ISSN 2220-3397

ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Программная инженерия

Том 14. № 4. 2023

Рисунки к статье **Д. В. Ефанова, Т. С. Погодиной**

«АНАЛИЗ ЭФФЕКТИВНОСТИ СХЕМ ВСТРОЕННОГО КОНТРОЛЯ НА ОСНОВЕ ОЦЕНКИ ПРИНАДЛЕЖНОСТИ ВЫЧИСЛЯЕМЫХ ФУНКЦИЙ КЛАССУ САМОДВОЙСТВЕННЫХ И ПРЕДВАРИТЕЛЬНОГО СЖАТИЯ СИГНАЛОВ С ПРИМЕНЕНИЕМ ЛИНЕЙНЫХ КОДОВ»

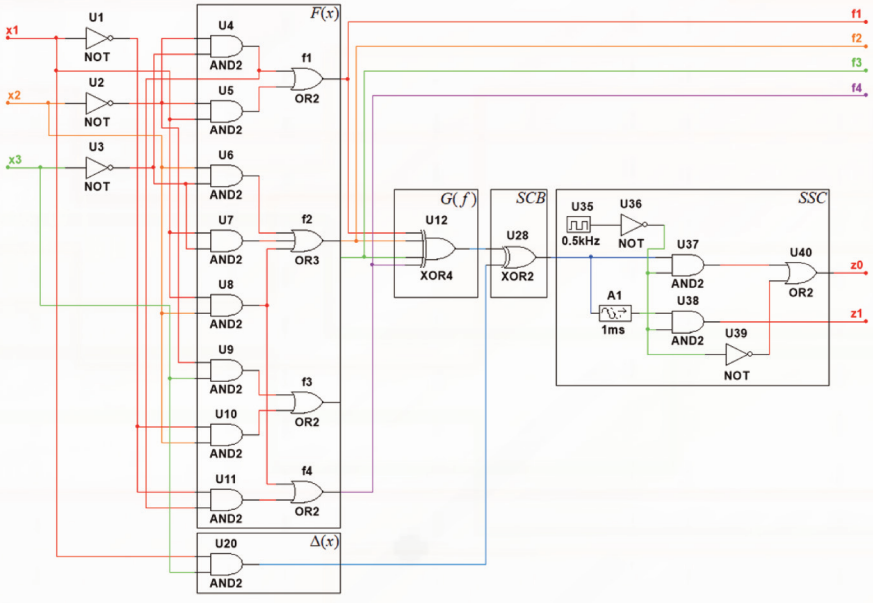


Рис. 4. Структура устройства с СВК на основе кода паритета

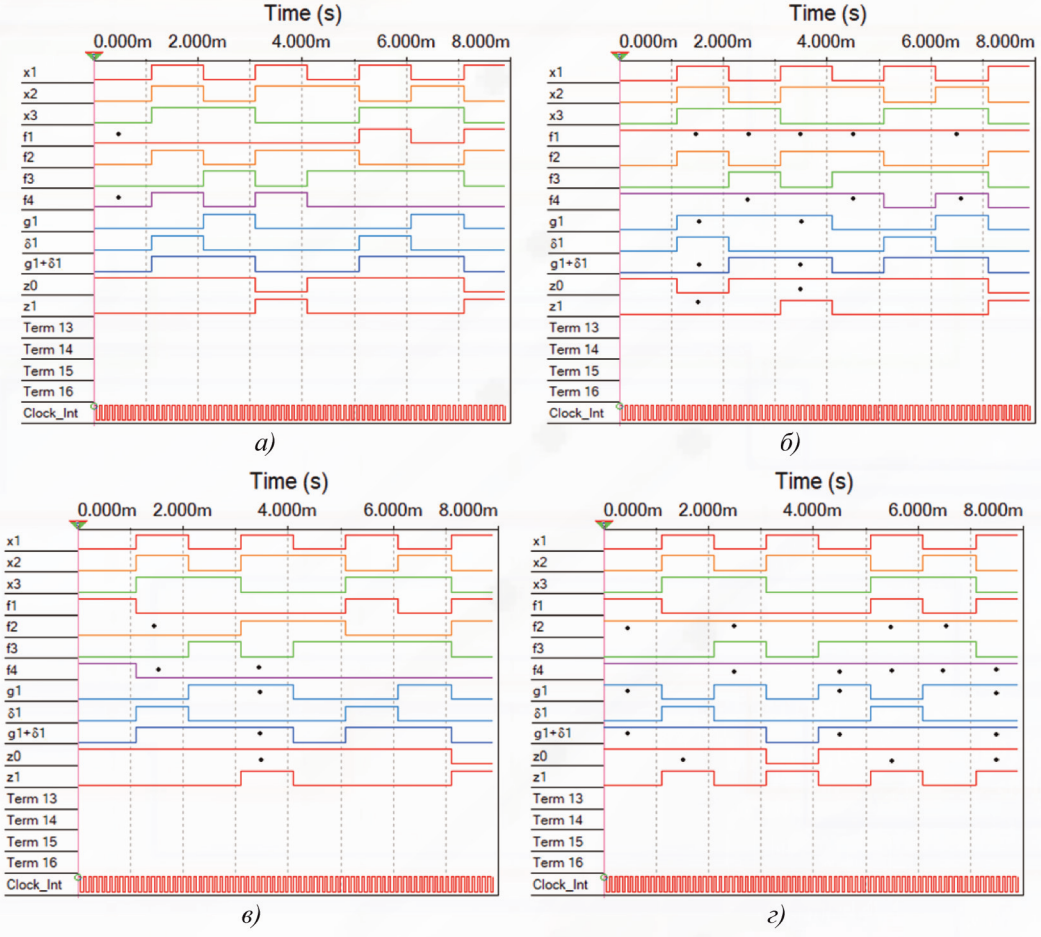


Рис. 5. Временные диаграммы работы устройства с СВК на основе кода паритета при внесении неисправностей: *а* – stuck-at-0 U4; *б*– stuck-at-1 U4; *в* – stuck-at-0 U8; *г* – stuck-at-1 U8

ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г. DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в подписных агентствах (индекс по Объединенному каталогу "Пресса России" — **22765**) или непосредственно в редакции (для юридических лиц).
Тел.: (499) 270-16-52.
Http://novtex.ru/prin/rus E-mail: prin@novtex.ru
Журнал включен в Российский индекс научного цитирования (РИНЦ) и Russian Science Citation Index (RSCI).
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 14
N 4
2023

Published since September 2010 **DOI 10.17587/issn.2220-3397** **ISSN 2220-3397**

CONTENTS

Information about the jornal is available online at:
http://novtex.ru/prin/eng e-mail:prin@novtex.ru

УДК 004.75
DOI: 10.17587/prin.14.155-164

Р. Э. Асратян, канд. техн. наук, вед. науч. сотр., rubezas@yandex.ru,
Институт проблем управления им. В. А. Трапезникова РАН, Москва

Детерминированная модель обработки пакета информационных запросов в многопоточном сервере

Поступила в редакцию 16.01.2023
Принята к публикации 02.02.2023

Рассмотрена аналитическая модель обработки пакета однотипных запросов в многопоточном сервере. Модель построена в расчете на высокую детерминированность длительности отдельных этапов обработки запросов и моментов событий, приводящих к переключению процессора с одной обрабатывающей программной нити на другую. В отличие от других детерминированных моделей, построенных на основе теории расписаний, в данном случае целью является не поиск оптимального распределения работ по процессорам, а изучение зависимости основных характеристик программного сервера (время обработки пакета, предельная производительность) от характеристик запроса и размеров пакета.

Ключевые слова: *детерминированная модель, пакетный запрос, пакетная обработка, многопоточный сервер, многоядерность, многопоточность*

Для цитирования:

Асратян Р. Э. Детерминированная модель обработки пакета информационных запросов в многопоточном сервере // Программная инженерия. 2023. Том 14, № 4. С. 155—164. DOI: 10.17587/prin.14.155-164.

Введение

Рассмотрим программу-клиент в составе распределенной информационной системы, которая последовательно отправляет на удаленный сервер 100 или более тысяч информационных запросов в день. Если время обработки каждого запроса превышает секунду, клиент не получит требуемого обслуживания даже при круглосуточной работе. В подобных случаях приходится принимать специальные меры для обеспечения нужной производительности. Одной из таких мер является введение средств обработки пакетных запросов — массивов однотипных информационных запросов, обрабатываемых за одно обращение к серверу. Введение таких средств, как правило, позволяет многократно повысить скорость обработки запросов в системе и очень часто становится эффективным средством, способным существенно изменить ситуацию [1, 2].

Одной из самых важных характеристик программных серверов (далее — серверов) является их многопоточность — способность обслуживать несколько запросов одновременно. Технически многопоточность обычно опирается на имеющиеся в операционной системе (ОС) средства распараллеливания вычислений: для обслуживания каждого поступившего запроса сервер создает отдельный программный процесс или отдельную обрабатывающую программную нить (поток) [3—5]. Такая организация нередко позволяет многократно повысить производительность сервера (число обработанных запросов в секунду) по сравнению с однопоточной организацией даже в условиях единственного одноядерного процессора [3, 6]. Этот выигрыш особенно четко проявляется при обработке пакетов запросов в распределенных информационных системах. Логика обработки запроса в таких системах обычно предполагает длительные периоды ожидания вследствие обращения к внешним источникам данных — серверам СУБД. В этих условиях средства распараллеливания ОС вполне эффективно решают свою главную задачу — сокращение простоев процессора за счет переключений его с одной программной нити на другую [3, 7].

Ранее автору пришлось заниматься экспериментальной оценкой производительности много-

поточного программного сервера, включающего средства пакетной обработки запросов. Обработка каждого запроса в пакете содержала три основные фазы: разбор параметров запроса; обращение к серверу СУБД для выборки релевантной информации; оформление результата выборки в формате XML-документа. Оказалось, что даже в относительно простом случае объяснить связь между длительностями этих трех фаз и временем обработки всего пакета достаточно непросто. Например, когда был найден способ в 2 раза сократить время выборки данных из базы данных (БД), оказалось, что в пакетах малого размера (т. е. включающих небольшое число запросов) скорость обработки запросов значительно выросла, а в больших пакетах почему-то практически не изменилась (рис. 1). Очевидно, что в отсутствие аналитической модели пакетной обработки нельзя получить содержательные ответы на ряд важных вопросов, связанных с «поведением» сервера. Например, при каких размерах пакета достигается предельная производительность (выраженная как отношение числа запросов в пакете ко времени его обработки). Или как связана эта предельная производительность с временными характеристиками обрабатывающей программной нити.

Представленные в настоящей статье исследования являются попыткой создания детерминированной аналитической модели обработки пакета запросов в многопоточном сервере на сетевом узле с одним процессором. В отличие от стохастических моделей, построенных на основе теории очередей [8, 9], эта модель должна учитывать особенности предметной области, а именно высокую детерминированность длительности отдельных этапов обработки запросов и моментов событий, приводящих к переключению процессора с одной нити на другую. В отличие от других детерминированных моделей, построенных на основе теории

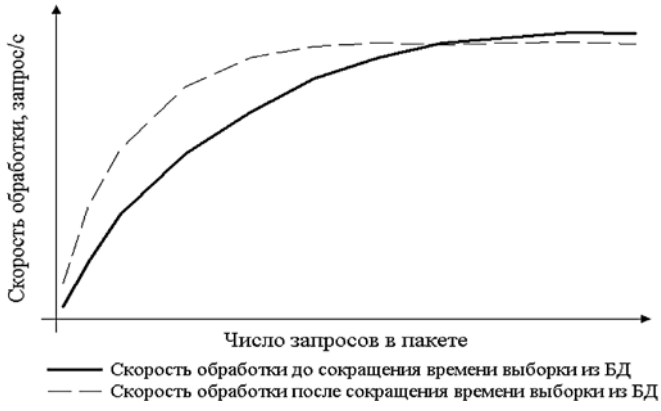


Рис. 1. Общий вид зависимости скорости обработки от размера пакета запросов

расписаний [10—12], в данном случае целью является не поиск оптимального распределения работ по процессорам, а изучение зависимости основных характеристик программного сервера (время обработки пакета, предельная производительность) от характеристик запроса и размеров пакета. Основным понятием модели является модельная программная нить, отражающая главные особенности поведения реальной программной нити. К числу таких особенностей относятся чередования состояний активной работы и состояний ожидания данных из внешних источников, а также периодов выполнения в процессоре и задержек (пауз), вызванных «конкуренцией» с другими нитями.

1. Модель обработки пакета запросов в одноядерной среде

Описываемая модель основана на следующих базовых предположениях.

1. В течение обработки пакетного запроса на сервере нет активных процессов или программных нитей, кроме тех, которые имеют отношение к обработке пакета запросов.

2. В каждый момент времен каждая нить может находиться в одном из трех состояний: активности (выполняется в процессоре), ожидания (нить не готова к работе в связи с выполнением операции сетевого обмена или ввода-вывода) и паузы (нить готова к работе, но процессор занят).

3. Каждая нить выполняется в процессоре до тех пор, пока не переключится из состояния активности в состояние ожидания (диспетчеризация «без вытеснения», *non preemptive*).

4. Все нити считаются равноприоритетными: если процессор занят, то каждая вышедшая из состояния ожидания или вновь созданная нить помещается в конец единственной очереди активных нитей, претендующих на процессорное время (состояние паузы).

5. Временные интервалы выполнения всех внешних по отношению к серверу операций, связанных с обращением к серверу БД, являются стабильными и не зависят от числа запросов в пакете.

6. Все запросы в пакете одинаковы, т. е. во всех обрабатывающих их нитях выполняется одна и та же программа. Поведение обрабатывающей нити будем считать абсолютно «прямолинейным»: оно заключается в простом чередовании периодов активности и периодов ожидания, причем число и длительности этих периодов у всех нитей одинаковы и детерминированы.

Так как целью моделирования является получение временных оценок, нас будут интересовать

только временные характеристики обрабатывающих нитей. Будем считать, что каждая нить полностью описывается набором чисел

$$\langle t_1,\ W_1,\ t_2,\ W_2,\ ...,\ t_m,\ W_m,\ t_{m+1}\rangle,$$

где m — число периодов ожидания в работе нити; t_j ($j = 1, 2, ..., m + 1$) — длительности периодов активности; W_j ($j = 1,2, ..., m$) — длительности периодов ожидания, а поведение нити во времени описывается последовательностью

$$t_1,\ W_1,\ t_2,\ W_2,\ ... .$$

Будем считать, что пакет запросов полностью описывается характеристиками обрабатывающей нити и числом запросов в пакете n:

$$\langle n, \langle t_1,\ W_1,\ t_2,\ W_2,\ ...,\ t_m,\ W_m,\ t_{m+1}\rangle\rangle.$$

Общая логика обработки пакета одновременных запросов в сервере включает три основные стадии.

1. Для обработки пакета сервер создает корневую нить, которая считывает запрос из сети и создает обрабатывающие нити в количестве, равном числу запросов в пакете. После этого корневая нить переключается в состояние ожидания до завершения работы всех обрабатывающих нитей.

2. Согласно принятым базовым предположениям нити первоначально получают управление в том порядке, в котором они были созданы (далее будем считать, что все нити перенумерованы именно в этом порядке). Когда работающая нить переключается из состояния активности в состояние ожидания, она уступает процессор другим нитям в очередности, определяемой базовыми предположениями. Эта стадия заканчивается после того, как все нити закончат свою работу.

3. Корневая нить получает управление, объединяет частные результаты выполнения обрабатывающих нитей в один массив результатов (пакетный ответ) и отправляет его клиенту по открытому сетевому соединению.

Так как первая и третья стадии выполняются строго последовательно, то суммарное время их выполнения можно приближенно оценить как nt_0, где n — число запросов в пакете; t_0 — время, необходимое для создания одной обрабатывающей нити на первой стадии и для обработки результата ее работы на третьей стадии. Время обработки второй стадии $T(n)$ оценить сложнее, так как в процессе обработки возможны периоды ожидания и паузы в работе нитей, а также периоды простоя в работе процессора.

Общее время обработки пакета запросов T_p представим в виде суммы

$$T_p = nt_0 + T(n) \qquad (1)$$

и попытаемся найти способы оценки $T(n)$, исходя из значений периодов активности t_i ($i = 1, 2, ..., m + 1$) и периодов ожидания W_i ($i = 1, 2, ..., m$).

Утверждение 1. Пусть n — число запросов в пакете ($n > 1$) и $1 \leqslant i < j \leqslant n$. Тогда обрабатывающая нить i раньше приступит к выполнению периода активности k, чем нить j при любом k ($1 \leqslant k \leqslant m + 1$), где m — число периодов ожидания.

Доказательство нетрудно провести с помощью индукции по значению k. Действительно, при $k = 1$ утверждение, очевидно, выполняется. Предположим, что $1 < k \leqslant m + 1$ и утверждение справедливо в отношении периода активности $k - 1$. Это означает, что нить i раньше закончит период активности $k - 1$, чем нить j (напомним, что рассматриваем диспетчеризацию «без вытеснения»), раньше закончит период ожидания $k - 1$ и раньше встанет в очередь активных нитей, претендующих на процессорное время. Этот факт, в свою очередь, означает, что нить i раньше приступит к выполнению периода активности k, чем нить j. Конец доказательства.

Следствие 1. Если $1 \leqslant i < j \leqslant n$, то обрабатывающая нить i раньше завершит период ожидания k, чем нить j при любом k ($1 \leqslant k \leqslant m$).

Доказательство. Согласно утверждению 1 нить i раньше приступит к выполнению k-го периода активности, чем нить j. Справедливость следствия вытекает из того, что для обеих нитей момент завершения k-го периода ожидания наступает через одинаковый интервал времени $t_k + W_k$ после начала выполнения k-го периода активности.

Разумеется, утверждение 1 отражает лишь самое общее свойство коллективного поведения обрабатывающих нитей и, в принципе, оставляет возможность для самых разных сценариев: от простого последовательного выполнения (сначала нить 1 выполняет все периоды активности вплоть до последнего, потом нить 2 и т. д.) до равномерного циклического выполнения (сначала все нити последовательно в порядке возрастания их номеров выполняют 1-й период активности, затем точно так же 2-й и т. д.) со множеством промежуточных вариантов. В общем случае все это разнообразие сценариев весьма затруднительно учесть в рамках сколько-нибудь компактной аналитической модели. Поэтому для получения аналитической оценки $T(n)$ понадобятся дополнительные ограничения на параметры нитей. Ниже рассмотрены два таких ограничения.

- Имеется лишь один период ожидания и два периода активности: начальный и конечный (т. е. $m = 1$). Это ограничение соответствует простому, но распространенному на практике случаю, когда обрабатывающей нити достаточно лишь одного обращения к БД для выполнения запроса.
- Все периоды активности, кроме первого и последнего, пренебрежимо малы по длительности ($t_k \approx 0$, $k = 2, 3, ..., m$). Это ограничение соответствует более сложному случаю, когда для обработки запроса требуется несколько обращений к БД, но работа нити в промежутках между обращениями сводится лишь к сохранению результата обращения (частного результата) во временной структуре данных. Как правило, время этой операции несравнимо меньше времени поиска данных в БД, а также времени выполнения первого и последнего периодов активности, в которых нить выполняет такие действия, как разбор и контроль параметров запроса, открытие сетевого соединения с сервером БД, формирование окончательного результата работы нити из множества частных результатов, закрытие сетевого соединения с сервером БД, освобождение ресурсов и завершение работы.

Утверждение 2. Если значения всех периодов активности, кроме начального и конечного, пренебрежимо малы, т. е. если $t_k \approx 0$ ($k = 2, 3, ..., m$), то

$$T(n) \approx \max\left(nt_1, W_1 + t_1\right) + \sum_{k=2}^{m} W_k + r(n), \qquad (2)$$

где $r(n) = \max(nt_{m+1}, t_{m+1} + nt_1 + W_1 - \max(nt_1, W_1 + t_1))$.

Доказательство утверждения приведено в Приложении 1. Здесь же приведем только пояснение.

Разделим все время работы нитей на три этапа.

Этап 1. Последовательное выполнение 1-го периода активности всеми нитями от 1-й до n-й (время выполнения этапа равно nt_1).

Этап 2. Обработка между моментом завершения этапа 1 и ближайшим моментом начала выполнения ($m + 1$)-го периода активности любой из нитей (время этапа определяется значениями W_k ($k = 1, 2, ..., m$). Важная особенность данного этапа заключается в том, что все попадающие в него периоды активности нитей (а значит, и состояния паузы) близки к нулю по длительности. Другими словами, можно считать, что в течение этого этапа все нити находятся в состоянии ожидания.

Этап 3. Обработка между моментом окончания этапа 2 до завершения работы всех нитей (как будет показано дальше, время выполнения этапа равно $r(n)$).

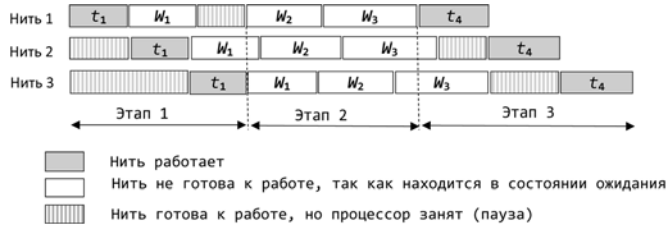


Рис. 2. Пример поведения нитей при соблюдении условия утверждения 2

На рис. 2 приведен пример поведения нитей (при соблюдении условия утверждения 2 для $n = 3$ и $m = 3$ в форме горизонтальной диаграммы с обозначением трех этапов обработки. Три полосы диаграммы, соответствующие трем нитям с номерами 1, 2 и 3, отображают последовательную смену состояний нитей слева направо в единой шкале времени.

Согласно утверждению 1 нить 1 первой завершит m-й период ожидания, а нить n — последней. Обозначим через τ_1 и τ_n величины временных интервалов от начала обработки (т. е. от начала этапа 1) до моментов завершения последнего m-го периода ожидания нитью 1 и нитью n соответственно. Легко видеть, что время τ_1 может быть выражено формулой:

$$\tau_1 \approx \max\left(nt_1, W_1 + t_1\right) \sum_{k=2}^{m} W_k. \qquad (3)$$

Действительно, нить 1 сможет приступить к выполнению 2-го периода активности через интервал времени $\max(nt_1, W_1 + t_1)$ после начала обработки, так как для этого все нити должны закончить выполнение первого периода активности (интервал nt_1), а нить 1 — закончить еще и первый период ожидания (интервал $W_1 + t_1$). Если нить 1 выйдет из 1-го периода ожидания слишком рано, то ей придется подождать освобождения процессора (именно этот случай изображен на рис. 2). Отметим, что попытка выполнить даже пренебрежимо малый период активности (t_2) может вызвать значительную паузу в работе нити, если процессор занят длительной работой. Как видно из определения τ_1 и на рис 2, суммарная длительность этапов 1 и 2 равна τ_1.

Обозначим через r длительность этапа 3. Очевидно, что r не может быть меньшей nt_{m+1} (все нити должны выполнить последний период активности). Вместе с тем нить n сможет приступить к выполнению последнего периода активности лишь после завершения последнего периода ожидания, т. е. через время τ_n после начала обработки. Другими словами, r не может быть меньше, чем $t_{m+1} + \tau_n - \tau_1$. Так как других ограничений на

длительность этапа 3 не имеется, можно выразить r с помощью формулы $r = \max(nt_{m+1},\ t_{m+1} + \tau_n - \tau_1)$. Как видно из определения τ_n и на рис. 2, значение τ_n может быть оценено простым выражением: $nt_1 + \sum_{k=1}^{m} W_k$. Подставив выражения (3) для τ_n и τ_1 в формулу для r, получим $r = \max(nt_{m+1},\ t_{m+1} + nt_1 + W_1 - \max(nt_1 + t_1))$, а $T(n) = \tau_1 + r \approx \max(nt_1,\ W_1 + t_1) + \sum_{k=2}^{m} W_k + r$.

Следствие 1. Если имеется единственный период ожидания, то

$$T(n) = \max(nt_1 W_1 + t_1) + r(n). \qquad (4)$$

Действительно, так как периоды ожидания с номерами больше 1 попросту отсутствуют, то формула (2) приобретает вид (4). Отметим, что в отличие от выражения (2) в данном случае имеет место строгое равенство, так как пренебрежимо малые периоды активности также отсутствуют.

Следствие 2. Если имеется единственный период ожидания и

$$n \geqslant \max\left(\frac{W_1 + t_1}{t_1},\ \frac{W_1 + t_2}{t_2}\right), \qquad (5)$$

то $T(n) = nt_1 + nt_2$ и $T_p = nt_0 + T(n) = nt_0 + nt_1 + nt_2$.

Доказательство. Если выполняется условие (5), то $r(n) = \max(nt_2,\ t_2 + nt_1 + W_1 - nt_1) = \max(nt_2,\ t_2 + W_1) = nt_2$. Подставив выражения для $r(n)$ в формулу (4), получим

$T(n) = \max(nt_1,\ W_1 + t_1) + nt_2 = nt_1 + nt_2$.

Подставив полученное значение в формулу (1), получим

$$T_p = nt_0 + T(n) = nt_0 + nt_1 + nt_2.$$

Практическое значение следствия 1 заключается в том, то оно позволяет определить размер пакетного запроса, при котором гарантированно достигается предельная производительность программного сервера для простого случая с одним периодом ожидания ($m = 1$). В самом деле, если выполняется условие (5), то скорость обработки достигает $\frac{n}{T_p} = \frac{1}{t_0 + t_1 + t_2}$ запросов в единицу времени. Как можно видеть, эта максимальная скорость обработки не зависит ни от n, ни от значения периода ожидания. Этот факт, в частности, означает, что за счет сокращения периода ожидания (т. е. за счет оптимизации запроса к СУБД) нельзя увеличить предельную производительность в этих условиях. Можно лишь добиться того, чтобы максимальная скорость достигалась при меньших значениях n.

Рис. 3. Общий вид зависимости модельной скорости обработки от размера пакета

На рис. 3 приведен общий вид зависимости скорости обработки $n/(nt_0 + T(n))$ от размера пакета n и показано, каким образом меняется вид кривой при сокращении периода ожидания. Если сравнить рис. 3 с рис. 1, то можно заметить определенное сходство в поведении кривых. Небольшое расхождение в предельной производительности у двух экспериментальных кривых на рис. 1 носит случайный характер и объясняется, по-видимому, несколько изменившейся нагрузкой на лабораторную сеть или на сервер БД в промежутке между двумя экспериментами.

Следствие 3. Если $n > 1$ и $W_k > 0$ ($k = 1, 2, ..., m$), то время выполнения пакета запросов, удовлетворяющего условию утверждения 2, T_1 больше или равно времени выполнения равного по размеру пакета простых запросов T_2 с двумя периодами активности, равными t_1 (начальный) и t_{m+1} (конечный) и единственным периодом ожидания, равным $\sum_{k=1}^{m} W_k$. При этом при некоторых сочетаниях параметров запросов имеет место строгое неравенство: $T_1 > T_2$.

Доказательство. Как следует из утверждения 2,

$$T_1 = \max\left(nt_1,\ W_1 + t_1\right) + \sum_{k=2}^{m} W_k + r_1,$$

$$T_2 = \max\left(nt_1,\ \sum_{k=1}^{m} W_k + t_1\right) + r_2,$$

где

$r_1 = \max(nt_{m+1},\ t_{m+1} + nt_1 + W_1 - \max(nt_1,\ W_1 + t_1))$,

$$r_2 = \max\left(nt_{m+1},\ t_{m+1} + nt_1 + \sum_{k=1}^{m} W_k - \right.$$
$$\left. - \max\left(nt_1,\ \sum_{k=1}^{m} W_k + t_1 \right) \right).$$

В отличие от выражения (2) здесь и далее будем использовать строгое равенство в выражении для T_1, т. е. фактически его нижнюю оценку, достигаемую при $t_k = 0$, $k = 2, 3,..., m$.

Разобьем множество возможных размеров пакета n на три непересекающихся интервала:

$$n < W_1/t_1 + 1,\ W_1/t_1 + 1 < n \leqslant \left(\sum_{k=1}^{m} W_k \right) \Big/ t_1 + 1$$

$$\text{и } n > \left(\sum_{k=1}^{m} W_k \right) \Big/ t_1 + 1.$$

Для каждого интервала значений n выражения для T_1 и T_2 могут быть упрощены путем несложных преобразований. В табл. 1 приведены результаты таких преобразований для каждого интервала значений и вытекающее из них соотношение T_1 и T_2.

На рис. 4 проиллюстрирована работа трех одинаковых нитей примерно с теми же параметрами, что и на рис. 2, но с одним существенным отличием: вместо трех периодов ожидания длительностью W_1, W_2 и W_3 имеется один период ожидания длительностью $W_1 + W_2 + W_3$. Ситуация на рис. 4 соответствует второй строке табл. 1. Если сравнить рис. 2 и 4, то легко заметить, что на последнем время обработки пакета заметно сократилось, и хорошо виден источник этого сокращения — отсутствие паузы после первого периода ожидания у нити 1.

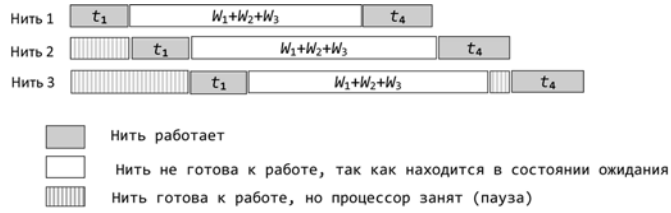


Рис. 4. Пример поведения нитей с одним большим периодом ожидания

Важно отметить, что, как следует из табл. 1, при больших размерах пакета, например, при $n \geqslant \max\left(\left(\sum_{k=1}^{m} W_k + t_1 \right) \Big/ t_1,\ (W_1 + t_{m+1}) \Big/ t_{m+1} \right)$, стабильно выполняется строгое неравенство $T_1 > T_2$. В самом деле, нетрудно заметить, что в этом случае все значения всех функций max в выражениях для T_1, r_1, T_2 и r_2 равны их первому аргументу. А значит $r_1 = r_2 = nt_{m+1}$, а $T_1 = nt_1 + \sum_{k=2}^{m} W_k + nt_{m+1}$ и $T_2 = nt_1 + nt_{m+1}$, т. е. $T_1 > T_2$.

Следствие 2 означает, что при обработке пакетов запросов один большой период ожидания предпочтительнее, чем несколько малых, равных ему в сумме, даже если последние разделены пренебрежимо малыми периодами активности.

2. Модель обработки запросов в многоядерном сервере

Использование многоядерных серверов позволяет многократно увеличить скорость обработки без значительных затрат за счет параллельной обработки сразу нескольких программных нитей

Таблица 1

Соотношение времен обработки T_1 и T_2 при различных размерах пакета n

Условие	Значения T_1 и T_2	Соотношение T_1 и T_2
$n < W_1/t_1 + 1$	$T_1 = t_1 + \sum_{k=1}^{m} W_k + t_{m+1} + (n-1)\max(t_1, t_{m+1})$ $T_2 = t_1 + \sum_{k=1}^{m} W_k + t_{m+1} + (n-1)\max(t_1, t_{m+1})$	$T_1 = T_2$
$W_1/t_1 + 1 < n \leqslant \left(\sum_{k=1}^{m} W_k \right) \Big/ t_1 + 1$	$T_1 = nt_1 + \sum_{k=1}^{m} W_k + \max(nt_{m+1} - W_1, t_{m+1})$ $T_2 = nt_1 + \sum_{k=1}^{m} W_k + \max(nt_{m+1} - (n-1)t_1, t_{m+1})$	Если $t_{m+1} > t_1$, то $T_1 > T_2$ иначе $T_1 = T_2$
$n > \left(\sum_{k=1}^{m} W_k \right) \Big/ t_1 + 1$	$T_1 = nt_1 + \sum_{k=1}^{m} W_k + \max(nt_{m+1} - W_1, t_{m+1})$ $T_2 = nt_1 + \sum_{k=1}^{m} W_k + \max(nt_{m+1} - \sum_{k=1}^{m} W_k, t_{m+1})$	Если $(n-1)t_{m+1} > W_1$, то $T_1 > T_2$ иначе $T_1 = T_2$

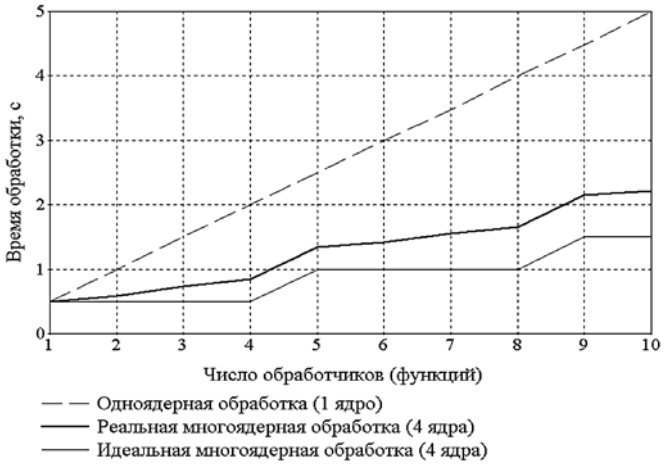


Рис. 5. Влияние многоядерности на скорость обработки пакета простых запросов

в одном процессоре. Это в полной мере относится и к пакетной обработке информационных запросов в многопоточном сервере. Однако представление о том, что R ядер способны сократить время обработки пакета в R раз, было бы слишком грубым упрощением.

На рис. 5 приведены результаты одного из экспериментов по оценке влияния многоядерности на скорость обработки программных нитей в многопоточном сервере с четырехядерным процессором Intel(R) Core(TM) i5-2410M 2.30 GHz. Поведение каждой обрабатывающей нити в эксперименте было нарочито простым: оно включало ровно 0,5 с активной работы с полной нагрузкой процессора без каких-либо периодов ожидания. Штриховая линия отображает зависимость времени обработки пакета от числа запросов в пакете в одноядерном сервере. Как и предполагалось, отсутствие периодов ожидания приводит к строго последовательной работе нитей и к линейной зависимости времени обработки от числа запросов в пакете.

Сплошная тонкая кривая линия отображает ту же зависимость в ситуации идеальной четырехядерной обработки, в которой время параллельной работы четырех нитей в четырех ядрах в точности равно времени работы одной нити (0,5 с). Как видно на графике, все n нитей разбиваются на $g = (n + 3)/4$ групп по 4 нити в каждой. Каждая группа при этом обрабатывается параллельно за 0,5 с, а полное время обработки выражается формулой $0{,}5g$ (здесь и дальше оператор «/» обозначает целочисленное деление с отбрасыванием остатка).

Сплошная полужирная кривая линия отображает ту же зависимость в ситуации реальной четырехядерной обработки, в которой время параллельной работы группы нескольких нитей в четырех ядрах заметно превышает время работы одной нити, причем приращение времени зависит (практически линейно) от числа нитей в группе. В частности, время работы группы из четырех нитей стабильно равнялось $1{,}66t$, где t — время работы одной нити, причем значение коэффициента (1,66) оставалось стабильным с точностью до двух знаков после запятой при t = 0,5, 1 и 2,5 с. Проигрыш в скорости в сравнении с идеальным случаем объясняется тем, что нити в одной группе пользуются общими ресурсами сервера (например, оперативной памятью) и несколько замедляют работу друг друга.

Если исходить из предположения, что время параллельного выполнения периода активности длительностью t группой обрабатывающих нитей в R ядрах линейно зависит от t и в процессорах других типов, то учет фактора многоядерности в рассматриваемой детерминированной модели может быть выполнен по следующим правилам.

- В формулах (2), (4) и (5) параметр n (число запросов в пакете) должен быть заменен на параметр g — число групп параллельно обрабатываемых запросов, определяемое формулой $g = (n + R - 1)/R$, где n — число запросов в пакете; R — число ядер в процессоре.
- Принцип одинаковости запросов должен быть перенесен и на группы. Другими словами, ограничимся рассмотрением пакетов, число запросов в которых кратно R (т. е. все группы, включая и последнюю, являются полными). Единственное исключение сделаем для n = 1: в этом случае многоядерный вариант модели, очевидно, не отличается от одноядерного. Для оценки времени обработки остальных пакетов придется воспользоваться какой-либо формой интерполяции, например, кусочно-линейной.
- При $n > 1$ все значения периодов активности t_j (j = 1, 2, ..., m + 1) должны быть помножены на поправочный коэффициент $C_R > 1$. Этот коэффициент отражает степень замедления выполнения периода активности одной параллельной группой нитей в R ядрах по сравнению с одной нитью (в рассмотренном выше примере он равнялся 1,66). Значения периодов ожидания W_j (j = 1,2,..., m) остаются без изменений.

Заключение

Приведенная аналитическая модель позволяет получить представление о поведении и характеристиках программного сервера при обработке пакетов однотипных запросов, не прибегая к трудоемким экспериментам или имитационному моделированию. Разумеется, она предполагает

использование оценок длительностей основных этапов обработки одиночного запроса (t_i, W_j, $i = 0, 1, ..., m + 1, j = 1, 2, ..., m$). В частности, следствие 2 из утверждения 2 позволяет определить предельную производительность сервера, а также размер пакета запросов, при котором она достигается для простого, но распространенного случая с одним периодом ожидания ($m = 1$).

Следствие 3 из утверждения 2 о преимуществе одного большого периода ожидания перед множеством малых периодов, равных ему в сумме, даже если последние разделены пренебрежимо малыми периодами активности, имеет важное практическое значение, связанное с использованием хранимых функций в СУБД [13—15]. В частности, оно означает, что, если интервалы ожидания в запросе связаны с обращениями к одной и той же БД, то с точки зрения минимизации времени обработки пакета выгоднее разместить всю последовательность обращений к БД в одной хранимой функции (если, разумеется, используемая СУБД и логика запроса допускают это), чем организовать эту последовательность обращений непосредственно в обрабатывающей программной нити. Этот вывод вытекает из того, что обращение к такой хранимой функции выливается в один большой интервал ожидания для нити, а последовательность непосредственных обращений к БД — в несколько малых. Важно подчеркнуть, что речь идет о концептуальном выигрыше, не связанном с какими-либо техническими подробностями реализации (например, с используемой СУБД, языком программирования, операционной средой и т. п.).

Приложение 1. Доказательство Утверждения 2

Обозначим через τ_i интервал времени от начала обработки (т. е. от начала работы нити 1) до момента завершения нитью i m-го периода ожидания W_m. Легко видеть, что τ_i можно выразить формулой

$$\tau_i \approx \max\left(nt_1, W_1 + it_1\right) + \sum_{k=2}^{m} W_k. \qquad (1.1)$$

В самом деле, значение $\max(nt_1, W_1 + it_1)$ отражает момент начала 2-го периода активности нити i. Интервал времени от начала обработки до завершения 1-го периода ожидания нити i включает в качестве слагаемых начальную паузу длительностью $(i - 1)t_1$, длительность 1-го периода активности t_1 и длительность 1-го периода ожидания (т. е. $(i - 1)t_1 + t_1 + W_1 = W_1 + it_1$). Если этот интервал больше или равен длительности этапа 1 (nt_1), то по завершению 1-го периода ожидания нить немедленно начнет выполнение 2-го периода активности, и $\tau_i \approx W_1 + it_1 + \sum_{k=2}^{m} W_k$. В противном случае в работе нити i возникнет пауза до завершения этапа 1, и значение $\tau_i \approx nt_1 + \sum_{k=2}^{m} W_k$.

Из выражения (1.1) очевидно вытекает, что $0 \leqslant \tau_{k+1} - \tau_k \leqslant t_1$ при всех $0 < k < n$. Отсюда при $i < j$ выполняется $\tau_i \leqslant \tau_j$. Поэтому момент завершения m-го периода ожидания нитью 1 (τ_1) является моментом начала этапа 3. Попробуем оценить длительность этого этапа r. Обозначим через r_i интервал времени от начала этапа 3 до момента завершения работы нити i. Как видно на рис. 2, значение r_i не может быть меньше, чем it_{m+1}. Вместе с тем оно не может быть меньше $t_{m+1} + \tau_i - \tau_1$, так как выполнение последнего периода активности нитью i не может начаться раньше τ_i. Докажем, что

$$r_i = \max(it_{m+1}, t_{m+1} + \tau_i - \tau_1) \qquad (1.2)$$

или, подставляя выражения (1.1) для τ_i и τ_1,

$$r_i \approx \max\left(it_{m+1}, t_{m+1} + \max\left(nt_1, W_1 + it_1\right) - \max\left(nt_1, W_1 + t_1\right)\right).$$

Доказательство проведем с помощью индукции по значению i. При $i = 1$ имеем $r_1 \approx \max(t_{m+1}, t_{m+1}) = t_{m+1}$, т. е. (1.2) дает правильный результат. Предположим, что (1.2) выполняется при $i = j$ и докажем, что

$$r_{j+1} = \max((j + 1)t_{m+1}, t_{m+1} + \tau_{j+1} - \tau_1). \qquad (1.3)$$

Для этого рассмотрим интервал времени от начала этапа 3 до момента завершения работы нити $j + 1$. Будем исходить из простого рекуррентного соотношения

$$r_{j+1} = \max(r_j, \tau_{j+1} - \tau_1) + t_{m+1}, \qquad (1.4)$$

которое отражает тот факт, что время начала выполнения последнего периода активности нитью $j + 1$ (считая от начала этапа 3) совпадает или с моментом завершения ею последнего периода ожидания ($\tau_{j+1} - \tau_1$) или с моментом завершения последнего периода активности нитью j (r_j). Согласно предположению индукции, подставим в (1.4) выражение для r_j из (1.2) при $i = j$ и получим

$$r_{j+1} = \max(\max(jt_{m+1}, t_{m+1} + \tau_j - \tau_1), \tau_{j+1} - \tau_1) + t_{m+1}. \qquad (1.5)$$

Таблица 1.1

Интервалы значений $\tau_{j+1} - \tau_j$

Значение $\tau_{j+1} - \tau_j$	Условие
0	$nt_1 \geqslant W_1 + (j+1)t_1$
$(0, t_1)$	$nt_1 < W_1 + (j+1)t_1$ и $nt_1 > W_1 + jt_1$
t_1	$nt_1 \leqslant W_1 + (j+1)t_1$ и $nt_1 \leqslant W_1 + jt_1$

Предположим, что $t_{m+1} \geqslant t_1$. Так как $0 \leqslant \leqslant \tau_{k+1} - \tau_k \leqslant t_1$ при всех $0 < k < n$, то $\tau_j - \tau_1 \leqslant (i-1)t_1$ и $jt_{m+1} \geqslant t_{m+1} + (j-1)t_1 \geqslant t_{m+1} + \tau_j - \tau_1$. Применив это соотношение к (1.5), получим: $r_{j+1} = \max(jt_{m+1}, t_{m+1} + \tau_{j+1} - \tau_1) + t_{m+1} = \max((j+1)t_{m+1}, t_{m+1} + + \tau_{j+1} - \tau_1)$, т. е. в этом случае (1.3) выполняется.

Предположим теперь, что $t_{m+1} < t_1$. Рассмотрим значение $\tau_{j+1} - \tau_j = \max(nt_1, W_1 + (j+1)t_1) - - \max(nt_1, W_1 + jt_1))$. Легко видеть, что эта величина может принимать всего три возможных интервала значений, которые приведены в табл. 1.1.

Легко убедиться, что, если выполняется условие из первой или второй строки таблицы, то $\tau_j - \tau_1 = 0$ (так как при $1 < k < j$ выполняется $\tau_{k+1} - \tau_k = = \max(nt_1, W_1 + (k+1)t_1) - \max(nt_1, W_1 + kt_1) = = nt_1 - nt_1 = 0$). Применим это соотношение в (1.5), получим: $r_{j+1} = \max((j+1)t_{m+1}, t_{m+1} + \tau_{j+1} - \tau_1)$, т. е. в этом случае (1.3) выполняется.

Если же выполняется условие из третьей строки, то $(\tau_{j+1} - \tau_1) - (\tau_j - \tau_1) = \tau_{j+1} - \tau_j = t_1$. Учитывая, что по нашему предположению $t_{m+1} < t_1$, получим $t_{m+1} + \tau_j - \tau_1 < \tau_{j+1} - \tau_1$. Применив это соотношение в (1.5), и в этом случае получим $r_{j+1} = = \max((j+1)t_{m+1}, t_{m+1} + \tau_{j+1} - \tau_1)$ (так как для любых a, b и c при $c \geqslant b$ выполняется $\max(\max(a, b), c) = = \max(a, c)$).

В результате имеем:

$$T(n) \approx \tau_1 + r_n \approx \max(nt_1,\ W_1 + t_1) + \sum_{k=2}^{m} W_k + r_n,$$

где $r_n = \max(nt_{m+1},\ t_{m+1} + \tau_n - \tau_1) \approx \max(nt_{m+1},\ t_{m+1} + \max(nt_1,\ W_1 + nt_1) - \max(nt_1,\ W_1 + t_1)) = = \max(nt_{m+1},\ t_{m+1} + W_1 + nt_1 - \max(nt_1,\ W_1 + t_1))$.

Конец доказательства.

Список литературы

1. **Muter I.** Exact algorithms to minimize makespan on single and parallel batch processing machines // European Journal of Operational Research. 2020. Vol. 285, No. 2. P. 470—483. DOI: 10.1016/j.ejor.2020.01.065.

2. **Ding S., Attenberg J., Baeza-Yates R., Suel T.** Batch query processing for web search engines // Proceedings of the fourth ACM international conference on Web search and data mining. NY: ACM, 2011. P. 137—146. DOI: 10.1145/1935826.1935858.

3. **Tannenbaum A., Bos H.** Modern Operating Systems. N. J.: Pearson Education, 2015, 1136 p.

4. **Келли-Бутл С.** Введение в Unix. М.: ЛОРИ, 1995. 596 с.

5. **Волосатова Т. М., Киселев И. А., Князева С. В.** Многопоточная обработка в POSIX стандарте // Восточно-Европейский научный журнал. 2021. № 12-3. С. 37—39. DOI: 10.31618/ESSA.2782-1994.2021.3.76.214.

6. **Larina T. B., Filipchenko A. S.** Experiments with multithreaded processing in RPA Robin // Information Innovative Technologies: International Scientific-Practical Conference, Prague. Moscow: Association of graduates and employees of AFEA named after prof. Zhukovsky, 2022. P. 245—252.

7. **Ткаченко В. И.** Перспективы развития операционных систем // Научные достижения и инновации: вопросы теории и практики: Материалы XIV Всероссийской научно-практической конференции, Ростов-на-Дону, 15 сентября 2022 г. Ростов-на-Дону: Параграф, 2022. С. 46—49.

8. **Mamonov A. A., Salpagarov S. I.** The distributed computing system as a specific case of queueing system // The 5th International Conference on Stochastic Methods (ICSM-5). Moscow: RUDN University, 2020. P. 102—106.

9. **Sahakyan V., Vardanyan A.** The Queue State for Multiprocessor System with Waiting Time Restriction // 2019 Computer Science and Information Technologies (CSIT): Revised selected papers. Yerevan: IEEE, 2019. P. 116—119. DOI: 10.1109/CSITechnol.2019.8895093.

10. **Коффман Э. Г.** Теория расписаний и вычислительные машины. М.: Наука, 1984. 336 с.

11. **Chernykh K. A., Servakh V. V.** Combinatorial structure of optimal solutions to the problem of a single machine with preemption// 15th International Asian School-Seminar Optimization Problems of Complex Systems, OPCS 2019. Novosibirsk, 2019. P. 21—26. DOI: 10.1109/OPCS.2019.8880148.

12. **Romanova A. A., Servakh V. V.** Complexity of cyclic job shop scheduling problem for identical jobs with no-wait constraints// Journal of Applied and Industrial Mathematics. 2019. Vol. 13, No. 4. P. 706—716. DOI: 10.1134/S1990478919040136.

13. **Прибыл Б., Фейерштейн С.** Oracle PL/SQL для профессионалов. СПб: Питер, 2015. 1024 с.

14. **Feuerstein S.** Oracle PL/SQL Programming. O'Reilly, 2014. 1392 p.

15. **Ferrari L., Pirozzi E.** Learn PostgreSQL. Birmingham: Packt, 2020. 650 p.

Deterministic Model of Information Queries Batch Processing in a Multithreaded Server

R. E. Asratian, Leading Researcher, rubezas@yandex.ru,
V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, Moscow, 117997, Russian Federation

Corresponding author:

Ruben E. Asratian, Leading Researcher,
V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, Moscow, 117997, Russian Federation
E-mail: rubezas@yandex.ru

Received on January 16, 2023
Accepted on February 02, 2023

The analytical model of processing a batch of similar requests in a multithreaded server is considered. The model is based on the high determinism of the duration of individual stages of request processing and the moments of events leading to the switching of the processor from one processing program thread to another. Unlike other deterministic models based on the theory of schedules, in this case the goal is not to find the optimal distribution of work across processors, but to study the dependence of the main characteristics of the software server (packet processing time, marginal performance) on the characteristics of the request and the size of the batch. The model concerns the situation when distributed system with intensive flows of information queries is developed and special measures to ensure the required performance must be taken. One of these measures is the introduction of batch query processing tools, allowing an array of similar information queries to be processed in a single service call. The introduction of such tools usually allows you to repeatedly increase the rate of processing queries in the system and very often becomes a real «lifesaver» for customers who issue, for example, hundreds of thousands of queries per day or more. It is quite likely that a sequence of ordinary single queries cannot provide a solution, and batch processing simply has no alternative

Keywords: *deterministic model, batch queries, batch processing, multithreaded server, multicore server, multithreading*

For citation:

Asratian R. E. Deterministic Model of Information Queries Batch Processing in a Multithreaded Server, *Programmnaya ingeneria,* 2023, vol. 14, no. 4, pp. 155—164. DOI: 10.17587/prin.14.155-164.

References

1. **Muter I.** Exact algorithms to minimize makespan on single and parallel batch processing machines, *European Journal of Operational Research*, 2020, vol. 285, no. 2, pp. 470—483. DOI:10.1016/j.ejor.2020.01.065.

2. **Ding S., Attenberg J., Baeza-Yates R., Suel T.** Batch query processing for web search engines, *Proceedings of the fourth ACM international conference on Web search and data mining*, NY, ACM, 2011, pp. 137—146. DOI: 10.1145/1935826.1935858.

3. **Tannenbaum A., Bos H.** *Modern Operating Systems*, N. J., Pearson Education, 2015, 1136 p.

4. **Stevens W., Rago S.** *Advanced programming in the UNIX environment*, Addison-Wesley Professional, 2013, 1032 p.

5. **Volosatova T. M., Kiselev I. A., Knyazeva S. V.** Multithreaded processing in POSIX standard, *Eastern European Scientific Journal*, 2021, no. 12-3, pp. 37—39. DOI: 10.31618/ESSA.2782-1994.2021.3.76.214 (in Russian).

6. **Larina T. B., Filipchenko A. S.** Experiments with multithreaded processing in RPA Robin, *Information Innovative Technologies: International Scientific-Practical Conference*, Prague, Moscow, Association of graduates and employees of AFEA named after prof. Zhukovsky, 2022, pp. 245—252.

7. **Tkachenko V. I.** Prospects for the development of operating systems, *Scientific achievements and innovations: questions of theory and practice: Materials of the XIV All-Russian Scientific and Practical Conference*, Rostov-on-Don, September 15, 2022, Rostov-on-Don, Paragraph, 2022, pp. 46—49 (in Russian).

8. **Mamonov A.A., Salpagarov S. I.** The distributed computing system as a specific case of queueing system, *The 5th International Conference on Stochastic Methods (ICSM-5),* Moscow, RUDN University, 2020, pp. 102—106.

9. **Sahakyan V., Vardanyan A.** The Queue State for Multiprocessor System with Waiting Time Restriction, *2019 Computer Science and Information Technologies (CSIT)*: Revised selected papers, Yerevan, IEEE, 2019, pp. 116—119. DOI: 10.1109/CSITechnol.2019.8895093.

10. **Coffman E. G.** *Computer and job-shop scheduling theory,* NY, Wiley, 1976, 299 p.

11. **Chernykh K. A., Servakh V. V.** Combinatorial structure of optimal solutions to the problem of a single machine with preemption, *15th International Asian School-Seminar Optimization Problems of Complex Systems, OPCS 2019*, Novosibirsk, 2019, pp. 21—26. DOI: 10.1109/OPCS.2019.8880148.

12. **Romanova A. A., Servakh V. V.** Complexity of cyclic job shop scheduling problem for identical jobs with no-wait constraints, *Journal of Applied and Industrial Mathematics*, 2019, vol. 13, no. 4, pp. 706—716. DOI: 10.1134/S1990478919040136.

13. **Feuerstein S.** *Oracle PL/SQL Programming,* O'Reilly, 2014, 1392 p.

14. **Worsley J., Drake J.** *PostgreSQL for professionals*, Saint Petersburg, Piter, 2003, 496 p. (in Russian).

15. **Ferrari L., Pirozzi E.** *Learn PostgreSQL*, Birmingham, Packt, 2020, 650 p.

УДК 004.052.4
DOI: 10.17587/prin.14.165-174

Д. С. Степанов, ст. препод., stepanov0995@gmail.com,
В. М. Ицыксон, канд. техн. наук, доц., itsykson@yandex.ru
Санкт-Петербургский политехнический университет Петра Великого

Поиск дубликатов ошибок компиляторов методом генерации программ-свидетелей

Поступила в редакцию 09.02.2023
Принята к публикации 27.02.2023

Компиляторы языков программирования — сложнейшие программные системы, от качества которых напрямую зависит качество создаваемых с их помощью программ. Поэтому предъявляются повышенные требования к таким характеристикам качества, как функциональная пригодность, надежность, производительность, безопасность и т. д. Для обеспечения качества используются различные средства: пользовательское и ручное тестирование, средства для автоматического поиска ошибок. При этом очень часто возникают ситуации, когда одна и та же ошибка обнаруживается многократно, а тестовые примеры, на которых она проявляется, друг на друга совсем не похожи. Такие тестовые примеры называют дубликатами, их определение — актуальная и острая проблема, так как их ручной поиск требует больших человеческих ресурсов. Алгоритмы автоматического поиска дубликатов помогли бы значительно упростить процесс поддержки компилятора.

Основная идея представленного в статье подхода для поиска дубликатов заключается в том, что причины возникновения одинаковых ошибок находятся в одном и том же месте исходного кода компилятора. Для поиска этого места используется метод генерации программ-свидетелей. Он состоит в том, что для каждой тестовой программы, содержащей ошибку, происходит генерация похожих программ, которые ее не содержат. После этого вычисляются метрики, основанные на покрытии исходного кода компилятора, и формируется список файлов, которые потенциально содержат причину неисправности компилятора. Если эти списки для двух тестовых программ похожи с точки зрения метрики близости, то тестовые программы считаются дубликатами.

Предлагаемый подход был разработан и реализован для компилятора языка программирования Kotlin. Тестирование показало применимость предлагаемого подхода для решения задачи поиска дубликатов ошибок компиляторов языков программирования.

Ключевые слова: *тестирование компиляторов, определение дубликатов ошибок, изоляция ошибок*

Для цитирования:

Степанов Д. С., Ицыксон В. М. Поиск дубликатов ошибок компиляторов методом генерации программ-свидетелей // Программная инженерия. 2023. Том 14, № 4. С. 165—174. DOI: 10.17587/prin.14.165-174.

Введение

Компиляторы языков программирования — сложнейшие программные системы, ошибки в которых могут привести к критическим последствиям и большим экономическим потерям. К сожалению, ошибки в них далеко не редкость, в чем можно убедиться, проанализировав багтрекер любого популярного компилятора[1].

Ошибка (или дефект) компилятора может проявляться не только в аварийном завершении работы с исключением (падение), но и в ситуациях мискомпляции (англ. *miscompilation*), когда результирующий бинарный код или байт-код создается, но не соответствует исходной семантике программы, или когда сгенерированный код соответствует семантике, но работает слишком медленно, а также в других случаях.

Рассмотрим процесс поиска сбоев. Несмотря на большой объем ручного тестирования и использова-

[1] См., например, https://bugs.java.com/bugdatabase/, https://bugs.llvm.org/buglist.cgi, https://youtrack.jetbrains.com/issues/KT

ние средств автоматического поиска ошибок, с дефектами чаще всего сталкиваются пользователи. При этом, учитывая большое число источников получения информации о дефектах, специалисты поддержки зачастую сталкиваются с ситуацией, когда им сообщают об одном и том же дефекте множество раз.

Поиск дублирующихся ошибок или дубликатов — одна из самых острых проблем в процессе поддержки компилятора. Под дубликатами в данной работе будем понимать разные программы на целевом языке программирования, приводящие к одинаковым ошибкам компилятора. Ошибки компиляции таких программ вызваны одной и той же причиной, сами же программы отличаются. При этом очевидно, что разработчику компилятора достаточно дать информацию только об одной ошибке либо указать, что это несколько проявлений одной и той же ошибки. В настоящее время поиск дубликатов обычно проводится вручную, что требует высокой компетенции специалистов и значительных временных затрат. При использовании автоматических средств обнаружения дефектов компилятора поиск дубликатов также является востребованной задачей, так как ежедневно могут генерироваться сотни приводящих к дефектам тестовых программ, подавляющее большинство из которых являются дубликатами.

В настоящей статье представлен разработанный авторами подход для обнаружения дубликатов ошибок компилятора, основанный на генерации программ-свидетелей. Основная идея метода базируется на том, что дублирующиеся ошибки находятся в одном и том же месте исходного кода компилятора. Для определения места, являющегося причиной дефекта, применяется метод изоляции ошибок, формирующий список файлов исходного кода компилятора, которые потенциально являются их причиной. При достаточном сходстве списков файлов можно считать тестовые программы дубликатами. Основным преимуществом данного подхода является его применимость для любых типов дефектов компилятора: падений, мискомпиляций, деградаций производительности и т. д.

Предлагаемый в работе подход был реализован для компилятора языка программирования Kotlin, для которого задача поиска дубликатов является острой и важной. На момент написания статьи (начало 2023 г.) баг-трекер компилятора[1] содержит 33 786 ошибок, 4542 из которых помечены командой поддержки дубликатами[2]. Тестирование показало, что предлагаемый подход способен эффективно выявлять одинаковые дефекты компилятора и является полезным средством для специалистов поддержки и незаменимым для автоматических средств тестирования.

1. Обзор предметной области

Основополагающей работой в области определения дубликатов ошибок компилятора можно считать работу [1]. В ней предложен метод для сортировки найденных ошибок. Входными данными для этого алгоритма является список тестовых программ, приводящих к дефектам компилятора. Выходом является отсортированный список, в котором тестовые программы, с наибольшей вероятностью приводящие к разным ошибкам, оказываются в его начале. В рассматриваемой работе авторы предлагают сортировать список, основываясь на сравнении векторов, состоящих из набора метрик, основанных на определенных статических и динамических характеристиках тестовых программ. После вычисления описанных векторов применяется алгоритм самой дальней точки [2], с помощью которого и формируется выходной список. Минусами данного подхода являются сложность в реализации и плохая применимость к дубликатам, имеющим разную структуру и минимальное синтаксическое сходство.

Холмс и Гроус [3] в своей работе предлагают метрику, основанную на мутациях, основная идея которой заключается в том, что, если две тестовые программы, приводящие к ошибке, перестают воспроизводиться на одинаковых мутантах исходного кода компилятора, то с высокой долей вероятности они являются дубликатами. Под мутацией понимается какое-либо изменение кода, например замена арифметических операторов или констант. Рассмотрим метод более подробно: сначала строится множество мутантов исходного кода компилятора, затем проводится детекция мутантов, на которых ошибки, к которым приводит пара тестовых программ, перестают воспроизводиться. Затем для пары потенциальных дубликатов вычисляется расстояние Жаккара, основанное на найденных мутантах. После этого также на основе алгоритма самой дальней точки проводится ранжирование тестов, при котором в начале списка находятся тесты, сгенерированные мутанты для которых максимально отличаются. Недостатками данного метода являются сложность генерации мутантов для исходного кода компилятора и объем необходимых вычислительных ресурсов для его работы.

[1] https://youtrack.jetbrains.com/issues/KT
[2] https://youtrack.jetbrains.com/issues/KT?q=%23 Duplicate%20

Еще одним подходом к поиску дубликатов является метод, основанный на сравнении сообщений об ошибках. Для сравнения сообщений используется алгоритм Майерса [4], суть которого состоит в поиске набора изменений для преобразования одного сообщения об ошибке в другое. Результатом работы алгоритма является набор совпадающих и различающихся символов для двух сообщений об ошибках. Далее сумма длин различающихся последовательностей делится на сумму длин совпадающих последовательностей. После этого на основе данной метрики при помощи алгоритма самой дальней точки сортируются найденные ошибки. Недостатком подхода является то, что сообщения об ошибках возникают только в случае падения компилятора, в случае других типов дефектов данный метод неприменим.

Смежной научной областью с поиском дубликатов является область изоляции ошибок, методы из которой также определяют место в исходном коде программы, ставшее причиной ошибки. Несмотря на большое число работ [5—7], методы, которые в них используются, плохо подходят для компиляторов, потому что не учитывают сложность таких программных проектов. Большинство существующих работ, направленных на изоляцию ошибок компилятора [8—10], основываются на визуализации ошибки или сообщении о ней.

Все рассмотренные методы либо не подходят для решения задачи обнаружения дубликатов, так как являются слишком ресурсоемкими, либо допускают ошибки: в приоритезированном списке рядом могут оказаться дубликаты и, наоборот, уникальная критическая ошибка может оказаться в его конце, а также не подходят для определенных видов дефектов или являются слишком медленными. Предлагаемый авторами способ подходит для любых видов ошибок и применим на практике ввиду его низкой ресурсоемкости.

2. Поиск дубликатов методом программ-свидетелей

В настоящей работе предлагается определять дубликаты ошибок компилятора с помощью метода локализации ошибок, основываясь на гипотезе о том, что причины появления дубликатов находятся в одном и том же месте исходного кода компилятора. В качестве подхода для локализации ошибок компилятора используется метод генерации программ-свидетелей, представленный в работе [11]. В этом подходе, названном DiWi, проблема локализации ошибки сводится к проблеме поиска набора программ-свидетелей, помогающих исключить файлы исходного кода компилятора, которые не являются причиной найденного дефекта. Программой-свидетелем является тестовый пример, схожий с исходным, но в отличие от него, не приводящий к ошибке компилятора. Генерация таких программ происходит с помощью применения различных мутаций к исходному тестовому примеру, приводящему к сбою. Под мутацией подразумевается небольшое изменение программы, при котором сохраняются ее синтаксическая и семантическая корректность. Примером мутации может служить замена арифметического оператора на другой или перестановка строк кода местами. После генерации набора программ-свидетелей проводится ранжирование файлов исходного кода компилятора на основе сравнения покрытия между сгенерированным набором и исходной тестовой программой с помощью метрик, предложенных в работе [12].

2.1. Генерация программ-свидетелей

Как было отмечено ранее, для успешной работы метода локализации ошибок для каждого приводящего к ошибке тестового примера необходимо сгенерировать набор программ-свидетелей, к которым предъявляются следующие требования:

1) программа-свидетель должна быть корректной и не приводить к ошибке компилятора;

2) программа-свидетель должна быть схожа с исходной;

3) программы-свидетели должны быть разнообразными с точки зрения покрытия исходного кода компилятора.

Для удовлетворения второго критерия все мутации, применяемые к исходной программе, не должны вносить глобальные изменения, существенно модифицировать ее структуру. Для удовлетворения третьего критерия мутации должны быть как можно более разнообразными.

Процесс генерации программы-свидетеля состоит из двух шагов:

- построение представления исходной тестовой программы, удобной для проведения мутаций;
- мутация построенного представления.

Как известно, компиляторы не работают напрямую с исходным кодом программы, а предварительно преобразовывают его в структурированное представление, как правило, в абстрактное синтаксическое дерево. Далее синтаксическая информация в дереве обогащается необходимой семантической информацией, например данными о типах. Такое представление удобно для мутации, так как оно позволяет точно и быстро оперировать программными конструкциями.

Всего в рамках разработанного подхода предлагается около 20 мутаций, каждая из которых направлена на генерацию программ-свидетелей, удовлетворяющих перечисленным в начале раздела критериям. Важно отметить, что применяемые мутации не гарантируют семантическую корректность результирующего мутанта: после применения мутации необходимо ее проверять и, в случае проблем, откатывать в предыдущее состояние. В данном подразделе будут описаны наиболее интересные мутации, а с полным списком можно ознакомиться в исходном коде инструмента, реализованного для компилятора языка программирования Kotlin[1].

2.1.1. Мутация абстрактного синтаксического дерева

Данная мутация предназначена для изменения абстрактного синтаксического дерева без использования какой-либо семантической информации. Мутации предполагают добавление, удаление или замену узлов в дереве. Замена узлов предполагается на узлы того же типа из исходного дерева или дерева, представляющего другую программу.

Пример замены узла на узел того же типа из другого дерева показан на рис. 1.

Для корректной работы этой мутации необходимо собрать базу данных абстрактных синтаксических деревьев, представляющих код на желаемом языке программирования. Чаще всего такая база собирается из тестовой выборки компилятора, для построения каждый тестовый пример из извлеченного множества конвертируется из текстового представления в структурированное, после чего собирается информация о типах каждого узла и том, где они находятся. Далее в исходном тестовом примере выбирается случайный узел, который заменяется на узел того же типа из другого дерева. После применения мутации проводится проверка синтаксической и семантической корректности результата, и в случае некорректности осуществляется откат в предыдущее состояние.

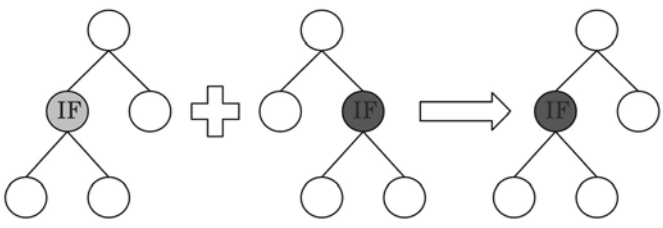


Рис. 1. Пример замены узла в исходном дереве на узел того же типа из другого дерева

[1] https://github.com/DaniilStepanov/bbfgradle

2.1.2. Замена выражений на константы

Данная мутация заменяет значение произвольного выражения на константу того же типа. Для корректной работы этой мутации необходима информация о типах выражений. Для языков программирования со статической типизацией информацию о типах всех выражений достаточно просто можно получить из компилятора, для языков с динамической типизацией можно воспользоваться дополнительными инструментами для вывода типов. Пример работы данной мутации для тестового примера на языке Kotlin представлен на рис. 2.

Для каждого выражения в исходном тестовом примере вычисляется его тип, дальше случайно выбранное выражение заменяется на константу того же типа. Константы, которые подставляются вместо выражений, имеют не только примитивные типы; при необходимости генерируется объект произвольного типа. Стоит отметить, что данная мутация может заменить и результаты вызова других функций.

Остальные мутации являются довольно простыми и представляют собой добавление или изменение модификаторов, операторов, удаление строк или «перемешивание» узлов в абстрактном синтаксическом дереве. На рис. 3 представлен тестовый пример, приводящий к сбою компилятора языка Kotlin, и сгенерированные для него программы-свидетели.

2.2. Определение дубликатов на основе сгенерированных программ-свидетелей

После формирования выборки программ-свидетелей происходит изоляция ошибки, т. е. выявление файлов исходного кода компилятора, потенциально содержащих в себе причину возникшего сбоя. Для изоляции ошибок используется метод, предложенный в работе [13], на основе которого для каждой покрытой при компиляции приводящего к ошибке тестового примера строки исходного кода компилятора вычисляется индекс подозрительности. По аналогии с работой [11] для этого используется индекс Отиай, который является одним из самых эффективных для изоляции ошибок. Формула для расчета индекса Отиай выглядит следующим образом:

$$sus(l)=\frac{ef_l}{\sqrt{(ef_l+nf_l)(ef_l+ep_l)}},$$

где ef_l и nf_l — число тестовых программ, приводящих к сбою, покрывающих и не покрывающих строку l исходного кода компилятора; ep_l — число

```
fun f() {
    val a = 1
    val b = 2
    val c = a + b
}
```
а)

```
fun f() {
    val a = [Int]
    val b = [Int]
    val c = [Int]
}
```
б)

```
fun f() {
    val a = 1
    val b = 2
    val c = b + 141
}
```
в)

Рис. 2. Пример замены выражения на константу для программы на языке Kotlin:
а — исходная программа; *б* — информация о типах используемых выражений; *в* — результирующий мутант

```
inline fun f(b: (i: Int) -> Int)  {
    for (i in "")
        try {
            b(test1())
        } finally {
            break
        }
}
fun test1(): Int = TODO()
fun box(): Any {
    f {  '1'.compareTo(return "") }
    return ""
}
```
а)

```
inline fun f(b: (i: Int) -> Int)  {
    for (i in "")
        try {
            b(test1())
        } finally {
            try {}
            catch (e :Exception)
            {}
        }
}
fun test1(): Int = TODO()
fun box(): Any {
    f {  '1'.compareTo(return "") }
    return ""
}
```
б)

```
inline fun f(b: (i: Int) -> Int)  {
    for (i in "")
        try {
            b(break)
        } finally {
            test1()
        }
}


fun test1(): Int = TODO()
fun box(): Any {
    f {  '1'.compareTo(return "") }
    return ""
}
```
в)

Рис. 3. Пример генерации программ-свидетелей для тестового примера, приводящего к сбою компилятора языка Kotlin:
а — исходный тестовый пример; *б* — сгенерированная программа-свидетель 1; *в* — сгенерированная программа-свидетель 2

тестовых программ, не приводящих к сбою, покрывающих строку l.

Так как программа, приводящая к ошибке, всего одна, формулу можно упростить:

$$sus(l)=\frac{ef_l}{\sqrt{(1+ep_l)}}.$$

Затем, согласно работе [12], вычисляется значение подозрительности для каждого файла исходного кода компилятора:

$$SUS(f)=\frac{\sum_{i=1}^{n_f} sus(l_i)}{n_f},$$

где n_f — число строк исходного кода компилятора, покрытых при компиляции тестового примера, приводящего к ошибке.

После этого для каждой пары потенциальных дубликатов f_i и f_j необходимо сравнить выборки файлов, используя косинусное расстояние:

$$k(i,\ j)=\frac{\sum_{k=1}^{n}\left(SUS\left(f_{ik}\right)SUS\left(f_{jk}\right)\right)}{\sqrt{\sum_{k=1}^{n}SUS\left(f_{ik}\right)^2}\sqrt{\sum_{k=1}^{n}SUS\left(f_{jk}\right)^2}},$$

где n — число файлов исходного кода компилятора, покрытых при компиляции тестового примера, приводящего к ошибке.

После получения коэффициента схожести ошибок k с помощью алгоритма самой дальней точки формируется ранжированный список тестовых примеров, в левой части которого с большой долей вероятности содержатся разные ошибки, а в правой — дубликаты.

Алгоритм самой дальней точки для ранжирования тестовых примеров представлен на рис. 4. Создаются результирующий список N, в которой помещается случайный тестовый пример, и список S, в который помещаются остальные тесты, приводящие к ошибке компилятора. Далее для каждого элемента S считается наибольший по коэффициенту сходства пример из N, после чего из S выбирается тест, у которого вычисленный максимальный коэффициент сходства минимален, далее он перемещается из списка S в список N. Алгоритм повторяется до тех пор, пока список S не станет пустым.

Вход: S – список тестовых примеров
Выход: N – ранжированный список тестовых примеров

```
randomElementFromS ← random(S)
N+ = randomElelemtFromS
S− = randomElelemtFromS
while length(S)! = 0 do
    maxSimilarities ← []
    for s ∈ S do
        maxSimilarity ← max(similarity(s, N))
        maxSimilarities+ = maxSimilarity
    end for
    sWithMinSimilarity ← minSimilarity(S, maxSimilarities)
    N+ = sWithMinSimilarity
    S− = sWithMinSimilarity
end while
return N
```

Рис. 4. Алгоритм самой дальней точки для ранжирования тестовых примеров

3. Разработка инструментария поиска дубликатов на основе описанного метода генерации программ-свидетелей

На основе описанного выше метода был разработан инструментарий для поиска дубликатов ошибок компилятора языка Kotlin. Этот язык является молодым и популярным языком программирования, в 2019 г. ставшим основным для разработки приложений платформы Android. Баг-трекер компилятора на момент написания статьи (начало 2023 г.) содержит 33 786 отчетов об ошибках, 4542 из которых помечены командой разработки компилятора как дубликаты.

Для реализации инструментария необходимо обеспечить возможность построения и модификации структурного представления кода на языке Kotlin, а также возможность запуска программы и получения покрытия с минимальными ресурсными затратами. Все необходимые механизмы уже реализованы внутри компилятора, поэтому было решено использовать его как библиотеку. Инфраструктура компилятора Kotlin позволяет использовать структурное представление кода, называющееся Program Structure Interface[1], которое кроме самого кода содержит информацию о типах, дескрипторах и т. д. и позволяет без труда модифицировать программу, проверять ее синтаксическую и семантическую корректность.

Для получения покрытия был разработан модуль для инструментирования компилятора Kotlin, который, в отличие от существующих инструментальных средств [14—16], не собирает лишней информации и работает только в необходимых модулях компилятора, чем практически не вносит задержек в процесс компиляции программ. Для инструментирования в байт-код компилятора вставляются специальные инструкции, которые сообщают о факте покрытия конкретной строки.

Типы ошибок, дубликаты которых может обнаруживать представленный инструментарий, не ограничиваются падениями компилятора. В нем также поддерживается детекция следующих видов ошибок: мискомпиляций, сбоев в плагинах, деградаций производительности и т. д. Необходимо отметить, что данный инструментарий был встроен в автоматическую систему поиска ошибок компилятора языка Kotlin [17], для работы которой критически необходим алгоритм определения дубликатов, так как число найденных ошибок исчисляется сотнями в день, при этом подавляющее большинство из них является дубликатами.

4. Апробация

Для проведения апробации инструментария были извлечены все приводящие к ошибкам тестовые примеры из баг-трекера компилятора Kotlin, актуальные для версии компилятора 1.5.10. Для этого была разработана утилита, разбирающая все отчеты об ошибках, извлекая тестовые примеры, которые к ним приводят, и далее проверяющая,

[1] https://plugins.jetbrains.com/docs/intellij/psi.html

воспроизводятся ли они на нужной версии компилятора. Из извлеченного тестового набора были выбраны три случайные выборки по десять тестовых примеров. Тот факт, что тестовые примеры из выборок являются дублирующимися, подтвержден специалистами команды разработки компилятора специальными метками в баг-трекере (например, дубликаты из выборки 1 — https://youtrack.jetbrains.com/issue/KT-46385/ и https://youtrack.jetbrains.com/issue/KT-46100/). Все три выборки составлены из ошибок разных типов: в первой находятся тесты, приводящие к падениям, во второй — приводящие к мископиляциям, в третьей —приводящие к деградациям производительности. На генерацию программ-свидетелей для каждого тестового примера было выделено 10 мин, как компромисс между качеством работы метода и его применимостью на практике. Для наглядности для каждой выборки было проведено попарное сравнение множеств подозрительных файлов, результаты которого приведены в табл. 1—3. Был также применен алгоритм самой дальней точки, который ранжировал ошибки согласно их уникальности. Эксперименты проводились на компьютере с процессором AMD Ryzen 7 4800H и 32 GB оперативной памяти.

В каждой из таблиц тестовые примеры пронумерованы в формате KT-*m*-*n*, где *m* — группа дубликатов; *n* — порядковый номер примера

Таблица 1

Результаты попарных сравнений тестовых примеров из выборки 1

Тестовые примеры	KT-1-1	KT-1-2	KT-1-3	KT-2-1	KT-2-2	KT-2-3	KT-3-1	KT-4-1	KT-4-2	KT-4-3
KT-1-1		0,97	0,96	0,27	0,27	0,27	0,83	0,88	0,88	0,86
KT-1-2	0,97		0,97	0,28	0,28	0,28	0,82	0,86	0,86	0,87
KT-1-3	0,96	0,97		0,26	0,26	0,26	0,81	0,84	0,84	0,85
KT-2-1	0,27	0,28	0,26		1	1	0,27	0,28	0,28	0,28
KT-2-2	0,27	0,28	0,26	1		1	0,27	0,28	0,28	0,29
KT-2-3	0,27	0,28	0,26	1	1		0,27	0,28	0,28	0,28
KT-3-1	0,83	0,82	0,81	0,27	0,27	0,27		0,95	0,86	0,87
KT-4-1	0,88	0,86	0,84	0,28	0,28	0,28	0,95		0,9	0,91
KT-4-2	0,85	0,86	0,84	0,28	0,28	0,28	0,86	0,9		1
KT-4-3	0,86	0,87	0,85	0,28	0,29	0,29	0,87	0,91	1	

Таблица 2

Результаты попарных сравнений тестовых примеров из выборки 2

Тестовые примеры	KT-1-1	KT-2-1	KT-2-2	KT-3-1	KT-3-2	KT-4-1	KT-4-2	KT-5-1	KT-5-2	KT-5-3
KT-1-1		0,96	0,96	0,95	0,95	0,93	0,93	0,97	0,97	0,97
KT-2-1	0,96		1	0,96	0,96	0,95	0,95	0,97	0,97	0,97
KT-2-2	0,96	1		0,96	0,96	0,95	0,95	0,97	0,97	0,97
KT-3-1	0,95	0,96	0,96		1	0,93	0,93	0,96	0,96	0,96
KT-3-2	0,95	0,96	0,96	1		0,93	0,93	0,96	0,96	0,96
KT-4-1	0,93	0,95	0,95	0,93	0,93		1	0,94	0,94	0,94
KT-4-2	0,93	0,95	0,95	0,93	0,93	1		0,94	0,94	0,94
KT-5-1	0,97	0,97	0,97	0,96	0,96	0,94	0,94		1	1
KT-5-2	0,97	0,97	0,97	0,96	0,96	0,94	0,94	1		1
KT-5-3	0,97	0,97	0,97	0,96	0,96	0,94	0,94	1	1	

Таблица 3

Результаты попарных сравнений тестовых примеров из выборки 3

Тестовые примеры	КТ-1-1	КТ-2-1	КТ-2-2	КТ-2-3	КТ-2-4	КТ-2-5	КТ-3-1	КТ-3-2	КТ-3-3	КТ-3-4
КТ-1-1		0,91	0,92	0,93	0,92	0,97	0,9	0,9	0,95	0,95
КТ-2-1	0,91		0,95	0,93	0,96	0,91	0,92	0,92	0,89	0,88
КТ-2-2	0,92	0,95		0,92	0,97	0,89	0,95	0,95	0,9	0,9
КТ-2-3	0,93	0,93	0,92		0,9	0,93	0,92	0,92	0,91	0,9
КТ-2-4	0,92	0,96	0,97	0,9		0,9	0,92	0,92	0,91	0,91
КТ-2-5	0,97	0,91	0,89	0,93	0,9		0,87	0,87	0,92	0,92
КТ-3-1	0,9	0,92	0,95	0,92	0,92	0,87		1	0,88	0,88
КТ-3-2	0,9	0,92	0,95	0,92	0,92	0,87	1		0,88	0,88
КТ-3-3	0,95	0,89	0,9	0,91	0,91	0,92	0,88	0,88		1
КТ-3-4	0,95	0,88	0,9	0,9	0,91	0,92	0,88	0,88	1	

Таблица 4

Результирующие ранжированные списки для тестовых выборок

Номер выборки	Элемент									
	1	2	3	4	5	6	7	8	9	10
1	КТ-1-1	КТ-2-1	КТ-3-1	КТ-4-2	КТ-1-3	КТ-4-1	КТ-1-2	КТ-4-3	КТ-2-2	КТ-2-3
2	КТ-1-1	КТ-4-1	КТ-3-2	КТ-2-1	КТ-5-1	КТ-3-1	КТ-5-2	КТ-4-2	КТ-5-3	КТ-2-2
3	КТ-1-1	КТ-3-2	КТ-2-1	КТ-2-3	КТ-2-2	КТ-3-4	КТ-2-5	КТ-2-4	КТ-3-1	КТ-3-3

в группе. То есть все тестовые примеры, у которых совпадает m, являются дублирующимися. Для наглядности все группы дубликатов, приведенные в таблицах, были выделены цветом.

Для всех экспериментов алгоритм самой дальней точки со 100 %-ной точностью определил уникальные ошибки в левую часть списка, а дубликаты — в правую. Полученные списки для всех выборок представлены в табл. 4. Разумеется, на более обширных выборках проявятся ложноположительные и ложноотрицательные срабатывания, однако проведение таких экспериментов требует огромной ручной работы. Анализ работы алгоритма на больших выборках тестовых примеров — одно из направлений будущих исследований. В табл. 1 алгоритм генерации программ-свидетелей достаточно точно определил группы дубликатов, коэффициенты сходства у разных групп сильно отличаются, чего нельзя сказать про результаты сравнения дубликатов в выборках с мискомпиляциями и деградациями производительности. Это связано с тем, что в первом случае компиляция проходит весь цикл, а не прерывается падением, и изменения, внесенные в покрытие программой-свидетелем, в масштабе всего исходного кода компилятора незначительны. Тем не менее дубликаты по-прежнему определяются верно и алгоритм самых дальних точек точно ранжирует файлы в списке.

Заключение

Представлен подход к поиску дубликатов ошибок компилятора методом генерации программ-свидетелей. Предложенный метод подходит как для использования специалистами поддержки компилятора, так и для интеграции в автоматическую систему поиска ошибок в нем. Подход основан на генерации программ-свидетелей, после которого на основе информации о покрытии формируется список файлов исходного кода компилятора, потенциально содержащих причину ошибки, далее они сравниваются и делается вывод о том, являются тестовые примеры дубликатами или нет.

На основе предложенного метода был разработан инструментарий для определения дубликатов ошибок компилятора языка Kotlin, который был протестирован на реальных тестовых примерах из

баг-трекера компилятора. Результаты показали применимость предлагаемого подхода для определения дубликатов. Помимо тестирования программной реализации алгоритма на больших выборках тестовых примеров, приводящих к ошибке, в будущем планируется проведение следующих работ:

- исследование других метрик для определения файлов исходного кода компилятора, которые являются причиной ошибки;
- создание мутаций, позволяющих расширить возможности для генерации программ-свидетелей;
- исследование возможностей применения гибридного подхода для улучшения точности;
- реализация инструментов по поиску дубликатов для компиляторов других языков программирования.

Список литературы

1. **Chen Y., Groce A., Zhang C.** et al. Taming compiler fuzzers // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, 2013. P. 197—208. DOI: 10.1145/2491956.2462173.
2. **Gonzalez T. F.** Clustering to minimize the maximum intercluster distance // Theor Comput Sci. 1985. Vol. 38. P. 293—306. DOI: 10.1016/0304-3975(85)90224-5.
3. **Holmes J., Groce A.** Causal Distance-Metric-Based Assistance for Debugging after Compiler Fuzzing // 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2018. P. 166—177. DOI: 10.1109/ISSRE.2018.00027.
4. **Myers E. W.** AnO(ND) difference algorithm and its variations // Algorithmica. 1986. Vol. 1, No 1—4. P. 251—266. DOI: 10.1007/BF01840446.
5. **Wong W. E., Gao R., Lo Y.** et al. A Survey on Software Fault Localization // IEEE Transactions on Software Engineering. 2016. Vol. 42, No. 8. P. 707—740. DOI: 10.1109/TSE.2016.2521368.
6. **Zakari A., Lee S., Abreu R.** et al. Multiple fault localization of software programs: A systematic literature review // Inf Softw Technol. 2020. Vol. 124. P. 106312. DOI: 10.1016/j.infsof.2020.106312.
7. **Soremekun E., Kirschner L., Bohme M.** et al. Locating faults with program slicing: an empirical analysis // Empir Softw Eng. 2021. Vol. 26, No. 3. P. 51. DOI: 10.1007/s10664-020-09931-7.
8. **Chang B.-Y. E., Chlipala A., Necula G.** et al. Type-based verification of sssembly language for compiler debugging // Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation. New York, NY, USA: ACM, 2005. P. 91—102. DOI: 10.1145/1040294.1040303.
9. **Hemmert K. S., Tripp J., Hutchings B.** et al. Source level debugger for the Sea Cucumber synthesizing compiler // 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM. IEEE Comput. Soc. 2003. P. 228—237. DOI: 10.1109/FPGA.2003.1227258.
10. **Krebs N., Schmitz L.** Jaccie: A Java-based compiler—compiler for generating, visualizing and debugging compiler components // Sci Comput Program. 2014. Vol. 79. P. 101—115. DOI: 10.1016/j.scico.2012.03.001.
11. **Chen J., Han J., Sun P.** et al. Compiler bug isolation via effective witness test program generation // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM, 2019. P. 223—234. DOI: 10.1145/3338906.3338957.
12. **Sohn J., Yoo S.** FLUCCS: using code and change metrics to improve fault localization // Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2017. P. 273—283. DOI: 10.1145/3092703.3092717.
13. **Abreu R., Zoeteweij P., van Gemund A. J. C.** On the Accuracy of Spectrum-based Fault Localization // Testing: Academic and Industrial Conference Practice and Research Techniques — MUTATION (TAICPART-MUTATION 2007). IEEE, 2007. P. 89—98. DOI: 10.1109/TAIC.PART.2007.13.
14. **JaCoCo** — Java Code Coverage Library. URL: https://www.jacoco.org/jacoco/trunk/index.html (дата обращения 09.02.2023).
15. **Clover** java and groovy code coverage tool homepage. URL: https://www.atlassian.com/software/clover/overview (дата обращения 09.02.2023).
16. Cobertura java code coverage utility homepage. URL: http://cobertura.github.io/cobertura/ (дата обращения 09.02.2023).
17. **Степанов Д. С., Ицыксон В. М.** Backend Bug Finder — a platform for effective compiler fuzzing // Информационно-управляющие системы. 2022. № 6. С. 31—40. DOI: 10.31799/1684-8853-2022-6-31-40.

Finding Compiler Bugs Duplicates by Generating Witness Programs

D. S. Stepanov, Senior Lecturer, stepanov0995@gmail.com,
V. M. Itsykson, PhD, Eng., Professor, itsykson@yandex.ru,
Peter the Great St. Petersburg Polytechnic University (SPbPU),
Saint Petersburg, 195251, Russian Federation

Corresponding author:

Daniil S. Stepanov, Senior Lecturer, Peter the Great St. Petersburg Polytechnic University (SPbPU)
Saint Petersburg, 195251, Russian Federation
E-mail: stepanov0995@gmail.com

Received on February 09, 2022
Accepted on February 27, 2022

Programming language compilers are complex software projects, the quality of which directly affects the quality of programs created by them. Therefore, compilers are subject to increased requirements for such software quality

characteristics as functional suitability, reliability, performance level, safety, etc. To ensure quality, various methods are used: user and manual testing, tools for automatic error detection. And due to many methods of searching for bugs often situations arise when the same error, especially if its cause is trivial, is detected repeatedly, and the test cases that lead to it can be completely different from each other. Such test cases are called duplicates, and their determination is an urgent and acute problem, since their manual search requires a large amount of human resources. Algorithms for automatically finding duplicates would greatly simplify the process of developing and maintaining a compiler.

The main idea of the approach for finding duplicates presented in the article is that the causes of the same errors are located in the same place in the compiler source code. To search for this place, the method of generating witness programs is used: for each test program containing an error, similar programs are generated that do not contain it. After that, metrics are calculated based on the source code coverage of the compiler and a list of source code files that are potentially causing the compiler to fail is formed. If these lists for two test programs are similar in terms of the proximity metric, then the test programs are considered duplicates.

The proposed approach was developed and implemented for the compiler of the Kotlin programming language. Testing has shown the applicability of the proposed approach for solving the problem of finding duplicate errors of compilers of programming languages

Keywords: *compiler testing, compiler bug duplicates, bugs isolation*

For citation:

Stepanov D. S., Itsykson V. M. Finding Compiler Bugs Duplicates by Generating Witness Programs, *Programmnaya Ingeneria*, 2023, vol. 14, no. 4, pp. 165—174. DOI: 10.17587/prin.14.165-174

References

1. **Chen Y., Groce A., Zhang C.** et al. Taming compiler fuzzers, *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA,ACM, 2013, pp. 197—208. DOI: 10.1145/2491956.2462173.

2. **Gonzalez T. F.** Clustering to minimize the maximum intercluster distance, *Theor Comput Sci.,* 1985, vol. 38, pp. 293—306. DOI: 10.1016/0304-3975(85)90224-5.

3. **Holmes J., Groce A.** Causal Distance-Metric-Based Assistance for Debugging after Compiler Fuzzing, *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE),* IEEE, 2018, pp. 166—177. DOI: 10.1109/ISSRE.2018.00027.

4. **Myers E. W.** AnO(ND) difference algorithm and its variations, *Algorithmica*, 1986. vol. 1, no. 1—4, pp. 251—266. DOI: 10.1007/BF01840446.

5. **Wong W. E., Gao R., Lo Y.** et al. A Survey on Software Fault Localization, *IEEE Transactions on Software Engineering*, 2016, vol. 42, no. 8, pp. 707—740. DOI: 10.1109/TSE.2016.2521368.

6. **Zakari A., Lee S., Abreu R.** et al. Multiple fault localization of software programs: A systematic literature review, *Inf Softw Technol*., 2020, vol. 124, pp. 106312. DOI: 10.1016/j.infsof.2020.106312.

7. **Soremekun E., Kirschner L., Bohme M.** et al. Locating faults with program slicing: an empirical analysis, *Empir Softw Eng.,* 2021, vol. 26, no. 3, pp. 51. DOI: 10.1007/s10664-020-09931-7.

8. **Chang B.-Y. E., Chlipala A., Necula G.** et al. Type-based verification of sssembly language for compiler debugging, *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. New York, NY, USA, ACM, 2005, pp. 91—102. DOI: 10.1145/1040294.1040303.

9. **Hemmert K. S., Tripp J., Hutchings B.** et al. Source level debugger for the Sea Cucumber synthesizing compiler, *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM 2003, IEEE Comput. Soc., 2003, pp. 228—237. DOI: 10.1109/FPGA.2003.1227258.

10. **Krebs N., Schmitz L.** Jaccie: A Java-based compiler—compiler for generating, visualizing and debugging compiler components, *Sci Comput Program*., 2014, vol. 79, pp. 101—115. DOI: 10.1016/j.scico.2012.03.001.

11. **Chen J., Han J., Sun P.** et al. Compiler bug isolation via effective witness test program generation, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* New York, NY, USA, ACM, 2019, pp. 223—234. DOI: 10.1145/3338906.3338957.

12. **Sohn J., Yoo S.** FLUCCS: using code and change metrics to improve fault localization, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA,ACM, 2017, pp. 273—283. DOI: 10.1145/3092703.3092717.

13. **Abreu R., Zoeteweij P., van Gemund A. J. C.** On the Accuracy of Spectrum-based Fault Localization, *Testing: Academic and Industrial Conference Practice and Research Techniques — MUTATION (TAICPART-MUTATION 2007),* IEEE, 2007, pp. 89—98. DOI: 10.1109/TAIC.PART.2007.13.

14. **JaCoCo** — Java Code Coverage Library, available at: https://www.jacoco.org/jacoco/trunk/index.html (date of access 09.02.2023).

15. **Clover** java and groovy code coverage tool homepage, available at:https://www.atlassian.com/software/clover/overview (date of access 09.02.2023).

16. **Cobertura** java code coverage utility homepage, available at:http://cobertura.github.io/cobertura/ (date of access 09.02.2023).

17. **Stepanov D., Itsykson V.** Backend Bug Finder — a platform for effective compiler fuzzing, *Information and Control Systems,* 2022, no. 6, pp. 31—40. DOI: 10.31799/1684-8853-2022-6-31-40.

УДК 681.518.5+004.052.32
DOI: 10.17587/prin.14.175-186

Д. В. Ефанов[1,2,3], д-р техн. наук, доц., проф., TrES-4b@yandex.ru,
Т. С. Погодина[2], студент, pogodina-ts@mail.ru
[1] Санкт-Петербургский политехнический университет Петра Великого
[2] Российский университет транспорта, Москва
[3] ООО НИПИ «ТРАНССТРОЙБЕЗОПАСНОСТЬ», Санкт-Петербург

Анализ эффективности схем встроенного контроля на основе оценки принадлежности вычисляемых функций классу самодвойственных и предварительного сжатия сигналов с применением линейных кодов

Поступила в редакцию 20.01.2023
Принята к публикации 15.02.2023

Описан метод контроля вычислений, направленный на определение принадлежности функций классу самодвойственных с применением известных линейных кодов для предварительного сжатия сигналов с выходов комбинационных устройств для уменьшения числа контролируемых сигналов. Предложена обобщенная структура организации контроля вычислений со сжатием сигналов с применением произвольных разделимых кодов, дополняющая известные структуры самодвойственного контроля вычислений по паритету («самодвойственного паритета») и самодвойственного контроля с дублированием каждой функции («самодвойственного дублирования»). В схеме встроенного контроля по представленному методу все устройства, кроме блока самодвойственного дополнения, являются стандартизированными (типовыми). Поэтому для синтеза схемы встроенного контроля необходимо получить только структуру этого блока в выбранном элементном базисе. Приведена методика синтеза блока самодвойственного дополнения при организации контроля вычислений с применением произвольных разделимых кодов. Описаны примеры реализации устройств со схемами встроенного контроля на основе известных линейных кодов — кодов паритета, Хэмминга и их модификаций. Даны некоторые результаты моделирования устройств со схемами встроенного контроля, отмечающие преимущества и недостатки применения каждого из рассмотренных кодов. Результаты, полученные в исследовании, могут быть применены при разработке самопроверяемых вычислительных устройств и систем.

Ключевые слова: *комбинационное устройство, контроль вычислений, контроль самодвойственности функций, код паритета, код Хэмминга, модифицированный код Хэмминга, самопроверяемое устройство, временна́я избыточность, самодвойственное дополнение*

Для цитирования:

Ефанов Д. В., **Погодина Т. С.** Анализ эффективности схем встроенного контроля на основе оценки принадлежности вычисляемых функций классу самодвойственных и предварительного сжатия сигналов с применением линейных кодов // Программная инженерия. 2023. Том 14, № 4. С. 175—186. DOI: 10.17587/prin.14.175-186.

Введение

Для контроля корректности реализации функций устройствами автоматики и вычислительной техники используются разнообразные методы [1—6]. В настоящей статье остановимся на рассмотрении вопросов организации схем встроенного контроля (СВК) [7—9].

При построении СВК используются методы теории информации и кодирования, а также булевой алгебры. Глубоко проработанными явля-

ются вопросы синтеза СВК, в основу которых положены кодовые методы [10—14]. Такие методы подразумевают отождествление сигналов на выходах объекта диагностирования с информационными символами с последующим контролем ошибок, возникающих в них вследствие действия неисправностей. Альтернативными являются методы синтеза СВК с применением временно́й избыточности, импульсного кодирования сигналов и контроля вычислений на выходах устройства по принадлежности формируемых функций особым классам булевых функций [15].

В контексте целей настоящей статьи особое внимание обратим на организацию контроля самодвойственности вычисляемых функций на выходах комбинационных устройств. Напомним, что *самодвойственная функция* принимает противоположные значения на ортогональных по всем переменным входных комбинациях: $f(x_1, x_2, ..., x_t) = f(\overline{x_1}, \overline{x_2}, ..., \overline{x_t})$, где t — число входных переменных [16]. Если некоторое устройство $F(x)$ имеет выходы $f_1, f_2, ..., f_m$, где m — число выходов устройства, описываемые самодвойственными функциями, то оно называется *самодвойственным устройством*.

В целом самодвойственные устройства не столь распространены, однако некоторые из них часто используются в автоматике и вычислительной технике, например, сумматоры по модулю $M = 2$ (XOR) и полные сумматоры одноразрядных двоичных чисел (*Full-Adders*) [17]. Любая булева функция может быть преобразована в самодвойственную с использованием всего одной дополнительной переменной. Такое преобразование связано со свойствами самодвойственных функций и возможно за счет использования формулы Шеннона: $f^\sigma = \bar{a}f \vee ag$, где f — исходная не самодвойственная функция; g — двойственная к ней функция; a — дополнительная переменная; f^σ — самодвойственная функция. Таким образом, любое несамодвойственное устройство может быть преобразовано в самодвойственное. Примеры самодвойственных преобразований и результаты экспериментов по тестированию таких устройств приведены в работе [18].

Контролировать вычисления по принадлежности формируемых функций классу самодвойственных булевых функций можно и на выходах несамодвойственных устройств. Для этого потребуется некоторое преобразование в СВК — каждая функция преобразуется к самодвойственной для контроля с помощью преобразования $f_i^\sigma = f_i \oplus \delta_i$, $i = \overline{1, m}$, где δ_i — это функция *самодвойственного дополнения* [19]. Такое преобразование возможно всегда, в том числе оно позволяет за счет выбора функции самодвойственного дополнения минимизировать структурную избыточность устройства с СВК.

Целью настоящей работы является представление результатов сравнительного анализа применения некоторых линейных двоичных кодов при синтезе СВК с контролем вычислений по принадлежности вычисляемых функций к классу самодвойственных булевых функций. Предложена обобщенная структура контроля вычислений по принадлежности функций к классу самодвойственных со сжатием сигналов с применением произвольных разделимых кодов. Такая структура является новым результатом по сравнению с известными структурами самодвойственного контроля вычислений с применением кодов паритета [20] и кодов с повторением [21]. В качестве основы схемы сжатия рассмотрено применение классических кодов паритета [22], Хэмминга [23] и одной их модификации, описанной в разд. 3 статьи [24].

1. Контроль комбинационных схем с применением самодвойственного дополнения до линейных кодов

Структура контроля вычислений с использованием признака самодвойственности функций со сжатием сигналов с применением кода паритета описана и исследована в работах [19, 20]. Данная структура названа ее авторами структурой *самодвойственного паритета* (рис. 1).

В структуре на рис. 1 блок $F(x)$ представляет собой исходное комбинационное устройство, снабженное t входами и m выходами. Устройство формирует значения булевых функций $f_1, f_2, ..., f_m$. Неисправности, обусловленные разнообразными внешними дестабилизирующими факторами, приводят к возникновению ошибок на линиях схемы устройства $F(x)$, транслируемых на выходы $f_1, f_2, ..., f_m$ в виде искажений сигналов, вызывая ошибки с различными кратностями и видами (по числу сочетаний искажений нулей и единиц), т. е. ошибки различных видов с различной кратностью. Для контроля вычислений и косвенно возникающих неисправностей используется специализированная СВК [7].

Контроль вычислений в СВК осуществляется путем проверки соответствия формируемой контрольной функции классу самодвойственных булевых функций. Для этого структура функционирует в импульсном режиме, а рабочие сигналы 0 и 1 представляются в виде последовательностей

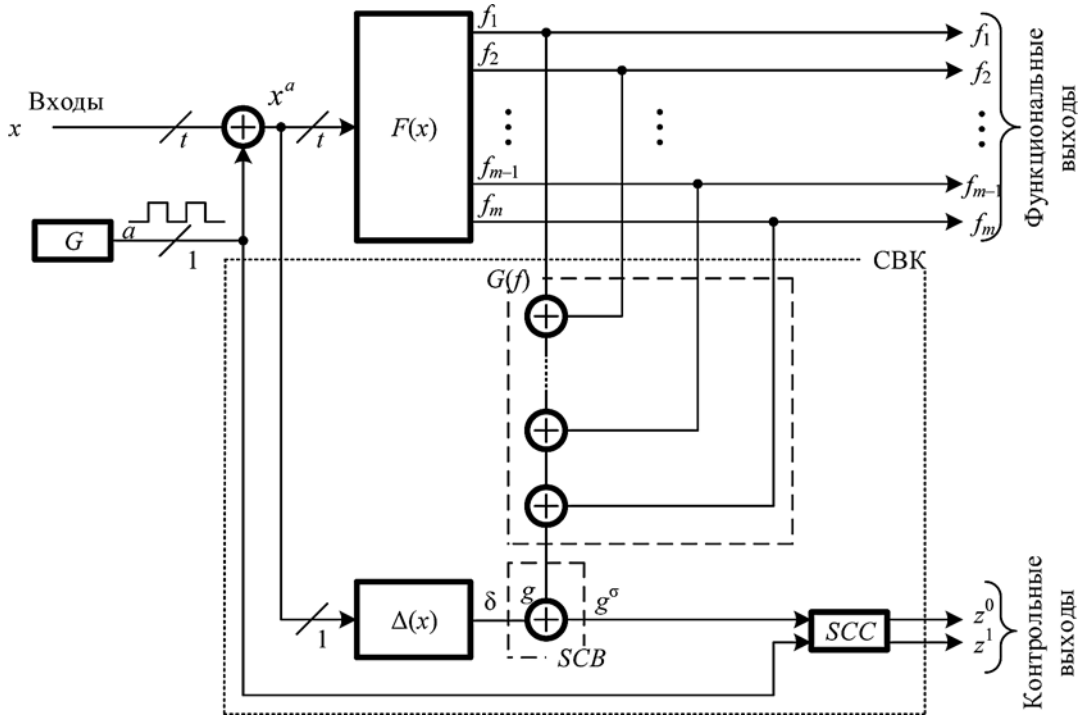


Рис. 1. Структура самодвойственного дополнения до кода паритета

импульсов: 0 — 0101...01, 1 — 1010...10. Входные сигналы x_1, x_2, ..., x_t преобразуются в последовательности сигналов с применением дополнительного импульсного сигнала (дополнительной переменной) a, формируемого генератором G: $x_i^a = x_i \oplus a$, $i = \overline{1, t}$.

В составе СВК использовано четыре блока: блок самодвойственного дополнения $\Delta(x)$; кодер кода паритета $G(f)$; блок коррекции сигналов *SCB* (*signal correction block*), включающий в себя в данном случае всего один элемент преобразования (XOR); тестер самодвойственности *SSC* (*self-checking self-dual checker*).

Кодер $G(f)$ реализует свертку по модулю $M = 2$ сигналов со всех рабочих функций блока $F(x)$ и вычисляет значение функции паритета $g = f_1 \oplus f_2 \oplus ... \oplus f_m$. Эта функция преобразуется в *SCB* в самодвойственную по правилу

$$g^\sigma = g \oplus \delta. \quad (1)$$

Функция δ определяется на этапе проектирования СВК и вычисляется блоком самодвойственного дополнения $\Delta(x)$. Контроль самодвойственности этой функции осуществляется тестером самодвойственности *SSC*. Последний имеет два входа (вход для подачи функционального сигнала и вход для подачи с генератора сигнала a) и два контрольных выхода z^0 и z^1. На контрольных выходах формируется парафазный сигнал при отсутствии ошибок в вычислениях любым блоком системы. Нарушение парафазности сигнализирует о наличии ошибки в вычислениях. Особенности реализации и функционирования тестера *SSC* описаны в работе [25].

Отметим, что структура, представленная на рис. 1, — это фактически структура с наиболее простой схемой сжатия, реализованной на основе единственной функции контроля четности — функции паритета.

На рис. 2 приведена обобщенная структура контроля вычислений со сжатием сигналов с применением произвольного разделимого кода и с использованием признака самодвойственности формируемых функций. В ней сжатие сигналов осуществляется с использованием кодера $G(f)$ выбранного разделимого кода. Затем сигналы с выходов кодера g_1, g_2, ..., g_k, где k — число выходов, преобразуются на элементах XOR в блоке *SCB* в сигналы самодвойственных функций g_1^σ, g_2^σ, ..., g_k^σ по правилу:

$$g_i^\sigma = g_i \oplus \delta_i, \; i = \overline{1, k}. \quad (2)$$

Она отличается от структуры, приведенной на рис. 1, тем, что блок $G(f)$ является многовыходным и реализует систему функций g_1, g_2, ..., g_k. Далее эти функции преобразуются в самодвойственные

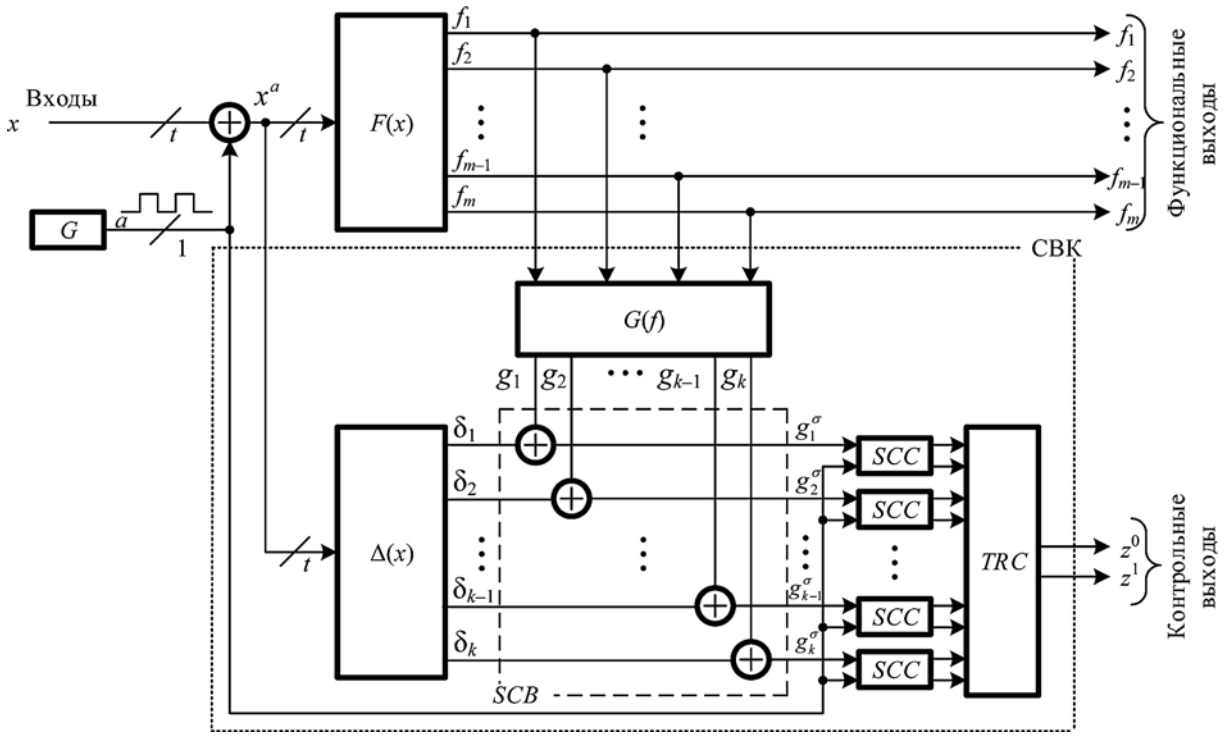


Рис. 2. Обобщенная структура контроля вычислений со сжатием сигналов с применением произвольного разделимого кода и с использованием признака самодвойственности формируемых функций

с применением k элементов XOR. Каждый самодвойственный сигнал g_1^σ, g_2^σ, ..., g_k^σ контролируется с помощью отдельного *SSC*. Выходы *SSC* объединяются на входах самопроверяемого компаратора *TRC*, реализуемого на основе каскадного подключения элементарных модулей сжатия парафазных сигналов [11—13]. Выходы *TRC* представляют собой контрольные выходы СВК.

Структура, приведенная на рис. 2, естественно, сложнее структуры, изображенной на рис. 1, так как кодер имеет бóльшее число выходов, что влияет и на число элементов преобразования и контрольных устройств. Кроме того, сложность кодера будет определяться и сложностью функций, реализуемых им.

В структуре, приведеннлй на рис. 2, в качестве схемы сжатия может быть использован любой разделимый код. Любую функцию g_i, $i = \overline{1, k}$, можно преобразовать в самодвойственную по правилу (2). В работе [26] рассмотрен способ реализации структуры, приведенной на рис. 2, со сжатием сигналов с применением классических и модульных кодов с суммированием. В работе [27] в качестве схемы сжатия рассмотрена возможность использования классических кодов Хэмминга. При этом было предложено применять их как при организации структуры контроля вычислений со сжатием всех сигналов от исходного комбинационного устройства, так и при организации контроля со сжатием сигналов от групп выходов. Можно применять сжатие сигналов по модифицированному коду с суммированием, пример использования которого при организации СВК есть в работе [28]. Данный код также является линейным (в настоящей статье этот код не рассматривается, так как имеет большую избыточность $k = m - 1$, практически сравнимую с использованием кодов с повторением при реализации СВК по методу дублирования).

Отметим важную особенность структур, приведенных на рис. 1 и 2. На функции $f_1, f_2, ..., f_m$, вычисляемые контролируемым устройством $F(x)$, не накладывается дополнительных ограничений — они могут быть и несамодвойственными. Несамодвойственными могут быть устройства $F(x)$, $G(f)$ и $\Delta(x)$. Это связано именно с наличием в структурах элементов преобразования сигналов и непосредственно блока *SCB*. Фактически функции g_1^σ, g_2^σ, ..., g_k^σ на выходах элементов преобразования в структуре на рис. 2 уже не являются функциями, описывающими проверочные символы выбираемого разделимого кода. Они являются функциями, полученными через функции $g_1, g_2, ..., g_k$, описывающие проверочные символы заданного кода. Аналогично и в структуре на рис. 1 функция g^σ не является функцией паритета для выходов контролируемого устройства.

Значения функций $\delta_1, \delta_2, ..., \delta_k$ получаются либо эвристическим способом, либо функциональным. Число самодвойственных функций от t аргументов равно $2^{2^{t-1}}$ (при $t = 2$ их 4, при $t = 3$ — 16 и т. д.), поэтому и значения функций g_1^σ, g_2^σ, ..., g_k^σ на каждой

входной комбинации могут быть получены различным числом способов. Эвристический способ подразумевает произвольное их получение по таблице истинности путем заполнения верхней или нижней половины таблицы и антисимметричным доопределением незаполненной части. При этом может быть упрощена каждая функция δ_1, δ_2, ..., δ_k, что требует анализа таблицы и при большом числе строк достаточно трудоемко. Функциональный подход связан с использованием функциональной зависимости между значениями функций f_1, f_2, ..., f_m и δ_1, δ_2, ..., δ_k. Некоторые методы расчета функций δ_1, δ_2, ..., δ_k приведены в разд. 3 статьи [19].

Из изложенного выше становится ясным, что для сжатия может использоваться произвольный двоичный разделимый код. При этом критериями выбора кода являются характеристики обнаружения ошибок в информационных символах, сложность технической реализации кодера, а также возможность обеспечения его полной самопроверяемости относительно выбранной модели неисправности. Далее рассмотрим применение только некоторых линейных кодов для построения структуры, изображенной на рис. 2. Тем не менее вопрос выбора кода может решаться обоснованно с учетом особенностей структуры устройства $F(x)$ и возможностей возникновения на его выходах ошибок с определенными видами и с различными кратностями [12, 13] при внесении в структуру неисправностей из наперед заданной модели [29].

2. Некоторые линейные коды

Рассмотрим три линейных кода: повсеместно используемые коды паритета и классические коды Хэмминга, а также одну из модификаций последних, описанную в работе [24].

Коды паритета имеют всего один проверочный символ, определяемый как свертка по модулю $M = 2$ значений всех информационных символов: $g = f_1 \oplus f_2 \oplus ... \oplus f_m$.

Коды Хэмминга строятся с использованием следующей проверочной матрицы:

$$M_H = \begin{pmatrix} 0 & 0 & ... & 1 & 1 \\ 0 & 0 & ... & 1 & 1 \\ ... & ... & ... & ... & ... \\ 0 & 1 & ... & 1 & 1 \\ 1 & 0 & ... & 0 & 1 \end{pmatrix}. \qquad (3)$$

В матрице перечисляются подряд слева направо все возможные двоичные числа от числа $[00...01]_2$ до числа $[11...11]_2$. Столбцы, соответствующие двоичным числам с десятичными аналогами $n = 2^i$, $i \in \mathbb{N}$, отводятся под проверочные символы (им соответствуют столбцы с одним символом 1). Остальные столбцы соответствуют информационным символам. Значение функции g_j, $j = \overline{1, k}$, описывающей j-й проверочный символ, вычисляется как сумма по модулю $M = 2$ сигналов тех информационных символов f_i, на пересечении столбцов которых с j-й строкой стоит 1. Число проверочных символов определяется как ближайшее целое, удовлетворяющее неравенству $m + 1 \leqslant 2^k - k$, где $m = n - k$ — число информационных символов; n — длина кодового слова; k — число проверочных символов.

Один из модифицированных кодов Хэмминга, применение которого эффективно при организации схем встроенного контроля, описан, как отмечено выше, в работе [24]. Он также строится по матрице (3). Однако в ней все столбцы соответствуют информационным символам. При построении модифицированного кода Хэмминга значение j-го проверочного символа $j = 1, \lceil \log_2 (m+1) \rceil$ вычисляется как сумма по модулю $M = 2$ тех информационных символов, которым соответствуют столбцы в проверочной матрице, в которых на пересечении j-й строки записана 1.

3. Структура контроля самодвойственных устройств

В работах [30, 31] показано, что применение линейных кодов при организации структур контроля вычислений имеет особенность, связанную с тем, что часть линейных кодов обладает свойством самодвойственности всех функций, по которым определяются значения проверочных символов. Такие коды характеризуются тем, что все контрольные функции содержат нечетное число аргументов, от которых они зависят существенно. Таким образом, в ряде специальных случаев структуры, приведенные на рис. 1 и 2, могут быть упрощены. Если устройство $F(x)$ является самодвойственным, т. е. все функции f_1, f_2, ..., f_m являются самодвойственными (а если часть из них или даже все не являются таковыми, то всегда возможно предварительное преобразование функции в самодвойственную с использованием только одной дополнительной переменной [32]), то возможно следующее упрощение.

При использовании кодов, все проверочные символы которых описываются самодвойственными функциями, и при самодвойственности устройства $F(x)$ в структуре на рис. 2 (структура на рис. 1 — частный случай структуры на рис. 2)

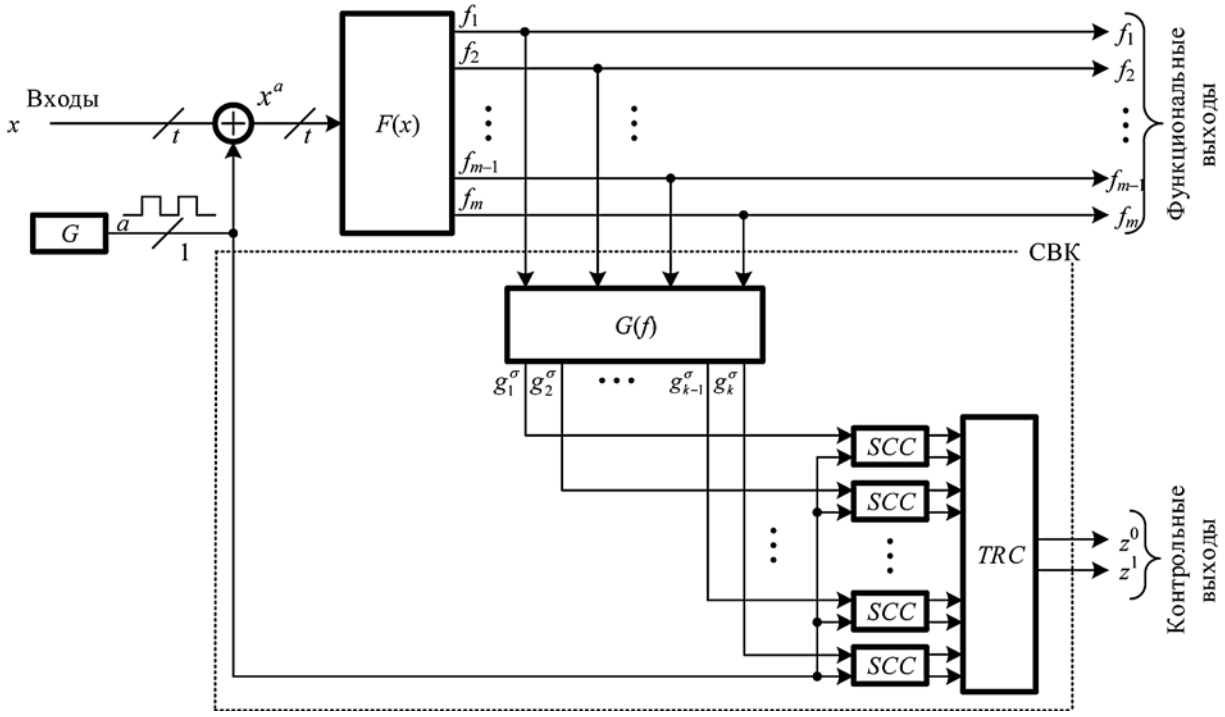


Рис. 3. Структура контроля вычислений со сжатием сигналов с применением произвольного разделимого кода, проверочные символы которого описываются самодвойственными булевыми функциями

не требуется блока самодвойственного дополнения $\Delta(x)$ и каскада элементов преобразования *SCB*, поскольку не требуется и само преобразование (2). Контроль вычислений осуществляется напрямую на выходах g_1, g_2, ..., g_k (рис. 3). Такая структура является фактически «вырожденной» структурой рис. 2, поскольку не применяется самодвойственных преобразований.

Известно [30, 31], что среди описанных выше линейных кодов при определенном числе информационных символов наблюдается свойство самодвойственности каждой функции, описывающей проверочные символы:

- коды паритета обладают свойством самодвойственности контрольной функции при нечетном значении m;
- коды Хэмминга со всеми контрольными самодвойственными функциями строятся при длинах кодовых слов $n = m + k = 3 + 4l$, $l \in \mathbb{N}_0$;
- модифицированные коды Хэмминга имеют все самодвойственные контрольные функции при $m = 2^k - 2$, $k \in \mathbb{N}$ $(k > 1)$.

Например, в табл. 1 отмечены рассматриваемые линейные коды, все проверочные символы для которых получаются по линейным функциям. Если исходное самодвойственное комбинационное устройство имеет число выходов, на пересечении которого стоит знак ×, то при использовании соответствующего кода строится структура, приведенная на рис. 3.

Таблица 1

Значения числа информационных символов у кодов с самодвойственными контрольными функциями

m	Коды паритета	Коды Хэмминга	Модифицированные коды Хэмминга
4		×	
5	×		
6			×
7	×	×	
8			
9	×		
10			
11	×	×	
12			
13	×		
14		×	×
15	×		
16			
17	×		
18		×	
19	×		
20			
21	×		
22		×	
23	×		
24			
25	×		
26		×	
27	×		
28			
29	×	×	
30			×

Отметив данную особенность структур с контролем вычислений с использованием признака самодвойственности функций, остановимся на общем случае контроля вычислений на выходах комбинационных устройств с выходами, описываемыми произвольными булевыми функциями.

4. Синтез схем контроля с использованием линейных кодов

Продемонстрируем особенности обнаружения ошибок в структуре, приведенной на рис. 2, синтезированной с применением кодов паритета, классических и модифицированных кодов Хэмминга, на элементарном примере.

4.1. Методика синтеза структуры блока самодвойственного дополнения и исходные данные

В структуре схемы контроля, изображенной на рис. 2, все элементы, за исключением блока $\Delta(x)$ являются стандартизированными. Структура блока $G(f)$ и число его выходов определяются заданным кодом, элементной базой и способом реализации кодера. Она фактически также является типовой. Другими словами, задачей синтеза является именно получение структуры блока $\Delta(x)$. *Методику его синтеза* составляют следующие шаги.

Шаг 1. Задаются устройство $F(x)$ и двоичный разделимый код.

Шаг 2. Описывается работа исходного устройства $F(x)$ на полном множестве рабочих входных комбинаций (это может быть как полное множество входных воздействий, так и неполное, что определяется заданием на синтез).

Шаг 3. На каждой входной комбинации по значениям функций $f_1, f_2, ..., f_m$ определяются значения функций $g_1, g_2, ..., g_k$, описывающих проверочные символы заданного кода.

Шаг 4. Определяются значения функций $g_1^\sigma, g_2^\sigma, ..., g_k^\sigma$ таким образом, чтобы на ортогональных по всем входным переменным наборах они имели противоположные значения (это можно сделать разными способами; в приведенных далее примерах использован табличный способ с эвристическим заполнением строк).

Шаг 5. Из выражения (2) определяются функции дополнения $\delta_i = g_i \oplus g_i^\sigma, \ i = \overline{1, k}$.

Шаг 6. Функции дополнения минимизируются.

Шаг 7. По минимизированным функциям синтезируется блок $\Delta(x)$.

Для демонстрации методики и некоторых преимуществ применения вместо кодов паритета других избыточных кодов рассмотрим произвольное комбинационное устройство с достаточно простой структурой, малым числом входов и выходов, к примеру, описываемое табл. 2.

Таблица 2

Таблица истинности функций, реализуемых устройством $F(x)$

№	x_1	x_2	x_3	f_1	f_2	f_3	f_4
0	0	0	0	1	0	0	1
1	0	0	1	0	0	1	0
2	0	1	0	0	1	1	0
3	0	1	1	0	0	1	0
4	1	0	0	1	1	0	0
5	1	0	1	1	0	1	0
6	1	1	0	0	1	0	1
7	1	1	1	0	1	0	1

Минимизация функций f_1, f_2, f_3, f_4 дает следующий результат:

$$\begin{cases} f_1 = \overline{x_2}\,\overline{x_3} \vee x_1\overline{x_2} = A \vee B; \\ f_2 = x_2\overline{x_3} \vee x_1\overline{x_3} \vee x_1x_2 = C \vee D \vee E; \\ f_3 = \overline{x_2}x_3 \vee \overline{x_1}x_2 = F \vee G; \\ f_4 = \overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1x_2 = \overline{x_1}A \vee E. \end{cases}$$

В приведенной выше системе функций использовано объединение переменных по конъюнкциям: $A = \overline{x_2}\,\overline{x_3}$, $B = x_1\overline{x_2}$, $C = x_2\overline{x_3}$, $D = x_1\overline{x_3}$, $E = x_1x_2$, $F = \overline{x_2}x_3$, $G = \overline{x_1}x_2$. Это необходимо для реализации схемы с ветвлениями для демонстрации особенностей обнаружения ошибок различными кодами.

Функции, описывающие проверочные символы рассматриваемых линейных кодов (контрольные функции), получаются следующим образом.

Код паритета:

$$g_1 = f_1 \oplus f_2 \oplus f_3 \oplus f_4.$$

Код Хэмминга:

$$\begin{cases} g_1 = f_1 \oplus f_2 \oplus f_4; \\ g_2 = f_1 \oplus f_3 \oplus f_4; \\ g_3 = f_2 \oplus f_3 \oplus f_4. \end{cases}$$

Модифицированный код Хэмминга:

$$\begin{cases} g_1 = f_1 \oplus f_3; \\ g_2 = f_2 \oplus f_3; \\ g_3 = f_4. \end{cases}$$

Используя приведенные исходные данные, синтезируем и смоделируем работу структур на рис. 2 для трех вариантов реализации.

4.2. Структура с СВК по коду паритета

Пользуясь представленной выше методикой, получим описание работы структуры на рис. 2 (в данном частном случае — структура на рис. 1) при использовании в качестве основы кода паритета (табл. 3).

Минимизация функции дополнения δ дает следующий результат:

$$\delta = x_1 x_3.$$

Синтезированное устройство со схемой встроенного контроля по коду паритета приведено на рис. 4 (см. вторую сторону обложки). Оно реализовано в программных средствах Multisim [33] для последующего моделирования неисправностей.

В структуре устройства $F(x)$ имеются два логических элемента, связанных путями с двумя выходами — это элементы U4 (связан путями с выходами f_1 и f_4) и U8 (связан путями с выходами f_2 и f_4). Рассмотрим одиночные константные неисправности выходов данных элементов (неисправности stuck-at-0 и stuck-at-1) и определим, как они обнаруживаются в СВК при контроле вычислений с применением кода паритета. Двукратные ошибки кодами паритета не обнаруживаются, но здесь используется контроль по самодвойственности вычисляемой функции, а входные комбинации подаются парами: (000, 111), (001, 110), (010, 101), (011, 100) [25]. Это позволяет обнаруживать вызываемые неисправностями ошибки на парах входных комбинаций.

На рис. 5 (см. вторую сторону обложки) представлены временные диаграммы работы устройства, приведенного на рис. 4 (см. вторую сторону обложки), при внесении последовательно неисправностей на выходах элементов U4 и U8. Черными точками отмечены такты, в которых неисправность проявляется в виде искажений на линиях схемы самого устройства. Интерес здесь представляет анализ контрольных выходов z^0 и z^1 СВК.

Рассмотрим, к примеру, неисправность stuck-at-0 U4. Наличие данной неисправности приводит к искажениям значений двух функций, однако парафазность значений на выходах z^0 и z^1 не нарушается ни на одной паре входных комбинаций. Эта ошибка маскируется на входах схемы сжатия сигналов. Аналогично, достаточно много ошибок маскируется и при внесении других рассматриваемых неисправностей, а число тестовых комбинаций и пар комбинаций среди рабочих воздействий достаточно мало: 37,5 % входных комбинаций одновременно являются и тестовыми для всех рассмотренных неисправностей при контроле вычислений на основе кода паритета, а доля тестовых пар среди рабочих равна 18,75 %.

Таким образом, становится понятной основная уязвимость структуры «самодвойственного паритета» — возможность маскировки ошибок с четной кратностью на входах схемы сжатия сигналов. Данный недостаток исправляется использованием кодов с большей избыточностью.

4.3. Структура с СВК по коду Хэмминга

По аналогии с предыдущим примером получим описание работы структуры по рассматриваемому коду Хэмминга (табл. 4).

Минимизация функций дополнения δ_1, δ_2, δ_3 дает следующий результат:

$$\begin{cases} \delta_1 = \overline{x_1}; \\ \delta_2 = x_1 \overline{x_3}; \\ \delta_3 = x_1 \overline{x_2 x_3}. \end{cases}$$

Таблица 3

Сигналы на линиях схемы в структуре с контролем вычислений по коду паритета

№	x_1	x_2	x_3	f_1	f_2	f_3	f_4	g	g^σ	δ
0	0	0	0	1	0	0	1	0	0	0
1	0	0	1	0	0	1	0	1	1	0
2	0	1	0	0	1	1	0	0	0	0
3	0	1	1	0	0	1	0	1	1	0
4	1	0	0	1	1	0	0	0	0	0
5	1	0	1	1	0	1	0	0	1	1
6	1	1	0	0	1	0	1	0	0	0
7	1	1	1	0	1	0	1	0	1	1

Таблица 4

Сигналы на линиях схемы в структуре с контролем вычислений по коду Хэмминга

№	x_1	x_2	x_3	f_1	f_2	f_3	f_4	g_1	g_2	g_3	g_1^σ	g_2^σ	g_3^σ	δ_1	δ_2	δ_3
0	0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	0
1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0
2	0	1	0	0	1	1	0	1	1	0	0	1	0	1	0	0
3	0	1	1	0	0	1	0	0	1	1	1	1	1	1	0	0
4	1	0	0	1	1	0	0	0	1	1	0	0	0	0	1	1
5	1	0	1	1	0	1	0	1	0	1	1	0	1	0	0	0
6	1	1	0	0	1	0	1	0	1	0	0	0	0	0	1	0
7	1	1	1	0	1	0	1	0	1	0	0	1	0	0	0	0

Таблица 5

Сигналы на линиях схемы в структуре с контролем вычислений по модифицированному коду Хэмминга

№	x_1	x_2	x_3	f_1	f_2	f_3	f_4	g_1	g_2	g_3	g_1^σ	g_2^σ	g_3^σ	δ_1	δ_2	δ_3
0	0	0	0	1	0	0	1	1	0	1	1	0	1	0	0	0
1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0
2	0	1	0	0	1	1	0	1	0	0	1	0	0	0	0	0
3	0	1	1	0	0	1	0	1	1	0	1	1	0	0	0	0
4	1	0	0	1	1	0	0	1	1	0	0	0	1	1	1	1
5	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1
6	1	1	0	0	1	0	1	0	1	1	0	0	1	0	1	0
7	1	1	1	0	1	0	1	0	1	1	0	1	0	0	0	1

Синтезированное в Multisim устройство со схемой встроенного контроля по коду Хэмминга приведено на рис. 6 (см. третью сторону обложки). На рис. 7 представлены временные диаграммы работы устройства, приведенного на рис. 6 (см. третью сторону обложки), при внесении последовательно неисправностей на выходах элементов U4 и U8.

Сравнивая диаграммы, приведенные на рис. 5 и рис. 7, отмечаем, что использование в качестве схемы сжатия кодера кода Хэмминга позволяет обнаруживать большее число ошибок, чем при использовании кодера кода паритета. Необнаруживаемые ошибки вообще отсутствуют, а число тестовых комбинаций значительно выше, чем при использовании кодов паритета. При контроле вычислений на основе кодов Хэмминга 68,75 % входных комбинаций одновременно являются и тестовыми для всех рассмотренных неисправностей, а доля тестовых пар среди рабочих равна 34,375 %.

4.4. Структура с СВК по модифицированному коду Хэмминга

Описание работы структуры, приведенной на на рис. 2, в качестве основы для которой выбран модифицированный код Хэмминга, представлено в табл. 5.

Минимизация функций дополнения δ_1, δ_2, δ_3 приводит к такому результату:

$$\begin{cases} \delta_1 = x_1\overline{x_2}\,\overline{x_3}; \\ \delta_2 = x_1\overline{x_3}; \\ \delta_3 = x_1\overline{x_2} \vee x_1x_3. \end{cases}$$

Синтезированное в Multisim устройство со схемой встроенного контроля по модифицированному коду Хэмминга приведено на рис. 8 (см. четвертую сторону обложки).

При контроле вычислений на основе модифицированных кодов Хэмминга показатели контролепригодности аналогичны показателям контролепригодности при контроле вычислений на основе классических кодов Хэмминга (рис. 9, см. четвертую сторону обложки): 68,75 % входных комбинаций одновременно являются и тестовыми для всех рассмотренных неисправностей, а доля тестовых пар среди рабочих равна 34,375 %.

Таким образом, при использовании кодов Хэмминга и их модификаций характеристики обнаружения ошибок в СВК с самодвойственным преобразованием улучшаются по сравнению с применением кодов паритета. При этом, естественно, увеличиваются значения показателей сложности технической реализации самих СВК, так как и число проверочных символов в данных кодах больше, чем у кодов паритета. Эффективность использования кодов Хэмминга (и других линейных кодов) требуется оценивать по сравнению с использованием метода дублирования. В рассматриваемом примере использование кодов Хэмминга и их модификаций по сложности соизмеримо с применением дублирования. Тем не менее показатели контролепригодности при использовании самодвойственного контроля по рассматриваемым кодам выше, чем при дублировании, что определяет и приоритет их использования.

Заключение

При применении в процессе синтеза самопроверяемого устройства обобщенной структуры контроля вычислений с использованием признака самодвойственности формируемых функций для сжатия сигналов может использоваться любой разделимый код, поскольку с помощью логического преобразования на элементах XOR для каждой контрольной функции $g_1, g_2, ..., g_k$ может быть получена самодвойственная функция $g_1^\sigma, g_2^\sigma, ..., g_k^\sigma$. При этом существует большое разнообразие способов получения контролируемых самодвойственных функций, что позволяет, в свою очередь, упрощать структуру блока самодвойственного дополнения $\Delta(x)$.

Перспективным является использование в качестве схемы сжатия кодеров линейных кодов. В статье приведен пример, показывающий преимущества применения кодов Хэмминга вместо кодов паритета при организации СВК: при незначительном усложнении структуры СВК достигается обнаружение большего числа ошибок на выходах объекта диагностирования, а также улучшаются показатели контролепригодности (конкретно — показатели наблюдаемости).

Однако отметим в качестве недостатка самого способа организации контроля вычислений с использованием признака самодвойственности функций то, что структуры функционируют в импульсном режиме. Это требует использования временно́й избыточности и, как следствие, снижения быстродействия по отношению к широко применяемым классическим методам синтеза СВК [7—9, 11—14].

Дальнейшие направления в исследованиях могут быть связаны с изучением особенностей применения в структуре, представленной на рис. 2, других преобразователей, в том числе позволяющих минимизировать число потенциально не обнаруживаемых ошибок, а также способов выделения групп выходов, сигналы от которых будут сжиматься для уменьшения числа наблюдаемых выходов. Интересными могут быть исследования контроля вычислений по двум диагностическим признакам, что потребует модификации структуры, представленной на рис. 2, по аналогии с предложенным в работе [34] способом.

Организация контроля вычислений с использованием признака самодвойственности булевых функций представляется авторам настоящей статьи перспективным подходом к синтезу надежных вычислительных устройств и систем.

Список литературы

1. **Fujiwara E.** Code Design for Dependable Systems: Theory and Practical Applications. John Wiley & Sons, 2006. 720 p.

2. **Дрозд А. В., Харченко В. С., Антощук С. Г., Дрозд Ю. В., Дрозд М. А., Сулима Ю. Ю.** Рабочее диагностирование безопасных информационно-управляющих систем / Под ред. А. В. Дрозда и В. С. Харченко. Харьков: Национальный аэрокосмический университет им. Н. Е. Жуковского «ХАИ», 2012. 614 с.

3. **Tshagharyan G., Harutyunyan G., Shoukourian S., Zorian Y.** Experimental Study on Hamming and Hsiao Codes in the Context of Embedded Applications // Proceedings of 15th IEEE East-West Design & Test Symposium (EWDTS'2017). Novi Sad, Serbia, September 29 — October 2, 2017. P. 25—28. DOI: 10.1109/EWDTS.2017.8110065.

4. **Стемпковский А. Л., Тельпухов Д. В., Жукова Т. Д.** и др. Синтез схемы функционального контроля на основе спектрального R-кода с разбиением выходов на группы // Микроэлектроника. 2019. Том 48, № 4. С. 284—294. DOI: 10.1134/S0544126919040094.

5. **Ярмолик В. Н.** Контроль и диагностика вычислительных систем. Минск: Бестпринт, 2019. 387 с.

6. **Бестемьянов П. Ф.** Обнаружение неисправностей в измерительных цепях систем железнодорожной автоматики и телемеханики для обеспечения безопасности движения поездов // Электротехника. 2022. № 9. С. 55—60. DOI: 10.53891/00135860_2022_9_55.

7. **Согомонян Е. С., Слабаков Е. В.** Самопроверяемые устройства и отказоустойчивые системы. М.: Радио и связь, 1989. 208 с.

8. **Nicolaidis M., Zorian Y.** On-Line Testing for VLSI — A Compendium of Approaches // Journal of Electronic Testing: Theory and Applications. 1998. No. 12. P. 7—20. — DOI: 10.1023/A:1008244815697.

9. **Mitra S., McCluskey E. J.** Which Concurrent Error Detection Scheme to Choose? // Proceedings of International Test Conference, 2000, USA, Atlantic City, NJ, 03-05 October 2000. P. 985—994. DOI: 10.1109/TEST.2000.894311.

10. **Piestrak S. J.** Design of Self-Testing Checkers for Unidirectional Error Detecting Codes. Wrocław: Oficyna Wydawnicza Politechniki Wrocławskiej, 1995. 111 p.

11. **Göessel M., Ocheretny V., Sogomonyan E., Marienfeld D.** New Methods of Concurrent Checking: Edition 1. Dordrecht: Springer Science + Business Media B. V., 2008. 184 p.

12. **Сапожников В. В., Сапожников Вл. В., Ефанов Д. В.** Коды Хэмминга в системах функционального контроля логических устройств. СПб.: Наука, 2018, 151 с.

13. **Сапожников В. В., Сапожников Вл. В., Ефанов Д. В.** Коды с суммированием для систем технического диагностирования. Том 1: Классические коды Бергера и их модификации. М.: Наука, 2020. 383 с.

14. **Сапожников В. В., Сапожников Вл. В., Ефанов Д. В.** Коды с суммированием для систем технического диагностирования. Том 2: Взвешенные коды с суммированием. М.: Наука, 2021. 455 с.

15. **Reynolds D. A., Meize G.** Fault Detection Capabilities of Alternating Logic // IEEE Transactions on Computers. 1978. Vol. C-27. Issue 12. P. 1093-1098. DOI: 10.1109/TC.1978.1675011.

16. **Яблонский С. В.** Введение в дискретную математику: учеб. пособие для вузов / под ред. В. А. Садовничева, 4-е изд., М.: Высшая школа, 2003. 384 с.

17. **Harris D. M., Harris S. L.** Digital Design and Computer Architecture. Morgan Kaufmann, 2012. 569 p.

18. **Гессель М., Мошанин В. И., Сапожников В.В, Сапожников Вл. В.** Обнаружение неисправностей в самопроверяемых комбинационных схемах с использованием свойств самодвойственных функций // Автоматика и телемеханика. 1997. № 12. С. 193—200.

19. **Гессель М., Дмитриев А. В., Сапожников В. В., Сапожников Вл. В.** Самотестируемая структура для функционального обнаружения отказов в комбинационных схемах // Автоматика и телемеханика. 1999. № 11. С. 162—174.

20. **Saposhnikov Vl. V., Dmitriev A., Goessel M., Saposhnikov V. V.** Self-Dual Parity Checking — a New Method for on Line Testing // Proceedings of 14th IEEE VLSI Test Symposium, USA, Princeton, 1996. P. 162—168.

21. **Saposhnikov Vl. V., Saposhnikov V. V., Dmitriev A., Goessel M.** Self-Dual Duplication for Error Detection // Proceedings of 7th Asian Test Symposium, Singapore, 1998. P. 296—300. DOI: 10.1109/ATS.1998.741628.

22. **Borecký J., Kohlík M., Kuátová H.** Parity Driven Reconfigurable Duplex System // Microprocessors and Microsystems. 2017. Vol. 52. P. 251—260. DOI: 10.1016/j.micpro.2017.06.015.

23. **Hamming R. W.** Error Detecting and Correcting Codes // Bell System Technical Journal. 1950. Vol. 29, No. 2. P. 147—160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

24. **Ефанов Д. В., Сапожников В. В., Сапожников Вл. В.** Синтез самопроверяемых комбинационных устройств на основе выделения специальных групп выходов // Автоматика и телемеханика. 2018. № 9. С. 79—94.

25. **Ефанов Д. В., Погодина Т. С.** Самодвойственный контроль комбинационных схем с применением кодов Хэмминга // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2022. № 3. С. 113—122. DOI: 10.31114/2078-7707-2022-3-113-122.

26. **Ефанов Д. В., Сапожников В. В., Сапожников Вл. В.** Синтез схем встроенного контроля для комбинационных цифровых устройств по методу самодвойственного дополнения до кода Бергера // Известия высших учебных заведений. Приборостроение. 2021. Том 64, № 9. С. 697—708. DOI: 10.17586/0021-3454-2021-64-9-697-708.

27. **Сапожников В. В., Сапожников Вл. В., Гессель М.** Самодвойственные дискретные устройства. СПб: Энергоатомиздат (Санкт-Петербургское отделение), 2001. 331 с.

28. **Ефанов Д. В., Сапожников В. В., Сапожников Вл. В.** Организация схем встроенного контроля на основе метода логического дополнения с предварительным преобразованием рабочих функций в контрольные векторы кодов Бергера // Информационные технологии. 2021. Том 27, № 6. С. 306—313. DOI: 10.17587/it.27.306-313.

29. **Багхдади А. А. А., Хаханов В. И., Литвинова Е. И.** Методы анализа и диагностирования цифровых устройств (аналитический обзор) // Автоматизированные системы управления и приборы автоматики. 2014. № 166. С. 59—74.

30. **Efanov D. V., Pogodina T. S.** Self-Dual Digital Devices with Calculations Testing by Modified Hamming Code // Proceedings of the 2023 Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), St. Petersburg, Russia, 24—27 January 2023. P. 72—77.

31. **Ефанов Д. В., Погодина Т. С.** Исследование свойств самодвойственных комбинационных устройств с контролем вычислений на основе кодов Хэмминга // Информатика и автоматизация. 2023. Том 22, № 2. С. 349—392. DOI: 10.15622/ia.22.2.5.

32. **Аксенова Г. П.** Восстановление в дублированных устройствах методом инвертирования данных // Автоматика и телемеханика. 1987. № 10. С. 144—153.

33. **Chen Y., Zhang M., Hao J.** The Circuit Design of Voltage-Controlled Color Changing Lamp Based on Multisim // 2020 IEEE International Conference on Power, Intelligent Computing and Systems (ICPICS), 28—30 July 2020, Shenyang, China. DOI: 10.1109/ICPICS50287.2020.9202148.

34. **Efanov D., Sapozhnikov V., Sapozhnikov Vl., Osadchy G., Pivovarov D.** Self-Dual Complement Method up to Constant-Weight Codes for Arrangement of Combinational Logical Circuits Concurrent Error-Detection Systems // Proceedings of 17th IEEE East-West Design & Test Symposium (EWDTS'2019). Batumi, Georgia, September 13—16, 2019. P. 136-143. DOI: 10.1109/EWDTS.2019.8884398.

Efficiency Analysis of Concurrent Error-Detection Circuits on the Basis of Assessment by Belonging of Calculated Functions to Self-Dual Class and Preliminary Compression of Signals Using Linear Codes

D. V. Efanov[1, 2, 3], D. Sc. (Engineering), Associate Professor, Professor, TrES-4b@yandex.ru,
T. S. Pogodina[2], Student, pogodina-ts@mail.ru
[1] Peter the Great St. Petersburg Polytechnic University, St. Petersburg, 195251, Russian Federation
[2] Russian University of Transport, Moscow, 127994, Russian Federation
[3] LLC Research and Design Institute for Transport and Construction Safety, St. Petersburg, 192102, Russian Federation

Corresponding author:

Dmitry V. Efanov, D. Sc. (Engineering), Associate Professor, Professor,
Peter the Great St. Petersburg Polytechnic University, St. Petersburg, 195251, Russian Federation,
Russian University of Transport, Moscow, 127994, Russian Federation, LLC Research and Design Institute for Transport and Construction Safety, St. Petersburg, 192102, Russian Federation
E-mail: TrES-4b@yandex.ru

Received on January 20, 2023
Accepted on February 15, 2023

A method is described for checking of calculations on whether functions belong to the class of self-dual with using well-known linear codes for preliminary compression of signals from the outputs of combinational devices in order to reduce the number of controlled signals. A generalized structure for organizing the checking of calculations with signal compression using arbitrary separable codes is proposed, which complements the well-known structures of self-dual check of parity calculations («self-dual parity») and self-dual check with duplication of each function («self-dual duplication»). In the concurrent error-detection circuit according to the presented method, all devices, except for the self-dual complement block, are standardized (typical). Therefore, to synthesize the concurrent error-detection circuit, it is necessary to obtain only the structure of this block in the selected elemental basis. The article presents a technique for synthesizing a block of self-dual complement when organizing the checking of calculations using arbitrary separable codes. Examples of the implementation of devices with concurrent error-detection circuits based on well-known linear codes — parity codes, Hamming codes and their modifications are given. Some results of modeling devices with concurrent error-detection circuits are given, noting the advantages and disadvantages of using each of the codes considered. The results obtained in the study can be applied in the development of self-checking computing devices and systems.

Keywords: *combinational device; checking of calculations; checking self-duality of functions; parity code; Hamming code; modified Hamming code; self-checking device; temporal redundancy; self-dual complement*

For citation:

Efanov D. V., Pogodina T. S. Efficiency Analysis of Concurrent Error-Detection Circuits on the Basis of Assessment by Belonging of Calculated Functions to Self-Dual Class and Preliminary Compression of Signals Using Linear Codes, *Programmnaya Ingeneria,* 2023, vol. 14, no. 4, pp. 175—186. DOI: 10.17587/prin.14.175-186.

References

1. **Fujiwara E.** *Code Design for Dependable Systems: Theory and Practical Applications*, John Wiley & Sons, 2006, 720 p.

2. **Drozd A. V., Kharchenko V. S., Antoshchuk S. G., Drozd Yu. V., Drozd M. A., Sulima Yu. Yu.** *Working diagnostics of safe information and control systems* / Eds. A. V. Drozd and V. S. Kharchenko, Kharkiv, National Aerospace University. N. E. Zhukovsky «KhAI», 2012, 614 p (in Russian).

3. **Tshagharyan G., Harutyunyan G., Shoukourian S., Zorian Y.** Experimental Study on Hamming and Hsiao Codes in the Context of Embedded Applications, *Proceedings of 15th IEEE East-West Design & Test Symposium (EWDTS'2017)*, Novi Sad, Serbia, September 29 — October 2, 2017, pp. 25—28. DOI: 10.1109/EWDTS.2017.8110065.

4. **Stempkovskii A. L., Tel'pukhov D. V., Zhukova T. D., Demeneva A. I., Nadolenko V. V., Gurov S. I.** Synthesis of a Concurrent Error Detection Circuit Based on the Spectral R-Code with the Partitioning of Outputs into Groups, *Mikrojelektronika*, 2019, vol. 48, no. 4, pp. 240-249, DOI: 10.1134/S0544126919040094 (in Russian).

5. **Yarmolik V. N.** *Control and Diagnostics of Computer Systems,* Minsk, Bestprint, 2019, 387 p. (in Russian).

6. **Bestemyanov P. F.** Detection of Malfunctions in Measuring Circuits of Railway Automation and Telemechanics Systems to Ensure the Safety of Train Traffic, *Jelektrotehnika*, 2022, no. 9, pp. 55-60, DOI: 10.53891/00135860_2022_9_55 (in Russian).

7. **Sogomonyan E. S., Slabakov E. V.** *Self-Checking Devices and Fail-Safe Systems*, Moscow, Radio and communication, 1989, 208 p. (in Russian).

8. **Nicolaidis M., Zorian Y.** On-Line Testing for VLSI — A Compendium of Approaches, *Journal of Electronic Testing: Theory and Applications*, 1998, no. 12, pp. 7—20. DOI: 10.1023/A:1008244815697.

9. **Mitra S., McCluskey E. J.** Which Concurrent Error Detection Scheme to Choose?, *Proceedings of International Test Conference*, 2000, USA, Atlantic City, NJ, 03-05 October 2000, pp. 985—994. DOI: 10.1109/TEST.2000.894311.

10. **Piestrak S. J.** *Design of Self-Testing Checkers for Unidirectional Error Detecting Codes,* Wrocław, Oficyna Wydawnicza Politechniki Wrocławskiej, 1995, 111 p.

11. **Göessel M., Ocheretny V., Sogomonyan E., Marienfeld D.** *New Methods of Concurrent Checking: Edition 1*, Dordrecht, Springer Science + Business Media B. V., 2008, 184 p.

12. **Sapozhnikov V. V., Sapozhnikov Vl. V., Efanov D. V.** *Hamming Codes in Concurrent Error Detection Systems of Logic Devices*. St. Petersburg, Nauka, 2018, 151 p. (in Russian).

13. **Sapozhnikov V. V., Sapozhnikov Vl. V., Efanov D. V.** *Sum Codes for Technical Diagnostics Systems*. Vol. 1: Classical Berger Codes and Their Modifications, Moscow, Nauka, 2020, 383 p. (in Russian).

14. **Sapozhnikov V. V., Sapozhnikov Vl. V., Efanov D. V.** *Sum Codes for Technical Diagnostics Systems*. Vol. 2: Weighted Codes with Summation, Moscow, Nauka, 2021, 455 p. (in Russian).

15. **Reynolds D. A., Meize G.** Fault Detection Capabilities of Alternating Logic, *IEEE Transactions on Computers*, 1978, vol. C-27, issue 12, pp. 1093—1098, DOI: 10.1109/TC.1978.1675011.

16. **Yablonsky S. V.** *Introduction to Discrete Mathematics: Proc. manual for universities* / Eds. V. A. Sadovnichev, 4th ed., Moscow, Higher School, 2003, 384 p. (in Russian).

17. **Harris D. M., Harris S. L.** *Digital Design and Computer Architecture*, Morgan Kaufmann, 2012, 569 p.

18. **Goessel M., Moshanin V. I., Sapozhnikov V. V., Sapozhnikov Vl. V.** Fault Detection in Self-Test Combination Circuits Using the Properties of Self-Dual Functions, *Avtomatika i Telemehanika*, 1997, no. 12, pp. 193—200 (in Russian).

19. **Goessel M., Dmitriev A. V., Sapozhnikov V. V., Sapozhnikov Vl. V.** A Functional Fault-Detection Self-Test for Combinational Circuits, *Avtomatika i Telemehanika*, 1999, no. 11, pp. 162—174 (in Russian).

20. **Saposhnikov Vl. V., Dmitriev A., Goessel M., Saposhnikov V. V.** Self-Dual Parity Checking — a New Method for on Line Testing, *Proceedings of 14th IEEE VLSI Test Symposium*, USA, Princeton, 1996, pp. 162—168.

21. **Saposhnikov Vl. V., Saposhnikov V. V., Dmitriev A., Goessel M.** Self-Dual Duplication for Error Detection, *Proceedings of 7th Asian Test Symposium*, Singapore, 1998, pp. 296—300. DOI: 10.1109/ATS.1998.741628.

22. **Borecký J., Kohlík M., Kubátová H.** Parity Driven Reconfigurable Duplex System, *Microprocessors and Microsystems*, 2017, vol. 52, pp. 251—260. DOI: 10.1016/j.micpro.2017.06.015.

23. **Hamming R. W.** Error Detecting and Correcting Codes, *Bell System Technical Journal*, 1950, vol. 29, no. 2, pp. 147—160. **DOI:** 10.1002/j.1538-7305.1950.tb00463.x.

24. **Efanov D. V., Saposhnikov V. V., Saposhnikov Vl. V.** Synthesis of Self-Checking Combination Devices Based on Allocating Special Groups of Outputs, *Avtomatika i Telemehanika*, 2018, no. 9, pp. 79—94 (in Russian).

25. **Efanov D. V., Pogodina T. S.** Self-Dual Control of Combinational Circuits with Using Hamming Codes, *Problemy razrabotki perspektivnyh mikro- i nanoelektronnyh sistem (MES)*, 2022, no. 3, pp. 113—122, DOI: 10.31114/2078-7707-2022-3-113-122 (in Russian).

26. **Efanov D. V., Sapozhnikov V. V., Sapozhnikov Vl. V.** Synthesis of Built-In Control Circuits for Combinational Digital Devices by the Method of Self-Dual Complement to the Berger Code, *Izvestija vysshih uchebnyh zavedenij. Priborostroenie*, 2021, vol. 64, no. 9, pp. 697—708, DOI: 10.17586/0021-3454-2021-64-9-697-708 (in Russian).

27. **Sapozhnikov V. V., Sapozhnikov Vl.V., Goessel M.** Self-Dual Discrete Devices, St. Petersburg, Energoatomizdat (St. Petersburg branch), 2001, 331 p. (in Russian).

28. **Efanov D. V., Saposhnikov V. V., Saposhnikov Vl. V.** The Concurrent Error-Detection Systems Organization Based on the Boolean Complement Method with the Preliminary Transformation of Operating Functions into Check Vectors of Berger Codes, *Informacionnye tehnologii*, vol. 27, no. 6, pp. 306—313, 2021, DOI: 10.17587/it.27.306-313 (in Russian).

29. **Baghdadi A. A. A., Hahanov V. I., Litvinova E. I.** Digital System Analysis and Diagnosis Methods (Analytical Review*), Avtomatizirovannye sistemy upravlenija i pribory avtomatiki*, 2014, no. 166, pp. 59—74 (in Russian).

30. **Efanov D. V., Pogodina T. S.** Self-Dual Digital Devices with Calculations Testing by Modified Hamming Code, *Proceedings of the 2023 Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, St. Petersburg, Russia, 24—27 January 2023, pp. 72—77.

31. **Efanov D. V., Pogodina T. S.** Properties Investigation of Self-Dual Combinational Devices with Calculation Control Based on Hamming Codes, *Informatika i Avtomatizatsiya*, 2023, vol. 22, no. 2, pp. 349—392. DOI: 10.15622/ia.22.2.5 (in Russian).

32. **Aksjonova G. P.** Restoration in Duplicated Units by the Method of Data Inversion, *Avtomatika i Telemehanika*, 1987, no. 10, pp. 144—153 (in Russian).

33. **Chen Y., Zhang M., Hao J.** The Circuit Design of Voltage-Controlled Color Changing Lamp Based on Multisim, *2020 IEEE International Conference on Power, Intelligent Computing and Systems (ICPICS)*, 28—30 July 2020, Shenyang, China, DOI: 10.1109/ICPICS50287.2020.9202148.

34. **Efanov D., Sapozhnikov V., Sapozhnikov Vl., Osadchy G., Pivovarov D.** Self-Dual Complement Method up to Constant-Weight Codes for Arrangement of Combinational Logical Circuits Concurrent Error-Detection Systems, *Proceedings of 17th IEEE East-West Design & Test Symposium (EWDTS'2019)*, Batumi, Georgia, September 13—16, 2019, pp. 136—143. DOI: 10.1109/EWDTS.2019.8884398.

УДК 004.75
DOI: 10.17587/prin.14.187-194

С. И. Жуков, канд. физ.-мат. наук, вед. программист, serge.zhukov@auriga.com,
НИВЦ МГУ,
К. А. Зубрилин, канд. физ.-мат. наук, разработчик программного обеспечения,
konstantin.zubrilin@auriga.com,
ООО “Аурига”, Москва

Использование технологии Amazon Kinesis для передачи данных с устройств интернета вещей в облачную инфраструктуру

Поступила в редакцию 31.01.2023
Принята к публикации 27.02.2023

Устройства интернета вещей, взаимодействующие с облаком Amazon Web Services (AWS), для подключения используют интерфейс IoT и протокол MQTT. В дополнение к этим базовым средствам коммуникации в AWS разработаны специализированные сервисы для передачи данных телеметрии с интеллектуальных устройств. Сервис Firehose позволяет загружать данные напрямую в хранилище данных AWS S3 (облачный аналог файловой системы). Сервис Kinesis предоставляет еще большие возможности: позволяет пользователю формально описать и программно реализовать внутреннюю логику для обработки данных, которая автоматически применяется к поступающим данным, а также анализировать данные телеметрии, используя методы искусственного интеллекта для обнаружения скрытых тенденций и закономерностей. В статье описана настройка IoT-устройства с помощью библиотеки AWS C++ SDK (Software Development Kit) для обработки данных его телеметрии сервисом Kinesis. Встроенное программное обеспечение IoT-устройства подключается к поставщику AWS с помощью API-интерфейсов SDK и отправляет данные телеметрии в виде пакетов данных сервиса Kinesis. На серверной стороне в облаке создается лямбда-функция, которая сохраняет полученные данные в файлохранилище S3 и после некоторой обработки отправляет их в хранилище (базу) данных DynamoDB. Сервис Kinesis позволяет использовать все богатство инструментария AWS для обработки и анализа данных (Amazon постоянно развивает этот инструментарий и дополняет его новыми возможностями).

Ключевые слова: *интернет вещей, облачные вычисления, IoT, AWS, Kinesis, AWS SDK*

Для цитирования:

Жуков С. И., Зубрилин К. А. Использование технологии Amazon Kinesis для передачи данных с устройств интернета вещей в облачную инфраструктуру // Программная инженерия. 2023. Том 14, № 4. С. 187—194. DOI: 10.17587/prin.14.187-194.

Введение

Задача передачи данных через сервис потоков данных Kinesis Amazon Web Services (AWS) возникла в процессе разработки встроенного программного обеспечения (ПО) интеллектуального контроллера для центров обработки данных. Конкретные функциональные возможности контроллера зависят от набора подключенных к нему сенсоров и контроллеров более низкого уровня, а также от набора правил, задаваемых пользователем. В частности, контроллер может заниматься управлением охлаждения, контролем и мониторингом механизмов доступа, сбором данных с сенсоров и их агрегированием для формирования массивов больших данных, оповещением о событиях и неисправностях, а также реализацией интерфейса интернета вещей (Internet of Things — IoT).

При этом интеллектуальное устройство подключается через Интернет к облачной инфраструктуре, поддерживающей интернет вещей, и взаимодействует с ней. Изначально для этого использовался протокол MQTT [1], однако в какой-то момент возникла необходимость передавать в облако и хранить там видеофреймы с видеокамеры низкого разрешения (232×160 пикселей, 5 FPS). Для целей последующего анализа видеопотока достаточно передавать и хранить в облаке один фрейм в секунду. На стороне облака фреймы подвергаются обработке перед сохранением. Сохраненный видеопоток может затем анализироваться средствами облачной инфраструктуры.

Для сохранения фреймов используется файлохранилище AWS Simple Storage Service (S3) [2, 3], аналогичное иерархически организованной файловой системе. Метаданные сохраняются в базе данных DynamoDB для быстрого поиска. DynamoDB [4, 5] — это реализованная в инфраструктуре AWS нереляционная (NoSQL) база данных. Для нее характерны бессерверная реализация, быстрый доступ к данным, высокая производительность, возможность хранения больших объемов данных. Данные хранятся в именованных таблицах и должны быть представлены в формате JSON. Одно или несколько полей структуры данных JSON используются как первичный ключ таблицы, по которому данные могут быть извлечены наиболее быстро. Для доступа к данным инфрастуктура AWS предоставляет как пользовательский, так и программный интерфейсы (реализованный на всех основных языках программирования).

Для передачи видеоданных от контроллера в облачную инфраструктуру AWS авторами было решено вместо протокола MQTT использовать сервис AWS Kinesis [6, 7], специально предназначенный как для передачи телеметрической и иной информации с устройств, так и для обработки этих данных на стороне облака в реальном времени и пересылки их в другие сервисы AWS.

Начальная настройка службы Kinesis в облачной инфраструктуре

Для передачи данных между устройством и облаком через службу Kinesis используются один или несколько потоков, которые нужно предварительно создать. Это можно сделать в web-сервисе AWS Console через раздел Amazon Kinesis → Data streams. При создании потока задаются его имя, параметры пропускной способности и время хранения невостребованных сообщений на сервере (по умолчанию 24 ч). Поток создается в определенном регионе AWS (регион можно выбрать и поменять в меню AWS Console), и в дальнейшем существует только в этом регионе. Примеры регионов: us-west-2, us-east-1, eu-central-1.

С точки зрения пропускной способности поток может создаваться в одном из двух режимов: provisioned и on demand.

В режиме provisioned потоку предоставляется фиксированная пропускная способность (инфраструктура статически выделяет необходимые ресурсы) и создается фиксированное число параллельных подпотоков (*shards*). При передаче данные распределяются по подпотокам и передаются и хранятся параллельно, что позволяет повысить суммарную пропускную способность потока. В этом режиме взимается фиксированная плата за обслуживание потока.

В режиме on demand предварительного выделения ресурсов не происходит, ресурсы назначаются динамически в зависимости от интенсивности передачи сообщений. Плата за обслуживание потока также берется динамически в зависимости от его загрузки.

В процессе разработки потоком Kinesis использовался режим on demand, что было более целесообразно, так как интенсивность потока сообщений была неравномерной.

После создания потока его имя можно использовать для работы с ним со стороны устройства и привязки к нему middleware-логики (например, лямбда-функций, обрабатывающих поступившие данные).

Программное обеспечение устройства для работы с инфраструктурой Kinesis

Для работы с облачными сервисами Amazon предоставляет библиотеку SDK для разных языков программирования, включая C++. Так как встроенное ПО устройства написано на языке C++ и работает под управлением ОС Linux, для работы с Kinesis использовалась интерфейсная библиотека AWS C++ SDK для ОС Linux.

Библиотека AWS SDK имеет модульную структуру, для экономии места на устройстве использовались только необходимые для работы Kinesis модули: aws-cpp-sdk-core, aws-cpp-sdk-kinesis, а также модуль aws-cpp-sdk-s3, который применялся для прямого доступа к файлохранилищу AWS (для записи объемных данных типа системных журналов).

Работа с модулем aws-cpp-sdk-kinesis происходит по представленной далее схеме.

1. Фаза инициализации:

a) инициализация SDK вызовом метода InitAPI();

b) создание объекта AWSCredentials, который содержит информацию для аутентификации пользователя AWS: идентификатор ключа доступа и секретный ключ;

c) создание объекта ClientConfiguration; этот объект включает имя региона, в котором определен поток (например, us-west-2);

d) создание объекта KinesisClient; в качестве параметров ему передаются ранее созданные объекты AWSCredentials и ClientConfiguration.

2. Фаза передачи данных. В течение этой фазы для каждого передаваемого элемента данных (или нескольких элементов данных) выполняются следующие действия:

a) создается объект PutRecordRequest, в котором заполняются следующие поля:

- имя потока Kinesis;
- ключ записи (*partition key*), который идентифицирует элемент данных и используется для равномерного распределения записей по подпотокам;
- данные в виде массива байтов;

b) созданный объект PutRecordRequest передается методу PutRecord() объекта KinesisClient, который возвращает объект PutRecordOutcome;

c) в объекте PutRecordOutcome вызывается метод IsSuccess(), который возвращает статус выполнения переданного запроса (успех/неудача); в случае неудачи методы GetError().GetErrorType() и GetError().GetErrorMessage() возвращают тип ошибки и сообщение об ошибке.

3. Фаза завершения работы, в течение которой:

a) разрушаются объекты, созданные в фазе инициализации (с вызовом деструкторов);

b) вызывается метод ShutdownAPI() для завершения работы SDK.

Далее представлены примеры кода для каждой фазы работы.

Фаза инициализации:

```
   options_ = "";
   Aws::InitAPI(options_);
   Aws::Client::ClientConfiguration configKinesis;
   Aws::Auth::AWSCredentials credentials;
   credentials.SetAWSAccessKeyId(accessKeyID);
   credentials.SetAWSSecretKey(secretKey);
   configKinesis.region = "us-west-2";
   kinesis_client_ = new Aws::Kinesis::KinesisClient(cre
dentials, configKinesis);
```

Фаза передачи данных:

```
Aws::Kinesis::Model::PutRecordRequest putRecord Request;
   putRecordRequest.SetStreamName("TofDataStream");
   std::string pkey = aws_root_dir_ + "/raw/" +
std::to_string(frame_data.frame_process_count) + ".bin";
   putRecordRequest.SetPartitionKey(pkey);

   int binary_length = frame_data.
serializeRawDataBinary(binary_buffer,
sizeof(binary_buffer));
   Aws::Utils::ByteBuffer bytes(binary_buffer,
binary_length);
   syslog(LOG_NOTICE, "Frame data size = %d,
partition key = %s\n", binary_length, pkey.c_str());
   putRecordRequest.SetData(bytes);
   Aws::Kinesis::Model::PutRecordOutcome
putRecordResult =
         kinesis_client_->PutRecord
(putRecordRequest);
…
   if (putRecordResult.IsSuccess()) {
      // Process successful sending
…
      return true;
   } else {
       syslog(LOG_NOTICE, "Put record failed,
error = %d, %s\n",
         int(putRecordResult.GetError().
GetErrorType()), putRecordResult.GetError().
GetMessage().c_str());
      return false;
   }
```

Фаза завершения работы:

```
   delete kinesis_client_;
   Aws::ShutdownAPI(options_);
```

Middleware для работы с потоками Kinesis

Для обработки данных из потока Kinesis со стороны облака используются, как правило, лямбда-функции [8]. Это программные компоненты, написанные на одном из поддерживаемых инфраструктурой языков и исполняемые в облаке в качестве реакции на какое-то событие. Для рассматриваемой задачи лямбда-функция написана на языке Python и привязана к используемому потоку Kinesis. В результате лямбда-функция запускается, когда в поток приходят данные, и обрабатывает все данные, которые буферизованы в потоке Kinesis на данный момент. После завершения обработки всех данных лямбда-функция завершается и запускается повторно, когда новые данные приходят в поток. Входным параметром лямбда-функции является массив записей, находящихся в буфере потока в данный момент.

В рассматриваемом случае обработка данных заключается в следующем:

1) чтение очередной записи из входного массива;

2) определение типа данных из ключа записи (*partition key*);

3) для видеофреймов:

a) обработка данных видеофрейма: вычленение черно-белой и цветной картинки, выделение метаданных;

b) сохранение видеоинформации в виде файлов в хранилище S3;

c) сохранение метаданных (включая имена файлов с видеоинформацией) в базу данных DynamoDB.

Далее представлен фрагмент кода лямбда-функции на языке Python.

```
s3 = boto3.resource("s3")
dynamodb_resource = boto3.resource('dynamodb',
region_name = 'us-west-2')

def lambda_handler(event, context):

   record_list = event["Records"]
   # record_list contains records handled by this
invocation of the lambda function
   count = 0
   for item in record_list:
       count += 1
       if item["eventSource"] = = «aws:kinesis»:
           partitionKey = item["kinesis"]
["partitionKey"]
           record_data = base64.
b64decode(item["kinesis"]["data"])
           upload_datetime = datetime.fromtimestamp
(item["kinesis"] ["approximateArrival
Timestamp"]).isoformat()

           # write record to an S3 bucket
           bucket_name = "tof-data"
           data_bucket = s3.Bucket(bucket_name)
           data_bucket.put_object(Key =
partitionKey, Body = record_data)

           # Convert image from proprietary
camera-specific format to
           # standard format
           environment, run_id, _, filename =
partitionKey.split('/')
           filename = filename.replace('bin', 'png')
           frame_number = int(filename.split('.')[0])
           data = royale.DepthData(record_data, False)
           # Create images and write them to
temporary files using
           # OpenCV tools
           gray_image = data.create_gray_image()
           rgb_image = data.create_rgb_image()
           cv2.imwrite("/tmp/bw.png", gray_image)
           cv2.imwrite("/tmp/orig.png", rgb_image)

           # Upload images to S3 next to the
original image
           data_bucket.upload_file("/tmp/bw.png", "{}/
{}/bw/{}.png".format(environment, run_id, frame_
number))
           data_bucket.upload_file("/tmp/orig.png",
"{}/{}/orig/{}.png".format(environment, run_id,
frame_number))

   # Add entry to the database
           frames_table = dynamodb_resource.
Table('Frames-test' if environment == "devel" else
'Frames')
           frame_info = {
             "frame-id": "{}_{:06d}".format(run_id,
frame_number),
             "bw-path": "s3://" + bucket_name + "/
{}/{}/bw/{}.png".format(environment, run_id, frame_
number),
             "run-id": run_id,
             "environment": environment,
             "filename": filename,
             "frame-number": frame_number,
             "is-labeled": False,
             "label-requested": False,
             "upload-date": upload_datetime
           }
           frames_table.put_item(Item = frame_info)
   print('Processed', count, 'items')
   return True
```

Привязку лямбда-функции к потоку Kinesis можно сделать через AWS Console при создании лямбда-функции в разделе Configuration → Add Triggers. В качестве триггера лямбда-функции указываются тип Kinesis и имя потока, созданного на предыдущем шаге (рис. 1).

Потоки Kinesis также предоставляют интерфейс к другим службам AWS, включая службы, предназначенные для обработки больших данных (Redshift) и машинного обучения (SageMaker). В дальнейшем планируется использовать эти службы для обработки данных телеметрии и видеоданных.

Оценка производительности потоков Kinesis

Размер передаваемого видеофрейма составляет 730 Кбайт (в собственном формате камеры), так что он укладывается в одну запись, размер которой в потоке Kinesis ограничен 1 Мбайтом.

Среднее время передачи одного видеофрейма составляет около 600 мс. Это соответствует требованиям задачи, по которым достаточна передача в среднем хотя бы одного фрейма в секунду.

Для более точного исследования скорости передачи были проанализированы три сеанса передачи данных через Kinesis. Результаты анализа представлены в таблице.

В каждом сеансе время доставки первого пакета существенно больше средних значений, что объясняется дополнительными затратами на установление соединения, поэтому первый пакет был исключен из рассмотрения.

По форме гистограмм видно, что распределение времени доставки напоминает нормальное

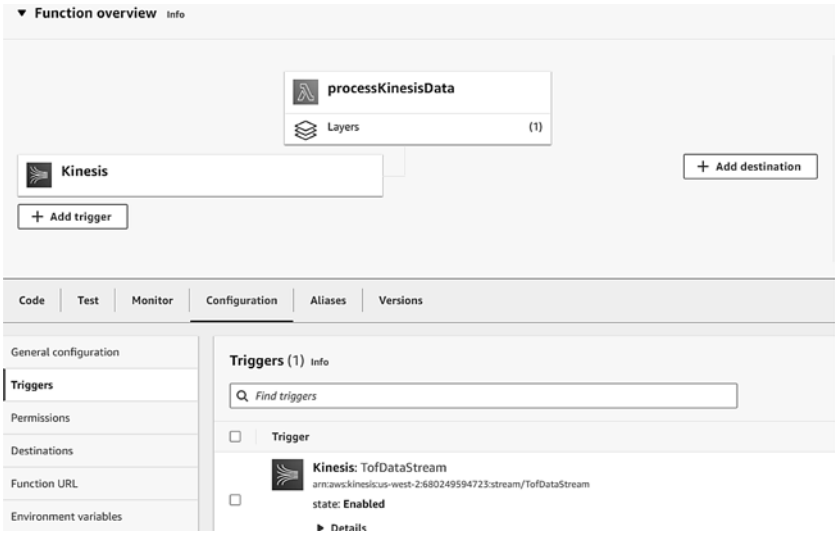


Рис. 1. Привязка лямбда-функции к потоку Kinesis

Результаты анализа трех сеансов передачи данных

Сеанс	Число пакетов	Среднее время передачи, мс	Стандартное отклонение, мс	Гистограмма
1	36	655,08	151,95	Рис. 2
2	50	552,58	159,16	Рис. 3
3	50	692,38	198,69	Рис. 4

со средним значением 500...600 мс, за исключением сравнительно немногочисленных выбросов в большую сторону.

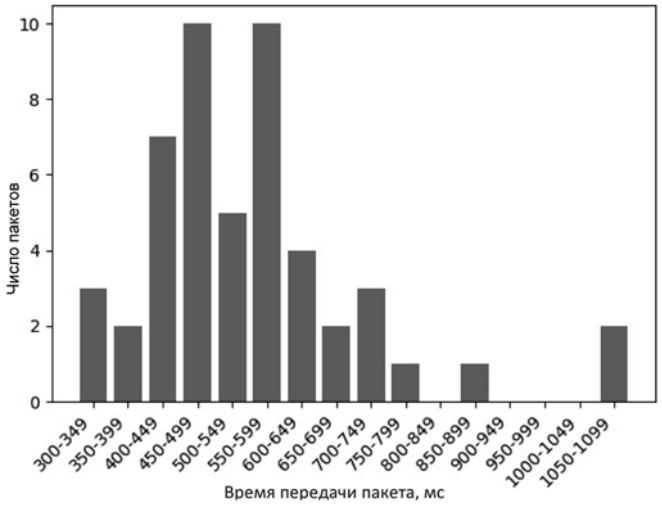


Рис. 3. Гистограмма времени передачи пакетов в сеансе 2

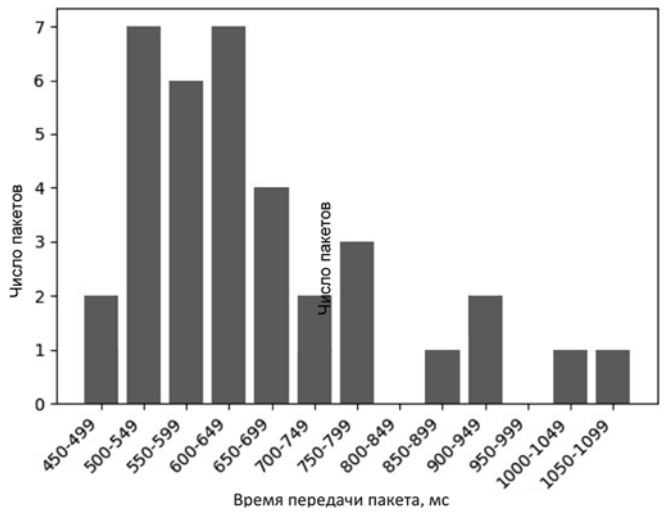


Рис. 2. Гистограмма времени передачи пакетов в сеансе 1

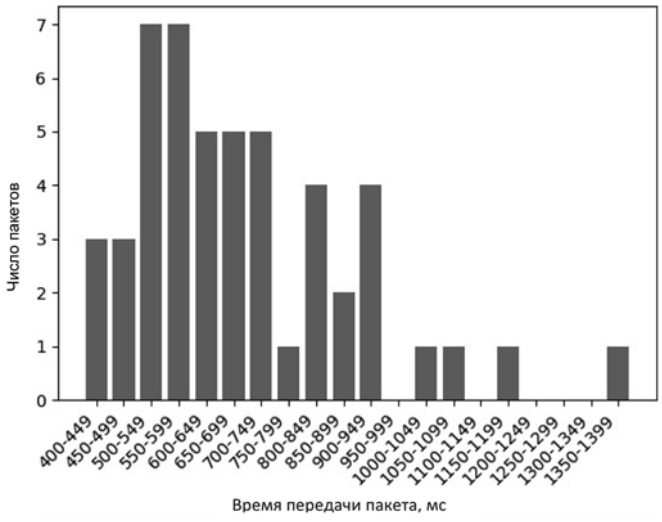


Рис. 4. Гистограмма времени передачи пакетов в сеансе 3

Рассмотрим типичный журнал передачи данных на устройстве.

```
Dec 22 23:33:20: Frame data size=732036, partition
key=field/2131N401035101418_2022-12-22_23:27:06/
raw/1830.bin

Dec 22 23:33:20: Put record successful,
shard=shardId-000000000170, 487 ms total, 6 ms
serialize, 480 ms transmit

Dec 22 23:33:20: Frame data size=732036, partition
key=field/2131N401035101418_2022-12-22_23:27:06/
raw/1833.bin

Dec 22 23:33:21: Put record successful, shard=
shardId-000000000156, 507 ms total, 5 ms serialize,
502 ms transmit

Dec 22 23:33:21: Frame data size=732036, partition
key=field/2131N401035101418_2022-12-22_23:27:06/
raw/1837.bin

Dec 22 23:33:21: Put record successful, shard=
shardId-000000000242, 382 ms total, 4 ms serialize,
377 ms transmit

Dec 22 23:33:21: Frame data size=732036, partition
key=field/2131N401035101418_2022-12-22_23:27:06/
raw/1838.bin

Dec 22 23:33:22: Put record successful,
shard=shardId-000000000162, 566 ms total, 4 ms
serialize, 561 ms transmit

Dec 22 23:33:22: Frame data size=732036, partition
key=field/2131N401035101418_2022-12-22_23:27:06/
raw/1841.bin

Dec 22 23:33:22: Put record successful,
shard=shardId-000000000210, 524 ms total, 5 ms
serialize, 518 ms transmit
```

На основании данных журнала можно сделать следующие выводы:

- большую часть времени доставки записи составляет время передачи ее по сети, при этом время обработки (сериализации) фрейма пренебрежимо мало;
- при передаче используются различные подпотоки (*shards*), что позволяет распараллелить процесс приема данных на стороне облака и хранить различные записи в различных кешах.

Адаптация для Yandex.Cloud

В текущей внешнеполитической ситуации существует вероятность прекращения доступа к инфраструктуре AWS для клиентов из России. Поэтому, а также в рамках работ по импортозамещению, была рассмотрена возможность переноса приложения на отечественную облачную инфраструктуру Yandex.Cloud, включающую:

- Data Streams — программно-совместимый аналог Kinesis;
- Object Storage — программно-совместимый аналог S3;
- Cloud Functions — программно-совместимый аналог AWS Lambda;
- Managed YDB — программно-совместимый аналог DynamoDB (для так называемых документных таблиц).

Перенос приложения был успешно осуществлен на уровне прототипа. Были реализованы передача данных от устройства к облачной инфраструктуре через поток Data Streams, создание облачной функции обработчика данных и ее привязка к потоку через триггер, запись данных из облачной функции в хранилище Object Storage и в документную таблицу Managed YDB. Однако для уменьшения необходимого объема работ из облачной функции были исключены специфическая для такой задачи обработка данных, создание некоторых объектов в хранилище Object Storage и заполнение некоторых полей в документной таблице.

При переносе обнаружены небольшие, но существенные различия в программном интерфейсе между инфраструктурами Yandex.Cloud и AWS, они описаны далее для каждой из составных частей решения.

1. Создание потока данных.

- При создании потока данных через Yandex Console имя потока не может содержать заглавных букв; имя потока пришлось изменить с TofDataStream на tofdatastream.

2. Приложение на устройстве.

- При инициализации необходимо задать регион ru-central1 и переопределить адрес точки доступа к инфраструктуре (*endpoint*):

```
   configKinesis.region = Aws::String("ru-central1");
   configKinesis.endpointOverride =
Aws::String("https://yds.serverless.yandexcloud.net");
```

- При передаче данных в качестве имени потока необходимо задавать его полный путь в инфраструктуре (доступен через Yandex Console):

```
   putRecordRequest.SetStreamName(
   "/ru-central1/b1g01hlkbh2a0p1k1bhu/
etnmacs59uhg3s8ilk81/tofdatastream");
```

- В отличие от AWS, триггер Data Streams вызывает заданную облачную функцию для принятой записи, только если эта запись находится в формате JSON. Произвольные бинарные данные можно передать через поток, но облачная функция для них не вызовется. Поэтому необходимо в приложении преобразовать передаваемые данные в формат Base64 и передавать их как поле JSON-объекта. Кроме того, JSON-объект должен содержать ключ записи, потому что в отличие

от AWS, ключ записи не передается в облачную функцию как отдельный элемент данных:

```
   int binary_length = frame_data.
serializeRawDataBinary(binary_buffer,
sizeof(binary_buffer));
   // Convert to base64
   std::string encoded_content = "{\"key\":\"" +
pkey + "\", \"data\":\"" + encode64(binary_buffer,
binary_length) + "\" }";
   Aws::Utils::ByteBuffer bytes((unsigned char *)
encoded_content.c_str(), encoded_content.size());
```

3. Облачная функция.

Облачная функция требует наибольшей переработки при переносе в Yandex.Cloud. Основные отличия заключаются в следующем:

- имя функции handler вместо lambda_handler;
- для использования библиотеки AWS (boto3) необходимо добавить файл requirements.txt, содержащий требуемую версию библиотеки, например:

```
boto3==1.13.15
```

- для доступа к службам Object Service и Managed YDB можно использовать сервисы boto3.s3 и boto3.dynamodb, но переопределив точки доступа к сервисам;
- формат объекта, передаваемого облачной функции от триггера Data Streams, отличается коренным образом от AWS; этот объект содержит единственное поле messages, представляющее собой массив принятых записей (которые обязаны быть в формате JSON).

Прототип облачной функции для Yandex.Cloud, реализующей сохранение данных в Object Storage и Managed YDB, приведен далее.

```
import json
import boto3
import base64

from datetime import datetime

session = boto3.session.Session()
s3 = session.client(
   service_name = 's3',
   endpoint_url = 'https://storage.yandexcloud.net')
   dynamodb_resource = boto3.resource('dynamodb',
   endpoint_url  =  'https://docapi.serverless.
yandexcloud.net/ru-central1/b1g01hlkbh2a0p1k1bhu/
etnmacs59uhg3s8ilk81')

def handler(event, context):
   record_list = event["messages"]
   count = 0
   print('Record count:', len(record_list))
   for item in record_list:
      count + = 1
      bucket_name = "tofdata"
   record_data = base64.b64decode(item["data"])
   print('Before put-object, data length = ',
len(item["data"]), ',', len(record_data))
   record_key = 'S3-' + item["key"]
   s3.put_object(Bucket = bucket_name, Key = record_
key, Body = record_data, StorageClass = 'COLD')
   print('After put-object')
   # put data into the table
   now = datetime.now()
   upload_datetime = now.strftime("%Y-%d-%m %H:%M:%S")
   frames_table = dynamodb_resource.Table('Frames')
   frame_info = {
      "frame-id": count,
      "bw-path": "s3://" + bucket_name + "/" + record_key,
      "filename": item["key"],
      "upload-datetime": upload_datetime
   }
   frames_table.put_item(Item = frame_info)
   print('After put_item')

print('Processed', count, 'items')
return True
```

Заключение

Потоки Kinesis оказались адекватным средством для передачи видеоданных низкого разрешения от IoT-устройства в облако, обработки этих данных в облачной инфраструктуре и сохранения их в файловом хранилище S3. При этом в базе данных DynamoDB хранятся метаданные для каждого фрейма и URL соответствующих файлов. Архитектура потоков расширяема и позволяет в дальнейшем подключить другие службы AWS (например, средства работы с большими данными или инструменты машинного обучения) в качестве обработчиков поступающих данных.

Кроме того, данное решение возможно перенести на отечественную облачную инфраструктуру Yandex.Cloud с сохранением функциональных возможностей и со сравнительно небольшими изменениями в исходном коде.

Список литературы

1. **MQTT Version** 3.1.1. OASIS Standard. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (дата обращения 31.01.2023).
2. **AWS** General Reference. Reference Guide. URL: https://docs.aws.amazon.com/general/latest/gr/Welcome.html (дата обращения 31.01.2023).
3. **Amazon** Simple Storage Service. User Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (дата обращения 31.01.2023).
4. **Amazon** DynamoDB. Developer Guide. URL: https://https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html (дата обращения 31.01.2023).
5. **The Architecture** of Amazon's DynamoDB and Why Its Performance Is So High. URL: https://medium.com/swlh/architecture-of-amazons-dynamodb-and-why-its-performance-is-so-high-31d4274c3129 (дата обращения 31.01.2023).
6. **Amazon** Kinesis Data Streams. Developer Guide. URL: https://docs.aws.amazon.com/streams/latest/dev/introduction.html (дата обращения 31.01.2023).
7. **Amazon** Kinesis Data Streams. API Reference. URL: https://docs.aws.amazon.com/kinesis/latest/APIReference/Welcome.html (дата обращения 31.01.2023).
8. **Beswick J**. Using AWS Lambda as a consumer for Amazon Kinesis. URL: https://aws.amazon.com/blogs/compute/using-aws-lambda-as-a-consumer-for-amazon-kinesis/ (дата обращения 31.01.2023).

Using Amazon Kinesis Service for Data Transfer from IoT Devices to Cloud Infrastructure

S. I. Zhukov, Leading Researcher, serge.zhukov@auriga.com,
RCC, Moscow State University, Moscow, 119991, Russian Federation,
K. A. Zubrilin, Software Developer, konstantin.zubrilin@auriga.com,
Auriga, Moscow, 117587, Russian Federation

Corresponding author:

Sergey I. Zhukov, Leading Researcher,
RCC, Moscow State University, Moscow, 119991, Russian Federation
E-mail: serge.zhukov@auriga.com

Received on January 31, 2023
Accepted on February 27, 2023

IoT devices connected to the Amazon Web Services (AWS) cloud use the IoT interface and the MQTT protocol for communication purposes. In addition to these basic communication tools, AWS has developed specialized services for transmitting telemetry data from smart devices. Firehose allows you to upload data directly to the AWS S3 data warehouse (cloud-specific file system). Kinesis service is even more powerful; it enables the user to build back-end data logic, which is automatically applied when data arrives. Also, Kinesis allows running analytics of the telemetry data using artificial intelligence techniques to discover hidden trends and consistent patterns. This article describes how to set up an IoT device using the AWS C++ SDK so that telemetry data it generates is processed by Kinesis services. The firmware of the device connects to the AWS provider using the SDK APIs and sends telemetry data as Kinesis service data packets; a Kinesis lambda function is created on the server side in the cloud to store the received data in S3 storage and send the data to the DynamoDB database after processing. Using Kinesis services allows the user to utilize all power tools provided by the AWS framework for processing and analyzing their data (Amazon constantly develops and enhances this set of tools).

Keywords: *IoT, AWS, Kinesis, cloud computing, telemetry, lambda functions*

For citation:

Zhukov S. I., Zubrilin K. A. Using Amazon Kinesis Service for Data Transfer from IoT Devices to Cloud Infrastructure, *Programmnaya Ingeneria,* 2023, vol. 14, no. 4, pp. 187—194. DOI: 10.17587/prin.14.187-194/

References

1. **MQTT Version** 3.1.1. OASIS Standard, available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (date of access 31.01.2023).

2. **AWS** General Reference. Reference Guide, available at: https://docs.aws.amazon.com/general/latest/gr/Welcome.html (date of access 31.01.2023).

3. **Amazon** Simple Storage Service. User Guide, available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (date of access 31.01.2023).

4. **Amazon** DynamoDB. Developer Guide, available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html (date of access 31.01.2023).

5. **The Architecture** of Amazon's DynamoDB and Why Its Performance Is So High, available at: https://medium.com/swlh/architecture-of-amazons-dynamodb-and-why-its-performance-is-so-high-31d4274c3129 (date of access 31.01.2023).

6. **Amazon** Kinesis Data Streams. Developer Guide, available at: https://docs.aws.amazon.com/streams/latest/dev/introduction.html (date of access 31.01.2023).

7. **Amazon** Kinesis Data Streams. API Reference, available at: https://docs.aws.amazon.com/kinesis/latest/APIReference/Welcome.html (date of access 31.01.2023).

8. **Beswick J**. Using AWS Lambda as a consumer for Amazon Kinesis, available at: https://aws.amazon.com/blogs/compute/using-aws-lambda-as-a-consumer-for-amazon-kinesis/ (date of access 31.01.2023).

ОТ РЕДАКЦИИ:

Статья **«Анализ отношений в виртуальном пространстве социальных сетей»** автора Алекперовой И. Я., опубликованная в № 1, 2023 г., была выполнена при финансовой поддержке Азербайджанского научного фонда — Грант № EF-BGM-4-RFTF-1/2017-21/08/1.

УДК 004.912
DOI: 10.17587/prin.14.195-202

А. С. Козицын, канд. физ.-мат. наук, вед. науч. сотр., alexanderkz@mail.ru,
Московский государственный университет имени М. В. Ломоносова

Алгоритмы поиска дубликатов научно-технических конференций и групп конференций в наукометрических системах

Поступила в редакцию 19.01.2023
Принята к публикации 07.02.2023

Представлены разработанные автором методы оценки близости описания конференций научно-технического содержания в целях обнаружения дубликатов, а также построения групп таких конференций. Дан обзор существующих каталогов конференций в Интернете, проанализированы их преимущества и недостатки. Обоснована необходимость разработки методов более точной верификации входных данных о конференциях. Приведены описание разработанных автором алгоритмов и их программной реализации, а также результаты их апробации на больших данных на примере реально существующей информационно-аналитической наукометрической системы. Описанные методы могут применяться при разработке каталогов конференций и наукометрических систем для верификации исходных данных в целях повышения качества построения рейтинговых оценок, ранжирования конференций и обеспечения тематического поиска.

Ключевые слова: *поиск дубликатов, наукометрия, информационные системы, конференция*

Для цитирования:

Козицын А. С. Алгоритмы поиска дубликатов научно-технических конференций и групп конференций в наукометрических системах // Программная инженерия. 2023. Том 14, № 4. С. 195—202. DOI: 10.17587/prin.14.195-202.

Введение

Своевременный анализ основных тенденций развития различных направлений науки и техники необходим в том числе для выбора эффективных способов оценки и стимулирования научной, технической и технологической (далее — научной) видов деятельности [1] и организации способствующих этому конкурсных процедур [2].

При оценке результатов научной деятельности в наукометрических системах необходимо использовать различного рода формальные показатели, учитывающие не только количественные характеристики научной продукции (число статей, монографий, докладов), но и различные индикаторы ее качества [3]. В настоящее время для оценки качества публикации принято использовать различные показатели авторитетности журналов, в которых она представлена. Одним из самых распространенных таких показателей является импакт-фактор, который рассчитывается по большим базам данных цитируемости публикаций. Наиболее известным каталогом зарубежных журналов с указанием импакт-факторов является JSR (*Journal Citation Reports*), который издается ежегодно компанией Clarivate Analytics на основе данных, представленных в системе Web of Science. Для российских журналов также существует система аналогичного назначения с указанием импакт-факторов — РИНЦ (Российский индекс научного цитирования). Наличие таких каталогов позволяет создавать на их основе наукометрические показатели оценки публикационной активности отдельных авторов. Однако использование только одного типа показателей не позволяет получить достаточно объективную оценку деятельности отдельного исследователя [4, 5]. Для построения более качественных

рейтинговых оценок научной деятельности необходимо учитывать показатели активности исследователей в других сферах, в том числе при представлении своих результатов на конференциях. Подобные показатели должны учитывать не только число докладов и презентаций каждого автора, но и авторитетность конференций, на которых он их представляет. Таким образом, возникает необходимость создания каталогов конференций, имеющих достаточно широкий охват как по тематике, так и по их географическому положению, позволяющих проводить ранжирование конференций по показателям (по степени) приоритетности.

Краткий обзор каталогов конференций

В Интернете представлено большое число проектов, которые в той или иной степени обеспечивают хранение, рубрикацию и поиск конференций. Значительное число таких проектов ставит первоочередной задачу продвижения и рекламы услуг по быстрой публикации платных статей. На веб-страницах подобных каталогов представлено, как правило, небольшое число конференций с низким уровнем их авторитетности в научном сообществе. Например, проект «научные-конференции.рф» (na-konferencii.ru) содержит всего семь конференций с датой начала в январе-феврале 2023 г., не имеет тематического поиска, а тематика представленных конференций не определяется по названиям (например, «Вектор науки», «Вопросы развития современной науки и техники»). Заявки на такие конференции принимаются вплоть до начала конференции, что свидетельствует об отсутствия возможности проведения какого-либо рецензирования представляемых материалов.

В проекте konferencii.ru в январе-феврале 2023 г. зарегистрировано более 250 конференций, для большинства которых срок окончания приема заявок совпадает с датой начала конференции, а названия конференций свидетельствуют об отсутствии выраженной тематической направленности мероприятия (например, «Актуальные вопросы развития современных технологий»; «Научный прорыв 2023»; «Наука и образование в контексте глобальной трансформации»). Значительную часть веб-страницы этих конференций занимают баннеры с рекламой срочной публикации статей. На этих страницах декларируется наличие расширенного поиска, в том числе по тематике мероприятия. Однако результат поиска по рубрике «математика» содержит всего четыре конференции «широкого профиля», например, «Современное образование: проблемы; решения; тенденции развития».

Проект «Научные конференции России» (www.kon-ferenc.ru) также представляет конференции «широкого профиля» для быстрой публикации статей с рекламой «Публикация научных статей за 3 дня», причем для многих конференций указан срок окончания приема заявок, но не указано время и место проведения мероприятия.

Основной недостаток отмеченных выше каталогов состоит в низком качестве предоставляемой информации. Большое число, по существу, фиктивных (а точнее — неэффективных) конференций не только затрудняет процесс информационного поиска реальных мероприятий, но и создает препятствия к наполнению каталога качественными данными. Организаторы авторитетных конференций не регистрируют свои мероприятия в подобном контексте.

Проект obshestvo.org специализируется на молодежных мероприятиях, но содержит описание только 2280 реальных конференций и олимпиад за все время работы проекта. Система поиска не позволяет проводить фильтрацию по времени, месту проведения и другим основным параметрам.

Проект konferen.ru позиционируется как календарь научных конференций России и содержит около тысячи планируемых мероприятий: олимпиад, школ, конференций и др. Поиск на сайте возможен по словам из названия и по одной из 39 рубрик.

Проект поиска мероприятий WorldExpo (worldexpo.pro) специализируется на представлении коммерческих выставок, форумов, семинаров и в меньшей степени конференций.

В Интернете представлены также зарубежные системы поиска конференций, например, проект conferencealerts.com предоставляет возможность поиска по рубрике и стране, а проект www.allconferences.com охватывает около 50 тыс. конференций и предоставляет возможность фильтрации по стране, городу, названию и рубрике. Однако большая часть конференций, зарегистрированных в таких системах, носит коммерческий характер. Например, в системе www.allconferences.com зарегистрировано более 1000 конференций по сетевым протоколам, более 600 конференций по электронной коммерции, около 100 конференций по компьютерным играм и только десять конференций по алгоритмике. Кроме того, в таких системах, как правило, отсутствуют не только российские конференции, но и большинство международных конференций, проходящих на территории России. Например, по данным conferencealerts.com в Рос-

сии в 2023 г. проходит только 15 различных конференций. Для осуществления качественной оценки научной деятельности необходимо учитывать в том числе русскоязычные источники информации, особенно в области гуманитарных наук [6].

Русскоязычные системы обзора конференций организаций (например, conf.msu.ru в МГУ им. М. В. Ломоносова, www.nstu.ru/science/scientific_events/search НГТУ, www.ruthenia.ru Тарусского университета) специализируются в большей степени на мероприятиях своей организации и имеют незначительный охват в масштабах России.

Существует отдельная категория систем, предназначенных для поддержки процесса организации конференций. Большая часть зарегистрированных в этих системах конференций имеет реально действующий программный комитет, рецензируемые доклады, и, как следствие, хороший научный уровень. Например, система EasyChair (easychair.org) содержит большое число конференций, но при этом не имеет системы поиска. Российский ее аналог «Конференции России» (ruconf.ru) имеет систему контекстного поиска и поиска по классификатору, однако число загруженных в нее конференций незначительно (машиностроение — 1, медицина — 2 и т. д.).

Большое число докладов в настоящее время можно найти в системе РИНЦ. Однако в этой системе отсутствует возможность работы с записями о конференциях. На страницах докладов и сборников может быть указано название конференции, но только в текстовом формате без функции перехода на страницу с описанием конференции и возможности анализа данных о конференции. Организаторы некоторых конференций обходят этот недостаток и создают фиктивный журнал (например, «Научный сервис в сети интернет»), к которому прикрепляются все материалы о докладах за весь период времени работы конференции разных лет. Однако такая запись представлена как журнал или сборник, и нет возможности указать, что это группа конференций или вывести список проведенных конференций по годам с указанием дат и мест.

На основании представленного краткого обзора можно сделать вывод, что в России на настоящее время не существует каталога конференций, который можно было бы эффективно использовать для реализации механизмов, позволяющих решать важные задачи подготовки управленческих решений на основе взаимодействия таких каталогов с системами наукометрии, в том числе для определения степени авторитетности конференций. Следует отметить, что решение задачи построения такого каталога конференций существенно сложнее, чем построение каталога журналов. При построении, например, каталога журналов в РИНЦ имеется возможность использовать административный ресурс и собирать данные централизовано от всех зарегистрированных издательств. Проведение конференций осуществляется большим числом оргкомитетов, которые не имеют регистрации, и, как следствие, их нельзя обязать подавать информацию о проводимых конференциях в единый регистрирующий орган. Таким образом, необходимо использовать механизмы сбора информации «снизу вверх», которые описаны в работе [1]. Это в свою очередь требует разработки и внедрения механизмов верификации данных о конференциях, выявления дубликатов и объединения конференций в группы.

Алгоритм поиска дубликатов

Выявление дубликатов в наукометрических системах является важным этапом «очистки» и подготовки входных данных. Без решения этой задачи нельзя обеспечить качественный уровень анализа данных, построения агрегированных наукометрических показателей, осуществления тематического поиска научной информации [7, 8] или поиска экспертов в заданной предметной области [9, 10].

Для определения степени сходства записей о конференции использовались следующие вводимые пользователем признаки: название конференции, период проведения, место проведения. Указанные признаки не могут использоваться для точного сравнения, поскольку при написании названий конференций пользователи меняют местами слова, включают в название места проведения, заменяют части названия аббревиатурами и делают другие изменения. Указание мест осуществляется с разным написанием названий организаций и с различным уровнем детализации, (например, «факультет журналистики МГУ», «Московский государственный университет», «Москва», «Россия»), даты проведения указывают с отличием в несколько дней. Кроме того, пользователи вводят данные с опечатками. В связи с этим необходимо использовать нечеткий поиск, учитывающий все три признака.

Выбор признаков обусловлен набором данных, которые содержатся в библиографических ссылках и которые пользователи вводят в наукометрическую систему. Существуют и другие признаки, например, состав организационных и программных комитетов. Однако эти данные отсутствуют

в библиографических описаниях тезисов и материалах докладов.

Наиболее ресурсоемкой задачей при таком подходе является сравнение схожести названий для каждой пары конференций. Для сравнения названий применяют два метода: использование хеш-функции и пословный индекс.

Схема работы алгоритма выявления дубликатов конференций представлена на рис. 1.

При построении хеш-функции из названия конференции удаляются все спецсимволы, арабские и римские числительные, высокочастотные стоп-слова, все буквы приводятся к верхнему регистру и для полученной строки вычисляется хеш-значение, по которому строится индекс для последующего быстрого поиска в блоках «Построение хеш-функции» и «Индексный поиск по хеш-функции». На вход этим модулям передается название конференции.

Сравнение при помощи пословного индекса проводится с использованием модели имени bag-of-words («мешок слов»). В блоке «Морфологический анализ» происходит построение морфем, и для каждой морфемы рассчитывается показатель частотности *idf* [11]. При индексном поиске по значимым словам мера близости вычисляется для всех пар конференций, которые имеют не мене двух совпадающих значимых морфем по формуле:

$$\frac{\sum\limits_{w \in A \cup B} idf_w}{\min\left(\sum\limits_{w \in A} idf_w, \sum\limits_{w \in B} idf_w\right)},$$

где A, B — множество слов w, принадлежащих названию каждой из двух конференций и имеющих $idf > \frac{1}{3000}$. На вход этим модулям также передается название конференции.

В блоках «Расширение места гипонимами и гиперонимами» и «Индексный поиск по месту проведения» при сравнении близости по месту используется словарь гипонимов мест, который дообучается в процессе работы на основе результатов выявления дубликатов конференций по другим признакам. В качестве входных данных для этих модулей используется название места проведения конференции.

Блок «Фильтрация и ранжирование по интервалу дат» для сравнения дат использует функцию для нечеткого сравнения пересекающихся интервалов:

$$\frac{1}{1 + \max\left(|s_1 - s_2|, |e_1 - e_2|\right)},$$

где s_i, e_i — дни начала и окончания i-й конференций соответственно.

Рис. 1. Схема работы алгоритма выявления дубликатов конференций

Кроме того, для учета ошибок указания месяца при вводе данных о конференции дополнительный ранг получают записи о конференциях, в которых совпадают числа начала и окончания конференции, но отличается месяц. На вход блоку передаются даты начала и окончания проведения конференции.

На выходе описанных выше модулей формируются четыре списка первичных ключей, хранящихся в системе конференций, найденных по соответствующему признаку, с указанием ранга соответствия заданному поисковому запросу.

На последнем этапе построенные списки передаются на вход блока «Ранжирование результатов поиска», который объединяет эти списки и осуществляет подсчет итоговой меры схожести каждой найденной пары конференций. Для такого подсчета используется взвешенное суммирование рангов, полученных на основе их сравнения по каждому из четырех описанных выше критериев. На выходе этого блока формируется итоговый отсортированный список найденных конференций с указанием меры сходства.

Программная реализация алгоритма выполнена на языке PL/SQL в виде двух программных модулей:

- FIND_SIMILAR_CONF — поиск конференций, похожих по названию, датам и месту;
- MERGE_EQUAL — поиск и слияние дубликатов по всему множеству конференций.

Модуль FIND_SIMILAR_CONF предназначен для первичного поиска дубликатов при вводе информации пользователем. На основе заполненной пользователем карточки описания доклада система предлагает список конференций, наиболее подходящих под описание пользователя, с возможностью выбора одной из уже зарегистрированных конференций или создания новой.

Модуль MERGE_EQUAL осуществляет регулярный просмотр зарегистрированных в системе конференций и объединяет найденные дубликаты в автоматическом режиме.

Алгоритм поиска групп конференций

Использование данных наукометрических систем позволяет создать поисковые механизмы для объединения проводимых в разные годы конференций в группы. Например, конференции «VIII Международная конференция ЗНАНИЯ—ОНТОЛОГИИ—ТЕОРИИ» и «V Международная конференция ЗНАНИЯ—ОНТОЛОГИИ—ТЕОРИИ» проводились в разные годы и являются разными конференциями. Однако они имеют одинаковое тематическое направление, сильно пересекающиеся список участников и состав программного комитета, и, следовательно, принадлежат одной группе. Как следствие, эти конференции должны объединяться при осуществлении тематического поиска конференций [12] и учитываться при коррекции результатов разрешения имен авторов докладов на конференции [13]. Поскольку названия конференций при регистрации в наукометрических системах могут отличаться не только цифрами, задача построения таких групп также должна решаться с использованием нечеткого поиска по названиям, датам проведения и местам. Основными отличаями этого алгоритма от алгоритма слияния дубликатов являются использование сравнения дат без учета года проведения, отсутствие нечеткого поиска по месяцу проведения и дополнительный учет связей уже построенных ранее групп.

Программная реализация алгоритма выполнена на языке PL/SQL в виде программного модуля CREATE_GROUP, осуществляющего поиск и объединение в группы по всему множеству конференций.

Апробация программных реализаций алгоритмов проводилась на данных **информационно-аналитической системы (ИАС) «ИСТИНА»** [1]. Для реализации интерфейса использовался фрэймворк Django. На странице с описанием конференции пользователь может перейти по ссылке «Все конференции этой группы» (рис. 2).

После перехода открывается страница с описанием группы конференции. Название группы совпадает с названием самой последней по дате конференции этой группы. Список конференций в группе отсортирован по датам проведения (рис. 3), и имеется возможность перейти по ссылке к описанию любой проводимой ранее конференции этой группы. Дальнейшая детализация возможна с использованием встроенного в ИАС «ИСТИНА» механизма построения отчетов SQLREPORTS [14].

Следует отметить, что создание удобного средства визуализации не является основной целью разработки представленных программных модулей. Результаты работы описанных алгоритмов могут использоваться для увеличения точности определения уровня (степени) авторитетности конференций, тематического анализа данных и ранжирования результатов научной деятельности.

Результатом работы программной реализации является автоматическое слияние конференций,

Рис. 2. Страница с описанием конференции

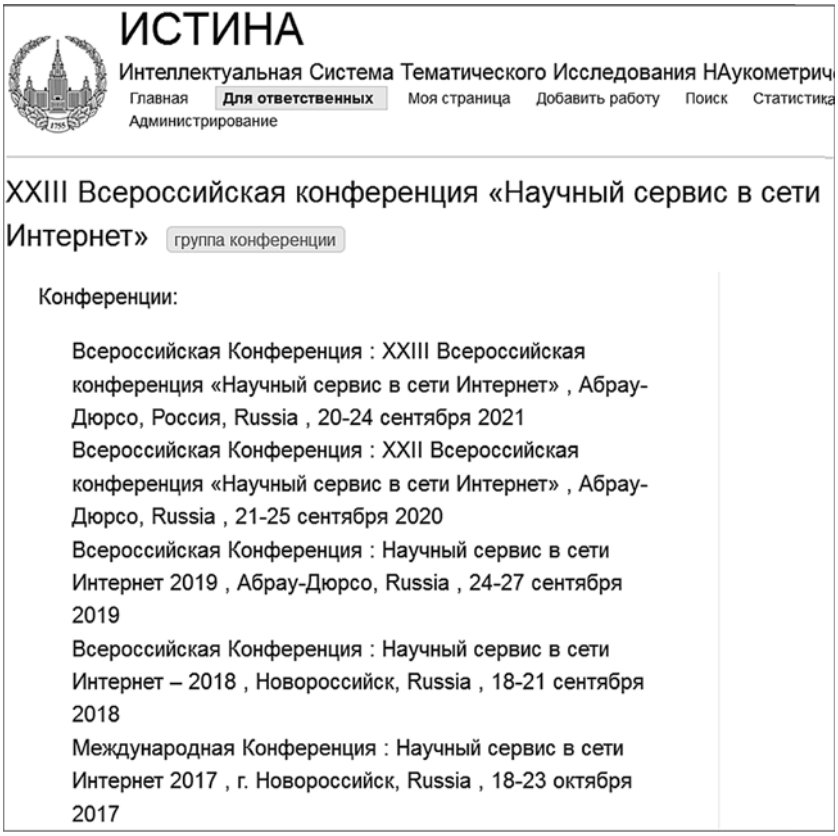


Рис. 3. Интерфейс с описанием группы конференций

распознанных как дубликаты, и объединение похожих конференций в группы. Из 144 тыс. конференций было выделено 14 тыс. групп. В указанных группах выявлено около 10 тыс. пар, которые являются кандидатами на слияние как дубликаты. Тестирование результатов проводилось по следующей методике. Для 100 случайно выбранных пар, входящих в одну группу конференций, была проведена экспертная оценка того, действительно ли каждая пара принадлежит одной группе конференций. Точность метода оценивалась как процент правильно распознанных пар и составила 91 %. Ошибки распознавания обусловлены наличием конференций с близкими названиями, например «II Международная научно-практическая конференция Актуальные проблемы коммуникации. Язык и перевод» и «XIII Международная научная конференция по актуальным проблемам теории языка и коммуникации. Язык, коммуникация, перевод». Эти конференции близки по тематике, но не принадлежат одной группе, поскольку имеют разных организаторов.

Заключение

Представленные в работе алгоритмы позволяют проводить первичный поиск похожих конференций при вводе информации о докладе в наукометрическую систему, автоматический поиск дубликатов среди уже зарегистрированных в системе конференций, а также объединение конференций в группы. Для удобства использования разработан специальный интерфейс, позволяющий осуществлять визуализацию связей между конференциями.

Использование представленных алгоритмов позволяет уменьшить число дубликатов, повысить качество исходных данных для проведения наукометрического анализа информации и проведения оценки авторитетности конференций.

Список литературы

1. **Интеллектуальная** система тематического исследования научно-технической информации (ИСТИНА) / Под ред. В. А. Садовничего. М.: Изд-во МГУ, 2014. 262 с.

2. **Васенин В. А., Зензинов А. А., Лунев К. В.** Использование наукометрических информационно-аналитических систем для автоматизации проведения конкурсных процедур на примере информационно-аналитической системы ИСТИНА // Программная инженерия. 2016. Том 7, № 10. С. 472—480. DOI: 10.17587/prin.7.472-480.

3. **Садовничий В. А., Васенин В. А., Афонин С. А.** и др. Информационная система «ИСТИНА» как big data — инструментарий в области управления на основе анализа наукометрических данных// Знания — Онтологии — Теории (ЗОНТ-2015). Материалы Всероссийской конференции с международным участием. Новосибирск. 2015. С. 115—123.

4. **Завражин А. В., Карманов М. В., Шубина И. В.** Наукометрия: панацея или беда?// Право и образование. 2022. № 9. С. 4—11.

5. **Полянин А. Д.** Недостатки индексов цитируемости и Хирша. Индексы максимальной цитируемости. URL: http://eqworld.ipmnet.ru/ru/info/sci-edu/Polyanin_IndexH_2014.html (дата обращения 25.01.2023).

6. **Жарова Е. Н.** Наукометрия в области социогуманитарных наук: проблемы и пути их решения // Научные и технические библиотеки. 2022. № 4. С. 34—53. DOI: 10.33186/1027-3689-2022-4-34-53.

7. **Козицын А. С., Афонин С. А., Шачнев Д. А.** Метод оценки тематической близости научных журналов // Программная инженерия. 2020. Том 11, № 6. С. 335—341. DOI: 10.17587/prin.11.335-341.

8. **Козицын А. С.** Алгоритмы тематического поиска данных в наукометрических системах // Программная инженерия. 2022. Том 13, № 6. С. 291—300. DOI: 10.17587/prin.13.291-300.

9. **Shachnev D. A.** Searching for activity results and experts in a given subject area, taking results significance into account // Программная инженерия. 2021. Том 12, № 5. С. 260—266. DOI: 10.17587/prin.12.260-266.

10. **Козицын А. С., Афонин С. А.** Метод поиска экспертов по данным наукометрических систем // Электронные библиотеки. 2021. Том 24, № 5. С. 870—888. DOI: 10.26907/1562-5419-2021-24-5-879-888.

11. **Маннинг К., Рагхаван П., Шютце Ч.** Введение в информационный поиск. М.: Вильямс, 2011. 520 с.

12. **Козицын А. С., Афонин С. А., Шачнев Д. А.** Методы тематического поиска конференций по наукометрическим данным // Научный сервис в сети Интернет. 2022. № 24. С. 332—339. DOI: 10.20948/abrau-2022-3.

13. **Козицын А. С., Афонин С. А.** Алгоритм разрешения неоднозначности имен авторов в ИАС ИСТИНА // Современные информационные технологии и ИТ-образование. 2020. Том 16, № 1. С. 108—117. DOI: 10.25559/SITITO.16.202001.108-117.

14. **Afonin S., Kozitsyn A., Astapov I.** Sqlreports: Yet another relational database reporting system // Proceedings of the 9th International Conference on Software Engineering and Applications. 2014. P. 529—534. DOI:10.5220/0005114205290534.

Algorithms for Finding Duplicate Conferences and Conference Groups in Scientometric Systems

A. S. Kozitsyn, Researcher, alexanderkz@mail.ru,
Lomonosov Moscow State University, Moscow, 119192, Russian Federation

Corresponding author:

Alexander S. Kozitsyn, Researcher,
Lomonosov Moscow State University, Moscow, 119192, Russian Federation
E-mail: alexanderkz@mail.ru

Received on January 19, 2023
Accepted on February 07, 2023

The article discusses the methods developed by the author for assessing the proximity of the description of conferences in order to detect duplicates, as well as building groups of conferences. An overview of the existing catalogs of conferences on the Internet is given. Their advantages and disadvantages are analyzed. The necessity of developing methods for more thorough verification of input data about conferences is substantiated. The description of the algorithms developed by the author and their software implementation and testing on big data is given on the example of the scientometric system IAS ISTINA. The developed algorithms make it possible to search for similar conferences by primary descriptions when registering a conference, search for duplicates in the database of the scientometric system, and combine conferences of different years into groups. The described methods can be used in the development of conference catalogs and scientometric systems to improve the quality of initial data verification.

Keywords: *duplicates search, scientometrics, information systems, conference*

For citation:

Kozitsyn A. S. Algorithms for Finding Duplicate Conferences and Conference Groups in Scientometric Systems, *Programmnaya ingeneria,* 2023, vol. 14, no. 4, pp. 195—202. DOI: 10.17587/prin.14.195-202.

References

1. **Intelligent** system of case study of scientific and technical information (ISTINA) / Eds. V. A. Sadovnichy, Moscow, Moscow University Press, 2014, 262 p. (in Russian).

2. **Vasenin V. A., Zenzinov A. A., Lunev K. V.** Using scientometric information-analytical systems to automate competitive procedures using the example of the information-analytical system ISTINA, *Programmnaya ingeneria* 2016, vol. 7, no. 10, pp. 472—480. DOI: 10.17587/prin.7.472-480 (in Russian).

3. **Sadovnichy V. A., Vasenin V. A., Afonin S. A.** et al. Information system «ISTINA» as big data — a tool in the field of control based on the analysis of scientometric data, *Knowledge — Ontologies — Theories (ZONT-2015), Materials of the All-Russian Conference with international participation*, Novosibirsk, 2015, pp. 115—123 (in Russian).

4. **Zavrazhin A. V., Karmanov M. V., Shubina I. V.** Scientometrics: salvation or death?, *Pravo i obrazovanie*, 2022, no. 9, pp. 4—11 (in Russian).

5. **Polianin A. D.** Disadvantages of citation indexes and Hirsch. Maximum Citation Indices, available at: http://eqworld.ipmnet.ru/ru/info/sci-edu/Polyanin_IndexH_2014.html (date of access 25.01.2023).

6. **Zharova E. N.** Scientometrics in the field of social and humanitarian sciences: problems and solutions, *Nauchnye i tekhnicheskie biblioteki*, 2022, no. 4, pp. 34—53. DOI: 10.33186/1027-3689-2022-4-34-53 (in Russian).

7. **Kozitsyn A. S., Afonin S. A., Shachnev D. A.** Metod otsenki tematicheskoi blizosti nauchnykh zhurnalov, *Programmnaya ingeneria*, 2020, vol. 11, no. 6, pp. 335—341. DOI: 10.17587/prin.11.335-341 (in Russian).

8. **Kozitsyn A. S.** Algorithms for thematic data search in scientometric systems, *Programmnaya ingeneria*, 2022, vol. 13, no. 6, pp. 291—300. DOI: 10.17587/prin.13.291-300 (in Russian).

9. **Shachnev D. A.** Searching for activity results and experts in a given subject area, taking results significance into account, *Programmnaya ingeneria* 2021, vol. 12, no. 5, pp. 260—266. DOI: 10.17587/prin.12.260-266.

10. **Kozitsyn A. S., Afonin S. A.** Expert search method based on scientometric systems data, *Elektronnye biblioteki,* 2021, vol. 24, no. 5, pp. 870—888. DOI: 10.26907/1562-5419-2021-24-5-879-888 (in Russian).

11. **Manning K.,Ragkhavan P., Shiuttse C.** *Introduction to information retrieval*, Moscow, Williams, 2011, 520 p. (in Russian).

12. **Kozitsyn A. S., Afonin S. A., Shachnev D. A.** Methods for thematic search of conferences based on scientometric data, *Nauchnyi servis v seti Internet*, 2022, no. 24, pp. 332-339. DOI: 10.20948/abrau-2022-3 (in Russian).

13. **Kozitsyn A. S., Afonin S. A.** Algorithm for resolving the ambiguity of the names of authors in the IAS ISTINA, *Sovremennye informatsionnye tekhnologii i IT-obrazovanie*, 2020, vol. 16, no. 1, pp. 108—117. DOI: 10.25559/SITITO.16.202001.108-117 (in Russian).

14. **Afonin S., Kozitsyn A., Astapov I.** Sqlreports: Yet another relational database reporting system, *Proceedings of the 9th International Conference on Software Engineering and Applications*, 2014, pp. 529—534. DOI: 10.5220/0005114205290534.

Институт Математики им. С. Л. Соболева СО РАН
Новосибирский национальный исследовательский государственный университет
Математический центр в Академгородке
Российская Ассоциация Искусственного Интеллекта
Российская Инженерная Академия
Institute of Electrical and Electronics Engineers Siberian Section (IEEE Siberian Section)

IX Международная конференция

ЗНАНИЯ — ОНТОЛОГИИ — ТЕОРИИ

2—6 октября 2023 г., Новосибирск

2—6 октября 2023 г. в Новосибирске состоится IX Международная конференция «Знания — Онтологии — Теории» (ЗОНТ-2023). Целью конференции является ознакомление с новейшими научными достижениями, обмен знаниями и передовым опытом в области математических методов представления и анализа данных, извлечения знаний и построения теорий предметных областей, анализа формальных понятий и извлечения информации из текстов естественного языка. Сборник трудов конференции будет проиндексирован в РИНЦ, избранные статьи будут проиндексированы в Scopus.

ТЕМАТИКА КОНФЕРЕНЦИИ

Обнаружение закономерностей и извлечение знаний, скрытых в структурированных и неструктурированных данных. Машинное обучение. Распознавание образов, анализ данных. Прогнозирование. Индуктивный вывод.

Систематизация знаний. Инженерия знаний. Управление знаниями. Извлечение знаний из текстов на естественном языке. Разработка онтологий предметных областей, технологии создания и применения онтологий.

Построение теорий предметных областей. Разработка семантических и онтологических моделей предметных областей. Анализ формальных понятий. Логическая семантика естественного языка. Нечеткие логики.

Работа конференции планируется в виде пленарных, секционных и стендовых докладов и круглых столов, рабочие языки — русский и английский. Конференция поддержана Математическим Центром в Академгородке, соглашение с Министерством науки и высшего образования Российской Федерации № 075-15-2022-282.

Контактные данные для переписки: zont@math.nsc.ru

16 июня 2023 г. в Санкт-Петербургском государственном электротехническом университете «ЛЭТИ» им. В. И. Ульянова (Ленина) состоится

IV Международная конференция по нейронным сетям и нейротехнологиям

(NeuroNT'2023)

ОРГАНИЗАТОРЫ КОНФЕРЕНЦИИ

- Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)
- Павловский центр «Интегративная физиология – медицине, высокотехнологичному здравоохранению и технологиям стрессоустойчивости»
- Институт физиологии им. И. П. Павлова Российской академии наук (ИФ РАН)
- Российская Северо-Западная секция Международного института инженеров электротехники и электроники IEEE

ОСНОВНЫЕ ТЕМЫ КОНФЕРЕНЦИИ

- Математические основы построения искусственного интеллекта
- Технологии искусственного интеллекта
- Сильный искусственный интеллект
- Гибридный интеллект
- Искусственные нейронные сети
- Нейроморфные вычисления и технологии
- Этика и безопасность применения искусственного интеллекта
- Прикладные системы с искусственным интеллектом
- Аппаратное обеспечение систем искусственного интеллекта

СЕКЦИИ

- Концепции построения гибридного интеллекта
- Технологии искусственного интеллекта и их приложения
- Управление данными и организация вычислений в интеллектуальных системах
- Рабочие языки конференции – русский, английский

Контактная информация
https://neuront.etu.ru/2023/ru/
E-mail: IRVC.eltech@mail.ru Тел.: +7 812 346-46-37

ООО "Издательство "Новые технологии". 107076, Москва, ул. Матросская Тишина, д. 23, стр. 2
Технический редактор *Е. В. Конова*. Корректор *А. В. Чугунова*.

Сдано в набор 28.02.2023 г. Подписано в печать 03.04.2023 г. Формат 60×88 1/8. Заказ PI423
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1. Сайт: **www.aov.ru**

Рисунки к статье **Д. В. Ефанова, Т. С. Погодиной**

«АНАЛИЗ ЭФФЕКТИВНОСТИ СХЕМ ВСТРОЕННОГО КОНТРОЛЯ НА ОСНОВЕ ОЦЕНКИ ПРИНАДЛЕЖНОСТИ ВЫЧИСЛЯЕМЫХ ФУНКЦИЙ КЛАССУ САМОДВОЙСТВЕННЫХ И ПРЕДВАРИТЕЛЬНОГО СЖАТИЯ СИГНАЛОВ С ПРИМЕНЕНИЕМ ЛИНЕЙНЫХ КОДОВ»

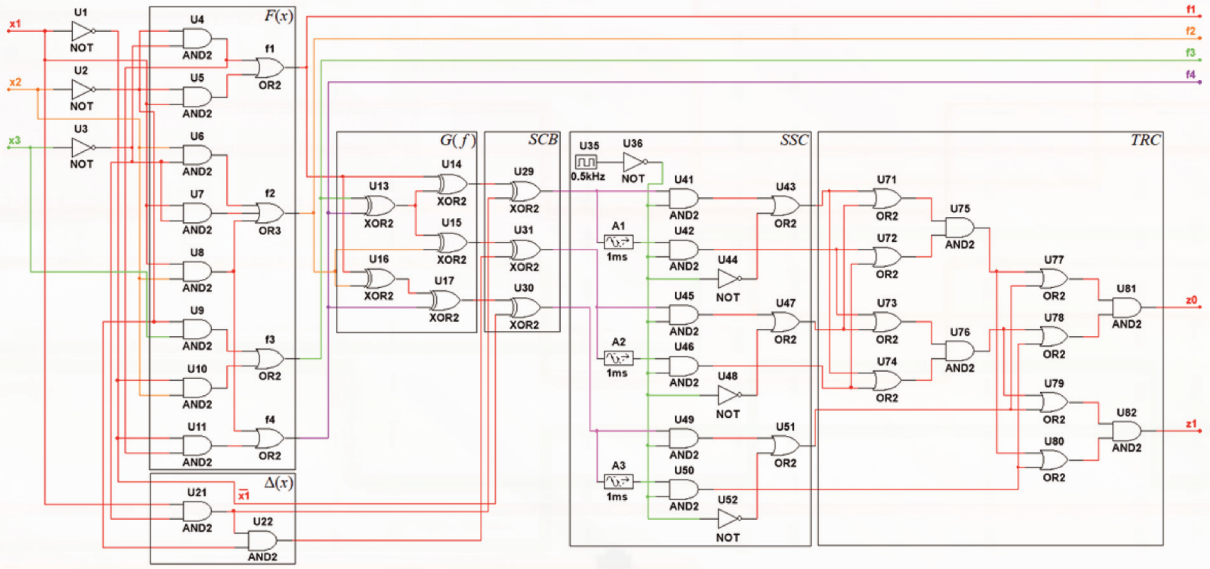
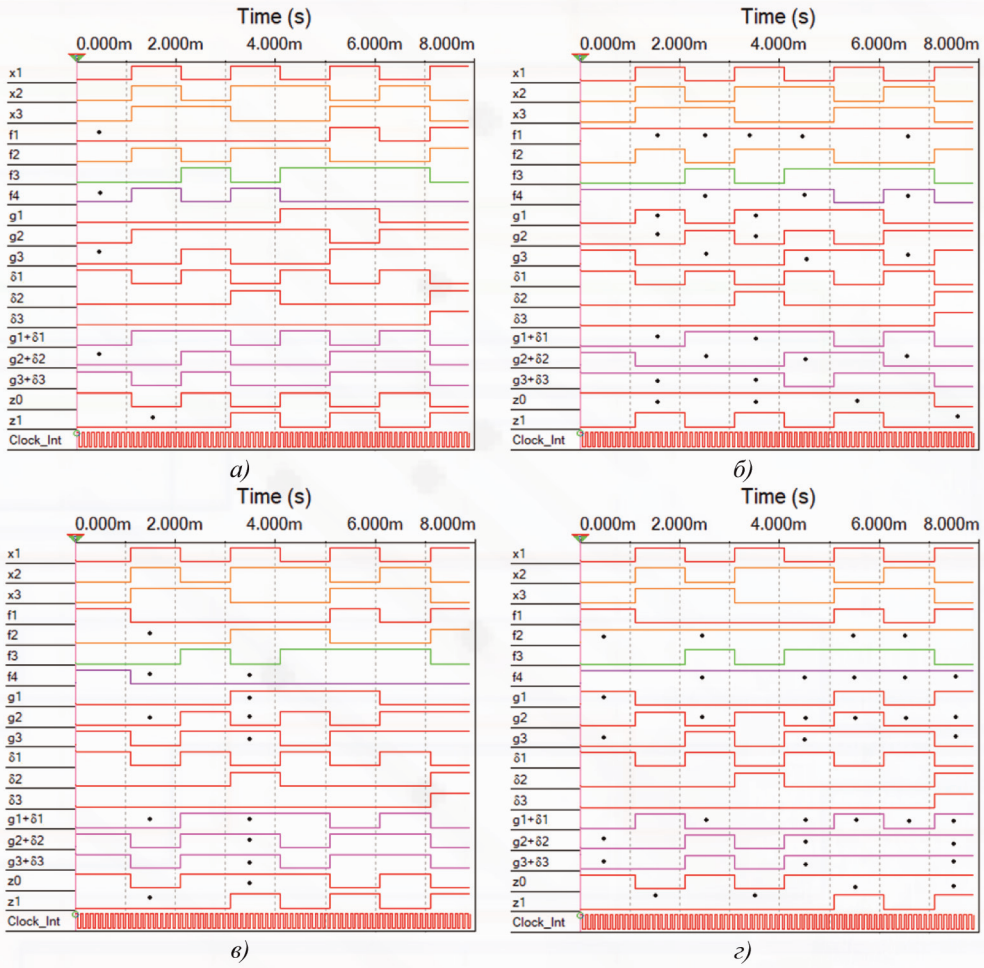


Рис. 6. Структура устройства с СВК на основе кода Хэмминга



а) *б)* *в)* *г)*

Рис. 7. Временные диаграммы работы устройства с СВК на основе кода Хэмминга при внесении неисправностей: *а* – stuck-at-0 U4; *б*– stuck-at-1 U4; *в* – stuck-at-0 U8; *г* – stuck-at-1 U8

Рисунки к статье **Д. В. Ефанова, Т. С. Погодиной**

«АНАЛИЗ ЭФФЕКТИВНОСТИ СХЕМ ВСТРОЕННОГО КОНТРОЛЯ НА ОСНОВЕ ОЦЕНКИ ПРИНАДЛЕЖНОСТИ ВЫЧИСЛЯЕМЫХ ФУНКЦИЙ КЛАССУ САМОДВОЙСТВЕННЫХ И ПРЕДВАРИТЕЛЬНОГО СЖАТИЯ СИГНАЛОВ С ПРИМЕНЕНИЕМ ЛИНЕЙНЫХ КОДОВ»

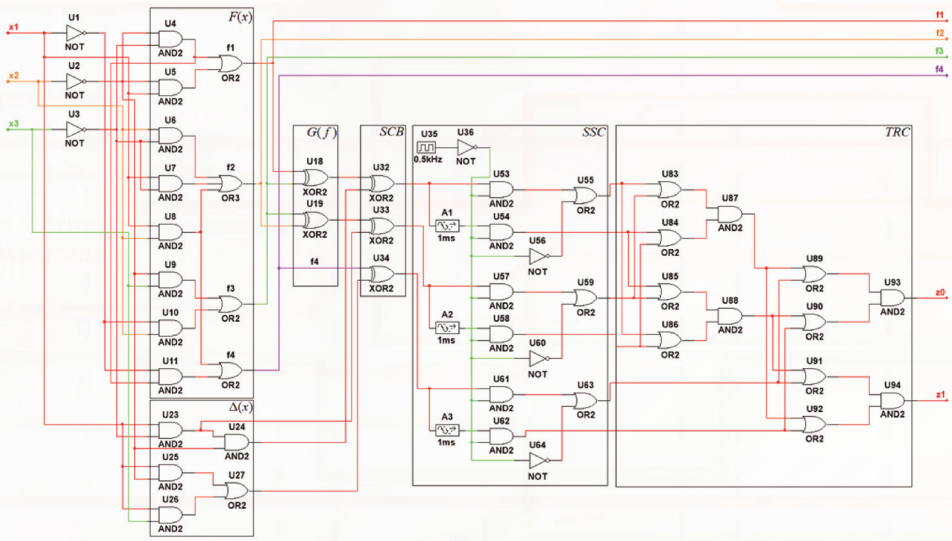


Рис. 8. Структура устройства с СВК на основе модифицированного кода Хэмминга

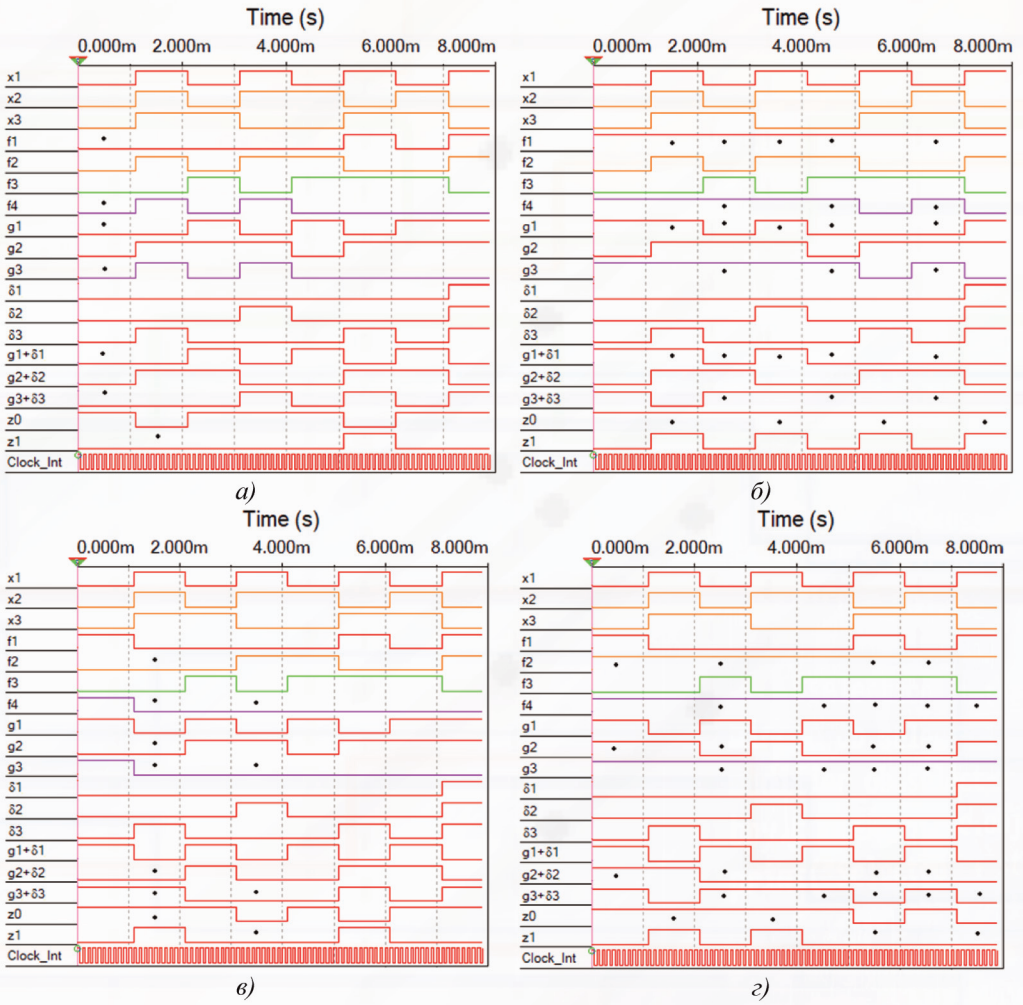


Рис. 9. Временные диаграммы работы устройства с СВК на основе модифицированного кода Хэмминга при внесении неисправностей: *а* – stuck-at-0 U4; *б*– stuck-at-1 U4; *в* – stuck-at-0 U8; *г* – stuck-at-1 U8

ISSN 2220-3397. Программная инженерия. 2023. Том 14. № 4. 153–204. Подписной индекс по Объединенному каталогу «Пресса России» – 22765