

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**

**Кудрявцева Андрея Александровича**

Тема работы:

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ СРЕДСТВ ДИРЕКТИВНОГО  
УПРАВЛЕНИЯ ПАМЯТЬЮ В СИСТЕМЕ LUNA**

**«К защите допущен»**  
Заведующий кафедрой,  
д.т.н., профессор  
Мальшкин В.Э./.....  
«31» мая 2023г.

**Руководитель ВКР**  
к.т.н.  
доц. каф. ПВ ФИТ НГУ  
Власенко А.Ю./.....  
«20» мая 2023г.

Соруководитель ВКР  
ст. преп. каф. ПВ ФИТ НГУ  
Перепёлкин В.А./.....  
«20» мая 2023г.

Новосибирск, 2023

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

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

.....  
«16» января 2023г.

**ЗАДАНИЕ**

**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

Студенту Кудрявцеву Андрею Александровичу, группы 19203

Тема Разработка и реализация средств директивного управления памятью в системе LuNA

утверждена распоряжением проректора по учебной работе от 07.11.2022 № 0330,  
скорректирована распоряжением проректора по учебной работе от 16.01.2023 № 0006

Срок сдачи студентом готовой работы 20 мая 2023 г.

Исходные данные (или цель работы): разработке директивных средств управления памятью для системы LuNA, которые будут способны приблизить расход памяти LuNA-программ к ее расходу программами, написанными с использованием низкоуровневых средств параллельного программирования.

Структурные части работы: введение, основная часть, заключение, список литературы и приложения.

Руководитель ВКР  
к.т.н.  
доц. каф. ПВ ФИТ НГУ  
Власенко А.Ю. /.....  
«16» января 2023г.

Задание принял к исполнению  
Кудрявцев А.А./.....  
«16» января 2023г.

Соруководитель ВКР  
ст. преп. каф. ПВ ФИТ НГУ  
Перепёлкин В.А. /.....  
«16» января 2023г.

## СОДЕРЖАНИЕ

Введение .....	4
1 Анализ предметной области .....	7
2 Алгоритм уничтожающего потребления .....	10
2.1 ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ .....	10
2.2 АНАЛИЗ ПРОБЛЕМЫ .....	12
2.3 ПОСТАНОВКА ЗАДАЧИ .....	14
2.4 ТРЕБОВАНИЯ К РЕШЕНИЮ .....	14
2.5 ОПИСАНИЕ РЕШЕНИЯ.....	15
3 Реализация .....	17
3.1 УСТРОЙСТВО СИСТЕМЫ LuNA .....	17
3.2 РЕАЛИЗАЦИЯ УНИЧТОЖАЮЩЕГО ПОТРЕБЛЕНИЯ .....	18
3.2.1 Расширение интерфейса класса DF .....	18
3.2.2 Расширение грамматики языка LuNA.....	19
3.2.3 Модификация транслятора системы LuNA .....	23
3.3 АНАЛИЗ ПРЕДЛАГАЕМОГО РЕШЕНИЯ.....	26
4 Тестирование .....	28
4.1 ТЕСТИРОВАНИЕ .....	28
4.1.1 Работоспособность уничтожающего потребления.....	28
4.1.2 Корректность проверки выполнения уничтожающего потребления .	30
4.1.3 Влияние уничтожающего потребления на нефункциональные характеристики программ.....	32
4.1.4 Работоспособность в распределенной среде .....	38
Заключение .....	41
Список использованных источников и литературы .....	43
Приложение А .....	45
Приложение Б.....	53

## ВВЕДЕНИЕ

Управление памятью – это важный аспект разработки любого программного обеспечения. От этого зависит не только работоспособность программы, но и ее эффективность, так как из-за недостаточно хорошего использования памяти могут ухудшиться нефункциональные характеристики программы. Например, возможно увеличение времени выполнения или потребление памяти. Последнее может привести к замедлению и даже прекращению работы не только данной программы, но и всей системы в целом.

Существуют различные подходы к управлению памятью. Одним из них является низкоуровневое ручное управление. Подобный способ предоставляет возможность достичь оптимальной (или близкой к оптимальной) скорости работы, но вместе с этим требует от программиста высокой квалификации и больших трудозатрат. Другим подходом является автоматическое управление памятью, например, сборка мусора. Этот вариант обычно уступает первому в скорости выполнения, но выигрывает в удобстве и скорости разработки программ. Стоит отметить, что такой способ зачастую тяжелее реализовать. В качестве третьего подхода можно выделить ручное управление памятью на достаточно высоком уровне абстракции. Этот вариант не требует высокой квалификации для использования, имеет приемлемую сложность реализации, может приближаться по скорости к низкоуровневым средствам, а также имеет потенциал для автоматизации. К последнему способу относится директивное управление памятью, то есть управление памятью, основанное на прямых предписаниях системе.

Проблема эффективного управления памятью не имеет общего решения, но возможны решения в частных случаях. Поэтому целесообразно будет рассмотреть проблему на примере системы фрагментированного программирования LuNA.

Система фрагментированного программирования LuNA [1, 2] предназначена для создания параллельных программ численного моделирования

на суперкомпьютерах, чтобы специалист мог создавать эффективные параллельные программы и при этом не обладать высокой квалификацией в области параллельного программирования. Это достигается благодаря автоматической сборке параллельной программы по ее высокоуровневому описанию. Система базируется на технологии фрагментированного программирования. Но LuNA обладает некоторыми недостатками, из-за которых производительность программ для нее уступает аналогичным программам, написанным с использованием низкоуровневых средств параллельного программирования. В частности, это связано с тем, что на данном этапе развития система недостаточно эффективно управляет памятью, а также предоставляет ограниченный набор инструментов для ручного управления памятью.

**Цель** работы заключается в разработке директивных средств управления памятью для системы LuNA, которые будут способны приблизить расход памяти LuNA-программ к ее расходу программами, написанными с использованием низкоуровневых средств параллельного программирования.

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

1. Выполнить анализ существующих средств директивного управления памятью в системе LuNA;
2. Сформировать список требований к разрабатываемым средствам директивного управления памятью;
3. Разработать и реализовать в системе LuNA средства директивного управления памятью;
4. Экспериментально определить нефункциональные характеристики результата.

**Научная новизна** данной работы заключается в реализации средства директивного управления памятью, основанного на механизме уничтожающего потребления и учитывающего специфику системы LuNA.

**Практическая ценность** состоит в том, что разрабатываемые в рамках работы средства директивного управления памятью обеспечивают системе LuNA возможность более эффективно использовать память, что положительно скажется на производительности программ, написанных для нее.

Работа представлена в четырех главах. В первой главе рассматриваются существующие решения и их применимость в системе LuNA. Во второй главе описывается проблема, формулируется список требований к решению и описывается предлагаемое решение. Третья глава посвящена деталям реализации и обсуждению полученных результатов. В четвертой главе описываются методы тестирования решения и приводятся его результаты.

## 1 Анализ предметной области

Рассмотрим некоторые из существующих решений, позволяющие пользователю создавать параллельные программы без необходимости применять низкоуровневые способы их параллельной реализации. Анализ будем осуществлять с точки зрения организации памяти, а также различных способов управления памятью. Так как система LuNA предназначена для создания программ, выполняющих распределенные вычисления, то основным требованием к разрабатываемым средствам выступает необходимость функционирования на вычислителях с распределенной памятью.

Язык программирования **SISAL** [3, 4, 5] – это функциональный язык с неявным параллелизмом и единственным присваиванием. Неявный параллелизм реализован в виде циклов, где каждая итерация может выполняться независимо. Компилятор SISAL конвертирует программу в графы потоков данных и выполняет возможные оптимизации. После этого запускается исполнительная система, при старте которой выделяется большой участок памяти и не возвращается системе до конца исполнения программы. Если же для выполнения программы требуется больше памяти, то программист должен специально указать необходимое количество, иначе программа завершится с ошибкой при превышении объема. Особенно важно отметить, что язык SISAL можно применять только в системах с разделяемой памятью, в то время как система LuNA разработана для запуска программ в системах с распределенной памятью.

**T-система** [6, 7, 8] – среда программирования с поддержкой автоматического динамического распараллеливания программ. Для написания программ используется язык T++, представляющий собой C++ с добавлением нескольких ключевых слов. По сути, программисту необходимо написать алгоритм в функциональном стиле с использованием функций без побочных эффектов («чистых» T-функций), а организацией параллельных вычислений займется T-система. Для обмена данными между узлами применяется модель объектно-ориентированной распределенной общей памяти [9], в основе которой

лежит обмен сообщениям с использованием MPI. Также в системе реализована сборка мусора на основе модифицированного алгоритма взвешенного подсчета ссылок. Однако средства управления памятью, используемые в T-системе, не могут быть применены для системы фрагментированного программирования LuNA, так как она поддерживает лишь модель с распределенной памятью.

**Charm++** [10, 11, 12, 13] – это система параллельного программирования на основе C++, реализующая модель мигрирующих объектов и поддерживаемая адаптивной исполнительной системой. Она позволяет реализовать параллелизм данных и параллелизм задач, а также эффективно распределять их по узлам благодаря динамической балансировке нагрузки. Базовой единицей параллельных вычислений, а также «мигрирующим объектом» является «чар» («chare»), который есть, по сути, C++ объект. У «чаров» есть особенные методы, называемые входными («entry»), которые могут вызывать другие «чары». При этом, если вызывающий «чар» и «чар», которому принадлежит вызываемый метод, находятся на разных узлах, то происходит удаленный вызов метода, подразумевающий сериализацию аргументов и управляющих структур и отправку сообщения на другой узел. Если же «чары» расположены на одном узле, то система оптимизирует общение между ними, используя общую память. Charm++ всегда передает аргументы по значению, что означает копирование данных на отправляющей стороне и на принимающей. Это особенно критично для массивов и структур. Вместо передачи аргументов во входной метод напрямую, можно упаковывать их в сообщения («message»), что положительно сказывается на потреблении памяти. После отправки экземпляра сообщения он автоматически удалится, однако управление полученным сообщением полностью лежит на программисте. Помимо этого, система предоставляет **Zero Copy Messaging API** на основе **RDMA**, позволяющий избавиться от копий на стороне отправителя и получателя. Средства управления памятью, применяемые в Charm++, специфичны для этой системы, поэтому не могут непосредственно использоваться в системе LuNA

На данный момент существующие в системе LuNA средства директивного управления памятью не позволяют достаточно эффективно использовать память, а другие реализации среди существующих решений либо не удовлетворяют основному требованию, либо не могут быть встроены непосредственно в систему LuNA. Вследствие чего возникает необходимость реализации средств директивного управления памятью, учитывающих специфику рассматриваемой системы.

## **2 Алгоритм уничтожающего потребления**

### **2.1 Термины и определения**

Рассмотрим термины и принципы, на которых основано фрагментированное программирование, а также некоторые особенности фрагментированного программирования в контексте системы LuNA. Фрагменты кода (ФК) – процедуры без побочных эффектов, из которых конструируется параллельная программа. ФК имеют конечное число входных и выходных аргументов. Структурированные ФК представляют собой фрагментированные подпрограммы, а атомарные ФК являются последовательными процедурами. Фрагмент данных (ФД) – один элемент из множества входных и выходных аргументов, к которым применяется ФК. Фрагмент вычислений (ФВ) – любое подобное применение. ФД являются переменными единственного присваивания и в каждый момент времени имеют одно из двух состояний: вычисленные или не вычисленные. ФВ готов к исполнению, как только все входные ФД вычислены. Во время исполнения соответствующий ФК применяется ко входным ФД, что приводит к вычислению выходных ФД. Фраgmentированный алгоритм (ФА) - это описание множеств ФВ и ФД. Фраgmentированный алгоритм можно условно представить в виде ориентированного графа, где узлами являются ФД и ФВ, а дугами – зависимости по данным между ними (Рисунок 1). Фраgmentированная программа (ФП) – это параллельная программа, реализующая ФА. ФП может быть описана на любом языке с использованием стандартных средств параллельного программирования.

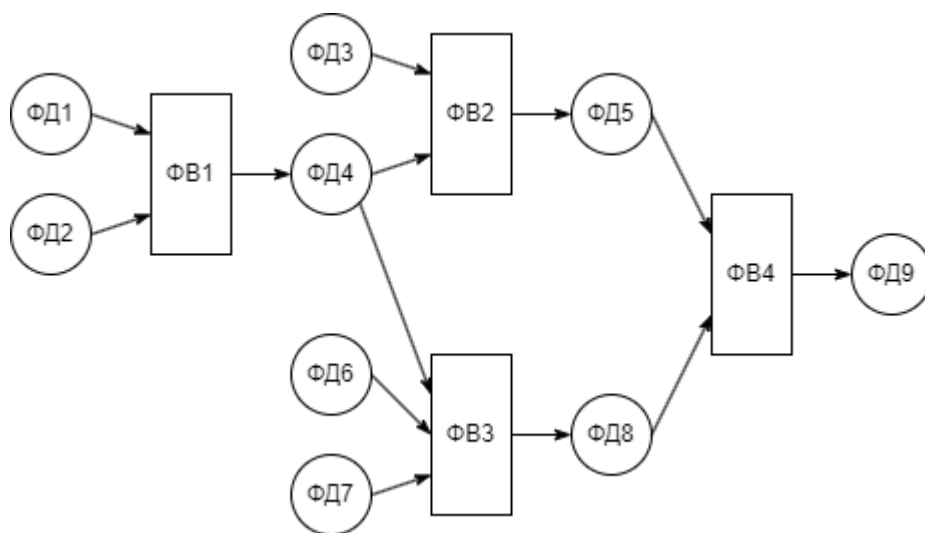


Рисунок 1 – Условное представление ФА

Стоит отметить, что ФВ и ФД являются крупноблочными объектами. Это означает, что ФВ может быть сложной процедурой, работающей с крупными структурами данных. ФД, например, может представлять подобные данные: его значением может быть не просто число, а, например, массив произвольной размерности, объект и т.п. Например, в системе LuNA поддерживаются следующие типы данных: `int` – целое число; `real` – вещественное число; `string` – строка; `value` – пользовательский тип, значение ФД этого типа соответствует непрерывному блоку в памяти конечного размера; `name` – ссылочный тип, используется для адресации входных и выходных ФД. Последние два типа являются примерами крупноблочных объектов. Использование таких крупноблочных объектов позволяет сократить накладные расходы на представление множества ФВ и ФД во время исполнения ФА.

ФД являются объектами единственного присваивания. Это означает, что ФД после инициализации может быть считан неограниченное число раз, но не может быть изменен. Если возникает потребность в модификации значения ФД, то результат должен быть записан в новый ФД.

Система LuNA позволяет конструировать ФП автоматически по описанию ФА. Это ускоряет и упрощает разработку, отладку и модификацию программ, так как система выполняет за пользователя задачи, связанные с низкоуровневым

параллельным программированием (например, коммуникации, конкурентный доступ к памяти и т.п.). Описание ФА выполняется на языке LuNA.

На текущем этапе развития системных алгоритмов эффективность программ, написанных для системы LuNA, может существенно уступать реализациям с применением низкоуровневых средств параллельного программирования. По этой причине в системе LuNA присутствуют инструменты управления процессом конструирования и исполнения ФП, называемые рекомендациями и директивами. Разница между ними заключается в том, что рекомендации являются безопасными подсказками системе, которые она может заменить или проигнорировать. Неправильное использование рекомендаций в худшем случае приведет к снижению эффективности программы. Директивы же, в свою очередь, представляют собой обязательные предписания системе, из-за которых работоспособность программы может быть нарушена. С помощью этих средств программист может задать размещение ФД на узлах, условия удаления ФД и т.п. В качестве примера рассмотрим директиву `req_count`. Она указывает, сколько раз данный ФД будет потреблен. Когда количество потреблений станет равным значению директивы, ФД будет удален.

Система LuNA исполняет ФА следующим образом. Она порождает множество ФВ и ФД, описанных в ФА, после чего распределяет их по узлам мультимпьютера. Система перемещает ФД таким образом, чтобы ФВ и его входные ФД оказались на одном узле. После того, как все входные для ФВ фрагменты данных вычислены и располагаются на одном узле с ним, происходит исполнение ФК, соответствующего ФВ. В результате вычисляются выходные ФД данного ФВ и распределяются по узлам мультимпьютера, а сам ФВ удаляется с узла. Такой подход позволяет выполнять ФВ одновременно в рамках информационных зависимостей.

## **2.2 Анализ проблемы**

Программы, написанные для системы LuNA, по эффективности могут значительно уступать программам, написанным с помощью низкоуровневых

средств параллельного программирования. Под эффективностью подразумеваются различные нефункциональные характеристики программы, например, время выполнения, потребление памяти и нагрузка на сеть. Сокращение подобного отставания является важной задачей развития системы. На момент написания работы в системе уже существуют различные подходы и инструменты для повышения эффективности реализуемых программ. В частности, к ним относятся средства директивного управления памятью, позволяющие использовать опыт программиста для написания прямых указаний системе. На основе данных указаний система осуществляет сборку мусора, что предотвращает утечки памяти. Но данных средств недостаточно для эффективного использования памяти, что особенно сильно проявляется в итерационных алгоритмах. Так как в системе LuNA ФД являются объектами единственного присваивания, то даже при необходимости увеличить значение ФД на единицу в ФА будут описаны два разных ФД – с прежним и увеличенным значением. Наивная реализация ФД подразумевает, что каждый из них занимает свой отдельный участок памяти. Это может привести к утечке памяти, если модификация ФД будет происходить итеративно. Разумеется, утечек памяти можно избежать, если вовремя удалять ФД с прежним значением. Но даже с учетом этого, выделение памяти под новое значение, копирование данных, удаление ФД и освобождение памяти занимает существенное количество времени. Кроме того, существует момент времени, когда и старый, и новый ФД расположены памяти, что может быть недопустимо для ФД большого размера.

Для более эффективного использования памяти в итерационных алгоритмах целесообразно будет использовать буфер памяти одного ФД другим. Под использованием буфера подразумевается операция по размещению значения одного ФД в тот же участок памяти, где было расположено значение другого ФД. Но нужно учитывать, что передача буфера потенциально может привести к негативным побочным эффектам. В качестве примера рассмотрим следующую ситуацию. После исполнения некоторого фрагмента вычислений

значение ФД с именем Б расположено в буфере, который прежде занимало значение ФД с именем А. Если после этого программа будет использовать этот буфер памяти, полагая, что в нем находится значение ФД с именем А, то ее работоспособность может быть нарушена. Чтобы система могла обнаружить подобные манипуляции и по возможности выполнить необходимые проверки, передачу буфера нужно совершать открыто для нее. Но нужно следить, чтобы информация, которой располагает система, соответствовала реальной ситуации в время вычислений. Иначе может произойти снижение эффективности программы и даже системная ошибка. В частности, в том случае, когда система принимает другие решения, основываясь на такой информации.

Перенос буфера целесообразно делать только для ФД типа value. Конечно, ФД строкового типа также могут использовать значительный объем памяти, но в случае необходимости их можно реализовать в виде ФД типа value.

При проектировании директивных средств управления памятью необходимо учитывать возможность их использования в автоматическом режиме. Такой способ может быть более перспективными, особенно с точки зрения удобства и ускорения разработки.

### **2.3 Постановка задачи**

В рамках системы LuNA задача состоит в расширении модели вычислений системы механизмом использования буфера одного ФД другим, причем информация о таком переносе должна быть открыта для системы.

### **2.4 Требования к решению**

С учетом проведенного анализа, можно предъявить следующие требования к решению:

1. Предлагаемое решение должно расширить модель вычислений системы LuNA механизмом использования буфера одного ФД другим;
2. Предлагаемое решение должно функционировать на вычислителях с распределенной памятью;

3. Предлагаемое решение должно предоставлять системе информацию о своем применении;
4. Информация о применении предлагаемого решения должна соответствовать реальной ситуации во время вычислений;
5. Предлагаемое решение должно поддерживать возможность автоматического использования в перспективе.

### 2.5 Описание решения

В качестве решения предлагается расширить модель вычислений системы LuNA механизмом, который позволяет записывать новое значение, принадлежащее выходному ФД, в участок памяти, занимаемый значением входного ФД. Назовем такой механизм уничтожающим потреблением, входной ФД – поглощаемым, а выходной ФД – поглощающим. Тогда модель вычислений может выглядеть следующим образом (Рисунок 2):

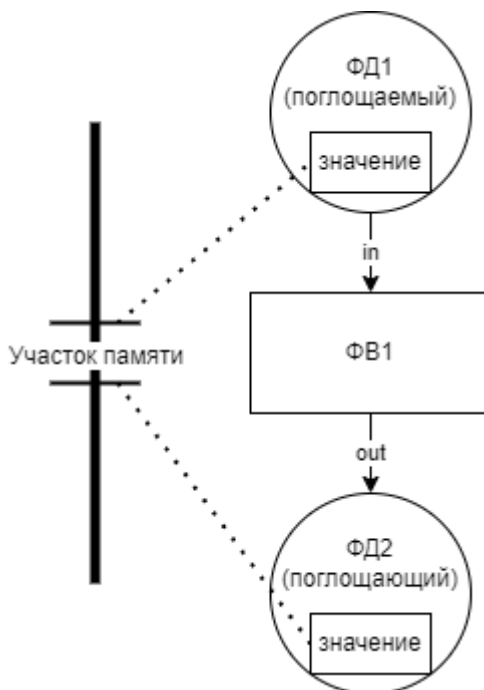


Рисунок 2 – Пример расширенной модели вычислений

Причем для каждого ФД должен существовать только один ФВ, в котором он является поглощаемым. Это необходимо, так как в противном случае каждое уничтожающее потребление одного и того же поглощаемого ФД может

приводить к неявной модификации поглощающих ФД из предыдущих уничтожающих потреблений. Поглощающий ФД одновременно является выходным, а значит, из-за принципа единственного присваивания ФД, для него всегда существует только один ФВ, в котором он поглощающий.

Помимо этого, предлагаемое решение предоставляет системе информацию о том, на каком этапе вычислений выполняется уничтожающее потребление и какие ФД в нем участвуют. Также решение предусматривает выполнение проверки того, что имеющаяся в системе информация об уничтожающем потреблении не противоречит реальному выполнению уничтожающего потребления.

Также предлагаемое решение не исключает возможность автоматического использования, так как ответственность за перенос буфера и генерацию системной информации в перспективе можно возложить на систему.

## 3 Реализация

### 3.1 Устройство системы LuNA

Прежде чем приступить к рассмотрению реализации уничтожающего потребления, стоит дать представление о некоторых существенных особенностях реализации системы LuNA, необходимых для дальнейшего понимания.

Чтобы написать программу для системы LuNA, пользователю необходимо описать ФА в файле с расширением `.fa`. Используемые в алгоритме атомарные ФК реализуются пользователем как процедуры на языке C++ в файлах с расширением `.cpp`. При запуске программы пользователь может указать как директории с такими файлами, так и конкретные файлы. По умолчанию файлы с реализацией атомарных ФК берутся из директории, где расположен файл с описанием ФА.

Абстракция ФД в системе LuNA реализована в классе DF (исходные файлы `df.h` и `df.cpp`). Интерфейс DF предоставляет конструкторы на основе различных типов значений ФД, арифметические операторы, методы для получения значений, свойств ФД, сериализации, десериализации и другие. Данный класс, помимо прочего, является умным указателем над значением ФД. Таким образом, он содержит указатель на это значение и счетчик ссылок на него.

Грамматика языка LuNA определена в исходном файле `grammar.ypp`. В этом файле задается синтаксис языка и определяются правила, согласно которым код будет преобразован транслятором в абстрактное синтаксическое дерево.

Сначала транслятор системы LuNA выполняет преобразование описания фрагментированного алгоритма в абстрактное синтаксическое дерево, представленное в формате JSON. Затем транслятор обходит это дерево, для каждого узла генерируя код итоговой фрагментированной программы на языке C++. Код преобразования абстрактного синтаксического дерева в C++ код написан на языке Python (исходный файл `fcmp2`). Особый интерес представляет метод транслятора, генерирующий код исполнения атомарного ФК: именно в

нем необходимо добавить генерацию кода проверки выполнения уничтожающего потребления, которая будет рассмотрена далее.

### 3.2 Реализация уничтожающего потребления

Процесс реализации уничтожающего потребления в системе LuNA можно разделить на несколько этапов:

1. Расширение интерфейса класса DF.
2. Расширение грамматики языка LuNA.
3. Модификация транслятора системы LuNA.

Далее каждый из этапов раскрывается более подробно. Подробнее о программе можно узнать в приложениях А и Б.

#### 3.2.1 Расширение интерфейса класса DF

Для предоставления программисту средства для переноса буфера между ФД интерфейс класса DF расширен соответствующими методами (DF::grab\_buf(DF &df) и DF::grab\_buf(const DF &df)). Они реализованы следующим образом. Выполняется проверка, что поглощаемый ФД имеет тип value. Затем уменьшается число ссылок на текущее значение поглощающего ФД. После этого ФД связывается с новым значением, а число ссылок на это значение увеличивается.

Модифицированный интерфейс ФД позволяет программисту осуществлять перенос буфера одного ФД в другой следующим образом (Рисунок 3):

```
1 void my_func(DF &df2, const DF &df1)
2 {
3     df2.grab_buf(df1);
4 }
```

Рисунок 3 – Листинг атомарного ФК с использованием уничтожающего потребления

Рассмотрим детали реализации методов переноса буфера (Рисунок 4):

```

1  void DF::grab_buf(DF &df)
2  {
3      assert(df.type_==TYPE_VALUE);
4      _unlink();
5      _link(df);
6  }
7
8
9  void DF::grab_buf(const DF &df)
10 {
11     assert(df.type_==TYPE_VALUE);
12     _unlink();
13     _link(df);
14 }

```

Рисунок 4 – Методы переноса буфера. Файл df.cpp

Проверка типа ФД представлена на строках 3 и 11. Уменьшение числа ссылок на текущее значение представлено на строках 4 и 12. Связывание с новым значением и увеличение числа ссылок на него представлено на строках 5 и 13.

### 3.2.2 Расширение грамматики языка LuNA

Для того, чтобы предоставить возможность указать системе на использование уничтожающего потребления, грамматика языка была расширена способом объявления атомарного ФК, снабженного директивами и/или рекомендациями. Также в грамматику добавлен способ указать имена формальных параметров атомарного ФК.

Модифицированная грамматика позволяет обозначать уничтожающее потребление в описании фрагментированного алгоритма следующим образом (Рисунок 5):

```

1  import my_func(name a, value b) as my_func @ {
2      | destruct b => a;
3  };

```

Рисунок 5 – Объявление атомарного ФК с уничтожающим потреблением в описании ФА

Такое объявление атомарного ФК преобразуется транслятором в следующее абстрактное синтаксическое поддерево (Рисунок 6):

```

1  "my_func": {
2      "type": "extern",
3      "code": "my_func",
4      "args": [
5          {
6              "type": "name",
7              "name": "a"
8          },
9          {
10             "type": "value",
11             "reuse": "False",
12             "name": "b"
13         }
14     ],
15     "rules": [
16         {
17             "ruletype": "map",
18             "begin": 102,
19             "type": "rule",
20             "property": "destruct",
21             "id": [
22                 "b"
23             ],
24             "expr": {
25                 "is_expr": true,
26                 "type": "id",
27                 "ref": [
28                     "a"
29                 ],
30                 "begin": 116
31             }
32         }
33     ],
34     "cuda_support": "false",
35     "nocpu": "false"
36 },

```

Рисунок 6 – Абстрактное синтаксическое поддерево, построенное по объявлению атомарного ФК

Как можно наблюдать, узел дерева содержит информацию о директивах (и/или рекомендациях) в свойстве `rules` (строка 15). В данном примере единственной директивой является описание уничтожающего потребления. Существенно важной информацией являются имя директивы в свойстве `property` и имена поглощающего и поглощаемого аргумента в свойствах `id` (строка 22) и

expr.ref (строка 28) соответственно. Остальные свойства необходимы для корректной работы прочих механизмов системы LuNA.

Детально рассмотрим изменения в грамматике языка (Рисунок 7):

```
1 sub_def:
2 <...>
3 | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id opt_behavior SCOLON
4 {
5     ostream os;
6
7
8     os << "\"" << $7 << "\" : {"
9         << "\"type\" : \"extern\","
10        << "\"code\" : \"" << $2 << "\",\"
11        << "\"args\" : [" << $4 << "\",\"
12        << "\"rules\" : [" << $8 << "\",\"
13        << "\"cuda_support\" : \"false\","
14        << "\"nocpu\" : \"false\""
15        << "}" << endl;
16
17
18    $$ = os.str();
19 }
20 <...>
21 opt_ext_params:
22     ext_params_seq
23     | { $$ = ""; }
24     ;
25
26
27 ext_params_seq:
28     type code_df
29     {
30         ostream os;
31
32
33         os << "{" << "type\":" << $1 << ","
34         << "\"name\":" << $2 << "\"}";
35
36
37         $$ = os.str();
38     }
39     | ext_params_seq COMMA type code_df
40     {
41         ostream os;
42
43
44         os << $1
45         << "," << "type\":" << $3 << ","
46         << "\"name\":" << $4 << "\"}";
47
48
49         $$ = os.str();
50     }
51     ;
```

Рисунок 7 – Модификация грамматики. Файл grammar.ypp

Разберем основные токены грамматики. Токен `sub_def` (строка 1) используется для описания фрагментов кода. Токен `KW_IMPORT` (строка 3) применяется для объявления атомарных ФК. Токен `opt_behaviour` (строка 3) как раз соответствует описанию директив и рекомендаций, одной из которых и является уничтожающее потребление. Токен `opt_ext_params` (строки 3 и 21) используется для описания формальных параметров атомарного ФК. Далее приведена таблица, задающая соответствие между токенами грамматики и лексемами объявления атомарного ФК из примера на рисунке 5 (Таблица 1).

Таблица 1 – Соответствие между токенами грамматики и лексемами объявления атомарного ФК

Токен грамматики	Лексема объявления атомарного ФК
<code>sub_def.KW_IMPORT</code>	<code>import</code>
<code>sub_def.code_id</code>	<code>my_func</code>
<code>sub_def.LB</code>	<code>(</code>
<code>sub_def.opt_ext_params</code>	<code>name a, name b</code>
<code>sub_def.RB</code>	<code>)</code>
<code>sub_def.KW_AS</code>	<code>as</code>
<code>sub_def.code_id</code>	<code>my_func</code>
<code>sub_def.opt_behaviour</code>	<code>@ { destruct b =&gt; a; }</code>
<code>sub_def.SCOLON</code>	<code>;</code>
<code>opt_ext_params.ext_params_seq.type</code>	<code>name</code>
<code>opt_ext_params.ext_params_seq.code_df</code>	<code>a</code>

### 3.2.3 Модификация транслятора системы LuNA

Для контроля выполнения объявленного уничтожающего потребления в транслятор системы LuNA была добавлена функциональность для преобразования информации об уничтожающем потреблении из абстрактного синтаксического дерева в C++ код проверки выполнения. Таким образом, если программист в коде LuNA программы отметил атомарный ФК как содержащий уничтожающее потребление, то после исполнения данного ФК система будет проверять, что захват буфера действительно произошел.

Модифицированный транслятор выполняет генерацию проверки выполнения уничтожающего потребления следующим образом (Рисунок 8):

```
1 // After requests: cf_l15: DC(consumer, consumed);
2 BlockRetStatus block_5(CF &self)
3 {
4     {
5         DF _out_0;
6         void *consumed_ptr_0 = self.wait(self.id(0)).get_data();
7         // EXEC_EXTERN cf_l15: DC(consumer, consumed);
8         destructive_consuming(
9             // name consumer
10            _out_0,
11            // value consumed
12            self.wait(self.id(0)));
13
14        {
15            void *consumer_ptr_0 = _out_0.get_data();
16            if (consumed_ptr_0 != consumer_ptr_0)
17            {
18                ABORT("Destructive consuming was declared, but not executed");
19            }
20        }
21        {
22            DF stored=_out_0;
23            self.store(self.id(1), stored);
24        }
25    }
26
27    // req_count consumer=1
28    { DF posted=self.wait(self.id(1));
29      self.post(self.id(1), posted, CyclicLocator(0), 1);
30    }
31    return EXIT;
32 }
```

Рисунок 8 – Сгенерированный транслятором C++ код, соответствующий применению ФК с уничтожающим потреблением

На строке 6 происходит инициализация указателя на данные поглощаемого ФД. Вызов ФК осуществляется на строках 8-12. На строке 15 инициализируется указатель на данные поглощающего ФД. Сравнение указателей и завершение программы с ошибкой представлено на строках 16-19.

Код генерации проверки на псевдокоде выглядит следующим образом (Рисунок 9):

```
1  START
2  |   FOR аргументы, к которым применяется атомарный ФК DO
3  |   |   IF аргумент имеет тип name
4  |   |   |   IF аргумент участвует в УП в роли поглощающего
5  |   |   |   |   сгенерировать код сохранения указателя на данные поглощающего аргумента
6  |   |   |   |   ENDIF
7  |   |   |   ELSE IF аргумент имеет тип value
8  |   |   |   |   IF аргумент участвует в УП в роли поглощаемого
9  |   |   |   |   |   сгенерировать код сохранения указателя на данные поглощаемого аргумента
10 |   |   |   |   ENDIF
11 |   |   |   ELSE
12 |   |   |   |   ...
13 |   |   |   ENDIF
14 |   |   ENDFOR
15 |   FOR пары, связанные УП DO
16 |   |   сгенерировать код проверки равенства указателей
17 |   ENDFOR
18 END
```

Рисунок 9 – Алгоритм генерации проверки выполнения уничтожающего потребления

Подробнее рассмотрим реализованные модификации (Рисунок 10):



Генерация кода проверки равенства указателей представлена на строках 54-57, что соответствует строкам 15-17 алгоритма.

Реализованный метод `get_dc_pairs` (Рисунок 11) для текущего атомарного ФК создает массив триплетов вида {номер триплета, имя поглощаемого аргумента, имя поглощающего аргумента} на основе данных об уничтожающем потреблении в AST-дереве.

```
1 def get_dc_pairs(sub, ja, scope):
2     if 'rules' not in sub or [x for x in sub['rules'] if x['property'] != 'destruct']:
3         return {}
4
5     dc_pairs = []
6     destruct_rules = [x for x in sub['rules'] if x['property'] == 'destruct']
7     for idx, rule in enumerate(destruct_rules):
8         consumed = rule['id'][0]
9         consumedArgNum = sub['args'].index(next(x for x in sub['args'] if x['name'] == consumed))
10        consumedRef = ref1(ja['args'][consumedArgNum]['ref'], scope)
11
12        consumer = rule['expr']['ref'][0]
13        consumerArgNum = sub['args'].index(next(x for x in sub['args'] if x['name'] == consumer))
14        consumerRef = ref1(ja['args'][consumerArgNum]['ref'], scope)
15
16        dc_pairs.append({'idx': idx, 'consumed': consumedRef, 'consumer': consumerRef})
17
18    return dc_pairs
```

Рисунок 11 – Модификации транслятора. Файл `fcmp2`

### 3.3 Анализ предлагаемого решения

Рассмотрим достоинства предлагаемого решения, а также его недостатки и направления дальнейших работ.

Среди плюсов решения можно выделить удобство использования: для применения уничтожающего потребления достаточно добавить соответствующую директиву к объявлению атомарного ФК, а в его реализации воспользоваться предоставленным методом переноса буфера. Также данное решение либо расширяет существующую функциональность, либо вносит модификации, не влияющие на нее. Таким образом обеспечивается обратная совместимость с программами, написанными для прошлых версий системы. Представленное решение расширяет грамматику языка таким образом, что появляется основа для использования других директив и рекомендаций при объявлении атомарных ФК в будущем. Помимо этого, решение выполняет проверку выполнения объявленного уничтожающего потребления. Это особенно

важно при развитии инструмента, так как в дальнейшем исполнительная система может принимать решения, опираясь на информацию об уничтожающем потреблении.

Из недостатков можно отметить тот факт, что на данном этапе решение не исключает возможность неправильного использования, из-за которого работоспособность программы будет нарушена. Ответственность за корректное использование уничтожающего потребления лежит на программисте. Система не сможет отследить любое ошибочное использование УП, так как C++ код не поддается анализу в общем случае. Но в некоторых частных случаях это возможно. Помимо реализованной мной проверки факта выполнения уничтожающего потребления, стоит исследовать возможность разработки других способов контроля использования данного решения. Например, система может проверять, будет ли поглощаемый ФД использован в других ФВ после уничтожающего потребления. Подобные проверки могут быть как динамическими, так и статическими. Причем последние могут анализировать не только описание ФА, но и атомарные ФК на C++.

Представленное решение необходимо применять вручную, тогда как автоматизация использования положительно бы сказалась на удобстве для пользователя. Поэтому стоит рассмотреть возможность использования разработанного средства автоматически исполнительной системой. Возможно, это удастся реализовать лишь частично, например, перенос буфера будет выполнять система, а от программиста потребуются только объявить уничтожающее потребление.

## 4 Тестирование

### 4.1 Тестирование

Для проверки работоспособности уничтожающего потребления был реализован ряд тестов на языке LuNA. В качестве примера приведены несколько основных тестовых программ.

#### 4.1.1 Работоспособность уничтожающего потребления

Данная программа тестирует работоспособность уничтожающего потребления. При штатной работе данные поглощенного и поглощающего ФД должны быть расположены в одном участке памяти. Также должны быть сгенерированы все метаданные об уничтожающем потреблении и проверка со стороны исполнительной системы.

Программа последовательно выполняет три ФК. Первый ФК инициализирует поглощаемый ФД целочисленным массивом, выводит его элементы и значение указателя на данные этого ФД. Второй ФК выполняет уничтожающее потребление, изменяет и выводит значение поглощающего ФД и значение указателя на его данные. Третий ФК выводит значение поглощающего ФД и значение указателя на его данные (Рисунки 12-13).

```
1  import init_var(name, int, int) as init_var;
2  import destructive_consuming(name a, value b) as DC @ {
3  |   destruct b => a;
4  };
5  import check_destructive_consuming(value) as DC_check;
6
7  sub main()
8  {
9  |   df consumer, consumed;
10
11 |   init_var(consumed, 2, 0) @ {
12 |     req_count consumed=1;
13 |   };
14
15 |   DC(consumer, consumed) @ {
16 |     req_count consumer=1;
17 |   };
18
19 |   DC_check(consumer);
20 }
```

Рисунок 12 – Листинг тестовой программы, описание ФА

Объявление уничтожающего потребления представлено на строке 3. Исполнение ФК с уничтожающим потреблением представлено на строке 16.

```
1  #include <cstdio>
2  #include "ucenv.h"
3
4  extern "C"
5  {
6
7  void init_var(DF &consumed, int val1, int val2)
8  {
9      printf("INIT_VAR\n");
10
11     int *arr = (int*) malloc(2 * sizeof(int));
12     arr[0] = val1;
13     arr[1] = val2;
14     consumed.create<int>(2);
15     memcpy(consumed.getData<int>(), arr, 2 * sizeof(int));
16     free(arr);
17
18     printf("Value in consumed: [%d, %d]\n", consumed.getData<int>()[0], consumed.getData<int>()[1]);
19     printf("Consumed DF data stored in: %p\n\n", (void*) consumed.getData<int>());
20 }
21
22 void destructive_consuming(DF &consumer, DF &consumed)
23 {
24     printf("DESTRUCTIVE CONSUMING\n");
25
26     consumer.grab_buf(consumed);
27     printf("Consumer DF grab buffer of consumed DF and increment values\n");
28     consumer.getData<int>()[0]++;
29     consumer.getData<int>()[1]++;
30
31     printf("Value in consumer:: [%d, %d]\n", consumer.getData<int>()[0], consumer.getData<int>()[1]);
32     printf("Consumer DF data stored in: %p\n\n", (void*) consumer.getData<int>());
33 }
34
35 void check_destructive_consuming(DF &consumer)
36 {
37     printf("CHECK DESTRUCTIVE CONSUMING\n");
38
39     printf("Value in consumer:: [%d, %d]\n", consumer.getData<int>()[0], consumer.getData<int>()[1]);
40     printf("Consumer DF data stored in: %p\n", (void*) consumer.getData<int>());
41 }
42
43 }
```

Рисунок 13 – Листинг тестовой программы, реализация ФК

Инициализация поглощаемого ФД представлена на строках 14-15. Передача буфера поглощаемого ФД поглощающему представлена на строке 26. Запуск программы дал следующие результаты (Рисунок 14):

```
INIT_VAR
Value in consumed: [2, 0]
Consumed DF data stored in: 0x7f4b94000fa0

DESTRUCTIVE CONSUMING
Consumer DF grad buffer of consumed DF and increment values
Value in consumer:: [3, 1]
Consumer DF data stored in: 0x7f4b94000fa0

CHECK DESTRUCTIVE CONSUMING
Value in consumer:: [3, 1]
Consumer DF data stored in: 0x7f4b94000fa0
```

Рисунок 14 – Результат запуска тестовой программы

Вывод программы позволяет установить, данные поглощенного и поглощающего ФД расположены в одном участке памяти. На этом основании можно сделать вывод, что уничтожающее потребление выполнено успешно. Также сгенерированные метаданные и код проверки совпадают с представленными на изображениях 6 и 8.

#### **4.1.2 Корректность проверки выполнения уничтожающего потребления**

Данная программа тестирует правильность реализованной проверки со стороны исполнительной системы. Если пользователь объявил атомарный ФК как содержащий уничтожающее потребление, но в реализованном ФК не выполнил передачу буфера поглощаемого ФД поглощающему, то выполнение программы должно прерваться с ошибкой.

Программа последовательно выполняет 2 ФК. Первый ФК инициализирует поглощаемый ФД целочисленным массивом, выводит его элементы и значение указателя на данные этого ФД. Второй ФК вместо уничтожающего потребления копирует данные поглощаемого ФД в поглощающий и выводит значение поглощающего ФД и значение указателя на его данные (Рисунки 15-16).

```

1  import init_var(name, int, int) as init_var;
2  import fake_destructive_consuming(name a, value b) as fake_DC @ {
3  |   destruct b => a;
4  };
5
6  sub main()
7  {
8  |   df consumer, consumed;
9
10 |   init_var(consumed, 2, 0) @ {
11 |     req_count consumed=1;
12 |   };
13
14 |   fake_DC(consumer, consumed);
15 }

```

Рисунок 15 – Листинг тестовой программы, описание ФА

Объявление поглощающего потребления представлено на строке 3. Исполнение ФК с копированием значения ФД представлено в строке 14.

```

1  #include <stdio>
2  #include "ucenv.h"
3
4  extern "C"
5  {
6
7  void init_var(DF &consumed, int val1, int val2)
8  {
9  |   printf("INIT_VAR\n");
10
11 |   int *arr = (int*) malloc(2 * sizeof(int));
12 |   arr[0] = val1;
13 |   arr[1] = val2;
14 |   consumed.create<int>(2);
15 |   memcpy(consumed.getData<int>(), arr, 2 * sizeof(int));
16 |   free(arr);
17
18 |   printf("Value in consumed: [%d, %d]\n", consumed.getData<int>()[0], consumed.getData<int>()[1]);
19 |   printf("Consumed DF data stored in: %p\n\n", (void*) consumed.getData<int>());
20 }
21
22 void fake_destructive_consuming(DF &consumer, DF &consumed)
23 {
24 |   printf("FAKE DESTRUCTIVE CONSUMING\n");
25
26 |   consumer.create<int>(2);
27 |   memcpy(consumer.getData<int>(), consumed.getData<int>(), 2 * sizeof(int));
28
29 |   printf("Value in consumer:: [%d, %d]\n", consumer.getData<int>()[0], consumer.getData<int>()[1]);
30 |   printf("Consumer DF data stored in: %p\n\n", (void*) consumer.getData<int>());
31 }
32
33 }

```

Рисунок 16 – Листинг тестовой программы, реализация ФК

Инициализация поглощаемого ФД представлена на строках 14-15. Копирование буфера поглощаемого ФД в поглощающий представлено на строках 26-27.

Запуск программы дал следующие результаты (Рисунок 17):

```
1  INIT_VAR
2  Value in consumed: [2, 0]
3  Consumed DF data stored in: 0x7f8ff8000fa0
4
5  FAKE DESTRUCTIVE CONSUMING
6  Value in consumer:: [2, 0]
7  Consumer DF data stored in: 0x7f8ffc000d70
8
9  luna: fatal error: run-time error: errcode=-6
10 err> 0 ERROR: Destructive consuming was declared, but not executed
11 /home/andrey/luna/scripts/./build/programs/home/andrey/diploma/DC_rts_check/DC_fa/test.cpp:112
12 err> 0 ABORT
13 err> terminate called after throwing an instance of 'RuntimeError'
14 err> what(): std::exception
15 err> [DESKTOP-U2LLOQD:06557] *** Process received signal ***
16 err> [DESKTOP-U2LLOQD:06557] Signal: Aborted (6)
17 err> [DESKTOP-U2LLOQD:06557] Signal code: (-6)
18 err> [DESKTOP-U2LLOQD:06557] [ 0] /lib/x86_64-linux-gnu/libc.so.6(+0x42520)[0x7f901251e520]
19 err> [DESKTOP-U2LLOQD:06557] [ 1] /lib/x86_64-linux-gnu/libc.so.6(pthread_kill+0x12c)[0x7f9012572a7c]
20 err> [DESKTOP-U2LLOQD:06557] [ 2] /lib/x86_64-linux-gnu/libc.so.6(raise+0x16)[0x7f901251e476]
21 err> [DESKTOP-U2LLOQD:06557] [ 3] /lib/x86_64-linux-gnu/libc.so.6(abort+0xd3)[0x7f90125047f3]
22 err> [DESKTOP-U2LLOQD:06557] [ 4] /lib/x86_64-linux-gnu/libstdc++.so.6(+0xa2bbe)[0x7f90127c8bbe]
23 err> [DESKTOP-U2LLOQD:06557] [ 5] /lib/x86_64-linux-gnu/libstdc++.so.6(+0xae24c)[0x7f90127d424c]
24 err> [DESKTOP-U2LLOQD:06557] [ 6] /lib/x86_64-linux-gnu/libstdc++.so.6(+0xae2b7)[0x7f90127d42b7]
25 err> [DESKTOP-U2LLOQD:06557] [ 7] /lib/x86_64-linux-gnu/libstdc++.so.6(+0xae518)[0x7f90127d4518]
26 err> [DESKTOP-U2LLOQD:06557] [ 8] /home/andrey/luna/scripts/./build/programs/home/andrey/diploma/DC_rts_check/DC_fa/libucodes.so
27 (+0x2709)[0x7f900942c709]
28 err> [DESKTOP-U2LLOQD:06557] [ 9] /home/andrey/luna/scripts/./lib/librts.so(+0x3b233)[0x7f9012ae2233]
29 err> [DESKTOP-U2LLOQD:06557] [10] /home/andrey/luna/scripts/./lib/librts.so(_ZN10ThreadPool7routineEv+0x17d)[0x7f9012ae9eed]
30 err> [DESKTOP-U2LLOQD:06557] [11] /lib/x86_64-linux-gnu/libstdc++.so.6(+0xdc2b3)[0x7f90128022b3]
31 err> [DESKTOP-U2LLOQD:06557] [12] /lib/x86_64-linux-gnu/libc.so.6(+0x94b43)[0x7f9012570b43]
32 err> [DESKTOP-U2LLOQD:06557] [13] /lib/x86_64-linux-gnu/libc.so.6(+0x126a00)[0x7f9012602a00]
33 err> [DESKTOP-U2LLOQD:06557] *** End of error message ***
34 err>
```

Рисунок 17 – Результат запуска тестовой программы

И действительно, программа завершилась с ошибкой именно из-за того, что данные поглощающего ФД расположены в участке памяти, отличном от участка, занимаемого поглощенным ФД.

#### 4.1.3 Влияние уничтожающего потребления на нефункциональные характеристики программ

Данная программа необходима для исследования влияния использования уничтожающего потребления на потребление памяти и время выполнения программ.

Программа последовательно выполняет ФК, количество которых равно числу итераций плюс один. Первый ФК инициализирует целочисленный массив размером 100 МБ. Все последующие ФК умножают все его элементы на 2 и вызывают утилиту top (Рисунок 18-19).

```

1  import init_var(name) as init_var;
2  import multiply_by_2(name out_df, value in_df, int i) as multiply_by_2 @ {
3  |      |      destruct in_df => out_df;
4  };
5
6  #define ITERS 10
7
8  sub main()
9  {
10     df arrays;
11
12     init_var(arrays[0]) @ {
13     |      req_count arrays[0]=1;
14     };
15
16
17     for i = 1 .. $ITERS {
18     |      multiply_by_2(arrays[i], arrays[i-1], i) @ {
19     |      |      req_count arrays[i]=1;
20     |      };
21     }
22 }

```

Рисунок 18 – Листинг тестовой программы, описание ФА

Объявление поглощающего потребления представлено на строке 3. Инициализация ФД целочисленным массивом представлена на строке 12. Итеративное умножение всех элементов массива на 2 представлено на строках 17-21.

```

1  #include <stdio>
2  #include <stdlib.h>
3  #include "ucenv.h"
4
5  #define SIZE 26214400 // 100 MB
6
7  extern "C"
8  {
9
10 void init_var(DF &in_arr)
11 {
12     printf("INIT_VAR\n");
13
14     int *arr = (int*) malloc(SIZE * sizeof(int));
15     for (int i = 0; i < SIZE; i++) {
16         arr[i] = i % 5;
17     }
18     in_arr.create<int>(SIZE);
19     memcpy(in_arr.getData<int>(), arr, SIZE * sizeof(int));
20     free(arr);
21 }
22
23 void multiply_by_2(DF &out_arr, DF &in_arr, int i)
24 {
25     printf("ITERATION %d\n", i);
26     // out_arr.create<int>(SIZE);
27     out_arr.grab_buf(in_arr);
28     for (int i = 0; i < SIZE; i++) {
29         out_arr.getData<int>()[i] = out_arr.getData<int>()[i]*2;
30     }
31     system("top -b -n 1 -e k -E k | grep 'rts\\|COMMAND' ");
32 }
33
34
35 }

```

Рисунок 19 – Листинг тестовой программы, реализация ФК

Инициализация ФД, значением которого является начальный массив, представлено на строках 18-19. Передача буфера текущего массива последующему представлена на строке 27. В реализации без уничтожающего потребления вместо передачи буфера создается новый ФД и выделяется память под новое значение, что представлено на строке 26. Вызов утилиты top представлен на строке 31.

Для подсчета потребляемой памяти программа была запущена на 10-ти итерациях в трех вариациях:

1. Без директив req\_count, из-за чего создаваемые ФД не удаляются до конца выполнения программы, и без уничтожающего потребления;
2. С использованием директив req\_count, из-за чего ФД удаляются после первого использования, и без уничтожающего потребления;
3. С удалением ФД и с уничтожающим потреблением.

Данные о потребляемой памяти получены из столбца VIRT вывода команды top (Таблица 2, Рисунок 20).

Таблица 2 – Потребляемая памяти тестовой программы

Номер итерации	Без удаления ФД, без УП (КБ)	С удалением ФД, без УП (КБ)	С удалением ФД, с УП (КБ)
1	739104	673568	636700
2	841508	739104	636700
3	943912	739104	636700
4	1046316	739104	636700
5	1148720	739104	636700
6	1251124	739104	636700
7	1353528	739104	636700
8	1455932	739104	636700
9	1558336	739104	636700
10	1660740	739104	636700

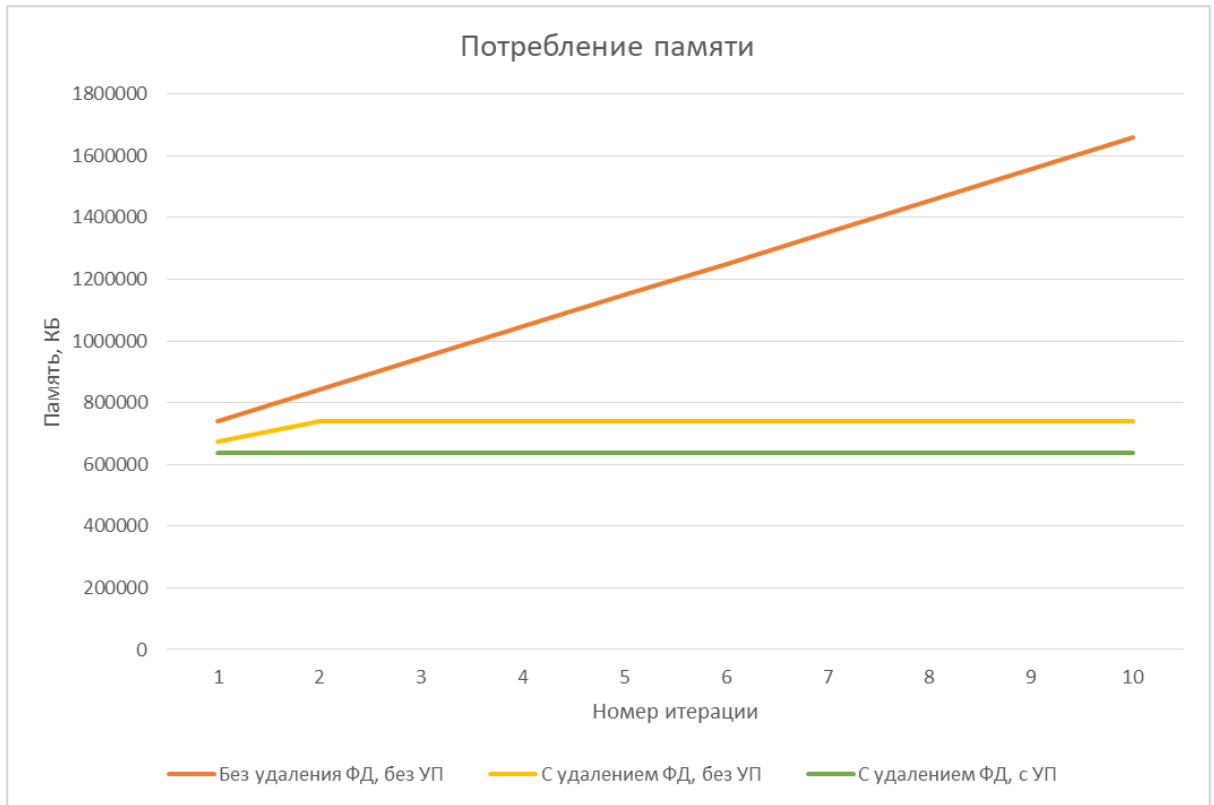


Рисунок 20 – График зависимости потребляемой памяти от номера итерации

Без удаления ФД и без УП потребление памяти закономерно постоянно растет. С удалением ФД и без УП потребление памяти сохраняется на одном уровне, при этом память расходуется на данные исполнительной системы и 2 массива по 100МБ, т.к. команда `top` вызывается после выделения памяти. Таким образом, в определенный момент времени в памяти находятся два массива - со старыми и измененными данными. С удалением ФД и с УП потребление памяти на одном уровне, при этом память расходуется на данные исполнительной системы и один массив размером 100 МБ. В этом случае в каждый момент времени в памяти расположен только один массив.

Для измерения времени исполнения программа была запущена на различном числе итераций в двух вариациях:

1. С удалением ФД и без уничтожающего потребления;
2. С удалением ФД и с уничтожающим потреблением.

Данные о времени исполнения получены из параметра `real` вывода команды `time`, вызов команды `top` был убран (Таблица 3, Рисунок 21).

Таблица 3 – Время исполнения тестовой программы

Число итераций	С удалением ФД, без УП (сек)	С удалением ФД, с УП (сек)
25	7,778	5,427
50	13,925	9,814
100	26,456	17,23
200	51,346	32,972
400	102,708	63,849
800	205,75	125,395

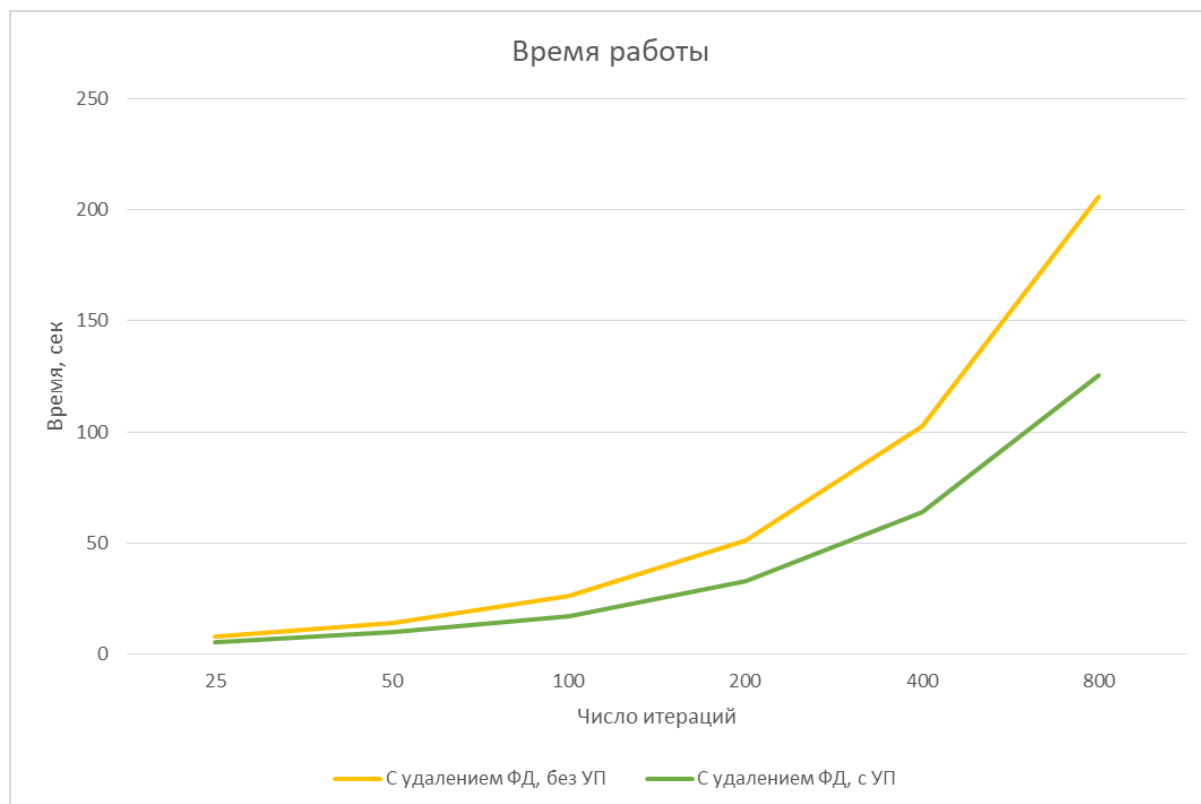


Рисунок 21 – График зависимости времени исполнения от числа итераций

Таким образом, вариант с использованием уничтожающего потребления выполняется на 30-40% быстрее.

#### 4.1.4 Работоспособность в распределенной среде

Так как подавляющее большинство программ для системы LuNA запускаются на вычислителях с распределенной памятью, то возникает необходимость протестировать работоспособность уничтожающего потребления в подобных условиях. Для этого была выбрана прикладная задача трехмерной фильтрации двухфазной жидкости при наличии скважин. Математическая модель данной задачи представляет собой СЛАУ, решаемую с помощью итерационного метода сопряженных градиентов. Из-за применения итерационного метода использование уничтожающего потребления должно положительно сказаться на нефункциональных характеристиках данной программы.

Была использована существующая реализация этой задачи в системе LuNA, которая была модифицирована и представлена в двух вариациях:

1. С использованием уничтожающего потребления (уничтожающее потребление добавлено в некоторые подходящие функции в итеративном процессе, например, в сложение умноженных на константу векторов с последующей записью значения в один из них).

2. С использованием копирования значения из одного ФД в другой.

Тестирование производилось на кластере НГУ со следующими параметрами: трехмерная сетка размером 100x100x100 ячеек; 20-ть итераций.

Данные о времени исполнения получены из параметра `real` вывода команды `time` (Таблица 4, Рисунок 22).

Таблица 4 – Время исполнения двух реализаций задачи

Кол-во процессов	Уничтожающее потребление, сек	Копирование, сек
1	16,9	18,5

2	15	16,7
4	8,6	10,7
6	11,9	12,4
8	9,8	11,8
12	16,4	19,8
16	20,6	21,2

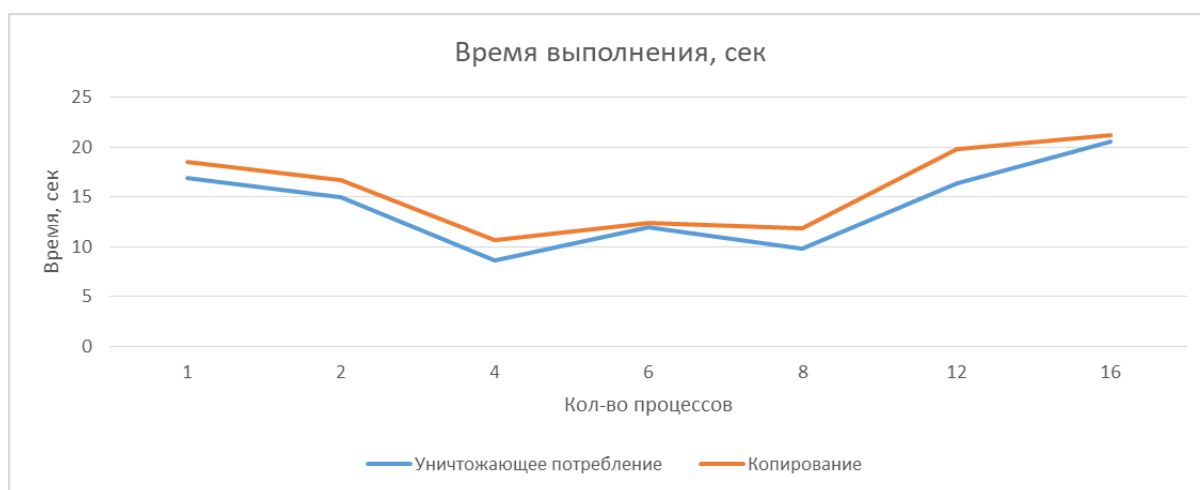


Рисунок 22 – Время исполнения двух реализаций задачи на различном числе процессов

Данные о потребляемой памяти получены из столбца VIRT вывода команды top (Таблица 5, Рисунок 23).

Таблица 5 – Потребляемая память двух реализация задачи

Кол-во процессов	Уничтожающее потребление, МБ	Копирование, МБ
1	1032,272	1032,272
2	1028,876	1071,252

4	1056,06	1079,296
6	980,36	1125,668
8	967,548	1098,62
12	999,304	1130,376
16	1022,944	1186,068

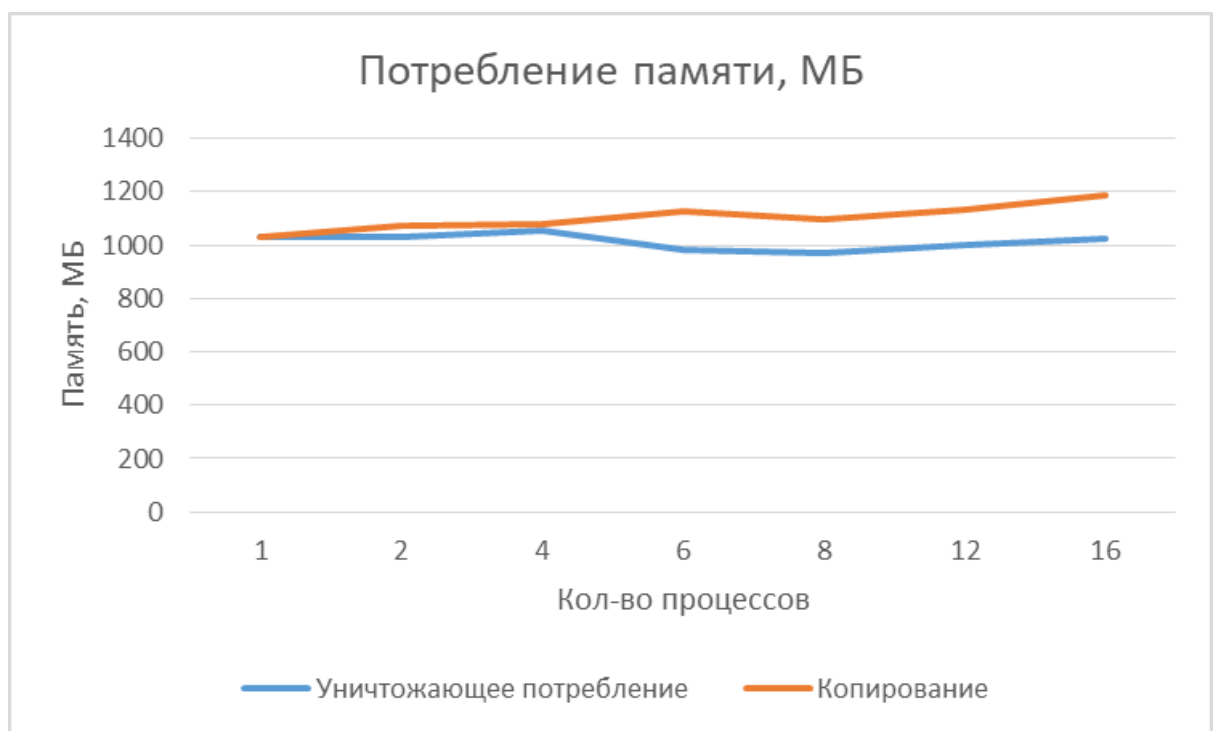


Рисунок 23 – Потребляемая память двух реализация задачи на различном числе процессов

Результат работы программы позволяет установить, что задача правильно выполняется на вычислителе с распределенной памятью, а также действительно улучшает ее нефункциональные характеристики.

## ЗАКЛЮЧЕНИЕ

Потребление памяти LuNA-программ было приближено к ее потреблению программами, написанными с использованием низкоуровневых средств параллельного программирования. Для этого было разработано средство директивного управления памятью, предоставляющее возможность использовать буфер памяти одного ФД другим. Данное средство было реализовано в системе LuNA посредством расширения грамматики языка и модификации транслятора.

Было проведено тестирование реализованного средства, в результате которого была проверена работоспособность решения, в том числе на прикладной задаче в среде с распределенной памятью. Также было экспериментально показано, что использование реализованного средства положительно влияет на нефункциональные характеристики программ, написанных для системы LuNA. В частности, происходит снижение времени выполнения и объема потребляемой памяти.

Полученные результаты частично опубликованы в статье для журнала «Проблемы информатики» [14].

Данная работа была опубликована на 61-й Международной научной студенческой конференции, г. Новосибирск, 2023 г. [15].

Защищаемые положения:

1. Разработано средство директивного управления памятью, предоставляющее возможность использовать буфер памяти одного ФД другим;
2. Разработанное средство реализовано в системе фрагментированного программирования LuNA;
3. Проведено экспериментальное исследование, которое показало работоспособность реализованного средства и его положительное влияние на нефункциональные характеристики программ, написанных для системы LuNA.

В дальнейшем планируется продолжить работу над реализованным средством директивного управления памятью. Возможными направлениями для развития являются:

1. Разработка способов контроля использования реализованного средства;
2. Полная или частичная автоматизация использования реализованного средства.

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 недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Кудрявцев Андрей Александрович \_\_\_\_\_

«\_\_» \_\_\_\_\_ 20\_\_ г.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Малышкин В. Э. Технология фрагментированного программирования //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.
2. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах //Проблемы информатики. – 2020. – №. 1. – С. 66-74.
3. Касьянов В. Н., Бирюкова Ю. В., Евстигнеев В. А. Функциональный язык SISAL 3.0 //Поддержка супервычислений и интернет-ориентированные технологии. – 2001. – С. 54-67.
4. Стасенко А. П. Внутреннее представление системы функционального программирования Sisal 3.0 //Препр./РАН. Сиб. Отд-ние. ИСИ. – 2004. – №. 110.
5. Касьянов В. Н. и др. Методы и средства параллельного программирования на основе языка Sisal //ИНФОРМАТИКА: ПРОБЛЕМЫ, МЕТОДОЛОГИЯ, ТЕХНОЛОГИИ. – 2015. – С. 169-174.
6. Абрамов С.М., Адамович А.И., Позлевич Р.В. Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ //Программные системы: Теоретические основы и приложения : сборник (под ред: А.К. Айламазян). — М. : Наука, Физматлит, 1999
7. Абрамов С. М. и др. Open TS: архитектура и реализация среды для динамического распараллеливания вычислений //Научный сервис в сети Интернет. – 2005. – С. 79-81.
8. Система параллельного программирования OpenTS. Описание языка Т++ — [Электронный ресурс] / — Режим доступа: <https://web.archive.org/web/20170521045548/http://www.opents.net/index.php/ru/lang-tp#tp-common> (дата обращения: 12.05.2023).

9. Новопашин В. С. Организация распределенной общей памяти //Известия СПбГЭТУ ЛЭТИ. – 2020. – №. 3. – С. 64-70.
10. Kalé L. V., Zheng G. The Charm++ Programming Model. – 2013.
11. Kale L. V., Bhatle A. (ed.). Parallel science and engineering applications: The Charm++ approach. – CRC Press, 2016.
12. Charm++ parallel programming framework — [Электронный ресурс] / – Режим доступа: <https://charmplusplus.org/> (дата обращения: 13.05.2023).
13. Charm++ Documentation — [Электронный ресурс] / – Режим доступа: <https://charm.readthedocs.io/en/latest/index.html> (дата обращения: 13.05.2023).
14. А.А. Кудрявцев, В.Э. Малышкин, Ю.Ю. Нуштаев, В.А. Перепёлкин, В.А. Спирин Эффективная фрагментированная реализация краевой задачи фильтрации двухфазной жидкости //Проблемы информатики. – 2023 (Отправлена в журнал).
15. Кудрявцев А. А. Разработка и реализация средств директивного управления памятью в системе LuNA / Кудрявцев А. А. //Информационные технологии : Материалы 61-й Междунар. науч. студ. конф. 17–26 апреля 2023 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2023 (Принято в печать).

## **ПРИЛОЖЕНИЕ А**

Средство директивного управления памятью в системе LuNA на основе  
уничтожающего потребления

Руководство оператора

Листов 8

Новосибирск 2023

## АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации средства директивного управления памятью в системе LuNA на основе уничтожающего потребления.

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.

В разделе «Сообщения оператору» приведены тексты сообщений, выдаваемых в ходе выполнения программы, описание их содержания и соответствующие действия оператора.

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

## СОДЕРЖАНИЕ

Аннотация.....	46
1 Назначение программы.....	48
1.1 ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПРОГРАММЫ.....	48
1.2 ЭКСПЛУАТАЦИОННОЕ НАЗНАЧЕНИЕ ПРОГРАММЫ.....	48
1.3 СОСТАВ ФУНКЦИЙ.....	48
2 Условия выполнения программы.....	49
2.1 МИНИМАЛЬНЫЙ СОСТАВ АППАРАТНЫХ СРЕДСТВ.....	49
2.2 МИНИМАЛЬНЫЙ СОСТАВ ПРОГРАММНЫХ СРЕДСТВ.....	49
2.3 ТРЕБОВАНИЕ К ОПЕРАТОРУ.....	49
3 Выполнение программы.....	50
3.1 ЗАГРУЗКА И ЗАПУСК ПРОГРАММЫ.....	50
3.2 ВЫПОЛНЕНИЕ ПРОГРАММЫ.....	50
3.3 ЗАВЕРШЕНИЕ РАБОТЫ ПРОГРАММЫ.....	50
4 Сообщения оператору.....	51
5 Лист регистрации программы.....	52

## **1 Назначение программы**

### **1.1 Функциональное назначение программы**

Программный компонент предоставляет средство для эффективного использования памяти при разработке программ для системы фрагментированного программирования LuNA, основанное на механизме уничтожающего потребления. Данный механизм позволяет записывать новое значение, принадлежащее выходному фрагменту данных (поглощающий ФД), в участок памяти, занимаемый значением входного фрагмента данных (поглощаемый ФД).

### **1.2 Эксплуатационное назначение программы**

Программный компонент должен эксплуатироваться как часть системы фрагментированного программирования LuNA. Конечными пользователями являются разработчики фрагментированных программ для системы LuNA.

### **1.3 Состав функций**

- Объявление атомарного фрагмента кода, содержащего уничтожающее потребление.
- Перенос буфера из одного ФД во второй.
- Проверка выполнения уничтожающего потребления.

## **2 Условия выполнения программы**

### **2.1 Минимальный состав аппаратных средств**

Минимальные требования к техническому средству:

- 1) монитор;
- 2) клавиатура;
- 3) процессор с частотой 2.4 ГГц и четырьмя физическими ядрами;
- 4) оперативная память объемом 2 Гбайт;
- 5) свободное дисковое пространство объемом 1 Гбайт.

### **2.2 Минимальный состав программных средств**

Минимальный состав программных средств:

- 1) компилятор g++;
- 2) библиотека OpenMPI;
- 3) интерпретатор языка Python версии не ниже 3.8;
- 4) генератор синтаксических анализаторов GNU Bison;
- 5) генератор лексических анализаторов flex.

### **2.3 Требование к оператору**

Конечный пользователь программы должен обладать практическими навыками разработки и запуска программ для системы фрагментированного программирования LuNA.

## 3 Выполнение программы

### 3.1 Загрузка и запуск программы

При разработке программы для системы LuNA в описании фрагментированного алгоритма при необходимости происходит объявление атомарных фрагментов кода, помеченных директивой уничтожающего потребления. В следующем примере аргумент “b” является поглощаемым, а аргумент “a” – поглощающим (Рисунок 24).

```
1 import my_func(name a, value b) as my_func @ {  
2   | destruct b => a;  
3   };
```

Рисунок 24 – Использование директивы уничтожающего потребления

В реализации таких фрагментов кода необходимо использовать предоставляемую компонентом функцию переноса буфера между фрагментами данных, связанных уничтожающим потреблением (Рисунок 25).

```
1 void my_func(DF &df2, const DF &df1)  
2 {  
3   | df2.grab_buf(df1);  
4 }
```

Рисунок 25 – Использование функции переноса буфера

Запуск разработанного программного компонента происходит во время выполнения стандартной LuNA-программы.

### 3.2 Выполнение программы

Выполнение программы не предполагает никаких дополнительных операций.

### 3.3 Завершение работы программы

Программа завершается автоматически при завершении соответствующей LuNA-программы.

## 4 Сообщения оператору

При завершении программы с ошибкой по причине отсутствия выполнения объявленного уничтожающего потребления предусмотрен вывод сообщения об ошибке в командную строку (Рисунок 26).

```
luna: fatal error: run-time error: errcode=-6
err> 0 ERROR: Destructive consuming was declared, but not executed
/home/andrey/luna/scripts/./build/programs/home/andrey/diploma/DC_rts_check/DC.fa/test.cpp:112
err> 0 ABORT
err> terminate called after throwing an instance of 'RuntimeError'
err> what(): std::exception
err> [DESKTOP-U2LLOQD:06557] *** Process received signal ***
```

Рисунок 26 – Сообщение об ошибке

В таком случае пользователь должен убедиться в корректности использования программы, произвести необходимые модификации, после чего перезапустить соответствующую LuNA-программу.

## 5 Лист регистрации программы

Таблица 6 – Лист регистрации изменений в программном документе  
«Руководство оператора»

Лист регистрации изменений									
Номер листов(страниц)					Всего листов (стр.) в док-е	№ док-а	Входящий № сопроводительно го док-а	Подпись	Дата
№ изм.	изменен ных	заменен ных	новых	аннулиро ванных					

## **ПРИЛОЖЕНИЕ Б**

Средство директивного управления памятью в системе LuNA на основе  
уничтожающего потребления

Описание программы

Листов 12

Новосибирск 2023

## **АННОТАЦИЯ**

В данном программном документе приведено описание средства директивного управления памятью в системе LuNA на основе уничтожающего потребления. Исходным языком программы является Python и C++. Средство разработки – редактор исходного кода CLion от компании JetBrains.

Программа предназначена для повышения эффективности использования памяти при разработке программ для системы LuNA.

Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

## СОДЕРЖАНИЕ

Аннотация.....	54
1 Общие сведения.....	56
1.1 ОБОЗНАЧЕНИЕ И НАИМЕНОВАНИЕ ПРОГРАММЫ.....	56
1.2 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ, НЕОБХОДИМОЕ ДЛЯ ФУНКЦИОНИРОВАНИЯ ПРОГРАММЫ.....	56
1.3 ЯЗЫКИ ПРОГРАММИРОВАНИЯ .....	56
2 Функциональное назначение .....	57
2.1 НАЗНАЧЕНИЕ ПРОГРАММЫ.....	57
2.2 СВЕДЕНИЯ О ФУНКЦИОНАЛЬНЫХ ОГРАНИЧЕНИЯХ НА ПРИМЕНЕНИЕ .....	57
3 Описание логической структуры .....	58
3.1 СТРУКТУРА ПРОГРАММЫ .....	58
3.2 АЛГОРИТМ ПРОГРАММЫ .....	58
3.3 СВЯЗИ МЕЖДУ СОСТАВНЫМИ ЧАСТЯМИ ПРОГРАММЫ .....	59
3.4 СВЯЗИ ПРОГРАММЫ С ДРУГИМИ ПРОГРАММАМИ.....	59
4 Используемые технические средства.....	60
5 Вызов и загрузка.....	61
6 Входные данные .....	62
7 Выходные данные .....	63
8 Лист регистрации программы .....	64

## **1 Общие сведения**

### **1.1 Обозначение и наименование программы**

Полное наименование программы – средство директивного управления памятью в системе LuNA на основе уничтожающего потребления.

### **1.2 Программное обеспечение, необходимое для функционирования программы**

Необходимое программное обеспечение:

- 1) компилятор g++;
- 2) библиотека OpenMPI;
- 3) интерпретатор языка Python версии не ниже 3.8;
- 4) генератор синтаксических анализаторов GNU Bison;
- 5) генератор лексических анализаторов flex.

### **1.3 Языки программирования**

Исходный код программы написан на языке C++ и Python. Средство разработки – редактор исходного кода CLion от компании JetBrains.

## **2 Функциональное назначение**

### **2.1 Назначение программы**

Программа предназначена для повышения эффективности использования памяти при разработке программ для системы LuNA.

### **2.2 Сведения о функциональных ограничениях на применение**

Программа поддерживает только фрагменты данных, имеющих тип `value`. Программа не гарантирует правильность выполнения при использовании одного и того же фрагмента данных в двух уничтожающих потреблении.

### 3 Описание логической структуры

#### 3.1 Структура программы

Программа состоит из 3-х модулей, реализованных на языке Python и C++ (Таблица 7).

Таблица 7 – Список модулей программы

Название модуля	Задействованные файлы	Описание
Грамматика	grammar.ypp	Задает грамматику языка LuNA и определяет правила, согласно которым код будет преобразован транслятором в абстрактное синтаксическое дерево
Фрагмент данных	df.h, df.cpp	Реализация абстракции фрагмента данных
Транслятор	fcmp2	Транслятор описания фрагментированного алгоритма в конечный код фрагментированной программы

#### 3.2 Алгоритм программы

1) Описание фрагментированного алгоритма, содержащее директивы уничтожающего потребления, транслируется системой LuNA в абстрактное синтаксическое дерево в соответствии в расширенным набором правил.

2) На основе абстрактного синтаксического дерева генерируется конечный код фрагментированной программы, содержащий проверки выполнения уничтожающего потребления.

3) После исполнения фрагмента вычислений с уничтожающим потреблением выполняется проверка факта выполнения уничтожающего потребления. При провале проверки программа завершается с ошибкой.

### **3.3 Связи между составными частями программы**

Связь между модулями программы осуществляется при помощи вызова функций и методов объектов, а также промежуточных файлов.

### **3.4 Связи программы с другими программами**

Для запуска программы необходимо наличие на устройстве компилятора g++, библиотеки OpenMPI, интерпретатора языка Python версии не ниже 3.8, генератора синтаксических анализаторов GNU Bison и генератора лексических анализаторов flex.

#### **4 Используемые технические средства**

Требования, предъявляемые к техническому средству:

- 1) монитор;
- 2) клавиатура;
- 3) процессор с частотой 2.4 ГГц и четырьмя физическими ядрами;
- 4) оперативная память объемом 2 Гбайт;
- 5) свободное дисковое пространство объемом 1 Гбайт.

## 5 Вызов и загрузка

При разработке программы для системы LuNA в описании фрагментированного алгоритма при необходимости происходит объявление атомарных фрагментов кода, помеченных директивой уничтожающего потребления. В следующем примере аргумент “b” является поглощаемым, а аргумент “a” – поглощающим (Рисунок 27).

```
1 import my_func(name a, value b) as my_func @ {
2   | destruct b => a;
3   };
```

Рисунок 27 – Использование директивы уничтожающего потребления

В реализации таких фрагментов кода необходимо использовать предоставляемую компонентом функцию переноса буфера между фрагментами данных, связанных уничтожающим потреблением (Рисунок 28).

```
1 void my_func(DF &df2, const DF &df1)
2 {
3   | df2.grab_buf(df1);
4 }
```

Рисунок 28 – Использование функции переноса буфера

Запуск разработанного программного компонента происходит во время выполнения стандартной LuNA-программы.

## **6 Входные данные**

Входные данные программы определяются входными данными соответствующей LuNA-программы.

## **7 Выходные данные**

Выходные данные программы определяются выходными данными соответствующей LuNA-программы.

## 8 Лист регистрации программы

Таблица 8 – Лист регистрации изменений в программном документе  
«Руководство оператора»

Лист регистрации изменений									
Номер листов(страниц)					Всего листов (стр.) в док-е	№ док-а	Входящий № сопроводительно го док-а	Подпись	Дата
№ изм.	изменен ных	заменен ных	новых	аннулиро ванных					