обращения к памяти для архитектур процессоров **Arm** и **POWER**. Несмотря на то, что наличие неопределенного поведения в языке Kotlin в общем нежелательно, на практике это труднодостижимо, так как гонки на неатомарных переменных уже имеют неопределенное поведение в LLVM [6], а LLVM является одной из целевых платформ языка Kotlin.

Среди класса моделей, сохраняющих программный порядок, наиболее хорошо изучена и полна с точки зрения поддержки различных примитивов модель **RC11** [21]. Она является модифицированной версией модели **C11** [5], в которую было добавлено сохранение программного порядка. **RC11** поддерживает надмножество режимов обращения к разделяемой памяти, доступных в **JMM** [3] и ее расширении **JAM** [4]. Эта модель очень близка к моделям памяти JavaScript [7] и LLVM [6], так как обе эти модели основываются на **C11**.

Все это делает модель **RC11** хорошей отправной точкой для разработки модели памяти для Kotlin.

8. ЗАКЛЮЧЕНИЕ

В данной работе мы представили обзор существующих моделей памяти языков программирования. Мы сравнили эти модели по ряду критериев и идентифицировали шесть основных классов моделей памяти. Также, мы предложили рекомендации по выбору эффективной модели памяти для различных языков программирования. Мы надеемся, что наша работа будет полезна для исследователей и разработчиков в области языков программирования и послужит введением в сложную тематику слабых моделей памяти.

На основе нашего анализа мы также можем сделать следующие предположения о будущих направлениях работы в данной области.

Проблема оптимальности схем компиляции и корректности локальных трансформаций кода на сегодняшний день относительно хорошо изучена. Более новые модели, такие как RC11 [21], OCaml MM [22], **Promising** [17, 18] и **Weakestmo** [19], поддерживают широкий диапазон локальных трансформаций и имеют ясные компромиссы относительно оптимальности схем компиляции. Исклю-

[11] https://kotlinlang.org/

чением являются локальные трансформации, использующие циклы или рекурсию, так как их корректность все еще недостаточно изучена. Глобальным трансформациям также уделялось мало внимания, за важным исключением работ [18, 25]. Влияние этих трансформаций на дизайн модели памяти еще предстоит изучить.

Глобальное свойство свободы от гонок на настоящий момент детально изучено [3, 17, 21, 55]. Напротив, локальное свойство свободы от гонок [22] является новой концепцией. Стоит ожидать, что оно, как и другие локальные гарантии [45, 67, 68], будет активно исследоваться в ближайшем будущем.

Смешанные обращения [30], используемые в модели памяти JavaScript [7], а также в некоторых приложениях, например, в кодовой базе ядра Linux [30], на сегодняшний день недостаточно изучены даже в контексте моделей памяти процессоров. Таким образом, более глубокое понимание семантики смешанных обращений является еще одним важным направлением исследований.

Модели памяти, сохраняющие семантические зависимости, по-прежнему являются темой активных исследований [17–20, 67, 68]. Следует ожидать, что они будут более детально разрабатываться и уточняться в ближайшем будущем. Интересным направлением работ в этой области является разработка новых гарантий, помимо свойства свободы от гонок, которые помогут усовершенствовать мета-теорию этих моделей и упростить рассуждение о корректности программ.

Наконец, детальное исследование накладных расходов на время исполнения программ с различными моделями памяти также являются очень важной задачей. Несмотря на наличие здесь некоторого количества работ [8, 22, 23, 33, 40, 41], полная картина все еще остается недостаточно ясной.

БЛАГОДАРНОСТИ

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 20-31-90088.

СПИСОК ЛИТЕРАТУРЫ

1. *Dijkstra E.W.* “Cooperating sequential processes,” in The origin of concurrent programming. Springer, 1968. P. 65–138.
2. *Lamport L.* “How to make a multiprocessor computer that correctly executes multiprocess programs,” IEEE Trans. Computers. 1979. V. 28. № 9. P. 690–691.
3. *Manson J., Pugh W., Adve S.V.* “The Java memory model,” in POPL 2005. P. 378–391, ACM, 2005.
4. *Bender J., Palsberg J.* “A formalization of Java’s concurrent access modes,” Proceedings of the ACM on Programming Languages. 2019. V. 3. № OOPSLA. P. 1–28.
5. *Batty M., Owens S., Sarkar S., Sewell P., Weber T.* “Mathematizing C++ concurrency,” in POPL 2011. 2011. P. 55–66, ACM.
6. *Chakraborty S., Vafeiadis V.* “Formalizing the concurrency semantics of an LLVM fragment,” in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2017. P. 100–110.
7. *Watt C., Pulte C., Podkopaev A., Barbier G., Dolan S., Flur S., Pichon-Pharabod J., Guo S.-y.* “Repairing and mechanizing the JavaScript relaxed memory model,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020. P. 346–361.
8. *Vollmer M., Scott R.G., Musuvathi M., Newton R.R.* “SC-Haskell: Sequential consistency in languages that minimize mutable shared heap,” ACM SIGPLAN Notices. 2017. V. 52. № 8. P. 283–298.
9. *Sewell P., Sarkar S., Owens S., Nardelli F.Z., Myreen M.O.* “x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors,” Commun. ACM. 2010. V. 53. № 7. P. 89–97.
10. *Alglave J., Fox A., Ishtiaq S., Myreen M.O., Sarkar S., Sewell P., Nardelli F.Z.* “The semantics of Power and ARM multiprocessor machine code,” in Proceedings of the 4th workshop on Declarative aspects of multicore programming. 2009. P. 13–24.
11. *Sarkar S., Sewell P., Alglave J., Maranget L., Williams D.* “Understanding POWER multiprocessors,” in PLDI 2011. ACM, 2011. P. 175–186.
12. *Alglave J., Maranget L., Tautschnig M.* “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” ACM Trans. Program. Lang. Syst. 2014. V. 36. № 2. P. 7:1–7:74.
13. *Chong N., Ishtiaq S.* “Reasoning about the ARM weakly consistent memory model,” in Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08). 2008. P. 16–19.
14. *Pulte C., Flur S., Deacon W., French J., Sarkar S., Sewell P.* “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8,” Proceedings of the ACM on Programming Languages. 2018. V. 2. № POPL. P. 1–29.
15. *Flur S., Gray K.E., Pulte C., Sarkar S., Sezgin A., Maranget L., Deacon W., Sewell P.* “Modelling the ARMv8 architecture, operationally: Concurrency and ISA,” in POPL 2016. 2016. P. 608–621, ACM.
16. *Boehm H.-J., Demsky B.* “Outlawing ghosts: Avoiding out-of-thin-air results,” in MSPC 2014. 2014. P. 7:1–7:6, ACM.
17. *Kang J., Hur C.-K., Lahav O., Vafeiadis V., Dreyer D.* “A promising semantics for relaxedmemory concurrency,” in POPL 2017, ACM, 2017.
18. *Lee S.-H., Cho M., Podkopaev A., Chakraborty S., Hur C.-K., Lahav O., Vafeiadis V.* “Promising 2.0: global optimizations in relaxed memory concurrency,” in Proceedings of the 41st ACM SIGPLAN Conference

on Programming Language Design and Implementation. 2020. P. 362–376.

19. *Chakraborty S., Vafeiadis V.* "Grounding thin-air reads with event structures," Proceedings of the ACM on Programming Languages. 2019. V. 3. № POPL. P. 1–28.
20. *Paviotti M., Cooksey S., Paradis A., Wright D., Owens S., Batty M.* "Modular relaxed dependencies in weak memory concurrency," in European Symposium on Programming. P. 599–625, Springer, Cham, 2020.
21. *Lahav O., Vafeiadis V., Kang J., Hur C.-K., Dreyer D.* "Repairing sequential consistency in C/C++11," in PLDI 2017, ACM, 2017.
22. *Dolan S., Sivaramakrishnan K., Madhavapeddy A.* "Bounding data races in space and time," ACM SIGPLAN Notices. 2018. V. 53. № 4. P. 242–255.
23. *Ou P., Demsky B.* "Towards understanding the costs of avoiding out-of-thin-air results," Proceedings of the ACM on Programming Languages. 2018. V. 2. № OOPSLA. P. 1–29.
24. *Jeffey A., Riely J.* "On thin air reads: Towards an event structures model of relaxed memory," in LICS 2016, IEEE, 2016.
25. *Pichon-Pharabod J. and Sewell P.*, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in POPL 2016. 2016. P. 622–633, ACM.
26. *Marlow S. et al.* "Haskell 2010 language report," Available on: https://www. haskell. org/onlinereport/haskell2010, 2010.
27. *Klabnik S., Nichols C.* The Rust Programming Language (Covers Rust 2018). No Starch Press, 2019.
28. *Boehm H.-J., Adve S.V.* "Foundations of the C++ concurrency memory model," ACM SIGPLAN Notices. 2008. V. 43. № 6. P. 68–78.
29. *Lahav O., Giannarakis N., Vafeiadis V.* "Taming release-acquire consistency," ACM SIGPLAN Notices. 2016. V. 51. № 1. P. 649–662.
30. *Flur S., Sarkar S., Pulte C., Nienhuis K., Maranget L., Gray K.E., Sezgin A., Batty M., Sewell P.* "Mixed-size concurrency: ARM, Power, C/C++ 11, and SC," ACM SIGPLAN Notices. 2017. V. 52. № 1. P. 429–442.
31. "C/C++11 mappings to processors," 2011. Available at https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html [Online; accessed 26-April-2021].
32. *Marino D., Singh A., Millstein T., Musuvathi M., Narayanasamy S.* "A case for an SC-preserving compiler," ACM SIGPLAN Notices. 2011. V. 46. № 6. P. 199–210.
33. *Liu L., Millstein T., Musuvathi M.* "A volatile-by-default JVM for server applications," Proceedings of the ACM on Programming Languages. 2017. V. 1. № OOPSLA. P. 1–25.
34. *Muchnick S.* Advanced compiler design and implementation. Morgan kaufmann, 1997.
35. *Lahav O., Namakonov E., Oberhauser J., Podkopaev A., Vafeiadis V.* "Making weak memory models fair," arXiv preprint arXiv:2012.01067, 2020.
36. *Ševčík J., Aspinall D.* "On validity of program transformations in the Java memory model," in European Conference on Object-Oriented Programming. P. 27–51, Springer, 2008.
37. *Wegman M.N., Zadeck F.K.* "Constant propagation with conditional branches," ACM Transactions on Programming Languages and Systems (TOPLAS). 1991. V. 13. № 2. P. 181–210.
38. *Maranget L., Sarkar S., Sewell P.* "A tutorial introduction to the ARM and POWER relaxed memory models," 2012. Available at https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf [Online; accessed 30-April-2021].
39. *Boudol G., Petri G.* "A theory of speculative computation," in European Symposium on Programming. Springer, 2010. P. 165–184.
40. *Singh A., Narayanasamy S., Marino D., Millstein T., Musuvathi M.* "End-toend sequential consistency," in 2012 39th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2012. P. 524–535.
41. *Liu L., Millstein T., Musuvathi M.* "Accelerating sequential consistency for Java with speculative compilation," in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019. P. 16–30.
42. *Marino D., Singh A., Millstein T., Musuvathi M., Narayanasamy S.* "DRFx: A simple and efficient memory model for concurrent programming languages," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2010. P. 351–362.
43. *Demange D., Laporte V., Zhao L., Jagannathan S., Pichardie D., Vitek J.* "Plan B: A buffered memory model for Java," in Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2013. P. 329–342.
44. *Boudol G., Petri G.* "Relaxed memory models: an operational approach," ACM SIGPLAN Notices. 2009. V. 44. № 1. P. 392–403.
45. *Dodds M., Batty M., Gotsman A.* "Compositional verification of compiler optimisations on relaxed memory," in European Symposium on Programming. Springer, 2018. P. 1027–1055.
46. *Doherty S., Dongol B., Wehrheim H., Derrick J.* "Verifying C11 programs operationally," in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. 2019. P. 355–365.
47. *Dang H.-H., Jourdan J.-H., Kaiser J.-O., Dreyer D.* "RustBelt meets relaxed memory," Proceedings of the ACM on Programming Languages. 2019. V. 4. № POPL. P. 1–29.
48. *Alglave J., Maranget L., McKenney P.E., Parri A., Stern A.* "Frightening small children and disconcerting grownups: Concurrency in the Linux kernel," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 2018. P. 405–418.
49. *Zhang Y., Feng X.* "An operational happens-before memory model," Frontiers of Computer Science. 2016. V. 10. № 1. P. 54–81.
50. *Huisman M., Petri G.* "The Java memory model: a formal explanation," VAMP. 2007. V. 7. P. 81–96.
51. *Jagadeesan R., Pitcher C., Riely J.* "Generative operational semantics for relaxed memory models," in European Symposium on Programming. Springer, 2010. P. 307–326.

52. *Sarkar S., Memarian K., Owens S., Batty M., Sewell P., Maranget L., Alglave J., Williams D.* “Synchronising C/C++ and POWER,” in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012. P. 311–322.
53. *Batty M., Memarian K., Owens S., Sarkar S., Sewell P.* “Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER,” ACM SIGPLAN Notices. 2012. V. 47. № 1. P. 509–520.
54. *Vafeiadis V., Balabonski T., Chakraborty S., Morisset R., Nardelli F.Z.* “Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it,” in POPL 2015. ACM, 2015. P. 209–220.
55. *Batty M., Memarian K., Nienhuis K., Pichon-Pharabod J., Sewell P.* “The problem of programming language concurrency semantics,” in ESOP. V. 9032 of LNCS. Springer, 2015. P. 283–307.
56. *Batty M., Donaldson A.F., Wickerson J.* “Overhauling SC atomics in C11 and OpenCL,” in Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2016. P. 634–648.
57. *Nienhuis K., Memarian K., Sewell P.* “An operational semantics for C/C++ 11 concurrency,” in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2016. P. 111–128.
58. *Crary K., Sullivan M.J.* “A calculus for relaxed memory,” in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2015. P. 623–636.
59. *Saraswat V.A., Jagadeesan R., Michael M., von Praun C.* “A theory of memory models,” in Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. 2007. P. 161–172.
60. *Inc S.I., Weaver D.L.* The SPARC architecture manual. Prentice-Hall, 1994.
61. *Lustig D., Trippel C., Pellauer M., Martonosi M.* “ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures,” in Proceedings of the 42nd Annual International Symposium on Computer Architecture. 2015. P. 388–400.
62. *Podkopaev A., Lahav O., Vafeiadis V.* “Bridging the gap between programming languages and hardware weak memory models,” Proceedings of the ACM on Programming Languages. 2019. V. 3. № POPL. P. 1–31.
63. *Pugh W.* “Fixing the Java memory model,” in Proceedings of the ACM 1999 conference on Java Grande. 1999. P. 89–98.
64. *Diwan A., McKinley K.S., Moss J.E.B.* “Type-based alias analysis,” ACM Sigplan Notices. 1998. V. 33. № 5. P. 106–117.
65. *Svendsen K., Pichon-Pharabod J., Doko M., Lahav O., Vafeiadis V.* “A separation logic for a promising semantics,” in Programming Languages and Systems (A. Ahmed, ed.), (Cham). P. 357–384, Springer International Publishing, 2018.
66. *Choi J.-D., Gupta M., Serrano M., Sreedhar V.C., Midkiff S.* “Escape analysis for Java,” Acm Sigplan Notices. 1999. V. 34. № 10. P. 1–19.
67. *Jagadeesan R., Jeffrey A., Riely J.* “Pomsets with preconditions: a simple model of relaxed memory,” Proceedings of the ACM on Programming Languages. 2020. V. 4. № OOPSLA. P. 1–30.
68. *Cho M., Lee S.-H., Hur C.-K., Lahav O.* “Modular data-race-freedom guarantees in the Promising semantics,” in Proceedings of the 42st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2021.

ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 51–61

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

УДК 004.92

МНОГООКОННАЯ ВИЗУАЛИЗАЦИЯ АВИАЦИОННОГО ДИСПЛЕЯ С ИСПОЛЬЗОВАНИЕМ АППАРАТНОГО УСКОРЕНИЯ

© 2021 г. Б. Х. Барладян[a], Н. Б. Дерябин[a], Л. З. Шапиро[a], Ю. А. Солоделов[b], А. Г. Волобой[a,*], В. А. Галактионов[a]

[a] *Институт прикладной математики им. М.В. Келдыша РАН, Москва, Россия*

[b] *Государственный научно-исследовательский институт авиационных систем, Москва, Россия*

** E-mail: voloboy@gin.keldysh.ru*

Поступила в редакцию 13.06.2021 г.
После доработки 16.07.2021 г.
Принята к публикации 22.07.2021 г.

Современный дисплей пилота гражданского самолета основан на новой идеологии интерфейса и позволяет улучшить восприятие полетной информации из нескольких источников за счет ее объединения на одном многофункциональном дисплее. В работе рассматриваются вопросы реализации многооконной визуализации дисплея пилота при использовании OpenGL SC с аппаратным ускорением. Предложен алгоритм компоновки информации на дисплее, позволяющий применять только одно GPU устройство, доступное на борту самолета. Подробно изложен подход адаптации и модификации пакета Mesa с открытым программным кодом для получения сертифицируемого драйвера GPU. Особое внимание уделено технологии адаптации открытых кодов пакета к операционной системе реального времени и к требованиям к системам, критичным для безопасности. Реализация предложенного подхода предназначена для работы под управлением операционной системы реального времени JetOS в системах визуализации бортовых комплексов гражданской авиации. Описанная реализация многооконной визуализации предполагает в дальнейшем ее сертификацию для систем, критичных для безопасности.

DOI: 10.31857/S0132347421060029

1. ВВЕДЕНИЕ

В последние десятилетия значительно усложнились кабины экипажей гражданских самолетов. Современная техника позволяет показывать пилоту больше информации об авиационном оборудовании и полетной ситуации. Увеличение вместимости пассажирских судов и продолжительности полетов в сложных метеоусловиях также увеличивают психологическую нагрузку на пилотов. Это потребовало разработки новой концепции дисплея. Соответственно, и индикаторы, и компоновка панели дисплея должны быть удобными для пользователя, чтобы улучшить взаимодействие между пилотом и летательным аппаратом [1, 2]. В связи с этим вопросы проектирования и внедрения приборов в кабину экипажа имеют большое значение с точки зрения обеспечения безопасной и эффективной работы пилотов.

С внедрением электронных систем управления полетом современный самолет получил “стеклянную кабину”. Эта новая идеология интерфейса позволяет улучшить восприятие полетных данных [3] за счет объединения важной информации на одном многофункциональном дисплее, который обеспечивает интегрированное, легко понятное изображение самолета. В настоящее время информация из нескольких источников визуализируется сразу на одном большом экране (например, приборная панель самолета Boeing 737 MAX, показанная на рис. 1). Кроме того, дисплей можно настраивать и отображать разную информацию в разных сегментах полета.

Современные бортовые системы проектируются на основе IMA (Integrated Modular Avionics) архитектуры [4]. Ключевой особенностью этой архитектуры является возможность выполнения нескольких функциональных приложений, реализующих программную часть систем авионики, на одном компьютере. Для этого необходимо совместное использование времени и ресурсов между приложениями. Этот режим обеспечивается бортовой операционной системой реального времени, под управлением которой и выполняются приложения. В [5] сформулированы требования к операционной системе реального времени (ОСРВ), предназначенной для работы с интегрированной модульной авионикой. В частности, ОСРВ должна соответствовать стандарту ARINC 653 [6]. Необходимость сертификации требует соблюдения

Рис. 1. Приборная панель Boeing 737 MAX.

корректных процессов разработки программного обеспечения (ПО) в соответствии с DO-178C [7], а также полного доступа к исходным кодам. Разрабатываемая нами система визуализации предназначена для работы под ОСРВ JetOS [8].

Важной составляющей системы визуализации бортового дисплея является библиотека OpenGL, которая для использования в авиационном ПО должна удовлетворять стандарту OpenGL SC (Safety Critical). В работах [9, 10] была представлена программная реализация графической библиотеки OpenGL SC, работающая под управлением ОСРВ JetOS. Программную реализацию OpenGL проще сертифицировать в соответствии с описанными выше требованиями, однако программные решения уступают по скорости визуализации аппаратной реализации библиотеки. Скорости визуализации, достигнутые в работах [9, 10], оказались удовлетворительными не для всех типичных авиационных приложений. Более того, для большинства анализируемых приложений удовлетворительные скорости были достигнуты только при использовании четырех ядер процессора. Однако использование четырех ядер процессора не всегда допустимо, поскольку в бортовой системе есть и другие потребители вычислительных ресурсов. Это также создает определенные проблемы для сертификации системы. В силу этих причин представляет большой практический интерес реализация библиотеки OpenGL SC с использованием аппаратного ускорения. Одной из широко используемых авиационных платформ является процессор i.MX6 с графическим ускорителем Vivante, для которых соответственно и необходимо разработать систему визуализации дисплея.

2. АЛГОРИТМ МНОГООКОННОЙ ВИЗУАЛИЗАЦИИ С ИСПОЛЬЗОВАНИЕМ АППАРАТНОГО УСКОРЕНИЯ

В случае, когда каждое приложение может использовать собственный экземпляр OpenGL, задача построения многооконного изображения решается путем использования компоновщика (compositor) [9, 11]. Для разработанной нами программной OpenGL SC мы также реализовали компоновщик [10], при использовании которого на многоядерных компьютерах была достигнута минимально приемлемая скорость визуализации для ряда не очень сложных авиационных приложений. Но проблема в том, что программная OpenGL на используемых в бортовом оборудовании не очень мощных компьютерах не может обеспечить необходимую скорость визуализации для сложных приложений.

При применении OpenGL с поддержкой аппаратного ускорения оба подхода, разработанные в [10] для многооконной визуализации, не могут быть использованы, поскольку они предполагают применение в каждом приложении отдельного экземпляра OpenGL. Такой подход невозможен при использовании аппаратного ускорения, поскольку в типичной авиационной платформе имеется только один графический процессор. Соответственно, не можем использовать отдельные экземпляры OpenGL в каждом разделе или экземпляре JetOS. Для решения этой проблемы мы разработали новый подход, когда все команды OpenGL выполняются в одном специальном разделе операционной системы JetOS.

Работу приложения, использующего библиотеку OpenGL для визуализации, можно разделить на две компоненты:

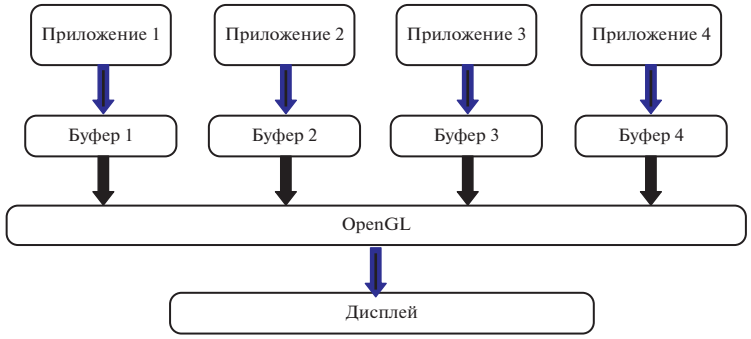


Рис. 2. Схема работы многооконной визуализации с использованием аппаратного ускорения.

1. Подготовка различных данных, необходимых для работы OpenGL – параметры геометрии, различные атрибуты, параметры камеры и т.д.

2. Непосредственный вызов функций OpenGL с подготовленными параметрами.

Для использования аппаратного ускорения в OpenGL в нашей реализации эти компоненты выполняются в разных разделах операционной системы JetOS. Каждое приложение выполняет подготовку необходимых данных в своем разделе и записывает необходимые для визуализации функции OpenGL с соответствующими параметрами в специальный массив. Этот массив находится в общей памяти с разделом, который осуществляет реальные вызовы функций OpenGL. Когда все вызовы, необходимые для генерации изображения одного кадра, подготовлены, то с помощью функций синхронизации запускается работа раздела, работающего с библиотекой OpenGL. Работу системы в целом можно представить с помощью блок-схемы, приведенной на рис. 2.

Код каждого приложения используется практически без изменений. Минимальные изменения на верхнем уровне приложения вносятся только для синхронизации работы приложений и раздела, осуществляющего реальные вызовы функций OpenGL.

Нами были созданы библиотеки OGLOUT и OGLIN. Вместо реальных вызовов функций OpenGL с помощью библиотеки OGLOUT осуществляется кодирование вызовов функций и запись необходимых данных в выходной массив соответствующего буфера. Эти кодирующие функции относительно просты. Если функция просто передает фиксированное число параметров, то в текущую позицию массива записывается индекс вызываемой функции, а за ним подряд значения параметров. Это относится к большинству функций OpenGL таких, как, например, glClearColor(), glClearDepthf() и т.д. В более сложных случаях, когда передаются указатели на массивы неопределенной в момент вызова длины, например glVertexPointer(), технология кодирования вызова несколько усложняется. В момент вызова таких функций вместо указателя записывается индекс зарезервированной позиции в выходном массиве, куда затем будут записываться используемые по данному указателю значения в функциях glMultiDrawArrays() или glDrawElements(). Только эти две функции работают с массивами неопределенной длины.

Декодирование записанной приложениями информации в вызовы функций OpenGL осуществляется с помощью библиотеки OGLIN последовательно для каждого массива данных, сгенерированного соответствующим приложением. Библиотека OGLIN состоит из одной функции process_ogl_input_array(), которая последовательно читает данные, записанные приложением. Затем по прочитанному индексу функции OpenGL переходит к участку кода, написанного для данной функции, извлекает из последующих элементов переданные данные, включая значения указателей, и вызывает с этими параметрами заданную функцию OpenGL.

Некоторым недостатком этого подхода является сложность в реализации функций OpenGL, которые возвращают значения параметров (такие как, например, glGenLists (), glGetError () и др.). Однако анализ практических приложений, применяемых в настоящее время в бортовом оборудовании, показал, что эти функции в них не используются. На практике эти функции используются, как правило, только при отладке приложений при однооконной визуализации. Вызовы этих функций также отсутствуют в библиотеке OGLX [12], которая в настоящее время используется при разработке большинства авиационных приложений с выводом на экран дисплея.

Синхронизация работы приложений и раздела, непосредственно вызывающего функции OpenGL, реализована с помощью специальных объектов, называемых событиями. События – это объекты, которые бывают в двух состояниях: взведен и сброшен. Поскольку эти объекты должны быть доступны одновременно из двух разделов (из приложения и из OpenGL), то они в нашем случае были реализованы через небольшую область памяти, общую для двух разделов. Детально реализация событий описана в работе [10]. Для синхро-

низации работы приложения и OpenGL раздела используется пара событий:

1. StartRender – взводится приложением, когда подготовлены вызовы всех функций OpenGL, необходимых для генерации изображения для данного кадра;

2. EndRender – взводится в OpenGL-разделе, когда генерация изображения для заданного кадра закончена.

Теперь алгоритм работы приложения можно описать следующим алгоритмом:

```
While (true)
{
  WaitEvent(EndRender);
  ResetEvent(EndRender);
  ResetOGLOutput();
  Генерация вызовов OpenGL;
  SetEndOGLOutput();
  SetEvent(StartRender);
}
```

Функция ResetOGLOutput() инициализирует начало работы с выходным массивом, а SetEndOGLOutput() его завершает. Генерация вызовов OpenGL не отличается от ее прямого использования. Только функции OpenGL заменены соответствующими функциями библиотеки OGLOUT.

Работу OpenGL раздела в случае двух приложений можно представить следующим алгоритмом:

```
processed_partitions[0] = false;
processed_partitions [1] = false;
client_num = 2;
while(1)
{
 for (int ic = 0; ic < client_num; ic++)
 {
  if (GetEventState(StartRender [ic]) == SIGNALED)
  {
   ResetEvent(StartRender [ic]);
   process_ogl_input_array(oglinp[ic]);
   processed_partitions[ic] = true;
  }
  if (processed_partitions[0] && processed_partitions [1])
   break;
 }
 if (!processed_partitions[0] || !processed_partitions [1])
   continue;
 SwapBuffers();

 SetEvent(EndRender [0]);
 SetEvent(EndRender [1]);
 processed_partitions[0] = false;
 processed_partitions [1] = false;
}
```

Раздел ждет пока приложения не подготовят свои выходные файлы с данными и последовательностью вызовов функций OpenGL. Как только это произошло (*StartRender* [ic]) = SIGNALED), то запускается вызов этих функций. После того как вызовы функций закончены для всех приложений, вызывается функция SwapBuffers(), которая завершает работу OpenGL для данного кадра и выводит результирующее изображение на экран. После этого взводятся события *EndRender* для всех приложений, и они начинают генерировать вызовы функций OpenGL для следующего кадра.

3. РАЗРАБОТКА GPU ДРАЙВЕРА С ОТКРЫТЫМ КОДОМ

В большинстве случаев драйверы, предоставляемые производителями GPU, не имеют открытого программного кода. Это означает, что такое ПО не может быть сертифицировано для использования в авиации. Также драйвер необходимо адаптировать для работы под управлением ОСРВ JetOS. Реализация различных версий OpenGL SC на базе коммерческих драйверов рассматривается в работах [13–15], однако использование бинарных драйверов в ОСРВ JetOS невозможно.

Для решения этой задачи мы использовали пакет Mesa [16] с открытыми исходными кодами. Mesa представляет собой программную реализацию с открытым исходным кодом OpenGL, Vulkan и других спецификаций графического API. Mesa переводит эти спецификации в драйверы конкретного графического оборудования. Некоторые производители процессоров, такие как, например, AMD или Intel, сами разрабатывают драйверы для пакета Mesa с открытым кодом для производимых ими процессоров. Другие производители, такие как Nvidia или Vivante, полностью заменяют Mesa, обеспечивая собственную реализацию библиотеки OpenGL. Для такого оборудования сообщество разработчиков Mesa создает альтернативные открытые драйверы (reverse-engineering), такие как Nouveau для Nvidia или Etnaviv для Vivante. В нашем случае при использовании платформы i.MX6 с графическим процессором Vivante единственной возможностью является использование драйвера Etnaviv. Первые шаги по использованию пакета Mesa в ОСРВ JetOS, изложены в докладе [17].

Пакет Mesa в настоящее время адаптирован только для использования под управлением ОС Linux, Android и Windows. Для этих операционных систем доступна технология компиляция и интеграция пакета MESA для большинства используемых в настоящее время графических процессоров. Разработка приложений (драйверов и библиотек) для работы под управлением ОСРВ JetOS существенно отличается от технологии разработки соответствующих программных продук-

тов для работы под управлением таких операционных систем как Linux или Windows.

Для использования пакета Mesa в ОСРВ JetOS, удовлетворяющей стандарту ARINC 653, необходимо было решить следующие основные проблемы:

1. В пакете Mesa используется большое количество системных функций ОС Linux, большая часть из которых отсутствует в JetOS. Эти функции необходимо заменить соответствующими функциями JetOS в тех случаях, когда имеются аналоги. В остальных случаях надо реализовать эквивалентные функции, функциональности которых достаточны для наших задач.

2. Динамическое выделение памяти, освобождение памяти запрещено. Выделение памяти разрешается только на этапе инициализации раздела. Это значит, что соответствующие вызовы в пакете Mesa должны быть исключены или переписаны с использованием собственного специального диспетчера памяти. Для выделения памяти можно использовать только функции JetOS.

3. Многопотоковость отсутствует согласно ARINC 653. Это означает, что соответствующие вызовы Mesa, использующие объекты мьютекс (mutex), должны быть исключены или переписаны.

4. В пакете Mesa присутствует большое количество кода, предназначенного для использования в различных операционных системах и для работы с различными графическими процессорами. Весь избыточный код необходимо исключить в соответствии со стандартом DO-178C. Очевидно, что чем меньше размер используемого кода, тем проще процесс сертификации. Размер исходного кода пакета Mesa составляет ~125 мегабайт и содержит ~6000 файлов. Большая его часть не имеет отношения к нашей задаче.

Разработка GPU драйвера с открытым программным кодом происходила в несколько последовательных этапов.

3.1. Адаптация пакета Mesa для ОС JetOS

На этом этапе пакет Mesa портировался в виде одной общей библиотеки **libmesa** с минимальными изменениями, необходимыми для компиляции и сборки графических приложений под управлением ОСРВ JetOS. При этом решались следующие основные задачи:

1. Добавление необходимых компонент. Пакет MESA требует использования внешних пакетов **libdrm** (DRM интерфейс) и **expat** (парсер XML файлов).

2. Исключение заведомо ненужных компонент. Были исключены все драйверы (amd, broadcom, intel и множество других) кроме etnaviv и imx. Также были исключены тесты (например, gtest), поддержка X-Windows (glx) и т.п.

3. Исключение ненужных исходных файлов. Некоторые исходные файлы предназначены для отдельно собираемых программ, таких как тесты, компилятор шейдеров и т.п. Для исключения таких файлов анализировались файлы сборки пакета Mesa.

4. Добавление генерируемых файлов. Пакет Mesa активно использует генерируемые исходные файлы, не входящие в состав поставляемого пакета. Генерируемые файлы создаются во время сборки при помощи специальных скриптов, написанных на языке Python. К счастью, генерируемые файлы слабо зависят от конкретной операционной системы. Поэтому были использованы файлы, полученные при сборке пакета Mesa под операционной системой Linux.

5. Исключение динамической загрузки. Пакет Mesa использует динамическую загрузку для подключения, например, требуемых графических драйверов (в случае i.MX6 – это etnaviv и imx). Так как в JetOS пакет портируется как одна общая статическая библиотека, то динамическая загрузка заменяется прямым вызовом соответствующей функции. Компонента динамической загрузки loader была исключена.

6. Добавление компоненты системной поддержки для JetOS, реализующая необходимые системные функции.

В результате такой адаптации пакета Mesa для работы в ОСРВ JetOS и на платформе i.MX6 была достигнута работа графических приложений с использованием аппаратного ускорения графики. Размер адаптированной библиотеки значительно уменьшился, но остался все еще большим: более 30 мегабайт исходного кода (почти 700 файлов *.c, более 700 файлов *.h, более 100 файлов *.cpp). Причинами этого является реализация большого количества интерфейсов, избыточных для нашей задачи. В частности, Mesa реализует все версии OpenGL с их многочисленными расширениями. А для авиационных приложений необходима поддержка только интерфейсов OpenGL SC стандартов 1.0.1 и 2.0.1. Пакет Mesa также содержит компилятор шейдеров, который согласно стандарту OpenGL SC 2.0.1 должен использоваться только на этапе подготовки приложений. Сами шейдеры должны загружаться на этапе выполнения приложений в двоичном виде.

3.2. Организация структуры драйвера

Для того чтобы получить качественный, поддерживаемый, сертифицируемый результат (драйвер с открытым программным кодом), прежде всего, важна хорошая структуризация всего проекта. К сожалению, исходный пакет Mesa плохо структурирован. Это связано с двумя причинами. Во-первых, пакет поддерживает большое количество интерфейсов графических библиотек: OpenGL ES

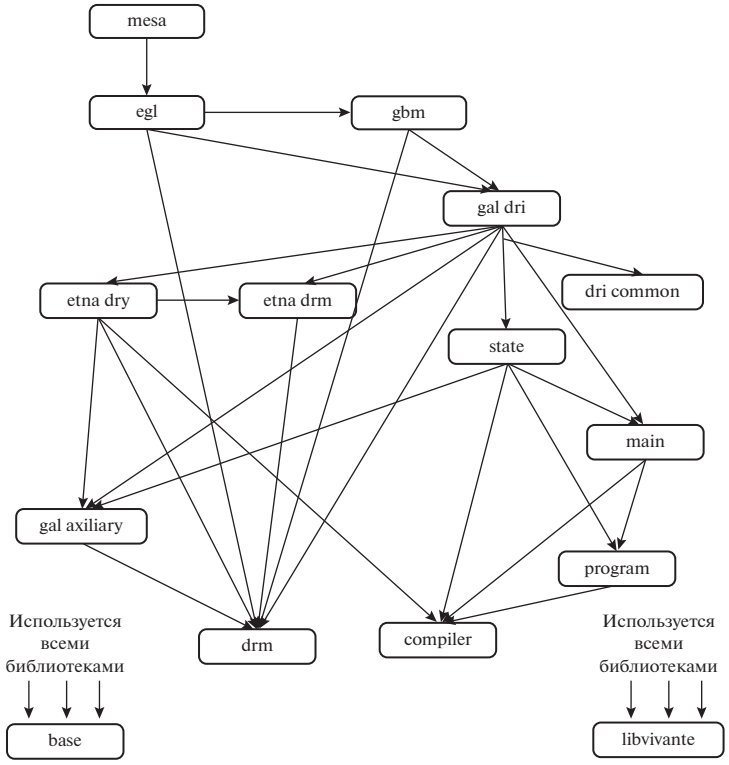


Рис. 3. Иерархия библиотек GPU драйвера.

(версии 1, 2, 3), OpenCL, OpenMAX, VDPAU, VA API, XvMC и Vulkan. Большую часть этих интерфейсов необходимо исключить. Во-вторых, в разработке пакета принимало участие большое количество разработчиков, которые преследовали различные цели и использовали различные программные трюки.

Поскольку нашей целью является поддержка двух стандартов OpenGL SC с достаточно жесткими ограничениями, то в качестве первого шага мы преобразовали адаптированный пакет, полученный на первом этапе, в иерархию библиотек. Под иерархией понимается, что библиотеки образуют дерево по отношению их использования: вышестоящие библиотеки используют функции и данные, определенные в нижестоящих библиотеках, но никак не наоборот. Более точно, для сборки (компиляции) каждой библиотеки в иерархии достаточно использовать только нижестоящие библиотеки. Для достижения этой цели ряд исходных файлов был разбит на части и разнесен по разным библиотекам. Такая же процедура была применена и к соответствующим заголовочным файлам. Чаще всего, одна из частей помещалась в “базовую” библиотеку (не использующую другие библиотеки).

Полученная иерархия библиотек и их зависимости показаны на рис. 3. Структура осталась достаточно сложной, но главный результат состоит в том, что иерархия все же была достигнута.

Для лучшего понимания полученной иерархии опишем функциональности некоторых библиотек на рис. 3:

• **mesa** – головная библиотека. Это один файл, который реализует интерфейс базовых функций, используемых графическими приложениями – инициализация буфера кадров, создания графического контекста, SwapBuffers() и др.

• **egl** – полная реализация стандартного интерфейса OpenGL с базовой оконной системой (EGL). Для работы библиотек OpenGL SC под управлением ОСРВ JetOS большая часть функций EGL не требуется, и они были исключены на следующих этапах работы.

• **gbm** – General Buffer Management. Компонента Linux, необходимая для работы с DRM (Rendering Manager).

• **etna_drv** – расширение драйвера Etnaviv для поддержки аппаратного ускорения.

• **etna_drm** – расширение драйвера Etnaviv для работы с DRM. Работа с буферными объектами и с буфером команд GPU.

• **state** – компонента, отслеживающая изменение состояния рендеринга OpenGL.

• **main** – содержит обработчики всех функций OpenGL.

• **program** – вспомогательная компонента, работающая с программами для шейдеров.

• **compiler** – компилятор шейдеров.

• **drm** – сторонняя библиотека, необходимая для работы с DRM.

• **base** – базовая библиотека, используемая всеми другими библиотеками. В ней собраны различные базовые функции, изъятые из других библиотек.

• **libvivante** – библиотека вывода на экран (Frame buffer library).

Разбивка на библиотеки сделала адаптированный код более управляемым, однако следует отметить, что поток управления в динамике в целом этой разбивке может не соответствовать, так как возможен вызов функций по указателю. Ряд компонент в пакете Mesa использует механизм виртуальных функций (как в C++), и такие вызовы могут не соответствовать иерархии рис. 3. Широко используется интерфейс (struct) pipe_context, через который вызываются аппаратно-зависимые функции драйвера, реализуемые, в основном, компонентой etna_drv, а также интерфейс Driver (struct dd_function_table), через который инициируются изменения состояния рендеринга, реализуемый библиотекой state.

3.3. Диспетчеризация OpenGL функций и генерация кода

Обработчики OpenGL функций в пакете Mesa вызываются через механизм диспетчеризации (dispatch). Имеется таблица обработчиков OpenGL функций, задаваемая указателем _glapi_Dispatch, которая может динамически переустанавливаться при помощи функции _glapi_set_dispatch(). Изначально установлена таблица пустых функций, так как до создания контекста рендеринга функции OpenGL не должны ничего делать. Когда устанавливается контекст рендеринга, устанавливается основная таблица диспетчеризации. Таблица диспетчеризации может также переключаться при выполнении функций glBegin() и glEnd().

Для каждой OpenGL функции gl*Func*() механизм диспетчеризации определяет три функции:

• GET_Func() – получить адрес обработчика функции из таблицы диспетчеризации;

• SET_Func() – установить новый обработчик функции в таблице диспетчеризации;

• CALL_Func() – выполнить обработчик функции, установленный для этой функции в таблице диспетчеризации (макро).

Это дает возможность манипулирования обработчиками для отдельных функций, и Mesa этим активно пользуется. Входные функции OpenGL просто вызывают соответствующий (по номеру) обработчик из текущей таблицы диспетчеризации. Таким образом, достигается гибкая настройка обработчиков OpenGL- функций для текущего состояния рендеринга.

В настоящее время пакет Mesa реализует более 1400 OpenGL функций (не считая синонимов). Для сравнения, спецификации OpenGL SC 1.0.1 и 2.0.1 используют немногим более 100 функций каждая. К тому же механизмы диспетчеризации в пакете Mesa перегружены избыточной для нашей задачи функциональностью и усложнены. Для наших целей эти компоненты было необходимо радикально переписать.

Рассматриваемые компоненты интенсивно используют генерируемые файлы. Как минимум, для каждой OpenGL функции gl*Func*() необходимы определения четырех функций – SET_*Func*(), GET_*Func*(), CALL_*Func*() и точки входа gl*Func*() – с соответствующими аргументами. Код этих функций и другие необходимые описания и определения генерируются в пакете Mesa при помощи специальных скриптов. Исходными файлами для их генерации служит большой набор XML-файлов. Этот набор принципиально настроен на полные стандарты OpenGL и OpenGL ES, а также на внутренние потребности пакета. Поэтому адаптировать этот генератор кода для целей нашей задачи оказалось практически невозможно.

Для решения этой проблемы компонента диспетчеризации была полностью переписана. Был реализован собственный генератор кода для компонент диспетчеризации. Генератор кода в виде скрипта на языке Python использует в качестве входа набор “официальных” заголовочных файлов OpenGL SC заданных стандартов 1.0.1 и 2.0.1. В результате генерируются только необходимые функции.

3.4. Библиотека аппаратного ускорения libhwgl

После реализации генератора кода для компоненты диспетчеризации стало возможным выделение из общей библиотеки libmesa, которая в текущем состоянии поддерживает стандарты OpenGL SC 1.0.1, 2.0.1, ES2 с некоторыми расширениями, библиотеки libhwgl, которая будет поддерживать только необходимые нам стандарты OpenGL SC 1.0.1 и 2.0.1. Для решения этой задачи в генераторе кода в качестве входа использовались только заголовочные файлы OpenGL SC заданных стандартов. Дальнейшая оптимизация кода производилась только для библиотеки libhwgl. Библиотека libmesa была оставлена в “замороженном” виде в основном для целей тестирования приложений, написанных по стандарту OpenGL ES2 под ОСРВ JetOS.

Как отмечалось выше, одной из основных проблем с исходным пакетом при приведении его к состоянию, удовлетворяющему требованиям к сертификации, состоит в его гигантском размере – более 30 мегабайт исходного кода после переноса

в JetOS, практически отсутствующей документации, сложности понимания и сопровождения. Для целей нашего проекта необходимо получить продукт, лишенный избыточного кода, с хорошей структурой, с соблюдением требуемых стандартов программирования и пригодный для будущей сертификации. В соответствии с DO-178C в программном обеспечении бортового оборудования не должно быть неисполняемого кода (Dead code), поэтому одной из важных задач было исключение кода, который не будет использоваться в OpenGL SC стандартов 1.0.1 и 2.0.1.

С этой целью был использован итерационный процесс, на каждом шаге которого достигается частичное снижение сложности и избыточности пакета. Первые шаги уже были сделаны на этапе разработки библиотеки libmesa, когда поддерживаемый ею интерфейс OpenGL был сведен к стандартам ES2, SC 1.0.1 и SC 2.0.1, а обработчики, не входящих в эти стандарты функций, удалены. Основные методы, применяемые в этом процессе, можно условно разбить на два класса – **формальные** и **содержательные.**

Формальные методы – это методы, которые не требуют понимания работы кода, а только понимания общих принципов программирования. Простейшим применением формального метода является удаление неиспользуемой функции. Если какая-то функция не используется – ее описание и определение удаляются.

После удаления неиспользуемой функции могут появиться другие неиспользуемые функции. Неиспользуемые статические функции, как правило, диагностируются компилятором. Для выявления неиспользуемых функций (или части кода) может также применяться такой инструмент JetOS как “Сбор покрытия по коду”. Он позволяет отследить все функции, которые не использовались в данном приложении. Разумеется, эта функциональность должна применяться с осторожностью и не совсем формально, поскольку некоторые функции могут не использоваться в данном приложении, но применяться в других.

Аналогично, можно удалить неиспользуемую переменную или неиспользуемый член структуры. Переменная (поле структуры) не используется, если она (оно) нигде не читается. При этом переменной может что-то присваиваться – это еще не делает ее используемой. Такие присваивания, естественно, тоже удаляются.

Иногда переменной (члену структуры) присваивается только одно значение. В этом случае ее тоже можно удалить, заменив чтения этой переменной ее значением. Формальное удаление неиспользуемой константы (обычно – макро-константы, определенной через #define) возможно, если значение этой константы только читается (сравнивается с чем-либо, используется как альтернатива в switch и т.п.), но ничему не присваивается.

При удалении сравнения с константой в условном операторе (if) условие может стать истинным или ложным и тогда можно удалить неиспользуемые ветви или даже весь условный оператор. Альтернативы в операторе выбора (switch) с использованием данной константы удаляются, удаляется и соответствующая ветвь кода, если для нее не осталось других альтернатив. Если в операторе выбора альтернатив не осталось, удаляется весь оператор выбора (заменяется действиями по умолчанию, если таковые есть). Если в процессе адаптации функция стала пустой, то ее и ее вызовы также можно удалить. Если функция стала возвращать единственное значение, ее тоже можно удалить, заменив вызовы этой функции данным значением. Если в файле не осталось внешних функций и переменных, он удаляется, и т.д.

Содержательные методы адаптации Mesa основаны на полном понимании работы отдельных компонент пакета. Суть содержательного метода состоит в замене кода компонент на более простой, понятный, поддерживаемый и поддающийся формальной верификации код.

В качестве примера применения содержательного метода можно рассмотреть удаление виртуального интерфейса. Исходный пакет Mesa имеет дело с самыми разнообразными драйверами и протоколами, а также с одновременной работой с разными устройствами. Поэтому он часто и оправданно использует виртуальные интерфейсы. Однако для нашей задачи виртуальные интерфейсы не нужны, так как по ним всегда вызывается один и тот же код. Более того, в нашем случае они часто оказываются вредными, так как будучи написанными на языке C с использованием специальных добавочных структур, существенно затрудняют понимание исходного кода (по вызову виртуальной функции бывает сложно определить реально вызываемую функцию). Удаление виртуального интерфейса состоит в замене вызовов виртуальных функций прямыми вызовами. Однако эта операция не так тривиальна, как может показаться, поскольку надо понимать все детали работы кода в данном месте.

Другим примером применения содержательного метода является полная замена компоненты диспетчеризации OpenGL простым эквивалентным кодом, описанная в разделе 3.3.

4. РЕЗУЛЬТАТЫ

Сначала разработанное аппаратное ускорение было протестировано на тех же авиационных приложениях, которые использовались для тестирования программной реализации библиоте-

ки в работе [9]. Это приложения S_PFD (основной дисплей полета SSJ-100), M_PFD (он же для MC-21), Counters, GlassCockpit, FlightDisplay и relief (визуализация рельефа). Сравнивались скорости визуализации на платформе i.MX6 (Wandboard) при использовании программной реализации прототипа OpenGL SC 1.0.1 (SWGL) из [9] и аппаратной реализации OpenGL (HWGL), разработанной нами на базе пакета Mesa. Результаты сравнения приведены в таблице 1. Программная реализация OpenGL позволяет использовать несколько ядер процессора, поэтому в колонке SWGL 1 представлена скорость при использовании одного ядра процессора, а в колонке SWGL 4 при использовании 4 ядер процессора.

Следует отметить, что указанная в таблице для некоторых приложений скорость в 60 кадров в секунду ограничена синхронизацией с частотой обновления дисплея. Скорость работы непосредственно библиотеки HWGL для этих приложений выше. Таким образом, использование аппаратной поддержки для реализации библиотеки OpenGL SC 1.0.1 в операционной системе JetOS позволяют достичь существенного ускорения визуализации при использовании графического процессора Vivante.

Из табл. 1 видно, что достигнутая скорость визуализации для приложения S_PFD недостаточна для практического использования в бортовых системах. Анализ выполнения кода показал, что причиной этого является неэффективный код приложения, сгенерированный с использованием пакета SCADE [12]. Когда нам удалось реорганизовать вызовы OpenGL внутри этого приложения, то скорость визуализации возросла до 60 кадров в секунду. То есть скорость визуализации стала теперь ограничиваться не OpenGL, а синхронизацией с дисплеем.

Разработанный алгоритм многооконной визуализации был протестирован на двух парах практических приложений, которые использовались при тестировании в работе [18]. Это PFD (Primary Flight Display) с DOORS (состояние дверей самолета) – рис. 4, и PFD с ND (Navigation Display) – рис. 5.

Таблица 1. Скорость визуализации библиотеки OpenGL SC 1.0.1 с аппаратной поддержкой и программной реализацией. Скорость указана в кадрах в секунду

	SWGL 1	SWGL 4	HWGL
S_PFD	5.9	13.8	10.8
M_PFD	6.3	15.6	20.0
Counters	23.9	35.4	60
GlassCockpit	10.5	28.7	29.7
FlightDisplay	9.7	26.4	60
SVS	7.7	20.9	60

Таблица 2. Скорость многооконной визуализации программной и аппаратной OpenGL SC библиотек. Скорость указана в кадрах в секунду

	SWGL 1	SWGL 3	HWGL
PFD + DOORS	4.5	6.2	14.9
PFD + ND	4.6	6.2	16.5

В табл. 2 представлено сравнение скоростей многооконной визуализации для программной (SWGL) и аппаратной (HWGL) OpenGL SC библиотек. Визуализация HWGL использует алгоритм, описанный в разделе 2. В столбце SWGL 1 представлена скорость при использовании 1 ядра процессора, в то время как SWGL 3 использует 3 ядра – по одному для каждого приложения и одно для компоновщика [18].

5. ЗАКЛЮЧЕНИЕ

Разработка системы визуализации данных о полете и отображения информации о состоянии самолета обладает своей спецификой, связанной

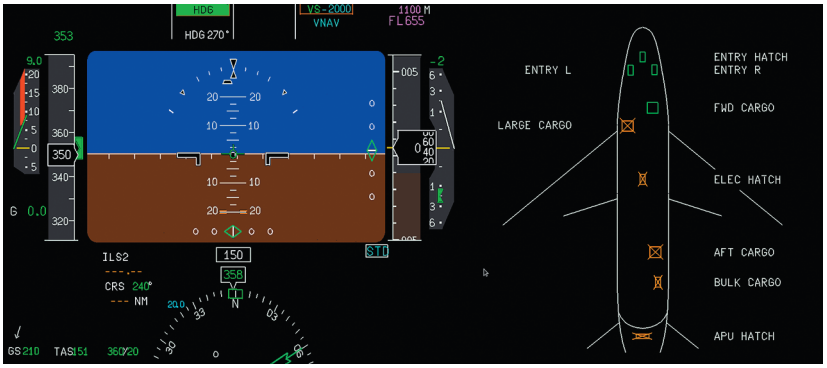


Рис. 4. Многооконная визуализация двух приложений – PFD и DOORS.

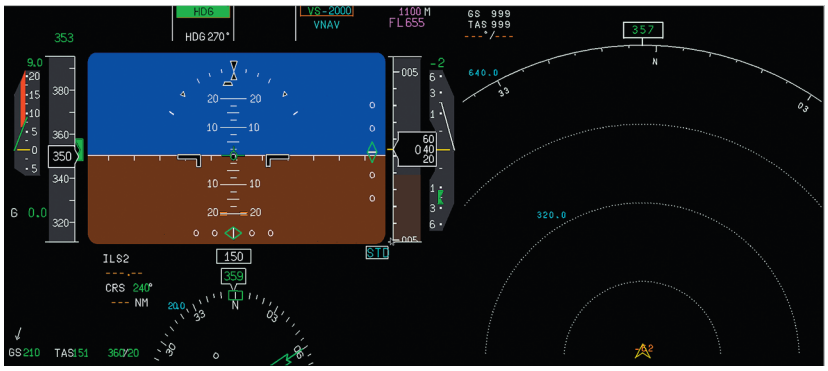


Рис. 5. Многооконная визуализация двух приложений – PFD и ND.

с критически важными вопросами безопасности. Эта специфика часто не позволяет применять готовые, известные решения для той или другой функциональности. Предложенный в работе подход позволяет эффективно реализовать многооконный режим визуализации под управлением ОСРВ JetOS и при использовании аппаратной поддержки. При исследованиях использовалась библиотека OpenGL SC, реализованная путем адаптации пакета Mesa с открытым исходным кодом. Хотя при создании аппаратной OpenGL мы ориентировались на платформу i.MX6 и графический процессор Vivante, но разработанные алгоритмы и подходы могут быть использованы и для другого оборудования.

Представленный в данной статье подход к визуализации (HWGL + новый многооконный алгоритм) в итоге обеспечивает скорость в 2–3 раза превышающую скорость визуализации при использовании программных библиотек. Учитывая, что дисплей пилота предназначен для визуализации информации о полете и состоянии систем самолета, скорость в 15–20 кадров в секунду можно считать приемлемой.

СПИСОК ЛИТЕРАТУРЫ

1. *Senol M.B.* A new optimization model for design of traditional cockpit interfaces, Aircraft Engineering and Aerospace Technology. 2020. V. 92. № 3. P. 404–417. https://doi.org/10.1108/AEAT-04-2019-0068
2. *Thomas P., Biswas P., Langdon P.* State-of-the-Art and Future Concepts for Interaction in Aircraft Cockpits, Lecture Notes in Computer Science. 2015. V. 9176. P. 538–549. https://doi.org/10.1007/978-3-319-20681-3_51
3. *Kal'avsky P., Rozenberg R., Mikula B., Zgodavova Z.* Pilots' Performance in Changing from Analogue to Glass Cockpits, In Proc. of the 22nd Int. Scientific Conf. on Transport Means (Transport Means). 2018. P. 1104–1109.
4. *Федосов Е.А.* Проект создания нового поколения интегрированной модульной авионики с открытой архитектурой. Полет. 2008. № 8. С. 15–22.
5. *Федосов Е.А., Ковернинский И.В., Кан А.В., Солоделов Ю.А.* Применение операционных систем реального времени в интегрированной модульной авионики. OSDAY 2015, http://osday.ru/solodelov.html
6. *Ananda C.M., Nair S., Mainak G.* ARINC 653 API and its application – An insight into Avionics System Case Study. Defence Science Journal. V. 63. № 2. P. 223–229. https://doi.org/10.14429/dsj.63.4268
7. DO-178C Software Considerations in Airborne Systems and Equipment Certification. http://www.rtca.org/store_product.asp?prodid=803
8. *Маллачиев К.М., Пакулин Н.В., Хорошилов А.В.* Устройство и архитектура операционной системы реального времени. Труды ИСП РАН. 2016. Т. 28. Вып. 2. С. 181–192. https://doi.org/10.15514/ISPRAS-2016-28(2)-12
9. *Барладян Б.Х., Волобой А.Г., Галактионов В.А., Князь В.В., Ковернинский И.В., Солоделов Ю.А., Фролов В.А., Шапиро Л.З.* Эффективная реализация OpenGL SC для авиационных встраиваемых систем // Программирование. 2018. № 4. С. 3–10. https://doi.org/10.31857/S013234740000519-5
10. *Барладян Б.Х., Шапиро Л.З., Малачиев К.А., Хорошилов А.И., Солоделов Ю.А., Волобой А.Г., Галактионов В.А., Ковернинский И.В.* Система визуализации для авиационной ОС реального времени JetOS. Труды Института системного программирования РАН. 2020. Т. 32. Вып. 1. С. 57–70. https://doi.org/10.15514/ISPRAS-2020-32(1)-3
11. EGL_EXT_compositor *http://www.coreavi.com/sites/default/files/coreavi_product_brief_-_egl_ext_compositor.pdf.*
12. Ansys SCADE Display Capabilities https://www.ansys.com/products/embedded-software/ansys-scade-display/scade-display-capabilities.
13. *Baek N. and Lee H.* OpenGL ES 1.1 Implementation Based on OpenGL, Multimedia Tools and Applications. 2012. V. 57. № 3. P. 669–685.
14. *Baek N., Lee H.* OpenGL SC Implementation over an OpenGL ES 1.1 Graphics Board, 2012 IEEE International Conference on Multimedia & Expo Workshops (ICMEW 2012). P. 671–671. https://doi.org/10.1109/ICMEW.2012.127

15. *Baek N. and Kim K.J.*, Design and implementation of OpenGL SC 2.0 rendering pipeline. Cluster Computing. 2019. V. 22. P. S931–S936. https://doi.org/10.1007/s10586-017-1111-1
16. The Mesa 3D Graphics Library. https://www.mesa3d.org/
17. *Barladian B., Deryabin N., Voloboy A., Galaktionov V., Shapiro L.* High Speed Visualization in the JetOS Aviation Operating System Using Hardware Acceleration, CEUR Workshop Proceedings. 2020. V. 2744. Proc. of the 30th International Conference on Computer Graphics and Vision. pp. short3-1–short3-9. https://doi.org/10.51130/graphicon-2020-2-4-3
18. *Barladian B.Kh., Shapiro L.Z., Mallachiev K.M., Khoroshilov A.V., Solodelov Y.A., Voloboy A.G., Galaktionov V.A., Koverninskiy I.V.* Multi-windows rendering using software OpenGL in avionics embedded systems // CEUR Workshop Proceedings. V. 2485. Proc. of the 29th International Conference on Computer Graphics and Vision, 2019. P. 28–31. https://doi.org/10.30987/graphicon-2019-2-28-31

ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 62–80

АНАЛИЗ ДАННЫХ

УДК 004.9

ПОИСК УЯЗВИМОСТЕЙ НЕБЕЗОПАСНОГО ИСПОЛЬЗОВАНИЯ ПОМЕЧЕННЫХ ДАННЫХ В СТАТИЧЕСКОМ АНАЛИЗАТОРЕ SVACE

© 2021 г. А. Е. Бородин[a,*], А. В. Горемыкин[a, b,], С. П. Вартанов[a,***], А. А. Белеванцев[a, b,****]**

[a] *Институт системного программирования им. В.П. Иванникова РАН, 109004 Москва, ул. А. Солженицына, д. 25, Россия*

[b] *Московский государственный университет имени М.В. Ломоносова, 119991 Москва, Ленинские горы, д. 1, Россия*

**E-mail: alexey.borodin@ispras.ru*

***E-mail: alexey.goremykin@ispras.ru*

****E-mail: svartanov@ispras.ru*

*****E-mail: abel@ispras.ru*

Поступила в редакцию 05.07.2021 г.
После доработки 16.07.2021 г.
Принята к публикации 22.07.2021 г.

В статье рассматривается поиск ошибок помеченных данных в исходном коде программ, т.е. ошибок, вызванных небезопасным использованием данных, полученных из внешних источников, которые потенциально могут быть изменены злоумышленником. В качестве основы использовался межпроцедурный статический анализатор Svace. Анализатор осуществляет как поиск дефектов в программе, так и поиск подозрительных мест, в которых логика программы может быть нарушена. Целью является найти как можно больше ошибок при приемлемой скорости и низком уровне ложных срабатываний (<20–35%). Для поиска ошибок Svace с помощью компилятора строит низкоуровневое типизированное промежуточное представление, которое подается на вход основному анализатору SvEng. Анализатор строит граф вызовов, после чего выполняет анализ на основе резюме. При таком анализе функции обходятся в соответствии с графом вызовов, начиная с листьев. После анализа функции создается ее резюме, которое затем будет использовано для анализа инструкций вызова. Анализ имеет как высокую скорость, так и хорошую масштабируемость. Внутрипроцедурный анализ основан на символьном выполнении с объединением состояний в точках слияния путей. Для отсеивания несуществующих путей для некоторых детекторов может использоваться SMT-решатель. При этом SMT-решатель вызывается, только если есть подозрение на ошибку. Анализатор был расширен возможностью поиска дефектов, связанных с помеченными данными. Детекторы реализованы в виде плагинов по схеме источник-приемник. В качестве источников используются вызовы библиотечных функций, получающих данные извне программы, а также аргументы функции main. Приемниками являются обращение к массивам, использование переменных как шага или границы цикла, вызов функций, требующих проверенных аргументов. Реализованы детекторы, покрывающие большинство возможных типов уязвимостей, для непроверенных целых чисел и строк. Для оценки покрытия использовался проект Juliet. Уровень пропусков составил от 46.31% до 81.17% при незначительном количестве ложных срабатываний.

DOI: 10.31857/S0132347421060030

1. ВВЕДЕНИЕ

В статье описывается реализация поиска ошибок помеченных данных с помощью статического анализатора Svace [1–4].

К наиболее важным особенностям анализатора Svace можно отнести следующее.

• Межпроцедурный анализ на основе резюме, при котором функции обходятся по графу вызовов, начиная с листьев. Каждая функция анализируется только один раз.

• Неконсервативный анализ. За счет отказа от консервативности анализ получает более высокую точность и производительность.

• Анализ внутри одной функции основан на анализе значений. Анализ отслеживает значения переменных и ячеек памяти и ассоциирует большинство свойств со значениями переменных.

В разд. 2 описывается, какие виды ошибок мы будем искать, в разд. 3 и 4 описывается устройство статического анализатора Svace и модуля

SvEng соответственно. Разд. 5 посвящен реализации анализа помеченных данных, и разд. 6 содержит оценку результатов для проектов Juliet и Tizen 6.

2. ПОМЕЧЕННЫЕ ДАННЫЕ

В статье рассматриваются ошибки небезопасного использования данных, не контролируемых программой, т.е. полученных из внешних источников. Такие данные могут быть изменены злоумышленником. Если эти данные используются без соответствующей проверки, то в программе присутствует уязвимость.

Неконтролируемыми данными являются данные из файлов, пользовательский ввод, данные, переданные по сети. Мы рассматриваем два вида помеченных данных: помеченные целые числа и помеченные строки.

Ниже перечислены виды уязвимостей, связанных с помеченными данными.

• При использовании для доступа к массиву происходит переполнение буфера, что может позволить злоумышленнику захватить контроль над устройством [5]. По данным национальной базы уязвимостей США (NVD) ошибки подобного рода являются причиной 9.49% всех уязвимостей, занесенных в базу CWE в 2018 году [6].

• Использование в циклах. Если помеченные данные используются в качестве ограничения цикла, то цикл может выполниться большее количество раз, чем предусмотрено. Это может приводить как к излишнему расходованию процессорного времени, так и к другим ошибкам, например, переполнению массива. Если помеченные данные используются в качестве шага итератора цикла, то можно так подобрать данные, чтобы цикл стал бесконечным.

• Использование для некоторых операций, например, для выделения памяти. Злоумышленник может заставить программу выделить излишне много памяти.

Листинг 1 иллюстрирует уязвимости, связанные с целыми числами. Для исправления уязвимости переполнения буфера, необходимо проверить диапазон переменной n, чтобы он принадлежал интервалу [0; 99]. Безопасный диапазон для других видов уязвимостей зависит от логики программы.

Помеченные строки могут как содержать произвольные символы, так и иметь произвольную длину. При копировании такой строки в массив фиксированного размера, может происходить его переполнение.

Фрагмент кода на листинге 2 иллюстрирует уязвимости с помеченными строками.

3. СТАТИЧЕСКИЙ АНАЛИЗАТОР SVACE

Анализатор Svace осуществляет поиск дефектов в исходном коде; при этом поиск происходит как для ошибок, возможных при выполнении программы, так и подозрительных мест, где, возможно, логика программы нарушена. Целью является найти как можно больше дефектов при приемлемом времени работы и высоком качестве результатов.

Инструмент может как пропускать реальные дефекты, так и выдавать ложные срабатывания, не соответствующие реальным дефектам. Для анализируемой программы не требуется никакой подготовки, достаточно, чтобы исходный код компилировался. Время анализа сравнимо со временем компиляции программы. Для больших программ целесообразно использовать Svace во время ночной сборки.

Задача поиска дефектов является алгоритмически неразрешимой [7], т.е. нельзя найти все ошибки определенного типа в произвольной программе и не выдать ложных срабатываний. Для решения этой задачи используются всевозможные компромиссы. Одним из подходов является поиск приближенного решения. Возможны два вида приближений.

• Приближать в сторону отсутствия ложных срабатываний. В этом случае выдаются только истинные сообщения, но возможен пропуск множества дефектов. Типичным примером является выдача ошибок компилятора, для которого недопустима возможность отказаться компилировать корректную программу.

• Поиск всех ошибок определенного типа. Процент ложных срабатываний при этом может быть высоким. Более качественный анализ может достигаться за счет существенного увеличения времени работы.

Такие приближения называются консервативными, так как они всегда округляют решение в одну сторону. В Svace не используется консервативное приближение. Это позволяет как ускорить анализ программ, так и существенно повысить уровень истинных срабатываний.

В статье [8] утверждается, что консервативный анализ алиасов необходим для оптимизации программ, но является опциональным для анализаторов. Когда авторы потоково- и контекстно-чувствительного анализа [9, 10] удалили один шаг, необходимый для обеспечения консервативности, общее время анализа значительно сократилось. В одном случае время уменьшилось от нескольких дней до нескольких минут.

На рис. 1 иллюстрируются возможные подходы. По оси ординат показан процент найденных ошибок, по оси абсцисс процент истинных. Подход, используемый в Svace, позволяет максими-

```
char buf[100];


int n;
scanf("%d", &n); //n - tainted


//переполнение буфера, если n меньше нуля
либо больше 99
buf[n] = 0;


//выделение памяти
char*p = (char*)malloc(n * sizeof(struct
Fmt));
int i = 99;


while(i > 0) {
  buf[i] = '0' + (i % 10);
  //бесконечный цикл, если n равно 0
  i -= n;
}
```

Листинг 1. Помеченные целые числа
Listing 1. Tainted integers

зировать площадь прямоугольника, показанного пунктирными линиями.

Подобный подход достаточно популярный и не является ноу-хау, используемым в Svace. В статье [11] предлагается термин *soundy*, означающий в целом консервативный анализ, который допускает отказ от консервативности для некоторых конструкций.

2.1. Архитектура Svace

Для анализа программы анализатору требуется исходный код и скрипт сборки. Svace осуществляет поиск дефектов в программах на языках C, C++, Java, Kotlin, Go[1].

Типичная схема анализа показана на рис. 2. Svace перехватывает команды запуска компилятора и компоновки [12]. Затем запускается модифицированный компилятор, который строит абстрактное синтаксическое дерево (АСД) и запускаются детекторы для поиска ошибок на АСД, а также генерируется промежуточное представление программы для последующего анализа. Промежуточное представление подается на вход анализатору SvEng[2], выполняющему межпроцедурный анализ.

Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил. С помощью АСД-анализаторов можно найти подозрительные паттерны на деревьях разборов, различные опечатки, но при этом

[1] Первый релиз Svace, поддерживающий язык Go, ожидается в январе 2021 г.

[2] Сокращение от **Sv**ace **Eng**ine – движок анализатора Svace.

```
char*p = getenv("aaa");

//потенциальное
переполнение буфера
//размер p может быть
меньше 10
char x = p[10];

char buf[10];
int n = *((int*)p);
//переполнение буфера
buf[n] = 0;
```

Листинг 2. Помеченные строки
Listing 2. Tainted strings

• сложно отследить зависимости между переменными,

• сложно выполнить анализ указателей,

• сложно сделать межпроцедурный и межмодульный анализ.

Поскольку поиск ошибок помеченных данных, как правило, требует анализа вышеперечисленных свойств, мы не будем реализовывать детекторы помеченных данных на этом уровне.

4. АНАЛИЗАТОР SvEng

4.1. Общая схема

Анализатор SvEng осуществляет глубокий межпроцедурный потоково- и контекстно-чувствительный анализ. Наиболее важными являются следующие особенности анализатора SvEng.

• Межпроцедурный анализ на основе резюме, при котором функции обходятся по графу вызовов начиная с листьев. Каждая функция анализируется только один раз. По завершению анализа функции создается ее резюме, которое затем будет использовано при анализе вызова функции. Резюме содержит описание свойств функции. Анализ сохраняет баланс между компактностью анализа и точностью описания семантики функции.

• Неконсервативный анализ. За счет отказа от консервативности анализ получает лучшую точность и более высокую производительность.

• Анализ внутри одной функции основан на анализе значений. Анализ отслеживает значение переменных и ячеек памяти и ассоциирует большинство свойств со значениями переменных.

• Запуск всех анализов одновременно. Невысокая стоимость одного детектора.

Дизайн SvEng позволяет находить множество типов дефектов: разыменование нулевых указателей, недостижимый код, переполнение массива, утечки памяти и ресурсов, неправильное использование библиотечных функций, двойные блокировки мьютексов.

Мы реализовали поиск уязвимостей помеченных данных таким образом, чтобы по максимуму использовать возможности анализатора. Детекторы реализованы в виде анализа источник-приемник, где источником являются функции, возвращающие помеченные данные, а приемником – операции, в которых эти данные должны быть проверены перед использованием. Анализатор хорошо обнаруживает ошибки, где источник и приемник не сильно удалены друг от друга на графе вызовов.

Анализ производится по следующем схеме:

1 – на вход анализатору подаются файлы с промежуточным представлением;

2 – строится граф вызовов;

3 – запускается предварительная фаза анализа, на которой доступны легковесные анализы; на этой фазе, в том числе, собирается информация об указателях на функцию и виртуальных вызовах;

4 – достраивается граф вызовов для виртуальных функций и вызовов по указателю;

5 – запускается основная фаза анализа.

4.2. Промежуточное представление

Для анализа используется собственное промежуточное представление, которое строится уже после запуска анализатора (Svace IR). На вход анализатору подаются файлы в промежуточном представлении, зависящие от анализируемого языка:

• LLVM-файлы для C/C++,

• Java-байт код для Java/Kotlin,

• собственный JSON-формат для Go.

После конвертации исходного кода в представление Svace IR, анализ производится одинаково для всех языков программирования.

Svace IR является низкоуровневым типизированным языком в SSA-форме. Язык имеет процедуры, в том числе возможность их вызова по указателю. Для моделирования виртуальных вызовов в C++ явно строятся виртуальные таблицы. Для моделирования исключений используются конструкции goto.

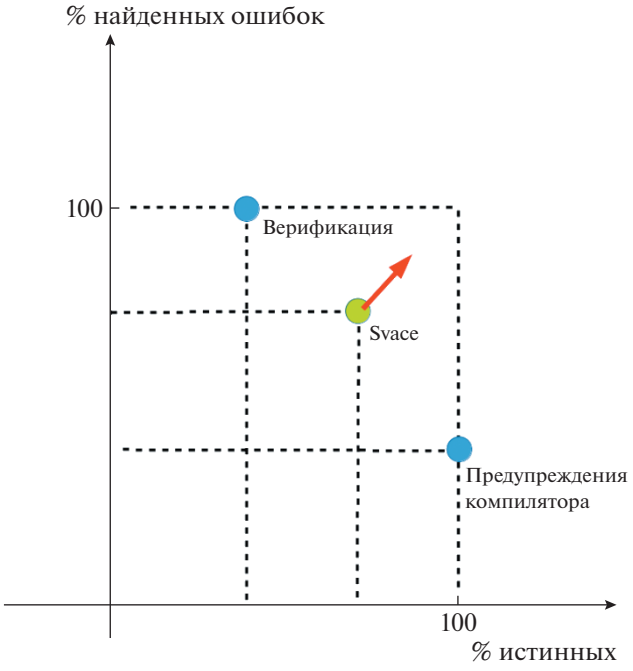


Рис. 1. Подходы к обнаружению ошибок.

Листинг 3 содержит пример кода на C. Соответствующий фрагмент на Svace IR приведен на листинге 4.

Использование низкоуровневого промежуточного представления имеет как свои плюсы, так и минусы. Недостатки такого представления:

• затруднен анализ высокоуровневых конструкций;

• нет информации об оригинальных синтаксических конструкциях;

• трансформация представления в эквивалентное компилятором.

Основным преимуществом является простота анализа. Семантика точно моделируется компилятором, язык имеет небольшое количество инструкций, которые редко меняются. Значительное количество языковых конструкций являются синтаксическим сахаром (исключения, конструкторы и деструкторы, циклы и др.) и не требуют добавления новых инструкций в IR.

Низкоуровневое IR для задачи поиска помеченных данных представляется хорошим выбором, т.к. для этого вида дефектов важна принципиальная возможность эксплуатации уязвимости, которая не меняется при переходе на низкоуровневый язык. Анализ синтаксических конструкций при этом не очень важен.

4.3. Межпроцедурный анализ

Используется межпроцедурный анализ на основе резюме.

При таком анализе для каждой функции строится резюме – краткое описание поведения функции. Резюме используется для анализа инструкции вызова функции и позволяет избежать повторного анализа тела функции. Резюме создается после анализа функции.

На рис. 3 показан граф вызовов для сборки двух программ. Наверху графа вызовов находятся функции main, которые вызывают другие функции. Листьями графа являются функции h и j, анализ начинается с листьев графа. После того как они будут проанализированы, станут доступны их резюме и возможность начать анализ функции foo.

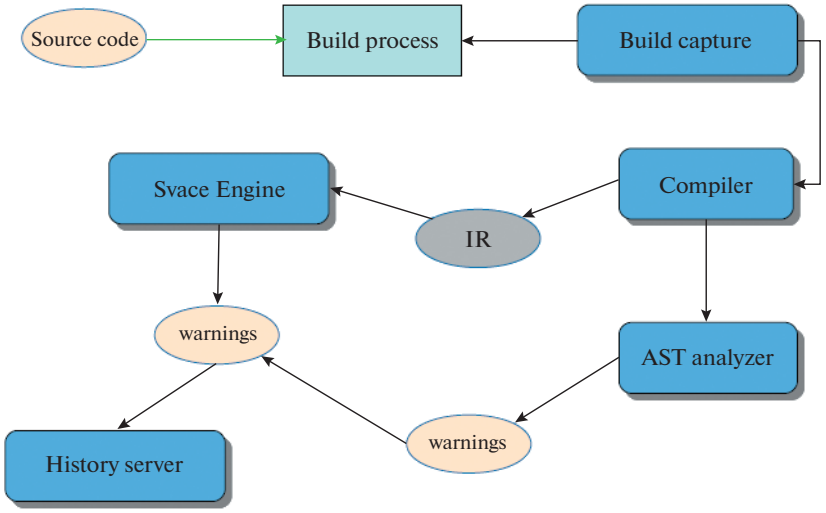


Рис. 2. Схема анализа.

```
int calc(int f)
{
    int a, b;
    int*p;
    a = 1;
    b = 2;
      p = f ?
&a : &b;

    int x = *p;
    *p = 3;

    int y = *p;

     return x +
y;
}
```

Листинг 3. Код на C
Listing 3. C code

Анализ имеет как высокую скорость, так и хорошую масштабируемость. Последняя достигается за счет того, что резюме создается достаточно компактным и не включает все детали поведения функции. Возможно ограничивать размер резюме. В этом случае резюме не будет описывать все интересующие эффекты вызова функции, но анализ вызова функции будет иметь константное время.

Благодаря своим преимуществам, этот анализ является довольно популярным и используется во многих инструментах статического анализа: Prefix [13], Saturn [14], Calysto [15], CSharpChecker [16].

4.4. Предварительная фаза

При использовании только анализа на основе резюме каждая функция посещается ограниченное количество раз. При этом для ряда анализов необходима некоторая информация обо всей программе, либо о функциях, находящихся выше по графу вызовов.

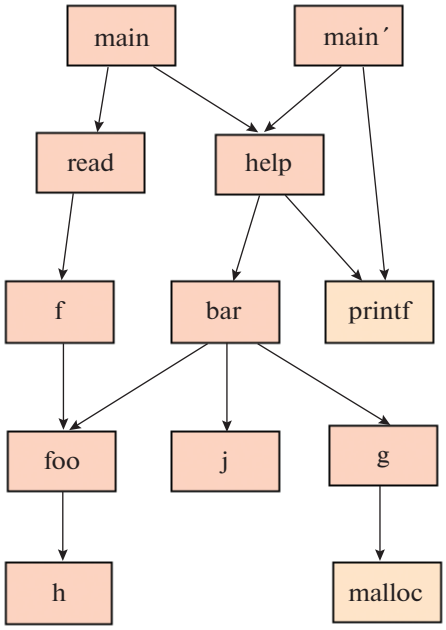


Рис. 3. Пример графа вызовов.

Для получения такой информации была создана предварительная фаза, которая включает следующие виды анализов:

- анализ присваиваний значений указателям на функции,
- анализ заполнения виртуальных таблиц C++,
- анализ иерархии классов для Java,
- анализ значений глобальных переменных.

Эти анализы на вход получают информацию о глобальных переменных и инструкции для каждой функции. Анализ производится параллельно для разных модулей.

Анализ не учитывает порядок инструкций, но доминирующая инструкция всегда анализируется первой. Потоково-нечувствительный анализ был выбран, чтобы не слишком замедлять время общего анализа, т.к. основные свойства будут проанализированы во время основного анализа.

Цель данных анализов – получить необходимую информацию, не затрачивая существенное количество времени.

4.5. Девиртуализация

Цель девиртуализации разрешить вызов процедур для указателей на функции и виртуальных таблиц. Таблица виртуальных функций представляет собой глобальную переменную типа структуры с полем – массивом, содержащим указатели на виртуальные функции, доступ к которым осуществляется через константные значения.

```
def calc(int f) int
{
    a = alloca();
    b = alloca();
     p = alloca ref
int();
    store 1, a;
    store 2, b;

    if(f != 0)
          goto true-
label;
    else
         goto false-
label;

true-label:
    store a, p;
    goto label2;
```

```
false-label:
    store b, p;
    goto label2;

label2:
    t1 = load p;
    x = load t1;
    store 3, t1;
    t2 = load p;
    y = load t2;
    t3 = x + y;
    ret t3;
}
```

Листинг 4. Код на Svace IR
Listing 4. Svace IR code

Анализ отслеживает значения переменных, структур и константные индексы массивов.

Для каждого указателя собираются все возможные записи. Результатом анализа для каждой пары “функция – указатель” является множество функций, которые могут быть вызваны, либо пустое множество, если анализ не смог определить всех кандидатов.

Результаты анализа используются двумя способами:

• при достраивании графа вызовов для добавления ребер,

• при обработке инструкции вызова по указателю.

Анализу на основе резюме нужна информация о графе вызовов. Поэтому после получения информации о виртуальных функциях достраивается граф вызовов. Эта процедура не является сложной, достаточно просто добавить ребро, где **может** быть вызвана функция в результате виртуального вызова. При анализе функции на основном анализе эту информацию в некоторых случаях можно будет уточнить. Более точный анализ на предварительной фазе добавит меньшее количество лишних ребер. До тех пор, пока основной анализ в состоянии убрать лишнее ребро, они не влияют на точность анализа. Но время анализа может увеличиваться, а потребление памяти возрастать, т.к. каждое ребро в графе вызовов накладывает ограничения на возможный обход функций и требует наличия резюме.

В основном анализаторе реализован плагин, отслеживающий возможных кандидатов для каждого указателя. Этот плагин получает данные о

кандидатах для каждого указателя, а затем используя потоково-чувствительный анализ распространяет эти данные внутри процедуры независимо. В некоторых случаях, используя информацию о значениях переменных и недостижимых инструкциях, анализ может получить более точную информацию, чем та, которая доступна после предварительной фазы.

Для случая, если указателю соответствуют несколько кандидатов, моделируется их условный вызов. Для каждого кандидата применяется резюме, а затем происходит объединение для каждого контекста использования. Таким образом каждое резюме применяется отдельно для каждого контекста.

4.6. Внутрипроцедурный анализ

Для анализа отдельной функции используется символьное выполнение с объединением состояний анализа в точках слияния путей. Для функции строится граф потока управления, и анализ начинает обход графа. С ребрами графа ассоциируются абстрактные состояния. На каждом шаге для инструкции по абстрактному состоянию на входных ребрах формируется абстрактное состояние на выходном ребре.

Для сильно связанных компонент (ССК) выполняются несколько итераций. Абстрактные состояния для выходных ребер ССК сохраняются после каждой итерации. После завершения анализа ССК, эти состояния объединяются. Во время анализа используются несколько эвристик для моделирования всевозможных путей выполнения ССК. В некоторых случаях моделирование происходит некорректно, если эвристики не сработали.

Все дополнительные анализы и детекторы запускаются одновременно, что позволяет уменьшить одновременно и время анализа, и потребляемую память. Время анализа сокращается за счет того, что общие для всех детекторов свойства анализируются только один раз. А потребление памяти уменьшается, т.к. в большинстве случаев после анализа инструкции, состояние анализа на входном ребре может быть удалено.

Для моделирования значений переменных и ячеек памяти Svace использует абстракцию, названную идентификатором значений. Двум переменным сопоставляется один идентификатор значений, если при выполнении эти переменные будут иметь одинаковые значения (решается задача нумерации значений).

Большинство свойств ассоциируются с идентификаторами значений. Сами свойства также можно описывать с помощью идентификаторов значений. Для описания свойств используются атрибуты. Атрибут обозначает некоторое анализируемое свойство.

Примеры атрибутов:

- Null – значение указателя имеет нулевое значение,
- ValueInterval – значения целочисленной переменной принадлежат отрезку [a;b],
- PointsTo – указатель указывает на ячейки памяти из множества. Сами ячейки памяти обозначаются идентификатором значений, который моделирует адреса ячеек памяти,
- Ness – необходимые условия достижения ребра в графе потока управления. Представляет собой формулу булевой логики, где в качестве переменных используются идентификаторы значений.

4.7. Использование SMT-решателя

Для реализации чувствительности к путям в Svace используются условные атрибуты, которые выражают свои свойства в виде формул булевой логики над идентификаторами значений. Условные атрибуты, описывающие свойства значений переменных, ассоциируются с идентификаторами значений.

Формулы имеют вид, показанный на рис. 4, где *Val* – идентификаторы значений, *Const* – константы. Для каждой литеральной формулы *Atom* определена инструкция отрицания, возвращающая другую литеральную формулу. В формулах могут использоваться конъюнкции, дизъюнкции от других формул, а также отрицание для литеральных формул.

SMT-решатель запускается только перед выдачей предупреждения для того, чтобы отсеять несуществующие пути. Таким образом SMT-решатель запускается не больше, чем количество неотфильтрованных предупреждений. В общем случае используется следующая формула:

$$Ness, \wedge Error_i,(v),$$

где *Ness* необходимое условие достижимости для ребра i, $Error_i,(v)$ – условие ошибки для значения, моделируемого идентификатором v.

4.8. Анализ потока данных

Для анализа недостижимого кода был реализован консервативный анализ потока данных (АПД[3]) [17]. Этот анализ запускается перед началом анализа каждой функции в основном анализе. Анализ помечает недостижимые ребра на графе потока управления. Основная причина, по которой он был реализован – недостаточная точность основного анализа. Недостижимое ребро сильно влияет на все остальные анализы, поэтому неконсервативность

[3] Под АПД в статье мы будем подразумевать движок анализа, основанный на анализе потока данных.

$\bowtie ::= > | < | = | \neq | \leq | \geq$
$\oplus ::= + | - | * | /$
Op ::= Val | Const
Atom ::= True | False | Op $\bowtie$ Op | Op = Op $\oplus$ Op
Conj = Atom | Conj $\wedge$ Conj
CFormula ::= Conj
SFormula ::= Conj $\wedge$ $\overline{\text{Conj}}$
FFormula ::= Atom | FFormula $\wedge$ FFormula | FFormula $\vee$ FFormula

Рис. 4. Используемые формулы.

в данном случае может приводить к значительно худшим результатам.

Позже были добавлены другие анализы:

- анализ интервалов,
- анализ исключений,
- анализ живых переменных.

Помимо получения консервативных результатов, этот анализ позволяет получить данные о функции перед запуском основного анализа. Поскольку в основном анализе все детекторы запускаются одновременно, возникает проблема предварительного получения свойств, которая решается АПД.

4.9. Анализ сверху вниз на предварительной фазе

Рассмотрим пример, показанный на листинге 5. Функция f получает помеченные данные и передает их в функцию g. Если функция g небезопасно использует свой аргумент, то надо выдать предупреждение. Для этого в анализе на основе резюме, надо протащить информацию о небезопасном использовании аргумента y через резюме.

В общем случае, сделать это не тривиально. При анализе функции g ничего не известно о контекстах, где она будет использована. Поэтому не известно надо ли сохранять информацию о небезопасном использовании. Резюме является компактным, сохранение всей возможной информации лишит анализ основных преимуществ.

Для решения описанной проблемы мы добавили анализ на предварительной фазе на основе АПД. На этой фазе процедуры обходятся в произвольном порядке; в рамках контекста процедуры отслеживаются помеченные данные и для каждого вызова процедуры в этом контексте запоминаются аргументы, которые являются помеченными. Таким образом, сохраненная информация относится только к рассмотренным контекстам.

```
void f() {
    int x = input();
    g(x);
}

void g(int y) {
    // Известен контекст, в котором
переменной y
      // было передано помеченное
значение
}
```

Листинг 5. Мотивация для предварительной фазы
Listing 5. Motivation for preliminary phase

Идея этого подхода в некотором смысле схожа с динамическим анализом, при котором происходит исследование свойств конкретного пути программы и все выводы которого остаются справедливыми только в рамках этого пути. При таком анализе производится исследование свойств, которые не обладают всеобщностью и справедливы для некоторого контекста вызова или цепочки вызовов функций.

Таким образом, предварительная фаза собирает информацию о контекстах использования функции. Данные о помеченных аргументах функции используются во время основного анализа функции. Соответствующие атрибуты, описывающие помеченные данные, устанавливаются на основе результатов предварительной фазы. Для конкретных детекторов ситуация выглядит так, как будто аргумент является результатом вызова функции, возвращающей помеченные данные. При этом возникает проблема, что информация о помеченных данных может попасть в резюме, что не является корректным. Резюме описывает результат вызова функции для произвольного контекста вызова, а предварительный анализ собирает данные для некоторых контекстов. Для решения этой проблемы был выбран следующий способ. Функция, у которой некоторые аргументы являются помеченными согласно предварительному проходу, анализируется два раза:

1 – используются данные от предварительного прохода, резюме не создается,

2 – обычный анализ без использования данных от предварительного прохода с созданием резюме.

Например, если для функции `input` известно, что она всегда является источником помеченных данных, то при анализе функции `f`, можно сделать вывод о существовании контекста вызова функции `g`, в котором ее аргумент имеет помеченное значение. На основании этого, верным будет сообщить об ошибке. Основная фаза принимает решение на основе только анализа функции `g`.

4.10. Спецификации в Svace

Для анализа библиотечных функций, поведение которых известно, используются спецификации. Спецификация в Svace это еще одно определение функции, написанное на анализируемом языке. Спецификация описывает поведение функции в компактной форме. Помимо конструкций языка, спецификации могут содержать вызовы специальных предопределенных функций, называемых спец-функциями.

Дистирбутив Svace содержит спецификации для широкоиспользуемых библиотек. Пользователь может добавлять свои собственные спецификации.

Для примера разберем спецификацию для функции `strcat` из стандартной библиотеки C. Исходный код спецификации:

```
char *strcat(char *s, const char *append) {
   char d1 = *s;
   char d2 = *append;
   sf_set_trusted_sink_ptr(s);
   sf_set_trusted_sink_ptr(append);
   sf_append_string(s, append);
     sf_vulnerable_fun("This function is unsafe, use strncat instead.");
   sf_null_terminated(s);
   return s;
}
```

В данном коде содержатся следующие спецфункции:

- `void sf_set_trusted_sink_ptr(const void* str)` – данная спецфункция показывает, что ее аргумент str должен быть из надежного источника, иначе данный указатель может вызвать уязвимость,
- `void sf_vulnerable_fun(const char*const reason)` – данная спецфункция показывает, что текущая функция небезопасна и имеет safe-аналоги,
- `void sf_append_string(char* dst, const char* src)` – данная спецфункция показывает, что выполнено добавление строки src к dst,
- `void sf_null_terminated(char *p)` – данная спецфункция показывает, что строка `p` заканчивается нулевым символом.

При анализе данной спецификации детекторы могут извлечь следующие свойства:

строки `s`, `append` были разыменованы, это может быть полезно для детекторов, которые ищут разыменование нулевых указателей,

строки `s`, `append` должны быть из надежного источника, информация используется детекторами помеченных данных,

функция `strcat` опасна и нужно использовать ее безопасную замену `strncat`,

строка s заканчивается нулевым символом.

5. ДЕТЕКТОРЫ ПОМЕЧЕННЫХ ДАННЫХ В SvEng

5.1. Плагины и детекторы

Анализ в SvEng разделен на ядро и плагины. Ядро анализа отслеживает граф указателей, выполняет сильные и слабые обновления ячеек памяти и вызывает обработчики для соответствующих ситуаций. Все дополнительные анализы реализуются внутри плагинов в виде обработчиков инструкций, оперирующих не переменными, а

соответствующими идентификаторами значений. Дополнительные анализы получают на вход абстрактное состояние на входном ребре инструкции, и формируют абстрактное состояние на выходном ребре. Детекторы также реализуются в плагинах и выдают предупреждения на основе входного абстрактного состояния и передаточной функции обрабатываемой инструкции.

Всем дополнительным анализам доступна информация на входных ребрах и недоступна на выходных. Различные анализы и детекторы не должны влиять друг на друга во время анализа инструкции. В текущей версии анализаторы и детекторы запускаются по очереди[4], но результаты должны быть такими же, как если бы они запускались параллельно.

В абстрактном состоянии для описания свойств используются атрибуты. Атрибуты могут ассоциироваться с идентификаторами значений и с ребрами графа потока управления для некоторого абстрактного состояния. Атрибут обозначает некоторое анализируемое свойство. Атрибуты должны иметь функцию объединения двух атрибутов $\sqcup$. Эта функция используется при объединении состояний в точках слияния путей. Атрибуты позволяют разделять абстрактное состояние между дополнительными анализами.

Атрибут может иметь произвольную структуру, но некоторые виды атрибутов используются достаточно часто. Можно выделить следующие виды атрибутов:

- двоичные атрибуты,
- тернарные атрибуты,
- интервальные атрибуты,
- условные атрибуты,
- множество идентификаторов значений.

Каждый вид атрибутов может также иметь трассу – односвязный список из пар “точка в программе – краткое текстовое описание события”. Трассы меняются при распространении атрибутов и используются в момент выдачи предупреждений, чтобы показать дополнительные точки в программе, позволяющие лучше понять, в чем ошибка.

Двоичные атрибуты. Имеют два значения: *true* и *false*. Значение *true* означает, что некоторая переменная точно имеет анализируемое свойство, значение *false* все остальное, т.е. либо переменная не имеет это свойство, либо недостаточно информации. В зависимости от функции объединения эти атрибуты могут быть двух видов:

- or-атрибуты – результат будет истинным, если хотя бы один аргумент является истинным,
- and-атрибуты – результат будет истинным, если оба аргумента является истинными.

Тернарные атрибуты могут принимать следующие значения:

true – для переменной выполняется свойство в данной точке для всех путей[5], проходящих через нее,

maybe – существует путь, возможно, недостижимый, для которого свойство выполняется,

false – остальные случаи: данное свойство либо не выполняется, либо недостаточно информации.

Функция объединения атрибута:

- $true \sqcup false = maybe$,
- $maybe \sqcup false = maybe$,
- $true \sqcup maybe = maybe$.

Фактически, тернарный атрибут является объединением or и and двоичных атрибутов. Значение *true* будет, если оба двоичных атрибута имеют истинное значение. Значение *maybe*, если or-двоичный атрибут имеет истинное значение, а and-атрибут не имеет, и *false* в остальных случаях.

Интервальные атрибуты связывают переменную с некоторым целочисленным интервалом, который описывает произвольное свойство. Например, возможный размер выделенной памяти для указателя или же значение, которое может принимать целочисленная переменная. Интервал может принимать следующие значения: $[a,b] - a \le b$; $a, b \in [MIN\_INT + 1, MAX\_INT - 1]$, данная запись означает, что свойство у переменной имеет значение в пределах интервала от a до b. Значения MIN_INT и MAX_INT зарезервированы для обозначения бесконечностей $-\infty$ и $+\infty$.

Цепочка интервалов представляет собой несколько интервалов. Цепочка интервалов позволяет моделировать интервалы с выколотыми точками.

Условные атрибуты хранят в себе формулу, описывающую выполнение некоторого свойства (см. 4.7). Выполнимость формулы проверяется SMT-решателем перед выдачей предупреждения. Данная формула состоит из следующих конъюнкций:

- условие достижимости, данная формула содержит условия при которых точка, где выдается срабатывание, будет достижима; данную формулу хранит атрибут *Ness*;
- условия помеченных данных, данная формула содержит условие, при котором указатель или зна-

[4] Распараллеливание запуска детекторов потенциально позволяет немного ускорить анализ, но при этом значительно усложняет внутрипроцедурный анализ из-за необходимости синхронизации. Поэтому выбор для распараллеливания был сделан на уровне графа вызовов: отдельные функции могут анализироваться параллельно, но анализ внутри функции последовательно.

[5] Для случая статического анализа учитываются все пути, которые анализ посчитал достижимыми. Чем точнее анализ, тем больше недостижимых путей он сможет отсеять.

чение целочисленной переменной будет получено из ненадежного источника; данную формулу отслеживают атрибуты *TaintedPtrIf* (для ненадежного указателя) и *TaintedIntIf* (для ненадежного значения целочисленной переменной), которые будут описаны позже;

• некоторое дополнительное свойство, зависящее от детектора, например, для обращения к массиву, проверяется что значение индекса меньше нуля, либо больше размера массива.

5.2. Межпроцедурное распространение атрибутов

Ядро анализа при создании резюме определяет какие идентификаторы значений туда попадут и для каждого вызывает обработчик *annotate*. В момент применения резюме ядро анализа сопоставляет формальные аргументы фактическим и вызывает обработчик *apply*. Для того, чтобы сделать атрибут межпроцедурным, достаточно подписаться на эти два обработчика, и реализовать соответствующую логику в зависимости от семантики атрибутов.

Для тернарных, двоичных, интервальных атрибутов резюме формируется путем передачи свойств без изменений (обработчик *annotate*).

Условные атрибуты – прежде, чем передать формулу в резюме, она упрощается:

1 – добавляем к условию конъюнкцию с условием достижимости,

2 – все атомарные условия, которые содержат в себе идентификаторы, добавленные ядром в резюме, сохраняются в специальное множество,

3 – берем не посещенное простое условие из формулы; если оно не содержится в списке, то преобразуем его в *False*, иначе помечаем условие как посещенное,

4 – для получившейся формулы применяем правила поглощения ($False \wedge Cond = False$, где Cond – некоторое условие),

5 – долучившаяся формула сохраняется в резюме.

Далее будем считать, что все стандартные атрибуты являются межпроцерурными и используют описанный выше механизм создания резюме, если не сказано иного.

5.3. Используемые атрибуты

Опишем вспомогательные атрибуты, предоставляющие нужную информацию для множества детекторов, которые используют помеченные данные.

Атрибут *MustTaintedInterval* хранит интервал возможных значений помеченной переменной. Значения из этого интервала должны быть проверены перед использованием. Функция объединения атрибута: пересечение интервалов ($[10, 20] \cap \cap \varnothing = \varnothing$, $[10,20] \cap [10,11] = [10, 11]$).

Атрибут *MightTaintedInterval* хранит максимально возможный помеченный интервал. Функция объединения атрибута: объединение с пустым интервалом) и пересечение с непустым ($[10,20] \sqcup [10,11] = [10,11]$).

Со значением помеченной переменной также связаны атрибуты, которые показывают осуществлялась ли проверка значения данной переменной:

MinIsTainted – двоичный or-атрибут, указывает на наличие проверки нижней границы. По умолчанию значение *true* (проверка отсутствовала);

MaxIsTainted – двоичный or-атрибут, указывает на наличие проверки верхней границы. По умолчанию значение *true* (проверка отсутствовала);

Эти атрибуты нужны, чтобы не выдавать бесполезное срабатывание для случаев, когда переменную сравнили с некоторым параметром функции:

```
char* allocate(int max) {
    unsigned int n;
    scanf("%d", n);

    if (n > max) {
        printf ("parameter too big,
use %d", max);
        return 0;
    }

return malloc(n);
}
```

Атрибут подавляет выдачу предупреждений для случаев, когда неизвестна точная безопасная граница. Например, для выделения памяти из кучи нет возможности определить такое ограничение статически. Для доступа к массиву с известным размером, такая граница известна, и атрибут не влияет на выдачу предупреждения.

5.4. Целочисленные помеченные значения

Значения целочисленных переменных могут контролироваться злоумышленником. Все реализованные детекторы являются подвидом источник-приемник, где в качестве источников используются функции получения данных из внешних источников, а приемников – операции, где данные необходимо проверить.

Источником являются все данные, которые получены извне программы (файл, сеть, пользовательский ввод). В большинстве случаев эти данные в Svace получаются из спецификаций.

Исключением являются параметры функции `main – argc` и `argv`. Svace с помощью атрибута *ArgvVarAttr* связывает `argv` с `argc`, данный атрибут позволяет детекторам избегать ложных срабатываний связанных с использованием параметров `main`. Например, когда указатель `argv` смещают, используя `argc`.

Приемником являются:

• использование как индекса массивов; перед использованием надо проверить диапазон,

• библиотечные функции,

• использование как шага цикла либо как ограничение цикла,

• использование как индекса для указателя; хотя неизвестен его точный размер, это не значит, что можно использовать любой.

Особенностью помеченных целочисленных переменных является то, что проверка их диапазона может производиться довольно сложным образом с использованием битовой арифметики.

Кроме этого, некоторые переменные могут быть связаны друг с другом какой-либо операцией, в этом случае проверяться может только одна переменная. При реализации анализа важно учитывать такие взаимосвязи. Для анализа взаимосвязей между переменными используется SMT-решатель: строятся формулы, описывающие эти взаимосвязи, а затем вызывается SMT-решатель, который определит может ли формула иметь решение.

В Svace реализованы следующие типы детекторов для поиска помеченных целых чисел:

• TAINTED_ARRAY_INDEX – доступ к массиву по непроверенному индексу,

• TAINTED.INT_OVERFLOW – приемником является потенциальное целочисленное переполнение,

• TAINTED_INT.PTR – доступ к указателю с помощью смещения,

• TAINTED_INT – доступ к функции, где входные параметры должны быть проверены,

• TAINTED_INT.LOOP – приемником помеченных данных является использование переменной как ограничения цикла, либо как шага цикла.

Для перечисленных детекторов могут использоваться суффиксы, имеющие следующее значение:

• .MIGHT – не на всех путях к опасному использованию данных эти данные помеченные,

• .COND – приемник находится в вызываемой функции и не достижим на всех путях внутри этой функции.

Например, TAINTED_ARRAY_INDEX.MIGHT – доступ к массиву в качестве индекса, где не все пути содержат помеченные данные.

5.4.1. Предупреждение TAINTED_INT. В коде может встретиться ситуация, когда программист использует переменную, значение которой получено из внешнего источника, в таких функциях как `strncpy`, `malloc` или в качестве условия выхода из цикла. Так как злоумышленник может передать любое значение, то использование таких переменных может повлечь за собой уязвимости: бесконечный цикл, переполнение массива.

```
void test(int fd) {
    int sizeBuf;
    //sizeBuf получен из ненадежного источника
    int ret = recv(fd, &sizeBuf, sizeof(sizeBuf), 0);
    if(ret<0)
        return;
    if (sizeBuf < 0) {
       return;
    }
    //использование помеченной переменной в calloc
    char*x = calloc(1, sizeBuf);//TAINTED_INT
}
```

В данном примере размер выделенной памяти зависит от помеченной переменной, *sizeBuf* может быть очень большим, что повлечет за собой выделение чрезмерно больших объемов памяти, которая не будет использоваться или же *sizebuf* может быть равен нулю, тогда обращение к *x* может вызвать ошибку переполнения массива.

Детектор ищет ситуации, когда помеченные целочисленные переменные передаются в функции, где они могут вызвать уязвимость.

Для идентификации помеченных целочисленных переменных используется атрибут *TaintedIntIf*, который хранит в себе формулу, при выполнении которой переменная будет содержать помеченное значение. Функция объединения атрибута: конъюнкция формул с веток.

Для межпроцедурного распространения свойств об использовании данных, которые должны быть из надежного источника, используется атрибут *TrustedIntSinkFlag*. Фактически, этот атрибут при применении резюме создает еще один приемник, для которого может быть выдано предупреждение. Заметим, что анализ спецификаций является частным случаем межпроцедурного анализа. При обработке спецификаций для функций типа *malloc* используется этот атрибут.

Условия выдачи срабатывания:

• после вызова функции ее аргумент имеет следующие свойство: *TrustedIntSinkFlag* – *true* или же переменная используется в условиях цикла,

• переменная имеет не пустой атрибут *MustTaintedInterval* или *MightTaintedInterval*,

• у переменной не проверяли нижнюю и верхнюю границу; атрибуты *MinIsTainted* и *MaxIsTainted* для переменной имеют значение *false*,

• конъюнкция формулы из *TaintedIntIf* с условием, что переменная лежит в интервале из *MustTaintedInterval* или *MightTaintedInterval* выполнима.

Использование атрибутов *MustTaintedInterval* и *MightTaintedInterval* является лишним для логики программы, но позволяют оптимизировать запросы к SMT-решателю.

5.4.2. Предупреждение TAINTED_INT.LOOP. Является подвидом TAINTED_INT для уязвимостей, связанных с циклами.

Выдается для двух случаев.

• Помеченное значение используется для ограничения количества итераций цикла. Ошибка заключается в том, что цикл может выполняться излишне много итераций. Для определения, что переменная ограничивает количество итераций цикла используется информация о графе потока управления. Обработчик условных инструкций проверяет, что инструкция принадлежит сильно связанной компоненте и входное ребро является ребром входа в эту компоненту.

• Помеченное значение используется как шаг цикла. В этом случае, если злоумышленнику удастся установить такое значение, чтобы переменная не менялась на разных итерациях, в программе будет бесконечный цикл. Определение шага цикла реализовано более сложно. На первом этапе на анализе потока данных вычисляются инварианты цикла, т.е. такие переменные, которые имеют одно и то же значение на всех итерациях. Для всех остальных переменных проверяется, что они используются в арифметических инструкциях, их диапазон допускает нежелательное значение[6] и условные инструкции, проверяющие значения этих переменных.

Возможны ситуации, когда переменная используется в цикле в вызываемой функции. Для анализа используется атрибут *LoopBoundFlag*, который устанавливается в истину в тех случаях, когда анализ посчитал переменную ограничителем количества итераций цикла. Далее этот атрибут распространяется межпроцедурно, и если в момент применения резюме формальный аргумент имеет этот атрибут, а фактический – атрибут *MustTaintedInterval*, то будет выдано предупреждение.

5.4.3. Предупреждение TAINTED_INT.PTR. Если использовать помеченную целочисленную переменную без каких-либо проверок в качестве смещения указателя, то можно выйти за пределы выделенной памяти, так как значение помеченной переменной может быть произвольным.

[6] Чаще всего это 0. В некоторых ситуациях к ошибке может дополнительно приводить целочисленное переполнение.

```
void test(int fd, int *ptr) {
    int index;
    //значение index помечено
    int ret = recv(fd, &index, sizeof(index), 0);
    //использование помеченной переменной index как смещение
    ptr[index] = 3;//TAINTED_INT.PTR
}
```

В данном примере значение переменной *index* может быть любым, поэтому возможна ошибка переполнения буфера.

Детектор ищет ситуации, когда помеченные целочисленные переменные не проверяются и используются в качестве смещения указателя.

Условия выдачи срабатывания:

инструкция получения доступа к указателю,

• смещение имеет непустой *MustTaintedInterval* или *MightTaintedInterval*,

• у смещения не проверяли нижнюю и верхнюю границу; атрибутов *MinIsTainted* и *MaxIsTainted* у смещения имеют значение *false*,

• размер выделенной памяти для указателя неизвестен, или же *MustTaintedInterval* или *MightTaintedInterval* интервал может превышать его размер.

5.5.4. Предупреждение TAINTED_ARRAY_INDEX. Предупреждение во многом похоже на TAINTED_INT.PTR с той разницей, что оно выдается при работе с массивами для которых статический анализатор знает размер.

```
void test(int fd) {
    int ptr [6];
    int index;
    //значение index получено из ненадежного источника
    int ret = recv(fd, &index, sizeof(index), 0);
    //использование помеченного значения в качестве индекса
    ptr[index] = 3;//TAINTED_ARRAY_INDEX
}
```

В данном примере нам известен размер массива *ptr*, и значение *index* может быть больше 5.

Детектор ищет ситуации, когда происходит доступ к массиву по индексу, где индекс получен из ненадежного источника и может иметь значение больше, чем размер массива. При этом размер массива известен анализатору.

Для определения размера массива используется атрибут *BufferSizeAttrVal*, который хранит возможный размер массива в виде интервала.

Условия выдачи срабатывания:

• доступ к массиву по индексу,

• нам известен размер массива; интервал из *BufferSizeAttrVal* не пустой и не [–∞, +∞],

• индекс имеет не пустой *MustTaintedInterval* или *MightTaintedInterval* атрибут,

• для индекса составляется формула с условием, что его значение может превышать интервал из *BufferSizeAttrVal*, эта формула выполнима.

5.5.5. Ошибки с целочисленным переполнением. Так как значение переменной, пришедшей из внешнего источника, может быть любым, то если осуществлять арифметические действия с данной переменной без предварительной проверки, то может произойти целочисленное переполнение и ее значение будет некорректным. Это может привести как к уязвимостям, так и к нарушению логики выполнения программы. Для таких ситуаций выдается предупреждение TAINTED.INT_OVERFLOW.

```
void test(int fd) {
    int ptr [6];
    int index;
    //значение index получено из ненадежного источника
    int ret = recv(fd, &index, sizeof(index), 0);
    //целочисленное переполнение
    index += 1; //TAINTED.INT_OVERFLOW
    if(index > 4) {
       return;
    }
    ptr[index] = 3;
}
```

В данном примере *index* может иметь максимальное значение для переменной типа *int*, поэтому при добавлении единицы может произойти целочисленное переполнение и ее значение станет некорректным.

Детектор проверяет ситуации, когда переменная из ненадежного источника участвует в арифметических операциях: сложение, умножение, вычитание. При помощи интервала из атрибута *MustTaintedInterval* проверяется, возможно ли переполнение.

Условия выдачи срабатывания:

• инструкция сложения, вычитания, умножения,

• атрибут *MustTaintedInterval* у одного из аргументов инструкции не пустой,

• интервал из *MustTaintedInterval* = $[-\infty; +\infty]$ или он может переполнить тип второго аргумента.

5.5. Помеченные строки

В некоторых ситуациях необходимо, чтобы указатель содержал в себе проверенные данные. Например, при открытии файла с помощью open или же когда осуществляется объединение или копирование строк. В случае `strcpy` и `strcat`, если строка `src` помечена, то ее размер может быть больше чем у строки `dst`, что вызовет переполнение буфера.

```
void test(int fd, int *ptr, int size) {
    //env получен из ненадежного источника
    char* env = getenv("PATH");
    char *buf = malloc(size);
    //строка в env может быть любого размера
    //поэтому возможно переполнение buf при копировании
    strcpy(buf, env);//TAINTED_PTR
}
```

В данном примере размер строки в `env` может быть любым, поэтому возможна ошибка переполнения буфера при ее копировании в `buf`.

Детектор ищет ситуации, когда ненаджедный указатель используется в функциях, где он может вызвать уязвимость.

Для идентификации испорченного указателя используются следующие атрибуты:

TaintedPtrIf – атрибут хранит условия, при которых указатель будет содержать помеченные данные. Представляет собой формулу логики высказываний. Функция объединения атрибута: конъюнкция формул с веток.

TaintedPtr – тернарный атрибут, показывает, содержит ли указатель помеченные данные.

С помощью тернарного атрибута *TrustedPtrSinkFlag* детектор находит указатели, которые использовались в опасных функциях, где помеченное происхождение указателя может вызвать уязвимость (`open`, `strcpy`, `strcat` и т.д.).

Условия выдачи срабатывания:

• атрибут *TaintedPtr* имеет значение *true* или *maybe*; указатель точно или же возможно получен из ненадежного источника,

• атрибут *TrustedPtrSinkFlag* имеет значение *true*, указатель использовался в функции, где он может вызвать уязвимость,

• формула в атрибуте *TaintedPtrIf* выполнима.

Также существуют ситуации, когда помеченный указатель не может вызвать уязвимость. Например, если при использовании `strcpy` известно, что для `dst`-строки выделено памяти достаточно, чтобы скопировать испорченную `src`-строку:

```
void test(int fd, int *ptr, int size) {
    //env содержит помеченные данные
    char* env = getenv("PATH");
    //размер buf зависит от длины строки в env
    char *buf = malloc(strlen(env) + 1);
    //переполнение невозможно
    strcpy(buf, env);//TAINTED_PTR
}
```

Таблица 1. Процент ложных срабатываний для проекта Tizen 6

Предупреждение	Количество предупреждений	Процент истинных срабатываний
TAINTED_ARRAY_INDEX	102	62.5
TAINTED_INT	137	65.5
TAINTED_INT.LOOP	137	76
TAINTED_INT.PTR	82	58
TAINTED_PTR	242	70.5
TAINTED.INT_OVERFLOW	796	85

В данном примере для массива *buf* выделено достаточно памяти для копирования туда строки *env*.

Для отсеивания таких ситуаций при копировании строки детектор узнает идентификатор, который является размером выделенной памяти для строки, после чего проверяет длины каких строк содержит данный идентификатор, если среди этих строк присутствует `dst`-строка, то ошибка не выдается. В примере, для переменной `buf` выделена память размера (*strlen*(*env*) + 1), данный размер выделенной памяти включает в себя длину строки `env`, поэтому срабатывание не выдается.

Похожая ситуация есть и со `strcat`. Разница в том, что при присоединении новой строки проверяется включает ли в себя размер выделенной памяти `src`-строку со множеством строк из которых состоит `dst`-строка.

В случае, если помеченную строку сравнивали с другой строкой при помощи функций сравнения (`strcmp`), то срабатывания так же не выдается. Для идентификации таких помеченных строк используется тернарный атрибут *SanitizationInvoked*. Он помечает переменные, которые использовались в функциях сравнения строк.

6. РЕЗУЛЬТАТЫ

6.1. Анализ проектов с открытым исходным кодом

Для оценки процента истинных срабатываний использовался исходный код проекта Tizen 6 [18]. Проект Tizen – это открытая операционная система на базе ядра Linux. Общий размер проанализированного кода с помощью Svace составил более 32 миллионов строк. Результаты анализа приведены в табл. 1. Для оценки было размечено как минимум по 40 предупреждений для каждого вида детекторов. При этом не оценивалось эксплуатируемость ошибки. Предупреждение считалось истинным, если код содержит передачу небезопасных данных в критические операции; проверка достижимости пути из точек входа в программу не производилась.

Детектор TAINTED.INT_OVERFLOW оказался довольно шумным. Возможно, для него требуется доработка, чтобы исключить малополезные предупреждения. Большинство найденных срабатываний выглядят следующим образом:

```
int count;
count = strtol(arg, NULL, base);
```

Функция *strtol* возвращает тип *long*, поэтому потенциально возможна потеря значимой части возвращаемого значения.

6.2. Оценка Juliet 1.3

Проект Juliet [19] является тестовым набором для проверки возможностей статических анализаторов. Он включает как корректные тесты, где анализатор должен выдавать данные, так и ошибочные тесты, где анализатор не должен выдавать предупреждение.

Таблица 2. Процент пропусков для проекта Juliet

CWE	Количество тестов	Покрытие	FN(%)
CWE680	384	206	46.35
CWE194	816	444	45.59
CWE195	816	444	45.59
CWE789	384	92	76.04
CWE127	240	104	56.67
CWE124	240	104	56.67
CWE126	390	104	73.33
CWE400	624	164	73.72
CWE134	1200	226	81.17
Всего	5094	1888	62.93

Общее покрытие тестов составило 38.32%.

Из набора типов ошибок в Juliet были взяты те, которые могут быть связаны с помеченными данными: CWE680, CWE194, CWE195, CWE789, CWE127, CWE124, CWE126, CWE134, CWE400. Данные тесты компилировались с опцией omitgood, которая скрывает все тесты, где ошибка отсутствует. Тесты используют специальную систему именования: имя теста можно разделить на части, каждая часть несет некоторую информацию, например, тип ошибки или же источник и приемник [20]. Основную информацию об используемых функциях в тесте можно получить из части под названием Functional Variant Name. Из получившейся выборки были отсеяны следующие тесты:

• тесты, которые компилируются только для windows; у таких тестов в Functional Variant Name содержатся w32, wchar;

• тесты, которые не содержат помеченные данные; у таких тестов в Functional Variant Name содержится не испорченный источник, а именно, функции: rand, new и другие.

На получившейся выборке было измерено покрытие тестов. Если один из помеченных детекторов выдавал ошибку в тесте, он считался покрытым. Все непокрытые тесты относятся к FN (False Negative). Табл. 2 содержит данные о проценте непокрытых текстов.

Для данной выборки также было измерено количество ложных срабатываний. Для этого компиляция тестов происходила с опцией omitbad, которая скрывает все тесты с ошибкой. Соответственно любое срабатывание будет ложным. Для тестовой выборки количество ложных срабатываний незначительно и составило 0.47%.

7. ЗАКЛЮЧЕНИЕ

Был описан межпроцедурный контекстно- и потоково-чувствительный анализ помеченных данных для программ на C, C++, Java, Kotlin и Go для поиска уязвимостей. В анализе используются хорошо известные и проверенные решения, которые можно встретить в других анализаторах.

Общая уникальная схема анализа была разработана за более, чем 10-летний опыт написания статических анализаторов. Получившееся решение не позволяет найти все уязвимости в программе, но процент найденных ошибок выше 38.32% для тестов Juliet.

8. БЛАГОДАРНОСТИ

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 20-01-00581 А.

СПИСОК ЛИТЕРАТУРЫ

1. *Belevantsev A., Borodin A., Dudina I. et al.* Design and development of Svace static analyzers. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018. P. 3–9.
2. *Бородин А.Е. и Белеванцев А.А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН. 2015. Т. 27. Вып. 6. С. 111–134. https://doi.org/10.15514/ISPRAS-2015-27(6)-8 / A. Borodin, A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN/Proc. ISP RAS. 2015. V. 27. Iss. 6. P. 111–134 (in Russian).
3. *Borodin A., Belevantsev A., Zhurikhin D., and Izbyshev A.* Deterministic static analysis. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018. P. 10–14.
4. *Иванников В.П., Белеванцев А.А., Бородин А.Е. и др.* Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН. 2014. Т. 26. Вып. 1. С. 231–250. https://doi.org/10.15514/ISPRAS-2014-26(1)-7 / V. Ivannikov, A. Belevantsev, A. Borodin et al. Svace: static analyzer for detecting of defects in program source code. Trudy ISP RAN/Proc. ISP RAS. V. 26. Iss. 1, 2014. P. 231–250 (in Russian).
5. Aleph One. Smashing the stack for fun and profit. Phrack magazine. 1996. V. 7. Iss. 49. P. 14–16.
6. National Vulnerability Database – CWE Over Time. 2020. URL: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. Accessed 15.01.2021.
7. *Landi W.* Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS). 1992. V. 1. № 4. P. 323–337.
8. *Hind M.* Pointer analysis: haven't we solved this problem yet? In Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2001. P. 54–61.
9. *Landi W.* Interprocedural aliasing in the presence of pointers. PhD Thesis, The State University of New Jersey, 1992, 268 p.
10. *Landi W., Ryder B.G.* A safe approximate algorithm for interprocedural aliasing. ACM SIGPLAN Notices. 1992. V. 27. № 7. P. 235–248.
11. *Livshits B., Sridharan M., Smaragdakis Y. et al.* In defense of soundiness: a manifesto. Communications of the ACM. 2015. V. 58. № 2. P. 44–46.
12. *Белеванцев А., Избышев А., Журихин Д.*. Организация контролируемой сборки в статическом анализаторе Svace. Системный администратор. 2017. Вып. 7–8. С. 135–139 / A. Belevantsev, A. Izbyshev, D. Zhurikhin. Monitoring program builds for Svace static analyzer. System Administrator. 2017. Iss. 7–8. P. 135–139 (in Russian).
13. *Bush W.R., Pincus J.D., and Sielaff D.J.* A static analyzer for finding dynamic programming errors. Software-Practice and Experience. 2000. V. 30. Iss. 7. P. 775–802.

14. *Aiken A., Bugrara S., Dillig I. et al.* An overview of the saturn project. In Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2007. P. 43–48.

15. *Babic D. and Hu A.J.* Calysto: scalable and precise extended static checking. In Proc. of the 30th international conference on Software engineering. 2008. P. 211–220.

16. *Кошелев В.К., Игнатьев В.Н., Борзилов А.И.* Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН. 2016. Т. 28. Вып. 1. С. 21–40. DOI: / V. Koshelev, V. Ignatiev, A. Borzilov, and A. Belevantsev. SharpChecker: static analysis tool for C# programs. Programming and Computer Software. 2017. V. 43. № 4. P. 268–276. https://doi.org/10.15514/ISPRAS-2016-28(1)-2

17. *Мулюков Р. Р., Бородин А.Е.* Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, 2016г., том 28, вып. 5, стр. 145–158 / R.R. Mulyukov, A.E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. Trudy ISP RAN/Proc. ISP RAS. 2016. V. 28. Iss. 5. P. 145–158 (in Russian). https://doi.org/10.15514/ISPRAS-2016-28(5)-9.

18. Tizen 6.0 Public M2 Release. URL: https://www.tizen.org/blogs/bighoya/2020/tizen-6.0-public-m2-release-0, accessed 15.01.2021.

19. *Black P.E.* Juliet 1.3 Test Suite: Changes from 1.2. US Department of Commerce, National Institute of Standards и Technology, 2018, 37 p.

20. Juliet Test Suite v1.2 for C/C++ User Guide. Center for Assured Software, National Security Agency, 2012, 41 p.