МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Мичурова Михаила Антоновича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ ПРОГРАММНЫХ СРЕДСТВ АВТОМАТИЗИРОВАННОГО ОБНАРУЖЕНИЯ СЕМАНТИЧЕСКИХ ОШИБОК ВО ФРАГМЕНТИРОВАННЫХ ПРОГРАММАХ ДЛЯ СИСТЕМЫ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /……
(ФИО) / (подпись)
«31» мая 2022 г.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./……...
(ФИО) / (подпись)
«20» мая 2022 г.

Новосибирск, 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
(название кафедры)
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)
…….
(подпись)
«21» января 2022 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у* Мичурову Михаилу Антоновичу, группы 18201
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация программных средств автоматизированного обнаружения семантических ошибок во фрагментированных программах для системы LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 20 янв 2022 № 0012

Срок сдачи студентом готовой работы 20 мая 2022 г.

Исходные данные (или цель работы):

Разработка модуля сбора трассы для системы LuNA и программного средства анализа трассы с целью обнаружения семантических ошибок

Структурные части работы:

Обзор существующих средств отладки параллельных программ, анализ семантических ошибок во фрагментированных программах и разработка требований к разрабатываемым средствам, описание реализации, тестирование

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./……...
(ФИО) / (подпись)
«20» января 2022 г.

Задание принял к исполнению
Мичуров М. А. /……..
(ФИО студента) / (подпись)
«20» января 2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка параллельных программ численного моделирования на суперЭВМ требует решения комплекса проблем, связанных с организацией параллельной обработки данных и требующих специальной квалификации и больших трудозатрат. Специалисты, в работе которых требуется выполнение численного моделирования, могут испытывать трудности с написанием эффективных параллельных программ.

Для решения этих проблем в ИВМ и МГ СО РАН разрабатывается технология фрагментированного программирования, а также экспериментальная система LuNA (Language for Numeric Algorithms) автоматического конструирования параллельных программ на мультикомпьютерах, поддерживающая данную технологию [1]. Технология фрагментированного программирования представляется перспективной и актуальной для высокопроизводительных вычислений [2].

Система LuNA позволяет автоматически конструировать и выполнять на мультикомпьютерах параллельные программы численного моделирования на основе высокоуровневой спецификации программы на одноименном языке. Однако при написании программ для системы LuNA программист может допускать семантические ошибки [3], не свойственные программам, написанным с использованием традиционных средств параллельного программирования, таких как MPI и OpenMP. Таким образом, проблема отладки LuNA-программ является актуальной.

Для упрощения отладки параллельных программ используются специальные инструментальные средства, реализующие различные подходы к обнаружению ошибок [4]. В рамках данной работы разрабатывалось средство автоматизированной отладки, реализующее подход анализа трассы.

Цель данной работы — разработка модуля сбора трассы для системы LuNA и программного средства анализа трассы с целью обнаружения семантических ошибок.

Для достижения этой цели были поставлены следующие задачи:

1. Провести обзор и анализ существующих методов отладки параллельных программ, а также программных средств, реализующих данные методы.
2. Провести анализ предметной области и выделить классы семантических ошибок, свойственных фрагментированным программам.
3. Сформулировать требования к разрабатываемому программному средству.
4. Разработать и реализовать обнаружение наиболее типичных для фрагментированных программ семантических ошибок в программном средстве.
5. Протестировать разработанное средство на ряде фрагментированных программ и оценить накладные расходы.

Научная новизна состоит в анализе семантических ошибок, свойственных фрагментированным программам, а также в реализации метода автоматизированного контроля корректности по собранной трассе применительно к фрагментированным программам.

Практическая ценность работы обусловлена тем, что использование разработанного средства может существенно упростить разработку и отладку программ для системы LuNA.

Работа изложена в четырех главах. В первой главе приводится обзор существующих средств и методов отладки параллельных программ. Вторая глава посвящена анализу предметной области и формулированию требований к разрабатываемому программному средству. В третьей главе описываются разработанные методы обнаружения семантических ошибок и их реализация. Четвертая глава посвящена тестированию разработанного программного средства и оценке производительности.

Глава 1. Обзор подходов к отладке параллельных программ

Во введении была поставлена проблема отладки LuNA-программ. В связи с этим был проведен обзор существующих средств и методов отладки параллельных программ. Целью обзора было исследование используемых средств и методов и выбор подхода, который будет реализован в разрабатываемом средстве отладки.

1.1 Ошибки в параллельных программах

Программист при написании программы может допускать различные ошибки. Ошибки можно разделить на [4]:

- синтаксические;
- семантические.

Под **синтаксическими** ошибками понимаются ошибки, обнаруживаемые компилятором при сборке программы и препятствующие генерации исполняемых файлов. **Семантические** же ошибки не препятствуют сборке программы, однако влияют на корректность результата выполнения. Далее под ошибками в параллельных программах будут пониматься именно семантические ошибки.

При написании параллельной программы программист может допускать специфические ошибки, не свойственные последовательным программам.

Ошибки в параллельных программах могут быть связаны с используемой технологией, реализующей параллельные вычисления. Так, например, при использовании MPI программист может указать различные типы данных при отправке и приеме MPI-сообщений [4].

Также ошибки могут быть связаны с параллелизмом как таковым, и не зависеть от конкретной реализации параллельного взаимодействия в программе. К таким ошибкам, например, можно отнести мертвые блокировки [5].

Таким образом, можно выделить следующие классы ошибок в параллельных программах:

- ошибки, связанные с некорректным использованием выбранной технологии параллельного программирования;
- ошибки, связанные с параллелизмом как таковым [6]:
    - бесконечное ожидание;
    - гонки данных (ошибки соревнования).

Бесконечное ожидание — ситуация, в которой вычисления не могут быть продолжены по причине ожидания события, которое никогда не наступит. В частности, к бесконечному ожиданию приводят дедлоки, или мертвые блокировки.

Гонки данных (ошибки соревнования) происходят тогда, когда от порядка выполнения операций зависит результат работы программы.

Приведенные классы ошибок не являются непересекающимися. Напротив, зачастую именно некорректное использование средств параллельного программирования может приводить к мертвым блокировкам и гонкам данных. Например, при использовании OpenMP отсутствие директивы reduction для переменной, изменяемой в цикле, может привести к гонке данных [7, 8] и вычислению некорректного значения этой переменной. При использовании MPI два процесса могут для обмена данными друг с другом вызывать сначала MPI_Send, а затем MPI_Recv, и, в зависимости от реализации MPI (если эти операции реализованы как блокирующие), это может приводить в мертвой блокировке [9].

Тому, как реализуются связанные с параллелизмом семантические ошибки в системе LuNA, будет посвящен пункт 2.2 главы 2.

1.2 Подходы к отладке параллельных программ

1.2.1 Проблемы отладки параллельных программ

Отладка параллельных программ может быть осложнена следующими факторами:

- “эффект наблюдателя”;
- недетерминированность выполнения программы;

- другие факторы.

“Эффект наблюдателя” заключается в том, что сама попытка наблюдения за ходом работы параллельной программы может влиять на результат ее работы. Так, например, при попытке обнаружить гонку данных с помощью выполнения по шагам, можно полностью ее избежать, так как действия параллельно взаимодействующих потоков будут разделены во времени [9].

Недетерминированность выполнения выражается в том, что параллельная программа на одних и тех же входных данных может вести себя по-разному, причем возможна ситуация, когда подавляющее большинство запусков будут оканчиваться вычислением верного результата. В частности, это может препятствовать обнаружению гонок данных.

1.2.2 Подходы к отладке

При разработке средств отладки параллельных программ используются различные подходы [10]:

- верификация модели программы;
- автоматизированный анализ корректности;
- диалоговая отладка;
- сравнительная отладка.

Эти подходы различаются как требованиями к квалификации программиста и его времени, так и наборами ошибок, в выявлении и исправлении которых они способны помочь.

1.2.3 Верификация модели программы

Метод верификации модели связан с формальной проверкой выполнения на модели свойств поведения, описываемых на языке формальной логики. Реальная система представляется моделью ее вычислений — системой переходов с конечным числом состояний. Проверка модели таким образом сводится к исчерпывающему анализу всего пространства состояний модели системы. Такой процесс может быть полностью автоматизирован [11].

К инструментальным средствам, реализующим подход верификации модели применительно к параллельным приложениям, можно отнести Spin [12, 13], а также его расширение, поддерживающее анализ MPI-программ — MPI-Spin [14]. Модель параллельной программы описывается на специальном языке Promela (PROcess MEta LAnguage). MPI-Spin предоставляет расширения языка Promela, упрощающие моделирование MPI-программ.

При использовании такого подхода программист вынужден помимо непосредственно параллельной программы описывать еще и ее модель, что требует дополнительной квалификации и времени. Кроме того, формальная модель программы не всегда учитывает все нюансы, и, как следствие, при ее анализе некоторые ошибки не могут быть обнаружены [4].

1.2.4 Автоматизированный анализ корректности

Автоматизированный анализ корректности подразумевает автоматическое выявление ошибочного поведения параллельной программы. Идея состоит в том, чтобы собирать информацию о различных событиях, происходящих во время работы программы, и на основании анализа этой информации делать вывод о наличии или отсутствии тех или иных ошибок.

Автоматизированный анализ корректности можно разделить на два различных подхода:

- анализ во время выполнения программы;
- анализ по трассе, или “посмертный анализ”.

К средствам, выполняющим анализ во время работы программы, можно отнести, например, Intel Trace Analyzer and Collector [15], MARMOT [16] и MPI-CHECK [17]. Анализ во время работы программы удобен тем, что информация об ошибках собирается за один запуск, а программисту не нужно дополнительно описывать модель программы и ждать ее анализа программным средством.

Подход анализа по трассе реализуют Intel Message Checker [18, 19], Distributed Virtual Machine [20]. При использовании данного подхода

информация о событиях не анализируется сразу, а записывается в файлы трассы для дальнейшего анализа отдельным программным средством.

Так, например, в DVM [20] данный подход реализован следующим образом. При выполнении программы пользователя трассировщик накапливает информацию об обращениях к функциям и параметрах этих обращений, о созданных задачей коммуникаторах, типах данных, стеках адресов возврата из процедур и т. д. Кроме этого при сборе трассы осуществляется динамический анализ обращений к MPI-функциям и выдача сообщений об ошибках, которые могут помешать работе трассировщика. Например, во всех функциях, имеющих параметром коммуникатор, проверяется его существование. По завершении работы задачи в рабочей директории создаются файлы с трассировкой обращений к MPI-функциям (по одному на каждый MPI-процесс), а также файл с описаниями трассировщика, содержащий общие для всех процессов описания коммуникаторов, описания ссылок на исходные тексты программы пользователя, значения констант и переменных MPI и другую информацию. В результате работы анализатора пользователю предоставляется довольно большой объем информации о работе программы, в том числе сообщения об обнаруженных ошибках. Для ошибок помимо прочего выводятся имя файла и номер строки точки, в которой произошла ошибка, фрагмент исходного текста программы, соответствующий этой точке, а также стек обращений к функциям для нее.

К недостаткам посмертного анализа можно отнести большие размеры файлов трассы при большом числе процессов и записываемых событий, и, как следствие, большое время анализа. Однако, с точки зрения разработки программного средства, реализующего посмертный анализ, такой подход имеет ряд преимуществ.

Во-первых, для сбора трассы требуются минимальные модификации в библиотеке/системе, реализующей параллельное взаимодействие. Часть

функциональности, отвечающая за сбор трассы, может быть относительно изолирована.

Во-вторых, отсутствие необходимости получения и анализа данных во время работы параллельной программы предоставляет большую свободу в выборе языков программирования и технологий, используемых при реализации средства анализа трассы.

1.2.5 Диалоговая отладка

Программные средства, ориентированные на диалоговую отладку параллельных программ, предоставляют как обычные для диалоговых отладчиков возможности (просмотр значения переменных, пошаговое выполнение, установка контрольных точек), так и дополнительные возможности, ориентированные на работу с параллельными программами: контроль над группами потоков и процессов.

К средствам диалоговой отладки, например, относятся TotalView [21] и PD [22]. Средства диалоговой отладки предоставляют пользователю доступ к большому количеству информации о работе параллельной программы, а также высокую степень контроля над выполнением. Так, например, PD позволяет отлаживать процессы и потоки программы, управлять точками прерывания и наблюдения, логически делить процессы программы на подмножества, управлять ими, изменять и просматривать переменные, а также выполнять профилирование отлаживаемой программы с использованием свободно распространяемых профилировщиков. Однако, в отличие от верификации модели и автоматического анализа корректности, диалоговая отладка требует непосредственного участия программиста, а главным их недостатком является невозможность обнаружения ошибок, связанных с недетерминированностью выполнения параллельной программы.

1.2.6 Сравнительная отладка

Идея сравнительной отладки состоит в том, чтобы проверить корректность параллельной программы, сравнив промежуточные результаты

выполнения в контрольных точках с некоторой эталонной программой. Эталонной программой может являться уже отлаженная последовательная версия той же программы, корректность которой не вызывает сомнений [23]. В результате сравнения пользователю может быть предоставлена информация о расхождениях, если таковые будут обнаружены.

Для сравнения выполнений программы может использоваться сравнительный анализ трассы [24]. В частности, при использовании подхода, описанного в [24], сначала накапливается трасса при выполнении параллельной программы, а затем выполняется ее проверка при запуске программы в последовательном режиме. С найденными отличиями пользователь может ознакомиться с помощью интерактивного визуализатора различий.

При использовании сравнительной отладки программист сталкивается с необходимостью иметь отлаженную и корректно работающую эталонную программу, однако такой программой программист обладает не всегда. Также существует проблема разрастания трассы при отладке больших параллельных программ. Для сокращения размеров трассы могут применяться методы сокращения числа контрольных точек [25], однако при таких компромиссах некоторые расхождения могут остаться незамеченными.

1.3 Выводы по обзору

Из обзора существующих средств и методов отладки параллельных приложений видно, что существует множество подходов, применимых в различных ситуациях, и имеющих как преимущества, так и недостатки. Методы и средства отладки различаются требованиями к квалификации и времени программиста, нагрузкой на вычислительную систему, предоставляемым уровнем обнаружения ошибок в параллельных программах.

Однако, парадигма фрагментированного программирования и поддерживающая ее система LuNA, рассматриваемые в данной работе, обладают своей спецификой, что делает применение рассмотренных средств отладки к отладке затруднительным, или вовсе неудобным. В связи с этим

актуальной является разработка узкоспециализированного средства отладки, которое реализовывало бы существующий подход к отладке, но было бы ориентировано на систему LuNA и учитывало ее специфику для улучшения пользовательского опыта.

Глава 2. Проектирование средств автоматизированного обнаружения семантических ошибок для программ в системе LuNA

2.1 Технология фрагментированного программирования

Одной из главных проблем при реализации больших численных моделей является тот факт, что написание эффективных параллельных программ требует дополнительных знаний и навыков наряду с умением реализовывать численный алгоритм в определенном языке программирования [26].

Для преодоления этой проблемы в ИВМ и МГ СО РАН разрабатывается технология фрагментированного программирования (ТФП) [1]. В основе этого подхода лежит представление прикладного алгоритма в виде не более чем счетного множества фрагментов вычислений (ФВ), выполняющих роль вычислительных задач в ТФП. **Фрагмент вычислений** — это триплет (*in*, *out*, *op*), где *in* и *out* — конечные множества входных и выходных объектов данных соответственно, называемых **фрагментами данных** (ФД), а *op* — последовательная операция, вычисляющая значения ФД из множества *out* по значениям ФД из множества *in* (рисунок 1).

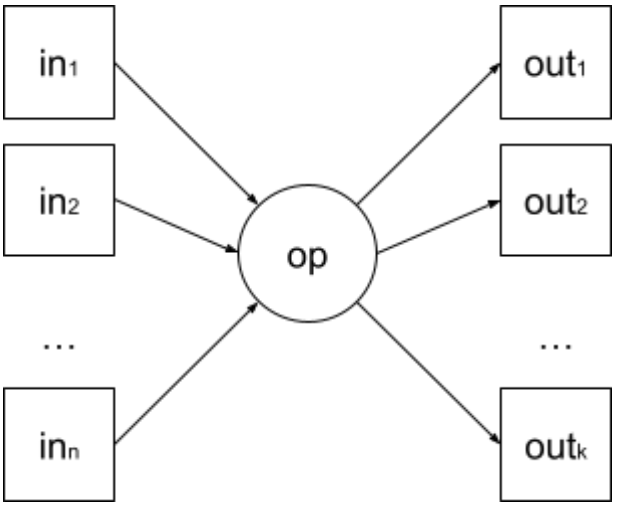

Рисунок 1 — Фрагмент вычислений

Такое представление прикладного алгоритма называется фрагментированным алгоритмом (ФА). Исполнение фрагментированного алгоритма описывается следующим базовым алгоритмом:

- фрагмент вычислений исполняется, если получили значения все его входные фрагменты данных;
- после выполнения фрагмента вычислений получают значения все его выходные фрагменты данных;
- выполнение фрагментированного алгоритма заканчивается, когда оказываются вычислены все фрагменты вычислений.

Операция *op*, называемая также *кодом ФВ*, реализуется последовательной процедурой и не имеет побочных эффектов. Это позволяет выполнять ФВ на любом вычислительном узле мультикомпьютера по выбору системы, в том числе динамически перераспределять ФВ по узлам.

Фрагменты данных в ТФП являются объектами единственного присваивания. Система может дублировать ФД, передавать их по сети или сохранять на диск.

Благодаря описанным выше свойствам ФВ и ФД система может автоматически распределять ФВ и динамически изменять распределение ФВ по узлам мультикомпьютера, выбирать порядок выполнения ФВ в рамках имеющихся зависимостей по данным (информационных зависимостей) с целью обеспечения нефункциональных свойств. Так, например, система, выполняющая ФА, может осуществлять перераспределение ФВ с целью балансировки вычислительной нагрузки на узлах и сокращения времени работы, выполнять копирование ФД с целью повышения доступности данных, сохранять контрольные точки выполнения.

Для расширения класса эффективно исполняемых алгоритмов в ТФП имеется возможность влиять на ход выполнения ФА с помощью т. н. *рекомендаций* — дополнительной информации об алгоритме и способе его отображения на ресурсы мультикомпьютера [1].

Рекомендации главным образом применяются для контроля над следующими аспектами выполнения:

- распределение ФД и ФВ по узлам мультикомпьютера;
- порядок выполнения ФВ;
- сборка мусора.

В ТФП рекомендации не оказывают влияния на вычисляемые значения, но влияют на нефункциональные свойства выполнения ФА.

Использование ТФП позволяет при разработке численной модели ограничиться написанием последовательного кода — процедур, реализующих код ФВ, а также описанием вычислительного процесса на высокоуровневом предметно-ориентированном языке, полностью абстрагировавшись от реализации параллельного взаимодействия с использованием низкоуровневых библиотек таких, как MPI, OpenMP, POSIX Threads и пр.

2.2 Система LuNA

Экспериментальная система автоматического конструирования параллельных программ на мультикомпьютерах LuNA (от Language for Numerical Algorithms) была разработана для поддержки ТФП [1, 26]. Входящий в ее состав одноименный язык позволяет описывать ФА как множество ФВ и ФД, а также использовать рекомендации для контроля над нефункциональными свойствами программы [27]. Операции в системе LuNA реализуются фрагментами кода (ФК). Фрагмент кода может быть как атомарным, то есть являться последовательной процедурой на языке программирования C++, использующей предоставляемый системой программного интерфейса для работы с фрагментами данных, так и структурированным, то есть реализовываться процедурой на языке LuNA. Система транслирует LuNA-программы во внутреннее представление, которое затем исполняется на мультикомпьютере исполнительной системой, причем во время исполнения фрагментированная структура программы сохраняется [26].

Рассмотрим простейший пример LuNA-программы, представленный на рисунке 2 (подробное описание языка LuNA приводится в [28]), а также ее вывод после одного из запусков на рисунке 3. Здесь *set* и *print* — атомарные фрагменты кода (последовательные процедуры на C++), *main* — структурированный.

```
1   C++ sub set(name x, string s) ${{ x = s; $}}
2
3   C++ sub print(string s) ${{ std::printf("%s\n", s); $}}
4
5   sub main() {
6     df msg;
7
8     let N = 10 {
9       for i = 1..N {
10        print(msg[i]);
11      }
12
13      for i = 1..N {
14        set(msg[i], "This is item " + i);
15      }
16    }
17  }
```

Рисунок 2 — Пример LuNA-программы

```
This is item 1
This is item 7
This is item 8
This is item 5
This is item 9
This is item 2
This is item 10
This is item 6
This is item 4
This is item 3
```

Рисунок 3 — Вывод LuNA-программы выше

Порядок выполнения ФВ, соответствующих операторам языка LuNA, не соответствует порядку этих операторов в коде LuNA-программы, что подтверждается, в частности, выводом (строка 10) программы. Это объясняется тем, что во время выполнения фрагментированная структура программы сохраняется, и вычисление ФВ происходит по мере их готовности, то есть по

мере удовлетворения их информационных зависимостей. Таким образом, при написании LuNA-программы, программист не задает *порядок вычислений*, как в императивных языках программирования, а описывает, *какие вычисления* должны быть произведены и над *какими данными*.

Эксперименты, проведенные на базе системы LuNA, демонстрируют автоматическое конструирование параллельных программ с заданными нефункциональными свойствами [2, 29].

Эксперименты показывают, что эффективность работы LuNA-программы обычно ниже эффективности аналогичной MPI-программы, что, однако, часто оказывается приемлемым благодаря простоте разработки фрагментированных программ для системы LuNA [1].

2.3 Семантические ошибки в LuNA-программах

Система LuNA позволяет абстрагироваться от реализации параллельного взаимодействия, однако это не значит, *что* при написании LuNA-программы разработчик не может допустить семантические ошибки, свойственные параллельным программам.

2.3.1 Бесконечное ожидание

Система LuNA оставляет несколько различных возможностей для создания ситуации бесконечного ожидания.

Программист может допустить ошибку при написании атомарного ФК и написать бесконечный цикл. В таком случае при выполнении ФА система зависнет. Например, в приложении А приводится полный текст LuNA-программы с бесконечным циклом в атомарном ФК *print_array* (рисунок 4):

```
1   C++ sub print_array(value x) ${{
2       // Получение размера массива
3       const auto data = reinterpret_cast<const size_t *>(x.get_data());
4       const auto size = data[0];
5
6       // Обращение к элементам массива
7       const auto array = reinterpret_cast<const double *>(data + 1);
8       for (auto i = 0; i < size; i = 1) {
9           printf("[%d] = %lf\n", i, array[i]);
10      }
11  $}}
```

Рисунок 4 — Некорректный атомарный ФК

Здесь допущена ошибка в объявлении цикла в строке 8: вместо инкремента счетчика *i += 1* или *i++* написано выражение *i = 1*, из-за чего в зависимости от значения переменной *size* программа может как отработать нормально и завершиться, так и зависнуть. Стоит отметить, что такой пример не является полностью искусственным, поскольку аналогичный подход, состоящий в использовании ФД как хранилища для пользовательской структуры данных, может использоваться и в других программах. Так, например, алгоритм блочного умножения матриц, входящий в набор тестов системы LuNA [30], реализован с использованием ФД для хранения частей (блоков) матриц как непрерывных массивов чисел с плавающей точкой, а атомарные ФК, реализующие операции над блоками, используют циклы для обхода этих массивов.

Возможна ситуация, когда для некоторых ФВ во время работы программы не создаются входные данные. В таком случае программа также зависнет, поскольку будет бесконечно ждать, когда ФД, необходимые для выполнения этих ФВ, получат значения. Это может произойти при простом использовании ФД, для которого не существует инициализирующего ФВ (рисунок 5).

```
1   C++ sub set(name x, int v) ${{
2     x.setValue<int>(v);
3   $}}
4
5   C++ sub print(int v) ${{
6     printf("%d\n", v);
7   $}}
8
9   sub main() {
10    df a, b, c;
11    set(b, a * a);
12    set(c, b + 1);
13    print(c);
14  }
```

Рисунок 5 — LuNA-программа, содержащая использование неинициализированного ФД

Здесь ФД *a* не получает значения, однако используется в качестве входного для ФВ, соответствующего выражению *set(b, a * a)*, что приводит к зависанию программы при исполнении.

Однако более вероятна ситуация, когда к использованию неинициализированного ФД приводит ошибка при индексации, как, например, в программе на рисунке 6.

```
1   C++ sub set(name x, string s) ${{ x = s; $}}
2
3   C++ sub print(string s) ${{ std::printf("%s\n", s); $}}
4
5   sub main() {
6     df msg;
7
8     let N = 10 {
9       for i = 0..N+3 {
10        print(msg[i]);
11      }
12
13      for i = 0..N-1 {
14        set(msg[i], "This is item " + i);
15      }
16    }
17  }
```

Рисунок 6 — LuNA-программа с ошибкой при индексации

Здесь получают значения ФД *msg[i]* для i от 0 до N-1 включительно (строки 13-14), однако используются в качестве входных *msg[i]* для i от 0 до N+3 включительно (строки 9-10). При выполнении такой программы система также зависнет в ожидании инициализации ФД, однако в данном случае принципиальная разница в том, что *формально* существует выражение, отвечающее за инициализацию отсутствующих данных (а именно *set(msg[i], "This is item " + i)*), хотя во время работы инициализирующие ФВ не выполняются.

Кроме того, в системе LuNA возможно возникновение классической мертвой блокировки в смысле цикла в графе ожидающих задач [8]. На рисунке 7 приводится пример программы, при выполнении которой в графе информационных зависимостей между ФВ образуется цикл, приводящий к мертвой блокировке и зависанию системы.

```
1   C++ sub set(name x, int v) ${{ x = v + 1; $}}
2
3   sub main() {
4     df x, y, z;
5     set(y, x);
6     set(z, y);
7     set(x, z);
8   }
```

Рисунок 7 — LuNA-программа с циклическими зависимостями по данным

В данном примере существуют выражения, соответствующие инициализации всех необходимых ФД, однако из-за взаимной зависимости инициализация не может быть выполнена.

Таким образом, хотя система LuNA и предоставляет высокий уровень абстракции и берет конструирование параллельной программы на себя, программист все равно может столкнуться с ошибками, свойственными параллельным программам, написанным вручную.

2.3.2 Ошибки соревнования

Язык LuNA, следуя принципам ТФП, допускает лишь единственное присваивание значения ФД. В исполнительной системе единственность присваивания обеспечивается тем, что при попытке сохранения нового значения для ФД, значение которого уже существует на узле, будет выброшено исключение, и работа программы будет аварийно завершена.

Однако, возможны ситуации, когда исполнительная система не сможет обнаружить повторное присваивание значения ФД. Проще всего такую ситуацию можно создать с использованием рекомендаций. Как было отмечено ранее, рекомендации могут использоваться для сборки мусора. На рисунке 8 приведен пример программы, использующей рекомендацию *delete* для удаления более ненужных ФД.

```
1   C++ sub init(name x) ${{ x = 0; $}}
2   C++ sub compute(name x, int v) ${{ x = v + 1; $}}
3   C++ sub save(int z) ${{ /* ... */ $}}
4
5   sub main() {
6       df x, y, z;
7
8       init(x);
9       compute(y, x) @ {
10          delete x;
11      };
12      compute(z, y) @ {
13          delete y;
14      };
15      save(z) @ {
16          delete z;
17      };
18  }
```

Рисунок 8 — LuNA-программа, содержащая использование рекомендации delete

Использование этой рекомендации позволяет сообщить исполнительной системе о том, что после выполнения ФВ, которому она соответствует, указанный ФД можно удалить.

Однако использование рекомендации *delete* может привести к нежелательным последствиям. Рассмотрим фрагмент программы на рисунке 9, полный текст которой приводится в приложении Б.

```
 9   sub main() {
10       df x, delay;
11
12       set_delays_random(delay[1], delay[2], 1, 2);
13
14       set_after_delay(x, 0, delay[1]);
15       set_after_delay(x, 1, delay[2]);
16
17       print(x) @ {
18           delete x;
19       };
20   }
```

Рисунок 9 — Фрагмент LuNA-программы, содержащей ошибку гонки данных

Здесь ФД *x* инициализируется с помощью процедуры *set_after_delay* дважды значениями 0 и 1 (строки 14 и 15). Эта процедура перед инициализацией переданным значением ожидает число секунд, переданное третьим аргументом, в данном случае — *delay[1]* и *delay[2]*. Сами *delay[1]* и *delay[2]* инициализируются либо значениями 1 и 2, либо 2 и 1, в процедуре *set_delays_random* (строка 12), которая, в соответствии с названием, выбирает вариант присваивания случайно (код приводится в приложении Б). Таким образом, при запуске программы *x* будет инициализирован дважды с интервалом в одну секунду значениями 1 и 2 в случайном порядке. Однако ФВ, соответствующий *print(x)* (строка 17), будет выполнен сразу после первой инициализации, а после его выполнения, следуя рекомендации, исполнительная система удалит значение *x*. Это позволит второй инициализации произойти без ошибок, а программе нормально завершиться. Многократные тесты показали, что от запуска к запуску выводимое процедурой *print* значение *x* действительно меняется.

В данном примере задержки перед инициализацией моделировались с помощью стандартной функции *sleep*, однако в реальных приложениях она может быть обусловлена долгими вычислениями, а инициализация одного и того же ФД может произойти, например, из-за ошибки при индексации в имени инициализируемого ФД, если индекс — вычисляемое выражение.

Таким образом, при написании LuNA-программы все же можно допустить ошибки, приводящие к гонке данных.

2.3.3 Ошибки несоответствия типов

Язык LuNA можно рассматривать как динамически типизированный, поскольку тип значения ФД определяется в момент присваивания. При этом при написании атомарных ФК программист может указывать конкретный примитивный тип аргумента, а не работать с объектом ФД, если в ФД не хранится пользовательская структура данных. Так, например, на рисунке 10 процедура *print_real* имеет единственный аргумент примитивного типа *real*, однако при вызове в качестве аргумента передается ФД *x* (строка 13). При генерации промежуточного представления система LuNA автоматически добавит код получения значения требуемого типа.

```
1   C++ sub set_int(name x, int v) ${{
2       x = v;
3   $}}
4
5   C++ sub print_real(real v) ${{
6       printf("%lf\n", v);
7   $}}
8
9   sub main() {
10      df x;
11
12      set_int(x, 42);
13      print_real(x);
14  }
```

Рисунок 10 — LuNA-программа, содержащая ошибку несоответствия типов

Однако в случае, когда требуемый тип не соответствует истинному типу ФД, будет выброшено исключение, а работа программы будет аварийно завершена. В сообщении об ошибке будет предоставлена информация о требуемом и истинном типе значения ФД, как показано на рисунке 11.

```
err> 0 ERROR:  get_real failed for type int
```

Рисунок 11 — Сообщение об ошибке несоответствия типов

2.4 Идея предлагаемого решения

Идея предлагаемого решения состоит в следующем: имея трассу выполнения фрагментированной программы в системе LuNA, можно провести ее анализ и обнаружить семантические ошибки, если такие имеются. Для этого необходим *модуль трассировки* для системы LuNA и *утилита анализа трассы*.

2.4.1 Модуль трассировки

Предлагается расширить исполнительную систему модулем, ответственным за сбор трассы исполнения на каждом узле. Этот модуль должен предоставлять интерфейс для уведомления его о различных событиях исполнительной системы (создание и удаление ФВ и ФД, передача ФВ между узлами и т. п.). Также собранная им информация может использоваться для обнаружения некоторых семантических ошибок во время работы программы, с возможностью последующего подробного анализа после завершения работы.

2.4.2 Утилита анализа трасс

Утилита анализа трасс, названная *luna_trace*, должна агрегировать трассы, собранные на узлах мультикомпьютера во время работы программы, и производить их анализ для выявления семантических ошибок. Утилита должна предоставлять информацию не только о факте наличия ошибок как таковых, но, по возможности, и о причинах возникновения ошибок с привязкой к исходным кодам LuNA-программы, а также о возможных способах исправления.

2.5 Требования к разрабатываемым средствам

Для удобства изложения требований будут введены следующие определения:

Зависший ФВ — ФВ, выполнение которого по тем или иным причинам не может завершиться в рамках нормальной работы исполнительной системы.

Первопричина зависания — зависший ФВ, не имеющий информационных зависимостей от других зависших ФВ.

Неинициализированный ФД — ФД, который никогда не получает значения во время работы программы.

Требования к разрабатываемым средствам изложены ниже.

В отладочной версии исполнительной системы должны быть реализованы:

- Модуль сбора трассы:
    - Выполняющий сбор и буферизованную запись трассы на диск на каждом узле;
    - Поддерживающий на актуальный список незавершенных ФВ на узле;
- Автоматическая остановка в ситуации, когда на всех узлах остались только ФВ, ожидающие входные данные;
- Возможность корректной записи трассы на диск при ручной остановке работы программы посылкой сигнала (для случаев, когда зависшая программа не может быть остановлена автоматически).

Утилита анализа трасс luna_trace должна обладать следующими функциональными возможностями:

- Автоматическая агрегация файлов трасс со всех узлов;
- Предоставление информации обо всех зависших ФВ;
- Предоставление информации только первопричинах зависания;
- Для каждого зависшего ФВ — предоставление информации о:
    - выражении в коде LuNA-программы, соответствующему данному ФВ;
    - стеке вызовов до данного выражения;
    - неинициализированных ФД среди его входных ФД;

- Для каждого неинициализированного ФД — предоставление информации о:
    - имени параметра, в качестве которого был использован данный ФД;
    - полном имени данного ФД, использованном при объявлении;
    - ФК, в котором данный ФД объявлен;
    - выражении, соответствующем использованию данного ФД в коде LuNA-программы;
    - возможных выражениях в коде LuNA-программы, в которых могла бы быть выполнена инициализация данного ФД, включая стек вызовов;
- Для ФВ, созданных в теле одного цикла, информация должна выводиться единожды с указанием числа таких ФВ;
    - Если неинициализированные входные ФД имели индексированное имя, то информация о них должна предоставляться в виде диапазона с указанием шага индекса;
- Для ФД, для которых существует более одного ФВ, имеющего данный ФД среди выходных, (повторно инициализированных ФД) — предоставление информации о:
    - имени при объявлении;
    - ФК, в котором этот ФД был объявлен;
    - всех ФВ, для которых данный ФД является выходным, включая стек вызовов и выражение в LuNA-программе;
- Возможность отключения поиска повторно инициализированных ФД.

2.6 Характеристики предлагаемого решения

Данное решение позволит автоматизировать обнаружение двух классов семантических ошибок в системе LuNA — бесконечного ожидания и гонок данных. Основным плюсом такого подхода будет минимальная необходимость во вмешательстве программиста — ему нужно будет только запустить программу в отладочном режиме, а после запустить утилиту анализа трассы,

которая самостоятельно найдет все файлы трассы и выполнит анализ, после чего сообщит обо всех найденных ошибках. Тот факт, что утилита анализа трассы существует отдельно от исполнительной системы, позволит вносить в нее изменения, расширять функциональность или изменять формат вывода информации об ошибках без вмешательства в исполнительную систему. Также, предоставление информации об ошибочных выражениях в коде LuNA-программ и подсказки о потенциальных местах инициализации данных помогут быстро и эффективно исправлять найденные ошибки.

Глава 3. Реализация

3.1 Схема работы

Схематически работа разработанных средств представлена на рисунке 12.

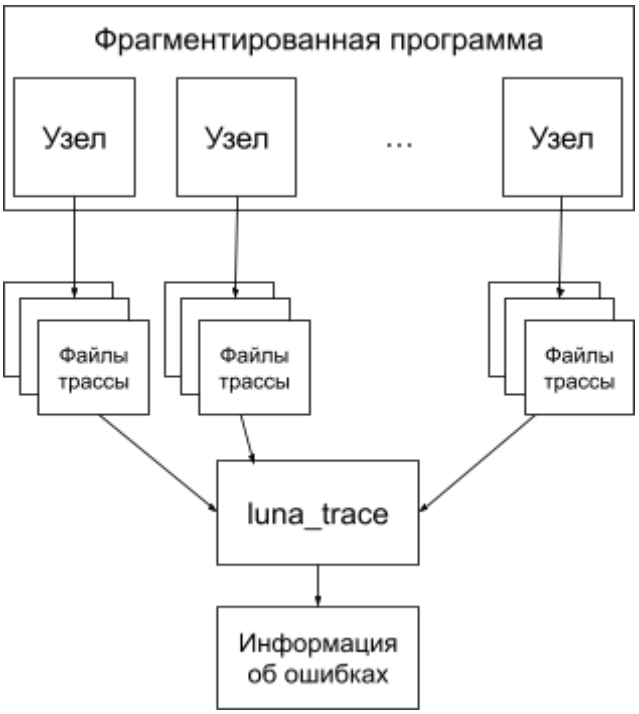

Рисунок 12 — Схема работы средств обнаружения семантических ошибок

Во время работы программы модуль WorkflowTracer на каждом узле собирает информацию о ФВ и ФД. Эта информация записывается в несколько различных файлов. После остановки программы (нормальной или вызванной пользователем) пользователь может использовать команду luna_trace в рабочей директории программы, после чего luna_trace загрузит информацию из всех обнаруженных файлов и произведет анализ.

3.2 Сбор информации во время работы

Для включения сбора трассы во время работы программы необходимо запускать/собирать программу с ключом -g (отладочный режим работы), поскольку сбор трассы и прочие изменения исполнительной системы были реализованы именно в отладочной версии.

За сбор информации во время работы программы отвечает класс WorkflowTracer. Его публичный интерфейс представлен на рисунке 13.

```
class WorkflowTracer final
{
public:
    WorkflowTracer(int rank, size_t max_id_info_size, size_t max_cf_info_size);

    void cf_created(const CF *);

    void cf_info_ready(
        const CF *,
        const DependencyInfo &
    );

    void cf_destroyed(const CF *);

    void id_created(
        const Id &,
        const std::string &name,
        int block_id
    );

    size_t cfs_remaining() const;

    void write_remaining();
};
```

Рисунок 13 — Публичный интерфейс модуля WorkflowTracer

В коде исполнительной системы фрагментам вычислений соответствуют объекты класса *CF*, содержащие значения аргументов примитивных типов, идентификаторы входных и выходных фрагментов данных, информацию о коде ФВ, а также другую специфичную для реализации исполнительной системы информацию. Для идентификации фрагментов данных используются объекты класса *Id*, представляющие собой последовательность чисел.

Для инстанцирования WorkflowTracer требуется передать номер узла, используемый в имени файлов трассы, а также размеры буферов отложенной записи в байтах для файлов с информацией о ФД и ФВ.

Размер буфера отложенной записи в байтах можно указать при запуске исполнительной системы с помощью аргумента *--log-buffer-size* (например, mpirun -np 2 $LUNA_HOME/bin/rts.dbg **--log-buffer-size=1000** build/libucodes.so).

Методы *cf_created* и *cf_destroyed* используются для учета существующих на узле ФВ. Метод *cf_info_ready* используется для сохранения информации о ФВ и его информационных зависимостях.

Для передаче информации об информационных зависимостях используется структура DependencyInfo (рисунок 14), содержащая наборы идентификаторов входных и выходных ФД.

```
struct DependencyInfo {
    std::vector<Id> in;
    std::vector<Id> out;
};
```

Рисунок 14 — Структура DependencyInfo

Метод *cfs_remaining* возвращает количество созданных ФВ, которые еще не были уничтожены (для которых был вызван *cf_created*, но не *cf_destroyed*). Этот метод используется для проверки наличия зависших ФВ после остановки системы. Если такие ФВ имеются, вызывается метод *write_remaining* для записи информации о них в отдельный файл. О назначении всех генерируемых файлов, а также об их формате будет подробно рассказано в пункте 3.4.

Система LuNA транслирует LuNA-программы в специальное промежуточное представление на языке C++. В этом представлении фрагментам вычислений соответствуют наборы функций (называемых блоками), ответственных за различные этапы жизни ФВ (запрос данных, ожидание данных, выполнение кода ФВ и т. д.), а выполнение ФВ подразумевает выполнение этих блоков. Блоки имеют уникальные номера, что позволяет во время работы хранить хранить лишь набор указателей на функции, идентифицируемые номерами. Контроль перехода между блоками осуществляется с помощью указания следующего блока (*self.NextBlock=<N>*) и возвращаемого значения. На рисунке 15 приводится пример блоков, сгенерированных для ФК main из программы, представленной на рисунке 8.

```
// MAIN
BlockRetStatus block_0(CF &self)
{

    self.NextBlock=1;
    return CONTINUE;
}

// STRUCT: sub main()
BlockRetStatus block_1(CF &self)
{
    Id _id_0=self.create_id(); // x
    Id _id_1=self.create_id(); // y
    Id _id_2=self.create_id(); // z

// GEN BODY: sub main()
    { // FORK_BI: cf _l8: init(x);
        CF *child=self.fork(2);
        child->id(0)=_id_0;
    }

    { // FORK_BI: cf _l9: compute(y, x);
        CF *child=self.fork(3);
        child->id(0)=_id_0;
        child->id(1)=_id_1;
    }

    { // FORK_BI: cf _l12: compute(z, y);
        CF *child=self.fork(6);
        child->id(0)=_id_1;
        child->id(1)=_id_2;
    }

    { // FORK_BI: cf _l15: save(z);
        CF *child=self.fork(9);
        child->id(0)=_id_2;
    }

    return EXIT;
}
```

Рисунок 15 — Промежуточное представление фрагмента LuNA-программы

Поскольку на этапе генерации промежуточного представления все еще сохраняется некоторая полезная при отладке информация информация, а

идентификаторы объектов, код создания которых генерируется здесь же, могут меняться от запуска к запуску, было решено внедрить код для сбора информации для файлов трассы именно в это промежуточное представление.

Для этого класс CF был расширен несколькими методами (рисунок 16).

```
const Id &cf_id() const;

const std::map<Id, std::string> &awaited() const;

const std::vector<int> &call_stack() const;

void log_dependencies(const DependencyInfo &) const;

DF wait(const Id &id, const char *name = nullptr);

Id create_id(
    const char *name,
    int block_id
);
```

Рисунок 16 — Дополнительные методы класса CF

Метод *cf_id* возвращает уникальный идентификатор данного ФВ. Используется для получения общей информации о ФВ для зависших ФВ.

Метод *awaited* возвращает информацию об идентификаторах ожидаемых ФД и их локальных именах.

Метод *call_stack* возвращает последовательность номеров блоков, позволяющую восстановить стек вызовов для данного ФВ. Номера блока при помощи метаинформации, генерируемой при сборке программы, можно соотнести с выражениями в коде LuNA-программы.

Описанные выше методы используются для получения информации, записываемой в файлы трассы. Далее речь пойдет об остальных методах, используемых для сохранения информации.

Метод *log_dependencies* используется для записи информации о ФВ и в конечном итоге приводит к вызову метода *cf_info_ready* у WorkflowTracer. На рисунке 17 представлен фрагмент сгенерированного кода, содержащий использование *log_dependencies*.

```
// BI_EXEC: cf _l9: compute(y, x);
BlockRetStatus block_3(CF &self)
{
    {
        DependencyInfo info;
        // Input DF Ids
        info.in.push_back(self.id(0));

        // Output DF Ids
        info.out.push_back(self.id(1));

        self.log_dependencies(info);
    }

    if (self.migrate(CyclicLocator(0))) {···

    // request x
    self.request(self.id(0), CyclicLocator(0));

    self.NextBlock=4;
    return CONTINUE;
}
```

Рисунок 17 — Использование *log_dependencies*

Метод *wait* является перегрузкой уже существующего метода, принимающего только идентификатор ФД. Как и оригинальный метод, *wait* используется для ожидания необходимых ФД, однако дополнительно принимает локальное имя ожидаемого ФД, чтобы связать его с идентификатором. На рисунках 18 и 19 показано использование перегрузок метода wait для версии программы, в которой трасса не собирается, и для версии, в которой собирается.

```
// Request : cf _l9: compute(y, x);
BlockRetStatus block_4(CF &self)
{
    // wait x
    if (self.wait(self.id(0)).is_unset()) {
        return WAIT;
    }


    self.NextBlock=5;
    return CONTINUE;
}
```

Рисунок 18 — Использование *wait* без указания имени ФД

```
// Request : cf _l9: compute(y, x);
BlockRetStatus block_4(CF &self)
{
    // wait x
    if (self.wait(self.id(0), "x").is_unset()) {
        return WAIT;
    }


    self.NextBlock=5;
    return CONTINUE;
}
```

Рисунок 19 — Использование *wait* с указанием имени ФД

Метод *create_id* также является перегрузкой. Существовавший метод *create_id* возвращал новый уникальный идентификатор для ФД. Перегрузка, используемая при сборе трассы, также принимает имя ФД при объявлении и номер блока, в котором создается идентификатор. При использовании этой перегрузки также вызывается метод *id_created* у WorkflowTracer с соответствующими параметрами. Номер блока позволяет найти объявляющий ФК в коде LuNA-программы. На рисунке 15 уже было показано использование обычной версии *create_id*. На рисунке 20 показано использование перегрузки, используемой при сборе трассы.

```
// STRUCT: sub main()
BlockRetStatus block_1(CF &self)
{
    Id _id_0=self.create_id("x", 1); // x
    Id _id_1=self.create_id("y", 1); // y
    Id _id_2=self.create_id("z", 1); // z

// GEN BODY: sub main()
    { // FORK_BI: cf _l8: init(x);···

    { // FORK_BI: cf _l9: compute(y, x);···

    { // FORK_BI: cf _l12: compute(z, y);···

    { // FORK_BI: cf _l15: save(z);···

    return EXIT;
}
```

Рисунок 20 — Использование *create_id* с указанием имени ФД и номера блока

3.3 Обнаружение зависания исполнительной системы

Пользователь может вручную остановить работу программы, послав сигнал Sigint (например, используя Ctrl+C). В этом случае вся информация, накопленная в буферах отложенной записи для файлов трасс, будет записана на диск.

В исполнительной системе за остановку системы отвечает модуль IdleStopper. Этот модуль реализует адаптированную версию алгоритма Дейкстры-Шольтена [31]. Алгоритм оперирует понятиями “наличия” и “отсутствия” работы на узлах. Когда на всех узлах не остается работы, система может быть остановлена.

В исполнительной системе LuNA наличие либо отсутствие работы на узле определяется значением счетчика задач на данном узле (невыполненные ФВ, ожидающие отправки сообщения). Каждый фрагмент вычислений (объект класса CF) увеличивает значение данного счетчика на единицу при создании и

уменьшает на единицу при уничтожении (при завершении выполнения или при передаче на другой узел).

Автоматическое обнаружение зависания, вызванного ожиданием неинициализированных данных, реализовано следующим образом:

- Исполнительная система останавливается, если все оставшиеся ФВ ожидают данные;
- Факт зависания после остановки системы обнаруживается по наличию невыполненных ФВ (используется WorkflowTracer).

Для автоматической остановки объектам CF был добавлен булев флаг *waiting*, изначально имеющий значение false. При переходе в состояние ожидания данных он устанавливается в true, а число задач на узле уменьшается на единицу. При получении *всех* необходимых данных (то есть когда ФВ может быть выполнен), если флаг *waiting* имел значение true, число задач увеличивается на единицу. Таким образом, ФВ, ожидающие данные, не учитываются при подсчете работы на узле, а значит, в ситуации, когда все оставшиеся ФВ ожидают данные, система будет остановлена вследствие отсутствия работы.

3.4 Файлы трассы и их формат

3.4.1 Информация о фрагментах вычислений

Информация о ФВ на каждом узле записывается в файл с именем *cf.<N>.json.list*, где <N> — номер узла (например, *cf.1.json.list*). Каждая строка представляет собой JSON-объект [32], обязательно имеющий следующие ключи:

- id — идентификатор ФВ, значение — массив целых чисел (значение идентификатора);
- h (сокращение history) — стек вызовов, значение — массив целых чисел (последовательность номеров блоков).

Опционально могут присутствовать следующие ключи:

- i (сокращение input) — информация о входных ФД, значение — массив массивов целых чисел (идентификаторов ФД);
- o (сокращение output) — информация о выходных ФД, значение — массив массивов целых чисел (идентификаторов ФД).

Данные ключи отсутствуют, если их значения пусты, для уменьшения размера файла. На рисунке 21 приводится пример содержимого файла cf.0.json.list для программы с рисунка 5, запущенной на одном узле.

```
1   {"id":[0,2],"h":[0,1,5],"i":[[0,1]],"o":[[0,2]]}
2   {"id":[0,1],"h":[0,1,2],"i":[[0,0]],"o":[[0,1]]}
3   {"id":[0,3],"h":[0,1,8],"i":[[0,2]]}
```

Рисунок 21 — Пример файла с информацией о ФВ

3.4.2 Информация о фрагментах данных

Информация о ФД на каждом узле записывается в файл с именем *id.<N>.map.list*, где <N> — номер узла (например, *id.1.map.list*). Каждая строка содержит информацию об одном базовом имени ФД. На рисунке 22 приводится пример содержимого файла id.0.map.list для программы с рисунка 5, запущенной на одном узле.

```
1   0 0>a 1
2   0 1>b 1
3   0 2>c 1
```

Рисунок 22 — Пример файла с информацией о ФД

Два числа перед символом “>” — идентификатор, создаваемый для ФД при вызове метода *create_id* у объекта CF. После “>” следуют имя ФД при объявлении и номер блока, в котором был создан соответствующий идентификатор, разделенные пробелом.

3.4.3 Информация о зависших фрагментах вычислений

Информация о зависших ФВ на каждом узле записывается в файл с именем *pending.<N>.json.list*, где <N> — номер узла (например, *pending.1.json.list*). Каждая строка представляет собой JSON-объект, обязательно имеющий ключ id (аналогично файлам с информацией о ФВ), а

также опционально ключ w (сокращение awaited), значение которого — массив объектов, описывающих ожидаемые ФД. Эти объекты имеют ключи:

- i (сокращение id) — идентификатор ФД, значение — массив целых чисел;
- n (сокращение name) — локальное имя ФД, значение — строка.

Аналогично файлу с информацией о ФВ, ключ w может быть опущен при отсутствии ожидаемых ФД (например, если причиной зависания был атомарный ФК). На рисунке 23 приводится пример содержимого файла pending.0.json.list для программы с рисунка 5, запущенной на одном узле.

```
1   {"id":[0,1],"w":[{"i":[0,0],"n":"a"}]}
2   {"id":[0,2],"w":[{"i":[0,1],"n":"b"}]}
3   {"id":[0,3],"w":[{"i":[0,2],"n":"c"}]}
```

Рисунок 23 — Пример файла с информацией о зависших ФВ

3.5 Функциональные возможности luna_trace

Утилита командной строки *luna_trace* реализована на языке Python. Для запуска пользователю доступен bash-скрипт luna_trace, использующий шаблоны имен файлов для автоматической передачи непосредственно программе-анализатору всех имен файлов трассы в качестве аргументов (id.*.map.list, pending.*.json.list и cf.*.json.list). Для использования luna_trace требуется информация, генерируемая при сборке LuNA-программы. Для этого необходимо собирать программу с использованием ключей *--no-cleanup* и *--build-dir=build*.

Утилита luna_trace имеет следующие аргументы командной строки:

--all — вывести информацию обо всех зависших ФВ, а не только о первопричинах;

--no-double-init — не искать случаи повторной инициализации ФД.

Вывод luna_trace соответствует требованиям, изложенным в пункте 2.5.

На рисунке 24 приведен пример вывода luna_trace для программы с рисунка 9 (код программы находился в файле с именем race.fa). В этой

программе имеется ошибка соревнования, вызванная множественной инициализацией ФД.

```
Following DFs are initialized multiple times:

x (declared in sub main) in
    set_after_delay(x, 0, delay[1]) [./race.fa:14]
        in sub main() [./race.fa:9]

    set_after_delay(x, 1, delay[2]) [./race.fa:15]
        in sub main() [./race.fa:9]


 -  -  -  -
```

Рисунок 24 — Вывод luna_trace для программы, содержащей ошибку гонки данных

Пользователю предоставляется информация о ФД, инициализируемом несколько раз, а также о выражениях, выполняющих инициализацию, со ссылкой на код LuNA-программы.

На рисунке 25 приводится вывод luna_trace для программы с рисунка 6. В этой программе используются неинициализированные ФД.

```
Following CFs appear to be the root cause of the system hang:

print(msg[i]) [./main_indexing_error.fa:10] never finished (4 instances)
main
    in for i = 0..N+3 [./main_indexing_error.fa:9]
    in sub main() [./main_indexing_error.fa:5]

    Awaited DFs msg[i] (where msg is msg, msg declared in sub main):
        msg[10]..msg[13] (msg[10]..msg[13])

    Maybe it should have been initialized here:
        set(msg[i], "This is item " + i) [./main_indexing_error.fa:14]
        in for i = 0..N-1 [./main_indexing_error.fa:13]
        in sub main() [./main_indexing_error.fa:5]

 -  -  -  -
```

Рисунок 25 — Вывод luna_trace для программы, содержащей использование неинициализированных данных

Пользователю предоставляется информация о выражениях, соответствующих зависшим ФВ (*print(msg[i])*), стек вызовов, а также

информация о неинициализированных данных в виде диапазона (*msg[10]..msg[13]*), поскольку зависшие ФВ соответствовали телу цикла. Также пользователю предоставляется информация о возможных местах в коде программы, где могла быть пропущена инициализация.

Циклы включаются в вывод стека вызовов, поскольку зачастую именно в объявлении цикла находится ошибочное указание границ итерации. Так, например, в приведенном выше примере, в выводе в стеке вызовов присутствует как цикл, в теле которого были созданы зависше ФВ (*for i = 0..N+3*), так и цикл, в теле которого по предположению утилиты должна была производиться инициализация (*for i = 0..N-1*).

В luna_trace реализована возможность поиска первопричин зависания. По умолчанию поиск производится, однако при использовании ключа *--all* будет выведена информация обо всех зависших ФВ. На рисунках 26 и 27 приводится вывод luna_trace для программы с рисунка 5 без использования ключа --all и с его использованием соответственно.

```
Following CFs appear to be the root cause of the system hang:

set(b, a * a) [./root.fa:11] never finished
main
    in sub main() [./root.fa:9]

    Awaited DF a
        full name a, a declared in sub main
 -  -  -  -

2 more CFs never finished but had data dependencies on other hang CFs
```

Рисунок 26 — Информация о первопричинах зависания

```
set(b, a * a) [./root.fa:11] never finished
main
    in sub main() [./root.fa:9]

    Awaited DF a
        full name a, a declared in sub main
 -  -  -  -

set(c, b + 1) [./root.fa:12] never finished
main
    in sub main() [./root.fa:9]

    Awaited DF b
        full name b, b declared in sub main
    Maybe it should have been initialized here:
        set(b, a * a) [./root.fa:11]
        in sub main() [./root.fa:9]

 -  -  -  -

print(c) [./root.fa:13] never finished
main
    in sub main() [./root.fa:9]

    Awaited DF c
        full name c, c declared in sub main
    Maybe it should have been initialized here:
        set(c, b + 1) [./root.fa:12]
        in sub main() [./root.fa:9]

 -  -  -  -
```

Рисунок 27 — Информация обо всех зависших ФВ

3.6 Поиск первопричин зависания

Для поиска первопричин зависания строится ориентированный граф информационных зависимостей между зависшими ФВ.

Пусть cf — фрагмент вычислений, тогда обозначим

in(cf) — множество входных ФД cf,

out(cf) — множество выходных ФД cf.

Пусть cf_1 и cf_2 — зависшие ФВ. Тогда в графе информационных зависимостей будет присутствовать ребро (cf_1, cf_2) тогда и только тогда, когда $in(cf_1) \cap out(cf_2) \neq \varnothing$.

Первопричинами будут ФВ, не имеющие исходящих ребер.

На рисунке 28 приведен пример графа информационных зависимостей между ФВ для программы с рисунка 5. Выделенный ФВ — первопричина зависания.

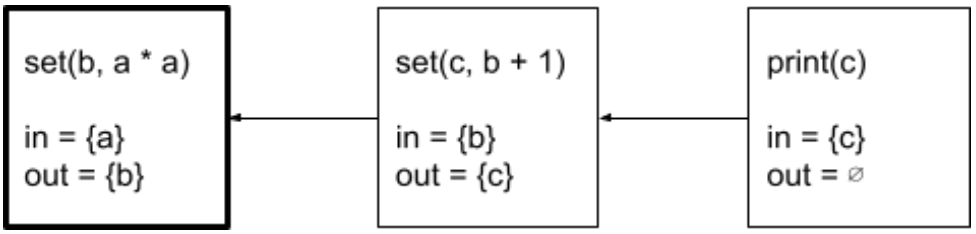

Рисунок 28 — Граф информационных зависимостей для зависших ФВ

Возможна ситуация, когда в непустом графе нет ни одной вершины, не имеющей исходящих ребер. В этом случае luna_trace выведет информацию обо всех зависших ФВ, а также сообщение о вероятном наличии циклических зависимостей между ФВ.

3.7 Поиск мест пропущенной инициализации данных

Поиск мест в коде программы, где могла быть пропущена инициализация ФД, основан на поиске мест инициализации *похожих* ФД.

Пусть существует множество неинициализированных ФД

$DF_1 = \{ \mathbf{name}[i] \mid i \in I \}$,

а также множество ФД

$DF_2 = \{ \mathbf{name}[j] \mid j \in J, I \cap J = \varnothing \}$,

которые были во время работы программы инициализированы множеством ФВ CF, причем все ФВ из множества CF выполняли один и тот же код (их выполнение начиналось с одного и того же блока). Тогда можно предположить, что ФД из множества DF_1 должны были быть инициализированы фрагментами вычислений, выполняющими тот же код, что и ФВ из множества CF. Имея номер блока, с которого начиналось выполнение ФВ из CF, можно найти соответствующее выражение в коде LuNA-программы.

Такой подход к поиску мест инициализации можно проиллюстрировать следующим примером:

Пусть x[1]..x[9] были инициализированы в одном цикле, а x[10] не был инициализирован. Высока вероятность, что x[10] должен был быть инициализирован в том же цикле, а в границах цикла была допущена ошибка. Вывод luna_trace, содержащий предположение о месте пропущенной инициализации для аналогичной ситуации, представлен на рисунке 25.

3.8 Вывод истории вызовов

При сборке LuNA-программы транслятор генерирует несколько файлов, содержащих метаинформацию о коде программы. В частности, генерируются файлы *preprocessed.fa.ti* и *cpp_blocks_info.json*. Файл *cpp_blocks_info.json* содержит информацию о соответствии блоков коду LuNA-программы, а также информацию о реализуемых блоками действий (выполнение кода ФВ, запрос данных и т. п.). Файл *preprocessed.fa.ti* содержит информацию о коде программы, которая в совокупности с информацией из *cpp_blocks_info.json* позволяет по номеру блока определить файл с исходным кодом, а также строку в нем, содержащую выражение, которому соответствует тот или иной блок.

Благодаря этой информации возможно хранить в файлах трассы лишь стек вызовов в виде последовательности номеров блоков, и при необходимости по номерам блоков восстановить как само выражение, соответствующее ФВ, так и последовательность вызовов, предшествующих ему.

3.9 Поиск множественной инициализации данных

В файлах трассы для создаваемых во время работы программы ФВ имеется информация об идентификаторах входных и выходных ФД. Благодаря информации о выходных ФД, можно произвести полный перебор пар ФВ и определить, существуют ли среди них те, что инициализируют один и тот же ФД.

Поскольку полный перебор при большом числе ФВ может занимать значительное время, в нем участвуют только ФВ, имеющие хотя бы один выходной ФД. Также, проверка на наличие повторяющихся идентификаторов

среди выходных для двух ФВ реализована с использованием множеств (set в Python), что позволяет использовать стандартную операцию пересечения.

Итогом поиска множественной инициализации является коллекция пар (ФД, коллекция ФВ), где коллекция ФВ — все ФВ, инициализирующие данный ФД. Этой информации достаточно, чтобы предоставить пользователю информацию о местах повторной инициализации в коде LuNA-программы для каждого ФД, инициализируемого несколько раз.

Глава 4. Тестирование

4.1 Тестирование обнаружения ошибок

Рассмотрим программу в приложении В. Эта программа реализует фрагментированный алгоритм вычисления скалярного произведения двух векторов. В программе намеренно допущены ошибки, приводящие к гонке данных (множественная инициализация) и зависанию (использование неинициализированных данных).

При запуске такой программы в отладочном режиме командой *luna -g --build-dir=build --no-cleanup scalar_product.fa*, она завершается (не зависает) с сообщением об ошибке (рисунок 29).

```
err> 0 ERROR:  System is idle, but unfinished CFs (2) exist. ./src/rts/rts.cpp:140
```

Рисунок 29 — Сообщение об ошибке для некорректной программы

Утилита анализа трасс luna_trace, запущенная в той же рабочей директории, что и программа, производит следующий вывод (рисунок 30):

```
Following DFs are initialized multiple times:

result (declared in sub main) in
    set(result, 0) [./scalar_product.fa:39]
        in sub main() [./scalar_product.fa:36]

    set(result, acc[size]) [./scalar_product.fa:23]
        in sub reduce(name arr, int size, name result) [./scalar_product.fa:12]
        in reduce(prod, size, result) [./scalar_product.fa:33]
        in sub product(name arr1, name arr2, int size, name result) [./scalar_product.fa:26]
        in product(a, b, N, result) [./scalar_product.fa:46]
        in sub main() [./scalar_product.fa:36]


 -  -  -  -

Following CFs appear to be the root cause of the system hang:

set(acc[i], acc[i - 1] + arr[i]) [./scalar_product.fa:20] never finished
main -> product -> reduce
    in for i = 1..size [./scalar_product.fa:19]
    in sub reduce(name arr, int size, name result) [./scalar_product.fa:12]
    in reduce(prod, size, result) [./scalar_product.fa:33]
    in sub product(name arr1, name arr2, int size, name result) [./scalar_product.fa:26]
    in product(a, b, N, result) [./scalar_product.fa:46]
    in sub main() [./scalar_product.fa:36]

    Awaited DF arr[100] (arr[i])
        full name prod[100], prod declared in sub product
    Maybe it should have been initialized here:
        fragment_product(arr1[i], arr2[i], prod[i]) [./scalar_product.fa:30]
        in for i = 0..size-1 [./scalar_product.fa:29]
        in sub product(name arr1, name arr2, int size, name result) [./scalar_product.fa:26]
        in product(a, b, N, result) [./scalar_product.fa:46]
        in sub main() [./scalar_product.fa:36]

 -  -  -  -

1 more CFs never finished but had data dependencies on other hang CFs
```

Рисунок 30 — Вывод luna_trace для некорректной программы

Утилита luna_trace корректно идентифицировала ФД, инициализируемый повторно (*result*), и инициализирующие выражения. Также корректно были определен ФВ, приведший к зависанию, и необходимый ему неинициализированный ФД. Корректно был определен цикл, в котором выполнялась инициализация других ФД из той же группы.

Утилита luna_trace также тестировалась на некорректных программах из пунктов 2.3.1 и 2.3.2. Автоматическое обнаружение зависания также тестировалось на программах, содержащих использование неинициализированных данных, из пункта 2.3.1.

4.1 Оценка накладных расходов при сборе отладочной информации

Была исследована зависимость времени работы программы и пикового использования памяти от значения ключа --log-buffer-size (размера буфера файла трассы).

Оценка времени и использования памяти проводились с помощью команды /usr/bin/time на ОС Ubuntu 20.04 LTS:

- Процессор: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz
- Число ядер: 2
- Число логических процессоров: 4
- RAM: 12,0 ГБ DDR3
- Число MPI-процессов: 2
- Число рабочих потоков в run-time системе: 4 (значение по умолчанию)

Для оценки использовалась программа, реализующая алгоритм блочного умножения квадратных матриц. Далее под размером и числом блоков будут пониматься эти значения вдоль одной из осей.

В таблице 1 и на рисунке 31 представлены результаты для размера блока 50 и числа блоков 40.

Таблица 1 — Время работы и использование памяти в зависимости от значения --log-buffer-size (50x40)

Размер буфера файла трассы, б	Время работы программы, с	Использование памяти, Мб
1 000	91,48	2 679,6
10 000	49,54	2 678,1
100 000	46,23	2 680,5
1 000 000	35,8	2 683,1
10 000 000	30,01	2 688,4

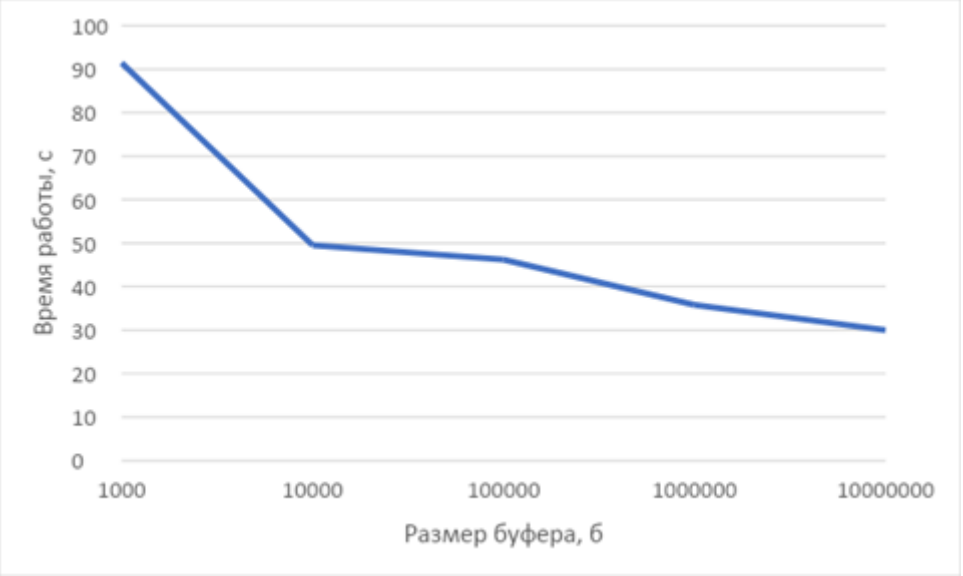

Рисунок 31 — Время работы в зависимости от значения --log-buffer-size (50x40)

В таблице 2 и на рисунке 32 представлены результаты для размера блока 100 и числа блоков 20.

Таблица 2 — Время работы и использование памяти в зависимости от значения --log-buffer-size (100x20)

Размер буфера файла трассы, б	Время работы программы, с	Использование памяти, Мб
1000	27,14	1 316,8
10000	21,62	1 318,6
100000	16,78	1 319,3
1000000	16,35	1 318,7
10000000	16,18	1 319,6

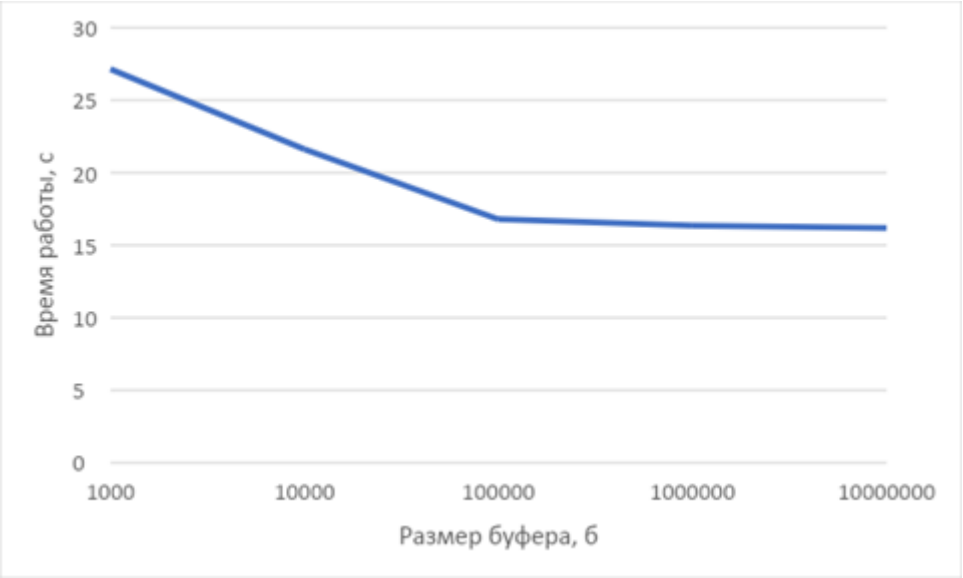

Рисунок 32 — Время работы в зависимости от значения --log-buffer-size (100x20)

Видно, что в зависимости от того, как задача разделена на ФВ, влияние размера буфера на время работы может различаться, хотя, вообще говоря, увеличение размера буфера коррелирует с уменьшением времени работы программы.

При этом на данной задаче при рассматриваемых вариантах значения размера буфера влияние на использование памяти оказывается незначительным.

4.2 Оценка времени работы luna_trace

Было обнаружено, что поиск повторной инициализации данных может занимать существенное время. Была произведена оценка времени работы luna_trace для одной и той же задачи при наличии/отсутствии ключа --no-double-init для различного числа ФВ, порождаемых в процессе работы. В качестве задачи вновь использовалось блочное умножение матриц размером 2000 на 2000 с варьируемым размером блока (2004 на 2004 для размера блока 167).

Оценка времени и использования памяти проводились с помощью команды /usr/bin/time на ОС Ubuntu 20.04 LTS:

- Процессор: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz

- Число ядер: 2
- Число логических процессоров: 4
- RAM: 12,0 ГБ DDR3
- Число MPI-процессов: 1
- Число рабочих потоков в run-time системе: 4 (значение по умолчанию)
- Значение --log-buffer-size: 1 Мб (значение по умолчанию)

В таблице 3 и на рисунке 33 приводятся результаты измерений. Под числом фрагментов вычислений понимается число записей о ФВ в файлах трассы, анализируемых luna_trace.

Таблица 3 — Время работы luna_trace для различного числа ФВ

Размер блока	Число блоков	Число фрагментов вычислений	Время работы программы, с	Время работы luna_trace без ключа --no-double-init, с	Время работы luna_trace с ключом --no-double-init, с
500	4	247	47,25	0,49	0,37
250	8	1483	17,93	0,68	0,37
167	12	4479	15,67	3,4	0,44
125	16	10003	14,05	12,24	0,63
100	20	18823	12,38	45,7	0,64

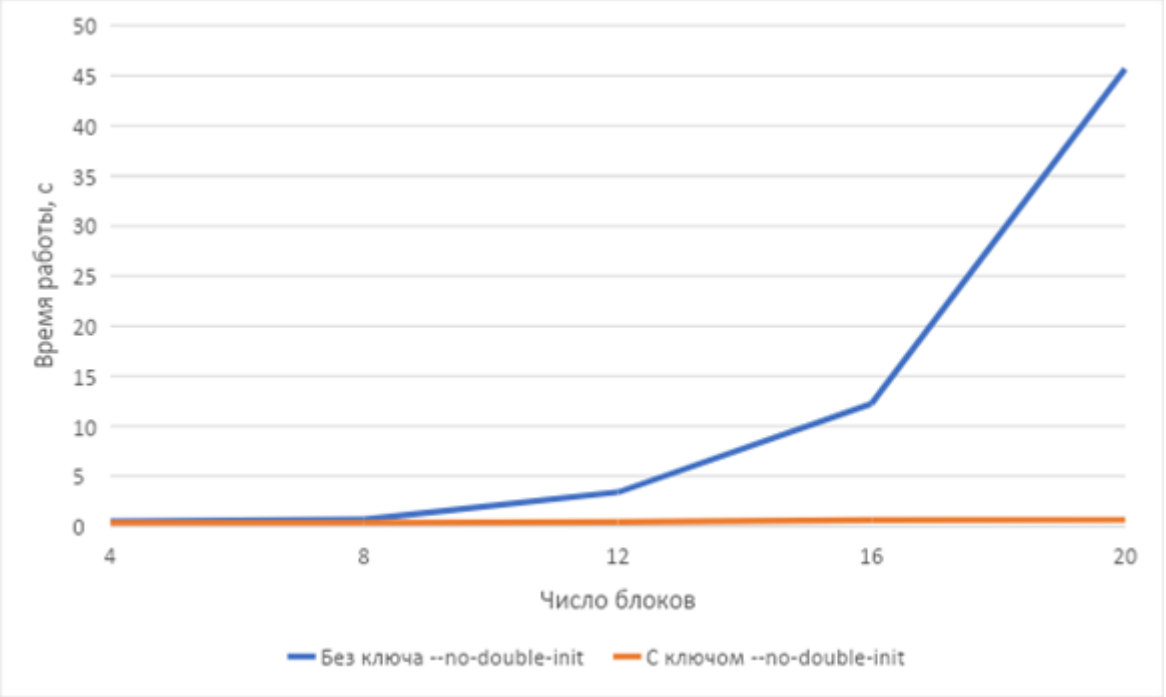

Рисунок 33 — Время работы luna_trace в зависимости от числа блоков

Поскольку при запуске программа не зависала и luna_trace, как следствие, не выполняла анализ причин зависания, время работы с ключом --no-double-init можно рассматривать как время на загрузку данных из файлов трассы.

Таким образом, при большом числе фрагментов вычислений, время на поиск случаев повторной инициализации, и, как следствие, время работы luna_trace может быть довольно большим, и даже превышать время работы самой программы. В связи с этим использование ключа --no-double-init может быть оправдано.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был проведен анализ семантических ошибок, свойственных фрагментированным программам. Для системы LuNA был разработан модуль сбора трассы, а также реализована возможность автоматической остановки при зависании, вызванном использованием неинициализированных данных. Было разработано средство анализа трассы, выполняющее автоматический поиск семантических ошибок. Разработанное средство протестировано на ряде некорректных LuNA-программ.

Выполнено тестирование с целью оценки влияния сбора отладочной информации на время работы программы, а также с целью оценки времени работы средства анализа трассы в зависимости от параметров задачи.

Разработанное средство в будущем может быть расширено функциональностью по обнаружению дополнительных классов семантических ошибок, а также другими методами и подходами к отладке.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Мичуров Михаил Антонович _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Perepelkin V. A. LuNA system for automatic construction of numerical parallel programs for multicomputers // Проблемы информатики. 2020. №1 (46).
2. Маркова В. П., Остапкевич М. Б. Сравнение возможностей MPI и LuNA на примере реализации модели клеточно-автоматной интерференции волн // Проблемы информатики. – 2017. – №. 2 (35). – С. 53-64.
3. Мичуров М. А. Средство анализа причин зависаний фрагментированных программ в системе LuNA // Инновации. Наука. Образование. 2021. № 40. С. 354-364.
4. Власенко А. Ю. и др. Автоматизированный контроль корректности MPI-программ на основе шаблонов ошибочного поведения. – 2014.
5. Иртегов, Д. В. Введение в операционные системы. — 2-е изд. — Санкт-Петербург: БХВ-Петербург, 2008
6. Кропачева М. С. Формальная верификация параллельных программ // Космические аппараты и технологии. – 2012. – №. 2. – С. 35-38.
7. Рыжков Е. А., Середин О. С. Применение технологии статического анализа кода при разработке параллельных программ // Известия Тульского государственного университета. Технические науки. – 2008. – №. 3. – С. 191-197.
8. Колосов А. П. Методы отладки параллельных программ // Известия Тульского государственного университета. Технические науки. – 2010. – №. 2-2. – С. 172-180.
9. Власенко А. Ю. Модель масштабируемой системы автоматического контроля корректности параллельных программ // Вестник Новосибирского государственного университета. Серия: Информационные технологии. – 2009. – Т. 7. – №. 4. – С. 53-65.

10. Афанасьев К. Е., Власенко А. Ю. Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки // Вестник Кемеровского государственного университета. – 2009. – №. 2. – С. 13-20.
11. Карпов, Ю. Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.
12. Spin, open-source verification software [Электронный ресурс]. URL: https://spinroot.com/spin/whatispin.html (дата обращения: 05.05.2022)
13. Можаров Г. П., Парфилов И. В. Подход к поиску взаимных блокировок в многопоточном программном обеспечении с помощью верификатора spin // Инженерный журнал: наука и инновации. – 2012. – №. 11 (11). – С. 7.
14. Siegel S. F. Verifying parallel programs with MPI-Spin //European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. – Springer, Berlin, Heidelberg, 2007. – С. 13-14.
15. Intel Trace Analyzer and Collector [Электронный ресурс]. URL: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-itac/top.html (дата обращения: 10.05.2022).
16. Krammer B. et al. MARMOT: An MPI analysis and checking tool // Advances in Parallel Computing. – North-Holland, 2004. – Т. 13. – С. 493-500.
17. Luecke G. et al. MPI‑CHECK: a tool for checking Fortran 90 MPI programs // Concurrency and Computation: Practice and Experience. – 2003. – Т. 15. – №. 2. – С. 93-100.
18. DeSouza J. et al. Automated, scalable debugging of MPI programs with Intel® Message Checker // Proceedings of the second international workshop on software engineering for high performance computing system applications. – 2005. – С. 78-82.
19. Samofalov V. et al. Automated Correctness Analysis of MPI Programs with Intel (r) Message Checker // PARCO. – 2005. – С. 901-908.

20. Отладка корректности MPI-программ [Электронный ресурс]. URL: http://keldysh.ru/dvm/dvmhtm1107/rus/deb_mpi/tranUG.html (дата обращения: 10.05.2022).
21. TotalView Debugger [Электронный ресурс]. URL: https://totalview.io/free-trial (дата обращения: 05.05.2022).
22. Киселев А. Б., Киселев С. Н. Отладчик параллельных программ для ОС Linux // Труды Института системного программирования РАН. – 2020. – Т. 32. – №. 4. – С. 97-114.
23. Крюков В. А. Разработка параллельных программ для вычислительных кластеров и сетей // Информационные технологии и вычислительные системы. – 2003. – Т. 1. – С. 2.
24. Бахтин В. А. и др. Сравнительная отладка OpenMP-программ // Препринты Института прикладной математики им. МВ Келдыша РАН. – 2009. – №. 0. – С. 76-24.
25. Алексахин В. А. и др. СРЕДСТВА ОТЛАДКИ OpenMP ПРОГРАММ В DVM-СИСТЕМЕ // Научный сервис в сети Интернет: решение больших задач. – 2008. – С. 281-285.
26. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.
27. Ахмед-Заки Д. Ж. и др. Автоматизация конструирования распределенных программ численного моделирования в системе LuNA на примере модельной задачи // Проблемы информатики. – 2019. – №. 4 (45).
28. Описание языка LuNA [Электронный ресурс].URL: https://gitlab.ssd.sscc.ru/luna/luna5/wikis/luna_lang_v01 (дата обращения: 05.05.2022)
29. Перепелкин В. А., Софронов И. В., Ткачева А. А. Автоматизация конструирования численных параллельных программ с заданными

нефункциональными свойствами на базе вычислительных моделей // Проблемы информатики. – 2017. – №. 4 (37). – С. 47-60.

30. Репозиторий системы LuNA [Электронный ресурс]. URL: https://gitlab.ssd.sscc.ru/luna/luna (дата обращения: 05.05.2022)
31. Fokkink, W. Distributed Algorithms: An Intuitive Approach / W. Fokkink. — : MIT Press, 2013. — 231 с.
32. Введение в JSON [Электронный ресурс]. URL: https://www.json.org/json-ru.html (дата обращения: 19.05.2022)

ПРИЛОЖЕНИЕ А

Код LuNA-программы, содержащей ошибку в атомарном ФК

Файл bad_atomic.fa:

```
C++ sub print_array(value x) ${{
   // Получение размера массива
   const auto data = reinterpret_cast<const size_t *>(x.get_data());
   const auto size = data[0];

   // Обращение к элементам массива
   const auto array = reinterpret_cast<const double *>(data + 1);
   for (auto i = 0; i < size; i = 1) {
      printf("[%d] = %lf\n", i, array[i]);
   }
$}}

C++ sub create_array(name x, int size) ${{
   // Выделение необходимой памяти (размер массива + сам массив)
   x.create(sizeof(size_t) + sizeof(double) * size);

   // Запись в выделенную память размера
   auto data = reinterpret_cast<size_t *>(x.get_data());
   *data = size;

   // Запись в выделенную память массива
   auto array = reinterpret_cast<double *>(data + 1);
   for (auto i = 0; i < size; i += 1) {
      array[i] = i * i;
   }
$}}

sub main() {
   df arr;

   let N = 10 {
      create_array(arr, N);
      print_array(arr);
   }
}
```

ПРИЛОЖЕНИЕ Б

Код LuNA-программы, содержащей ошибку гонки данных

Файл race.fa:

```
import set_after_delay(name, int, int) as set_after_delay;
import set_delays_random(name, name, int, int) as set_delays_random;

C++ sub print(int x) ${{
   printf("%d\n", x);
$}}


sub main() {
   df x, delay;

   set_delays_random(delay[1], delay[2], 1, 2);

   set_after_delay(x, 0, delay[1]);
   set_after_delay(x, 1, delay[2]);

   print(x) @ {
      delete x;
   };
}
```

Файл race.cpp:

```
#include <stdlib.h>
#include <unistd.h>
#include <ucenv.h>


extern "C" {
   void set_after_delay(
      OutputDF &x,
      int value,
      int delay
   ) {
      sleep(delay);
      x = value;
   }

   void set_delays_random(
      OutputDF &delay1,
      OutputDF &delay2,
      int delay1_value,
      int delay2_value
   ) {
      srand(time(nullptr));
      auto reverse_order = rand() > (RAND_MAX / 2);
```

```
        delay1 = reverse_order ? delay2_value : delay1_value;
        delay2 = reverse_order ? delay1_value : delay2_value;
    }
}
```

ПРИЛОЖЕНИЕ В

Код LuNA-программы, содержащей использование неинициализированных данных и ошибку гонки данных

Файл bad_atomic.fa:

```
import c_fill_fragment(name, int) as fill_fragment;
import c_fragment_product(value, value, name) as fragment_product;
import c_print(real) as print;
import c_set(name, real) as set;

sub init_arr(name arr, int size, int fragment_size) {
   for i = 0..size-1 {
      fill_fragment(arr[i], fragment_size);
   }
}

sub reduce(name arr, int size, name result) {
   df acc;

   if size > 0 {
      set(acc[0], arr[0]);
   }

   for i = 1..size {
      set(acc[i], acc[i - 1] + arr[i]);
   }

   set(result, acc[size]);
}

sub product(name arr1, name arr2, int size, name result) {
   df prod;

   for i = 0..size-1 {
      fragment_product(arr1[i], arr2[i], prod[i]);
   }

   reduce(prod, size, result);
}

sub main() {
   df a, b, result;

   set(result, 0);
   print(result) @ { delete result; };

   let N = 100, FragmentSize = 5 {
      init_arr(a, N, FragmentSize);
      init_arr(b, N, FragmentSize);
```

```
        product(a, b, N, result);
        print(result);
    }
}
```

Файл scalar_product.cpp:

```
#include <stdlib.h>
#include <unistd.h>
#include <ucenv.h>

using namespace luna::ucenv;

struct ArrayFragment {
    size_t size;
    double *elements;

    static const ArrayFragment attach(const InputDF &df) {
        const auto data = reinterpret_cast<const size_t *>(df.get_data());
        const auto size = *data;
        const auto elements = reinterpret_cast<const double *>(data + 1);

        return { size, const_cast<double *>(elements) };
    }

    static ArrayFragment create(OutputDF &df, const size_t size) {
        df.create(sizeof(size) + size * sizeof(double));
        auto data = reinterpret_cast<size_t *>(df.get_data());
        *data = size;
        auto elements = reinterpret_cast<double *>(data + 1);

        return { size, elements };
    }
};

extern "C" {
    void c_fill_fragment(OutputDF &df, int size) {
        auto fragment = ArrayFragment::create(df, size);

        for (auto i = 0; i < fragment.size; i += 1) {
            fragment.elements[i] = i;
        }
    }

    void c_fragment_product(const InputDF &first, const InputDF &second, OutputDF &result) {
        const auto f1 = ArrayFragment::attach(first);
        const auto f2 = ArrayFragment::attach(second);

        assert(f1.size == f2.size);
```

```
        auto sum = 0.0;
        for (auto i = 0; i < f1.size; i += 1) {
            sum += f1.elements[i] * f2.elements[i];
        }
        result = sum;
    }

    void c_print(double v) {
        printf("%lf\n", v);
    }

    void c_set(OutputDF &result, double v) {
        result = v;
    }
}
```

ПРИЛОЖЕНИЕ Г

Средство автоматизированного обнаружения семантических ошибок luna_trace

Руководство оператора

Листов 6

Новосибирск 2022

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации средства автоматизированного обнаружения семантических ошибок luna_trace.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

1 Назначение программы

1.1 Функциональное назначение программы

Разработанная программа предназначена для анализа файлов трассы, генерируемых при выполнении фрагментированных программ исполнительной системой LuNA, с целью поиска наиболее типичных семантических ошибок и анализа их причин.

1.2 Эксплуатационное назначение программы

Средство обнаружения ошибок должно использоваться разработчиками фрагментированных программ для системы LuNA во время отладки программ.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Автоматическая агрегация файлов трассы с узлов мультикомпьютера;
- Предоставление информации о причинах зависания фрагментированной программы;
- Предоставление информации о возможных местах инициализации отсутствующих данных;
- Предоставление информации о случаях повторной инициализации данных.

2. Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере или вычислительном кластере. Устройство, на котором запускается программа, должно иметь:

- Оперативная память объемом не менее 4 Гб;
- Жёсткий диск объёмом не менее 64 Гб;
- Монитор;
- Клавиатура.

2.2 Минимальный состав программных средств

- Операционная система, для которой существует реализация интерпретатора языка Python;
- Интерпретатор языка Python версии не ниже 3.8;
- Командный интерпретатор bash.

2.3 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки.

3. Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется вызовом команды luna_trace в рабочей директории LuNA-программы.

Пользователь может указать следующие опциональные параметры:

- --all — вывести информацию обо всех зависших фрагментах вычислений;
- --no-double-init — не выполнять поиск случаев повторной инициализации.

3.2 Выполнение программы

После запуска luna_trace выполнит анализ обнаруженных файлов трассы и напечатает в стандартный поток вывода информацию обо всех найденных ошибках. Пример вывода представлен на рисунке 1.

```
Following DFs are initialized multiple times:

x (declared in sub main) in
    set_after_delay(x, 0, delay[1]) [./race.fa:14]
        in sub main() [./race.fa:9]

    set_after_delay(x, 1, delay[2]) [./race.fa:15]
        in sub main() [./race.fa:9]


 -  -  -  -
```

Рисунок 1 — Вывод luna_trace для программы, содержащей ошибку гонки данных

3.3 Завершение работы программы

Программа завершается автоматически после вывода результата анализа файлов трассы. Программу можно прервать в любой момент, используя сигнал SIGINT. Для этого можно использовать сочетание клавиш Ctrl+C.