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

Олимпиева Юрия Юрьевича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ УПРАВЛЕНИЯ ПАМЯТЬЮ В СИСТЕМЕ АКТИВНЫХ ЗНАНИЙ LUNA

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

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Перепёлкин В. А./……...
(ФИО) / (подпись)
«……»……20…г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Матвеев А. С./……..
(ФИ О) / (подпись)
«…»……20…г.

Новосибирск, 2025

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)

«17» января 2025г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту*(ке)* Олимпиеву Юрию Юрьевичу, группы 21210
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация алгоритмов управления памятью в системе активных знаний LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 21.10.2024 №0377, скорректирована распоряжением проректора по учебной работе от 17.01.2025 №0020

Срок сдачи студентом готовой работы 20 мая 2025 г.

Исходные данные (или цель работы):

Разработка и реализация алгоритмов управления памятью в системе активных знаний LuNA

Структурные части работы:

обзор литературы, постановка задачи, формулирование требований к системе, разработка и реализация системы, тестирование

Руководитель ВКР
доц. каф. ПВ ФИТ НГУ,
к.т.н.
Перепёлкин В. А. /……..
(ФИ О) / (подпись)

«17» января 2025г.

Задание принял к исполнению
Олимпиев Ю. Ю. /……..
(ФИО студента) / (подпись)

«17» января 2025г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,
Матвеев А. С. /……..
(ФИ О) / (подпись)

«17» января 2025г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Проблема автоматического управления памятью возникает во многих вычислительных системах — в компиляторах и трансляторах языков программирования, в фреймворках машинного обучения и обработки данных, в операционных системах и многих других областях.

Управление памятью может преследовать разные цели — упрощение написания кода через сокрытие механизмов управления ресурсами вычислителя (Java, Golang) от программиста, уменьшение расходов памяти (оптимизации компиляторов) в ходе исполнения программы, управление общей или распределенной памятью в системах распределенных вычислений (управление рассылками данных между узлами, синхронизация данных в системе) и многое другое. Специфика управления памятью в конкретной системе определяется теми задачами и условиями эксплуатации, которые сложились в этой системе.

Исследование проблемы автоматического управления памятью требует рассмотрения конкретных частных случаев, поскольку на текущем этапе развития области не существует общепринятой универсальной модели, охватывающей все аспекты этой задачи. Унификация подходов затруднена как разнообразием вычислительных архитектур, так и различиями в требованиях прикладных систем.

Система активных знаний LuNA, разрабатываемая в ИВМиМГ СО РАН [1], представляет собой подходящее окружение для такого исследования: она сочетает в себе декларативную постановку задач, автоматическую генерацию параллельного кода и взаимодействие с распределенной вычислительной средой. Эти особенности позволяют рассматривать проблему управления памятью не абстрактно, а в контексте конкретных механизмов и сценариев использования.

Следовательно, допустимо и обоснованно проводить исследование проблем автоматического управления памятью на примере системы LuNA, как

на репрезентативном примере современных инструментов генерации программ из декларативных описаний.

В системе активных знаний LuNA, поддерживающей концепцию активных знаний, специфика управления памятью также может быть рассмотрена с нескольких точек зрения — управление распределенной и общей памятью на суперкомпьютере, распределенная сборка мусора, выбор близких к оптимальным решений поставленных задач по расходу ресурсов вычислителя.

Общей задачей управления памятью в системе активных знаний LuNA является одновременное сокрытие деталей реализации решений управления ресурсами от пользователя или предоставление возможностей декларативного изложения этих деталей при формальной постановке задачи и удовлетворительное для практического применения управление ресурсами вычислительной системы для обеспечения пригодных нефункциональных свойств генерируемых програм.

Целью работы является разработка и интеграция механизмов управления памятью вычислителя в системе активных знаний LuNA.

Для достижения этой цели в работе были решены следующие задачи:

1. Анализ требований к управлению памятью в вычислительных системах и выявление особенностей, влияющих на архитектуру решения;
2. разработка алгоритмов управления памятью для снижения избыточного выделения памяти в типичных вычислительных задачах системы активных знаний LuNA;
3. реализация модуля управления памятью вычислителя на основе разработанных алгоритмов в виде фрагментов вычислений;
4. оценка эффективности предложенной реализации, сравнительный анализ.

Научная новизна заключается в формализации принципов управления переменными и их размещения в памяти в системах автоматической генерации программ. Предложенные алгоритмы реализуют формальную модель, обеспечивая автоматизированное управление памятью.

Практическая ценность работы заключается в разработке и тестировании модуля управления памятью, реализующего предложенную модель для систем автоматической генерации параллельных программ. Несмотря на то что интеграция в систему LuNA пока не выполнена, модуль готов к встраиванию и может быть использован как в LuNA, так и в аналогичных платформах, использующих высокоуровневые описания вычислений. Его применение позволяет структурировать работу с переменными, отслеживать их состояния и управлять оперативной памятью вычислителя.

1 Анализ предметной области

Цель данного обзора — проанализировать существующие подходы к автоматическому управлению памятью в современных вычислительных системах и определить их применимость к системе автоматической генерации программ, подобной LuNA, с учетом требований к эффективному использованию памяти.

Критерии оценки включают:

- автоматизацию управления памятью: степень автоматизации процессов выделения и освобождения памяти.
- эффективность использования памяти: способность систем минимизировать использование памяти и предотвращать утечки.
- поддержку параллелизма: возможность эффективного управления памятью в многопоточных и распределенных средах.

1.1 Анализ существующих решений

В языках C и C++ управление памятью осуществляется вручную — разработчик самостоятельно выделяет и освобождает память с помощью функций и операторов. Такой подход обеспечивает детальный контроль над ресурсами и потенциально высокую производительность. Современные компиляторы применяют оптимизации на уровне кода, например, выравнивание данных, размещение переменных в регистрах и устранение повторяющихся вычислений [2, 3]. Однако они не решают проблему автоматического управления памятью, что требует от программиста дополнительной дисциплины для обеспечения безопасности и эффективности кода и увеличивает вероятность ошибок [4]. В многопоточных системах возникает риск состояния гонок и ошибок синхронизации, что усложняет управление памятью. Эти особенности ограничивают возможность применения такого подхода в системах автоматической генерации программ, где требуется минимальное вмешательство пользователя.

Языки программирования с автоматическим сбором мусора, например Java и C#, реализуют управление памятью с помощью периодического отслеживания и освобождения неиспользуемых объектов. Это устраняет необходимость ручного освобождения памяти и снижает вероятность утечек. Однако сбор мусора приводит к неопределённым паузам в выполнении программ, что негативно влияет на детерминизм и производительность. В многопоточных средах синхронизация доступа к куче увеличивает накладные расходы и может создавать задержки. Современные алгоритмы сборщика мусора, такие как поколенческий или параллельный сбор, уменьшают время пауз, но полностью исключить их не удаётся [5]. В результате языки с таким управлением памяти менее подходят для систем, требующих высокого контроля над ресурсами и предсказуемого поведения.

Julia — высокоуровневый язык программирования с открытым исходным кодом с динамической типизацией, созданный для математических вычислений [6, 7] — применяет сборщик мусора для автоматического управления памятью [8, 9], но при этом допускает активное вмешательство пользователя в процесс оптимизации. Для достижения высокой производительности рекомендуется избегать лишних аллокаций — выделений памяти — и использовать строгую типизацию [10]. Эти меры требуют знаний внутренней архитектуры языка и ручной настройки кода. Автоматизация оптимизации распределения памяти в Julia ограничена, что снижает степень автоматизации, необходимую для систем, подобным LuNA, где управление памятью должно быть прозрачным и эффективным.

Решения, такие как TensorFlow [11] — комплексная платформа с открытым исходным кодом для машинного обучения, PyTorch — библиотека для программ на Python, которая поддерживает программы глубокого обучения [12], реализуют управление памятью с использованием пулов и асинхронного выделения ресурсов, что снижает накладные расходы и увеличивает производительность при работе с GPU и CPU [13]. Они применяют

статистические методы и машинное обучение для предсказания времени жизни объектов и оптимизации их размещения [12]. Эти решения ориентированы на специфические задачи — выполнение вычислительных графов нейросетей и научных моделей. Архитектура фреймворков ограничивает их применимость для более общих вычислительных задач и автоматической генерации программ, так как управление памятью ориентировано под узкий класс вычислений.

Использование промежуточных представлений, таких как MLIR в LLVM [13] — многоуровневое промежуточное представление, предназначенное для решения растущей сложности современных компиляторов, особенно тех, которые работают с гетерогенным оборудованием (CPU, GPU, TPU и т. д.) [14] или требуют нескольких уровней абстракции [15], позволяет реализовать анализ времени жизни объектов и трансформации программ, которые включают автоматическое распределение памяти. Анализ зависимостей и возможность размещать объекты на стеке, в куче или избегать их создания полностью позволяют повышать эффективность программ. Аналогичные методы применяются в системах автоматического дифференцирования, например JAX с XLA-компилятором [16], где статический анализ вычислительных графов улучшает управление памятью. Эти технологии могут служить основой для интеграции в системы автоматической генерации кода, однако их использование требует значительных усилий по адаптации и интеграции в целевую архитектуру.

Rust использует систему владения и заимствований, которая позволяет компилятору на этапе компиляции определять время жизни объектов и обеспечивать автоматическое освобождение памяти без сборщика мусора [17]. Такая модель исключает ошибки, характерные для ручного управления памятью, и улучшает безопасность программ. Требования к пользователю по контролю ссылок и времени жизни делают язык сложным в использовании. Тем не менее, при генерации кода из декларативных моделей правила владения могут быть созданы автоматически на основе анализа зависимостей.

Исследования в области автоматической генерации кода на Rust показывают потенциал этого подхода для систем, ориентированных на высокую безопасность и производительность.

В системах, использующих MPI , CUDA и другие модели вычислений, ответственность за выделение, синхронизацию и освобождение памяти лежит на разработчике [18]. Некоторые механизмы, например Unified Memory в CUDA, упрощают совместное использование памяти CPU и GPU, но имеют ограничения по производительности [19]. Модели с глобальным адресным пространством (PGAS) [20] обеспечивают логическое объединение памяти, но не автоматизируют контроль времени жизни объектов. В таких системах необходимы дополнительные слои управления памятью для применения в контексте автоматической генерации программ.

1.2 Выводы

Рассмотренные подходы к управлению памятью обладают преимуществами и недостатками, которые определяют их применимость к системам автоматической генерации программ, подобным LuNA. Ручное управление памятью обеспечивает детальный контроль, но требует значительных усилий и может приводить к ошибкам. Автоматический сбор мусора упрощает разработку, но снижает предсказуемость и производительность. Оптимизации в языках вроде Julia требуют вмешательства пользователя, а специализированные фреймворки ориентированы на узкие задачи. Методы статического анализа времени жизни и модели владения, реализованные в современных компиляторах и языках, предлагают перспективные пути для создания эффективных и безопасных механизмов управления памятью в автоматических системах. Для их интеграции необходимы дальнейшие исследования и разработка специализированных инструментов, способных работать в условиях генерации кода с учетом параллелизма и разнообразия архитектур.

2 Алгоритм управления памятью

Как уже было упомянуто выше, проблемы управления памятью могут иметь различную специфику в зависимости от задач, которые решает вычислительная система. Каждая вычислительная и информационная система, работающая с данными, вынуждена решать какие-либо задачи управления памятью в соответствии со своими потребностями — обеспечение целостности данных, компактное их расположение в памяти, обеспечение согласованности данных, управление распределенной или общей памятью и многое другое.

В этом разделе более подробно опишем систему активных знаний LuNA и определим какого рода задачи управления памятью возникают в ней, выделим класс проблем, на решение которых нацелена данная работа. Затем перейдем к описанию предлагаемого теоретического решения выявленных проблем управления памятью.

2.1 Задачи управления памятью в системе активных знаний luna

В рамках исследуемой проблемы нет необходимости подробно разбирать концепцию активных знаний, но для анализа системы LuNA на выявление проблем, связанных с управлением памятью, необходимо построить базовое представление о концепциях, которые лежат в основе системы активных знаний.

Активное представление знаний должно обеспечивать извлечение знаний из его описания и использование знаний для решения определенной прикладной задачи.

Концепция активных знаний описывает способ решения проблемы активного представления, понимания и применения знаний в четыре этапа:

– разработка достаточно полной аксиоматической теории (AT),

– формулировка прикладной задачи в терминах теории,

– вывод желаемого алгоритма решения прикладной задачи (APS),

– генерация программы, реализующей производный алгоритм APS. [21]

В системе LuNA построение достаточно полной теории сводится к формализации некоторой насыщенной предметной области путём построения вычислительной модели — двудольного графа, одна доля которого содержит переменные предметной области, а вторая доля — операции предметной области — и базы активных знаний — множества реализаций описанных в вычислительной модели операций. (можно привести пример, кажется, что он тут не помешает пониманию).

Постановка задачи при таком подходе сводится к построению VW-задачи — разделению множества всех переменных предметной области на известные и неизвестные переменные. Тогда известные переменные определяются как входные данные, необходимые для вычисления неизвестных выходных данных, что формализует постановку задачи по входу и выходу.

Если в вычислительной модели существует подмножество операций, упорядоченное применение которых к заданным значениям переменных позволяет получать значения новых переменных до тех пор, пока все переменные из W не получат значения, то это подмножество будем называть VW-планом. Любая VW-задача может иметь, вообще говоря, ноль или более VW-планов. Очевидно, что для заданной VW-задачи можно ставить задачу поиска VW-плана, и в случае успеха вычислить значения всех переменных из W, имея значения переменных из V.

Так как мы ходим решать сложные вычислительные задачи, то и результаты поиска VW-плана хотим получать эффективные (или хотя бы пригодные для практического применения). (На этом этапе могут возникать задачи управления памятью — выбор менее требовательных по памяти фрагментов вычислений из базы активных знаний ("жадное" решение) …).

После получения (или в процессе при интерпретации VW-задачи) VW-плана необходимо его "исполнить".

Задача управления памятью, которую мы перед собой ставим, заключается в разработке инструмента, позволяющего на этапе исполнения

VW-плана обеспечить управление переменными с допустимыми при использовании на практике расходами ресурсов вычислителя.

С одной стороны, возможность обеспечить нулевые дополнительные расходы на использование памяти на вычислителе вполне реальна — ручная реализация для отдельно взятой задачи, оптимизированная под конкретный вычислитель, для некоторого класса задач позволяет размещать данные в памяти эффективно, избегая дополнительных расходов.

С другой стороны, в общем случае в условиях, которые диктуют необходимость автоматически генерировать параллельные высокопроизводительные программы, такая задача является неразрешимой на данном этапе развития систем активных знаний — каждой VW-задаче необходимо будет поставить помимо пригодных для практического применения реализаций операций еще и большой объём метаинформации, которая позволила бы оптимизировать код под каждый возможный контекст исполнения программы — тип и конфигурация вычислителя, размеры входных данных [22]. Всё это усложняет процесс разработки программ в системе активных знаний и идёт вразрез с концепциями активных знаний в плоскости практических приложений. Кроме того, не все данные, необходимые для оптимального решения, могут быть заданы на этапе описания VW-плана и входящих в него модулей, а могут быть выявлены только на этапе исполнения, что также вводит в процесс выбора оптимального решения дополнительные, в общем случае неразрешимые, сложности.

На практике получение оптимальных решений не требуется — достаточно показать достаточно хорошее решение поставленной задачи, применимое в реальных эксплуатационных условиях. Таким образом, наша задача сводится к снижению расходов на ресурсы памяти вычислителя в некотором подклассе задач с сохранением пригодности использования решения на практике.

Далее будем считать, что в рамках некоторой VW-задачи некоторым образом был выбран VW-план, нефункциональные свойства — использование

ресурсов памяти вычислителя, — который будем считать достаточно хорошими. Это допущение является отправной точкой данной работы.

Возможны различные подходы к исполнению VW-плана: интерпретация VW-плана, генерация кода на основе VW-плана, исполнение VW-плана в рантайм системе.

Существует два ключевых подхода к выполнению VW-плана, а также возможные их комбинации: интерпретация и генерация исполняемого кода. В первом случае используется виртуальная машина, способная принимать на вход описание вычислительной модели, постановку задачи и входные данные (значения переменных множества V). На основе этих данных она формирует VW-план в процессе выполнения и запускает требуемые модули на целевом вычислительном оборудовании, последовательно исполняя операции до тех пор, пока все значения переменных из множества W не будут получены.

Во втором подходе применяется автоматическая генерация программы, способной вычислить значения переменных W при наличии исходных данных из множества V. Такая программа создаётся специальным генератором, который использует описание VW-задачи, модель вычислений, характеристики целевой вычислительной системы и, при необходимости, конкретные входные данные. В этом случае VW-план формируется заранее, до исполнения, а итоговая программа представляет собой управляющую структуру, в которой предусмотрен вызов нужных модулей в строго определённом порядке, заданном планом.

Интерпретация предпочтительна, когда построение VW-плана невозможно выполнить полностью заранее или эффективность его исполнения зависит от данных, доступных только во время выполнения. К подобным случаям можно отнести, например, адаптацию выбора операций под текущие свойства входных данных или необходимость динамического перераспределения нагрузки между вычислительными узлами в условиях мультикомпьютерной архитектуры. Основной недостаток интерпретируемого

подхода заключается в высоких накладных расходах, возникающих из-за необходимости обработки вычислительных сущностей и принятия решений непосредственно в процессе выполнения. Это особенно критично, когда в задаче присутствует множество малых по ресурсоемкости операций и переменных, либо когда предъявляются жесткие требования к производительности. В таких ситуациях предпочтение отдают генерации статической программы, в которой управление и ресурсы заранее распределены максимально эффективно.

Смешанный подход — так называемая полуинтерпретация — реализует компромисс между гибкостью интерпретации и эффективностью сгенерированного кода. Он предполагает, что часть вычислений описывается в виде традиционного императивного кода (например, с использованием MPI или потоков), а другая часть остаётся динамически управляемой. При этом динамическая часть охватывает лишь те аспекты вычислений, где это действительно необходимо, что позволяет сохранить адаптивность при снижении общего объема накладных расходов.

В данной работе рассматривается возможность управления переменными и данными в памяти вычислителя в рамках обоих подходов и их комбинации. Это может быть достигнуто путем реализации модулей взаимодействия с памятью вычислителя, которые могли бы использоваться интерпретатором и генератором в рамках концепции активных знаний, но были бы сокрыты от пользователя и не фигурировали в верхнеуровневом описании вычислительной модели.

Рассмотрим идею решения на примере простой модельной предметной области — умножения матриц.

Формальная постановка задачи в виде VW-графа представлена на рисунке ниже (Рисунок 1).

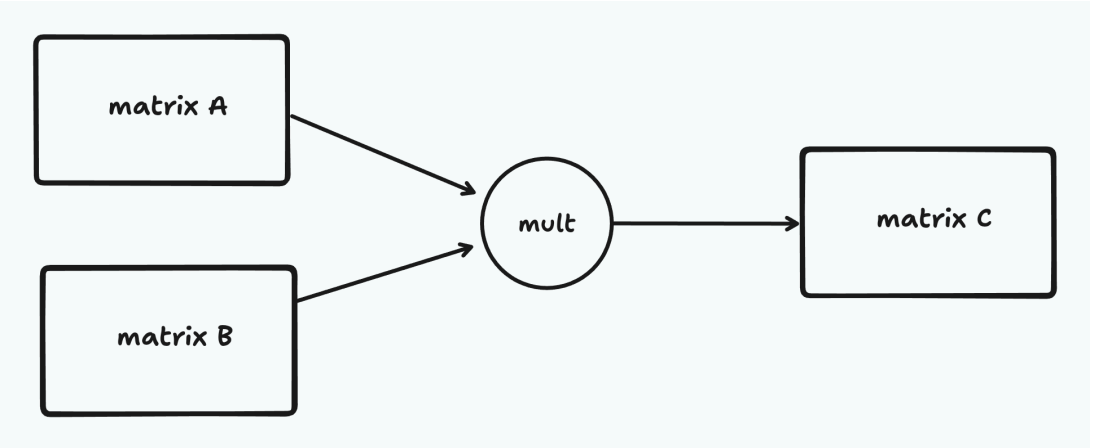

Рисунок 1 — Граф вычислительной модели для задачи умножения матриц

На вход подаются две матрицы, на выходе ожидается результат перемножения этих двух матриц. Решение о выборе алгоритма умножения матриц может производиться как на этапе работы со структурой вычислительной модели, так и на более поздних этапах.

Допустимо усложнить вычислительную модель, предположив, что в качестве алгоритма умножения выбрано блочное умножение. Тогда вход и выход для задачи останутся теми же, а структура фрагмента вычисления для умножения развернется в некоторую вложенную вычислительную модель (Рисунок 2) — входные матрицы должны быть разделены на блоки, которые впоследствии будут использованы (потреблены) для вычисления блоков итоговой матрицы, результирующие блоки будут агрегированы в итоговый ответ.

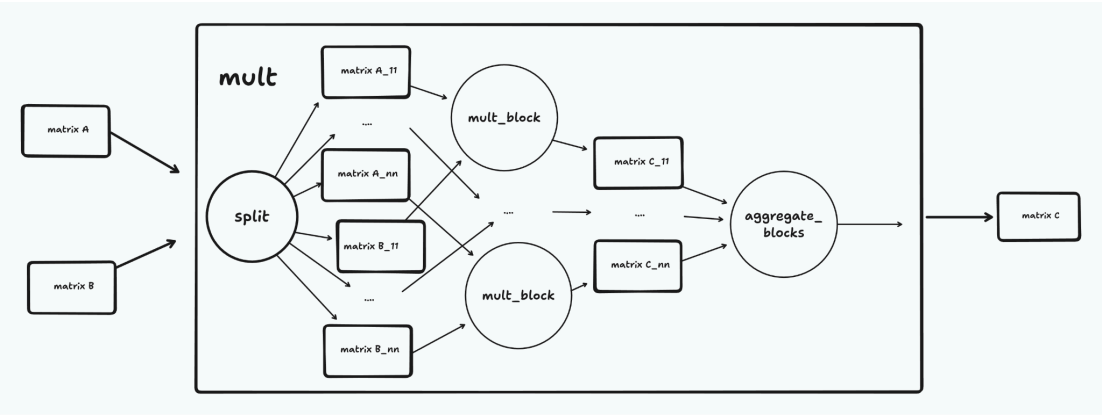

Рисунок 2 —Граф вычислительной модели для задачи блочного умножения матриц

Разберем теперь возможность внедрения в эту уточненную вычислительную модель алгоритмов работы с памятью на примере фрагмента вычислений промежуточных блоков (Рисунок 3). Модуль умножения матриц может получать на вход имена обеих матриц и индекс блока, для которого непосредственно необходимо произвести вычисления. По именам переменных предметной области и именам скрытых переменных можно получить адреса, по которым располагаются необходимые для вычислений данные, затем получить указатели на сами блоки с данными, перемножить их и вернуть результат. В этом примере разобрана работа с получением данных из памяти системы, но опущены подробности загрузки данных в память, которая возникает после умножения блоков — промежуточные результаты необходимо куда-либо выгрузить или передать на дальнейшую обработку через модули управления памятью. Суть решения при выгрузке данных в память на текущем уровне абстракции имеет структуру, симметричную чтению, поэтому подробный разбор механизмов записи может быть опущен.

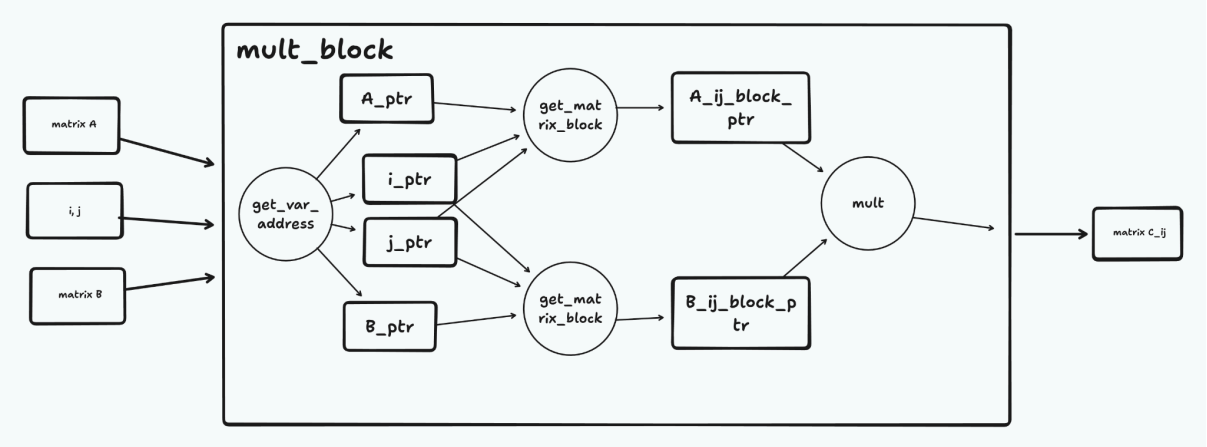

Рисунок 3 — Граф вычислительной модели для задачи блочного умножения матриц с операциями управления памятью

В базе активных знаний может присутствовать несколько реализаций работы с данными в памяти — в зависимости от требований и специфики задачи можно выбирать более подходящие — работать с удаленным источником данных в распределенной системе, использовать стратегии кэширования данных для увеличения скорости доступа и многое другое.

Таким образом, решение для управления памятью, воплощенное в сокрытых модулях базы активных знаний, позволяет уже на этапе планирования, то есть построения VW-плана, встраивать подходящие решения работы с данными. Такой подход применим как в случае интерпретации вычислительной модели — интерпретатор на основе имеющихся данных способен принимать решения об использовании того или иного механизма взаимодействия с памятью, — так и в случае генерации кода — полученный VW-план будет преобразован в код, содержащий фрагменты кода взаимодействия с памятью в соответствии с описанием вычислительной модели.

Даже на столь детальном уровне рассмотрения проблемы управления памятью возникают различные потенциальные направления решения в зависимости от особенностей вычислительной задачи. Дальнейшее рассмотрение предлагаемого решения ограничим следующими условиями:

- Модуль должен поддерживать объявление, чтение и запись скалярных и составных типов — массивов и двумерных матриц в оперативной памяти для вычислений на процессоре вычислителя;
- модуль должен предоставлять возможности работы со срезами и подматрицами;
- в структуре модуля необходимо заложить возможности взаимодействия с внешними веб-сервисами для получения и передачи данных;
- модуль должен поддерживать возможность работы вычислителя в условиях параллельного исполнения с общей памятью.

Такие свойства разрабатываемого решения хоть и не способны закрыть все возникающие потребности в условиях эксплуатации системы, но всё же позволяют решить достаточно большой пласт задач, возникающих в системе активных знаний LuNA в контексте управления данными в памяти.

2.2 Формальная модель переменных и операций над ними

Ниже приводится формальное описание модели переменных, их типов и состояний. Также приведен набор допустимых операций над переменными, включая их создание, чтение, запись, извлечение срезов, синхронный и асинхронный доступ к значениям.

Определим множество переменных $V= \{v_1, \ldots, v_n\}$, для элементов этого множества определим отображение:

$s: V \rightarrow S$, где

S - множество кортежей вида (name, type, shape, state).

name $\in \Sigma^*$ - уникальное символьное имя, где Σ — конечное множество допустимых символов. Например, Σ = {a, …, z, A, …, Z, 0, …, 9, _}.

type $\in$ {int, float, double},

shape $\in \{\varnothing, (n), (n, m)\}$, $m,n \in R$.

state $\in$ {ready, undefined}.

Следует отметить, что в вышеописанной формализации переменные и их состояния описаны как неизменяемые математические объекты. Однако в реальности значения и состояния переменных меняются во времени в процессе исполнения программы. Для отражения этого процесса необходимо ввести дискретную модель времени, в которой вычисление рассматривается как последовательность шагов исполнения. На каждом шаге фиксируются изменения состояний переменных и выполняемые над ними операции. Такая модель позволяет описывать эволюцию вычислений, фиксировать зависимости между операциями и задавать поведение системы при параллельном и асинхронном исполнении.

Обновим описание переменных с учетом дискретного времени. Пусть:

$V = \{v_1, \ldots, v_n\}$ — множество переменных,

$T = \{t_0, t_1, \ldots, t_m\}$ — дискретное множество шагов исполнения программы,

s: $V \times T \to S$, которое задает состояние переменной v в момент времени t.

Таким образом, состояние переменной может изменяться от шага к шагу. Например, на шаге t_0 переменная может быть создана, на шаге t_1— записано значение, а на t_2 — произведено чтение.

Введем определение допустимых операций доступа с учетом времени. Каждая операция теперь определяется на моменте времени и может порождать изменение состояния:

create(v, t): s(v, t):= (name, type, shape, undefined).

write(v, d,t): если s(v, t).state ≠ ready, s(v, t`):= ready, t` > t.

read(v, t) → d блокирует до ∃ t`≤ t: s(v, t`).state = ready

wait(v, t): блокирует выполнение до тех пор,

пока ∃ t` ≤ t: s(v, t`).state = ready.

consume([v_1, …, v_1], f, t): если ∀i: ∃ t_i ≤ t: s(v, t_i).state = ready, то выполняется функция f(...), которая создает изменения в состояниях. Выполняет её со значениями v1, …, vn, когда они все готовы.

slice(v, i_1, i_2, t): shape(v) = (n), $0 \le i_1 \le i_2 \le n$,∃ t` ≤ t: s(v, t`).state = ready, возвращает подмассив v` = v[i_1: i_2].

submatrix(v, i_1, i_2, j_1, j_2): если shape(v) = (n, m), $0 \le i_1 \le i_2 \le n$,

$0 \le j_1 \le j_2 \le m$, ∃t` ≤ t: s(v, t`).state = ready. Возвращает подматрицу v` = v[i_1:i_2, j_1:j_2].

Также отображение размещения переменной в памяти:

μ: $V \times T \to A \times Offset$

показывает, где именно размещены данные переменной в момент времени t. Это позволяет учитывать перемещения данных между аренами, а также реализацию стратегий повторного использования памяти.

Контроль доступности значений также зависит от времени:

ready: $V \times T \to \{0, 1\}$, что эквивалентно s(v, t).state ∈ {ready, undefined}.

Введение дискретного времени позволяет:

- фиксировать изменения состояния переменных во времени;
- формализовать причинно-следственные связи между операциями;
- учитывать порядок исполнения при управлении памятью;
- поддерживать смешанные режимы доступа (синхронный, асинхронный).

Таким образом, дополнение модели переменных формальной временной структурой делает её пригодной для описания процессов исполнения, что особенно важно в системах, генерирующих код автоматически и работающих с частичной информацией о будущих вычислениях.

Ниже приведена сводная таблица методов, предусловий и возможных ошибок при использовании вышеописанного интерфейса взаимодействия с переменными (Таблица 1).

Таблица 1 - Допустимые операции, предусловия и возможные ошибки

Операция	Предусловие	Возможные ошибки
create(v)	$v \notin V$	переменная уже существует
read(v)	$v \in V$	переменная не существует
consume($[v_1, \ldots, v_l]$, f)	$\forall i\ v_i \in V$	переменная не существует
$write(w,\ d)$	$v \in V$, d соответствует типу переменной	переменная не существует
$wait(v)$	$v \in V$	переменная не существует
slice(v, i_1, i_2)	shape(v) = (n), $0 \leq i_1 \leq i_2 \leq n$	переменная не существует, недействительный индекс, неверная форма переменной
submatrix(v, i_1, i_2, j_1, j_2)	shape(v) = (n, m), $0 \leq i_1 \leq i_2 \leq n$, $0 \leq j_1 \leq j_2 \leq m$	переменная не существует, недействительный индекс, неверная форма переменной

2.3 Алгоритмы, поддерживающие набор допустимых операций модели переменных

Для реализации описанного набор допустимых операций модели переменных необходимо формальное описание внутреннего механизма размещения, хранения и обращения с данными в памяти вычислителя. Основное требование, предъявляемое к реализации — обеспечение эффективного повторного использования памяти, управление временем жизни данных и уменьшение накладных расходов на аллокации. Это актуально в контексте автоматической генерации программ в системе активных знаний, где структура задач может быть заранее неизвестна.

Рассмотрим аренную модель управления памятью как внутреннюю реализацию. Пусть имеется множество арен (пулов памяти):

$A = \{A_1, \ldots, A_k\}$

каждая арена представляет собой буфер фиксированного или динамически расширяемого размера, выделенный под размещение данных переменных. Пусть функция

$$\mu: V \times T \to \bigcup_{i=1}^{k} A_i$$

задает отображение переменной на конкретный блок памяти внутри арены. То есть, для каждой переменной $v \in V$ отображение $\mu(v, t)$ определяет смещение внутри соответствующей арены Ai, где хранятся данные.

Каждая арена управляется локальным аллокатором, который поддерживает следующие функции:

allocate(size) → offset — выделяет блок требуемого размера,

deallocate(offset) → Unit — помечает блока как освобожденный,

reuse(offset) → Unit — разрешает переиспользование блока.

Таким образом, возникает отображение:

alloc: $V \times N \to Offset \subset A$, где Offset содержит информацию о позиции и длине блока в арене.

Дополнительное отображение

map: Offset → RAM ∪ VRAM ∪ SharedMem

отвечает за физическое связывание логического блока в арене с конкретной памятью вычислителя: основной ОЗУ, видеопамятью или общей памятью при распределенном хранении.

Для предотвращения гонок используется структура контроля доступности — булева функцияready: V → {0, 1} или, эквивалентно, булево поле state каждой переменной. Это позволяет реализовывать семантику операций wait, read, consume через проверку флага готовности.

Наличие аренной модели позволяет:

- выделять память быстро, без системных вызовов;
- переиспользовать ранее освобождённые блоки без их дефрагментации;
- реализовать слабую локальность — несколько переменных могут размещаться последовательно в рамках одной арены, улучшая кэширование;
- отображать данные на разные уровни памяти (включая GP).

В контексте автоматической генерации программ (как в системе активных знаний LuNA), это особенно важно. Программы генерируются динамически, по мере поступления данных и управляющих событий, и заранее определить структуру размещения невозможно. Аренная модель предоставляет механизм, где:

- выделение и освобождение памяти можно привязать к структуре вычислений;
- можно явно или неявно строить зависимости между переменными через совместное размещение;
- можно контролировать время жизни данных через жизненный цикл арены.

Таким образом, представленная модель переменных, дополненная аренной моделью размещения и управления данными в памяти, позволяет реализовать задачи управления памятью с учетом:

- произвольного порядка исполнения (возможность синхронного и асинхронного доступа);
- автоматической обработки состояний готовности переменных;
- повторного использования и адаптивного выделения памяти в условиях частичной или полной неизвестности о будущем вычислении.

Эта архитектура может быть формализована в виде композиции трех отображений:

$V \to s, S \to \mu A \to \text{map Memory}$

Такая композиционная структура делает возможным эффективное управление памятью в условиях автоматической генерации исполнения, где пользовательский код полностью отделен от механики аллокации и трансляции данных.

Ниже представлен пример использования модели на простой задаче. Цель — обработать два массива данных, сложить их поэлементно, и выделить из результата подмассив (Рисунок 4). Это отражает часто встречающуюся модель вычислений в системах научных расчётов, потоковых трансформаций и ML-пайплайнов. В задаче создаются три переменные a, b и c.

Далее в a и b записываются массивы чисел. После этого запускается операция consume(a, b, c, add), которая должна дождаться, пока a и b будут готовы, выполнить над ними операцию сложения (поэлементно), и записать результат в переменную c. Затем выделяется переменная c_slice, и в нее

записывается срез данных из c (элементы с индексами 2 по 4 включительно).

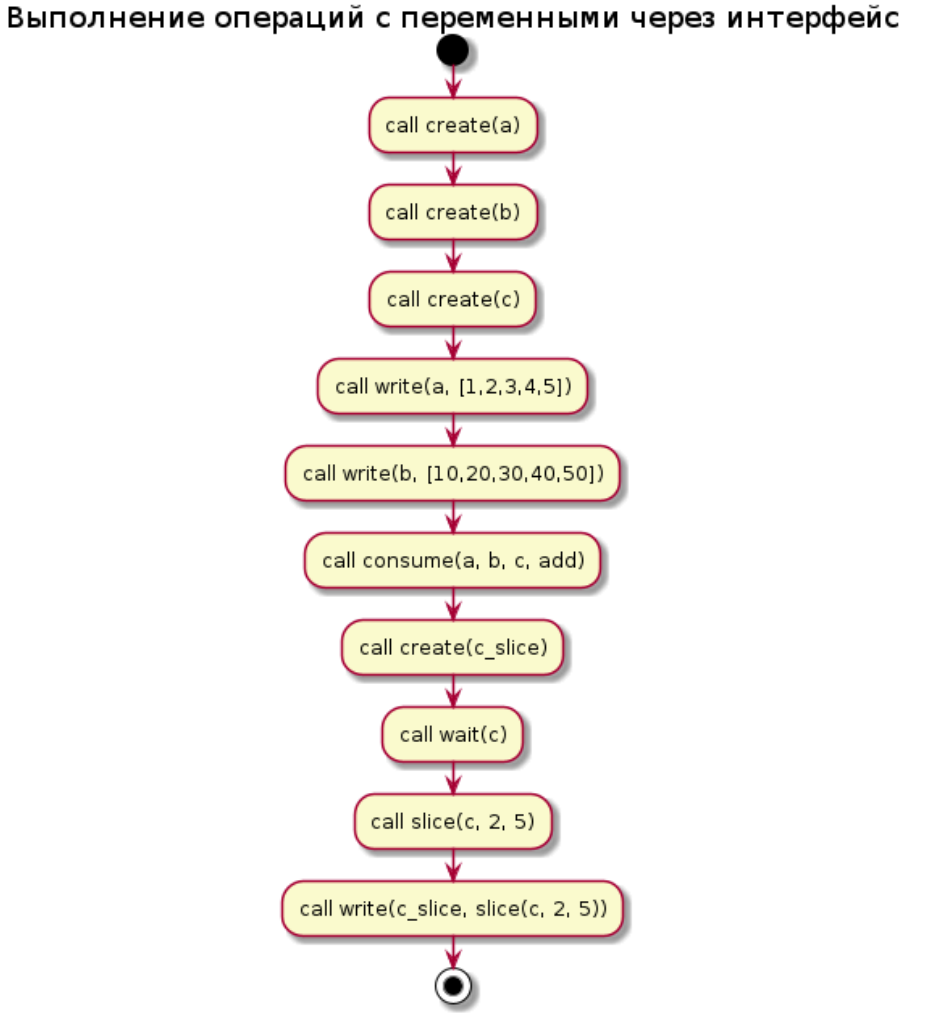

Рисунок 4 — Последовательность операций обработки двух массивов через операции модели управления памятью

Диаграмма ниже демонстрирует последовательность операций в рамках задачи, моделирующей асинхронное выполнение операции сложения над двумя массивами и извлечение среза результата (Рисунок 5).

В данной модели взаимодействуют несколько сущностей: пользователь (User), переменные-массивы (Var a, Var b, Var c, Var c_slice), аренная модель памяти (Memory Arena) и диспетчер задач (Scheduler). Пользователь является инициатором операций, переменные представляют собой объекты, над которыми производится вычислительная работа, Memory Arena управляет выделением и хранением памяти, а Scheduler обеспечивает асинхронное выполнение операций и координацию состояний.

На первом этапе происходит инициализация. Пользователь создаёт три переменные: a, b и c. После создания в переменные a и b записываются значения. Каждая из этих переменных взаимодействует с Memory Arena, которая выделяет необходимую память и сохраняет соответствующие данные. После записи переменные переводятся в состояние ready, что означает их готовность к дальнейшему использованию в вычислениях.

На втором этапе запускается асинхронная операция. Пользователь инициирует выполнение команды consume(a, b, c, add), которая передается диспетчеру задач Scheduler. Планировщик отслеживает состояние переменных a и b и ожидает их перехода в состояние ready. После этого выполняется операция поэлементного сложения содержимого массивов a и b, а результат записывается в переменную c. Запись результата также производится через Memory Arena, в которой выделяется новая область памяти. Переменная c переводится в состояние ready после успешного завершения записи.

На третьем этапе осуществляется извлечение среза из результата. Пользователь создаёт переменную c_slice, предназначенную для хранения части результата. Перед тем как приступить к извлечению, вызывается операция wait(c), обеспечивающая блокировку до тех пор, пока переменная c не перейдет в состояние ready. После этого пользователь вызывает операцию slice(2,5), указывая желаемые границы среза. Memory Arena вычисляет соответствующее смещение в памяти, извлекает данные и возвращает срез {33, 44, 55}. Полученный фрагмент записывается в переменную c_slice, для которой также производится выделение памяти, а состояние переводится в ready, что завершает цикл обработки.

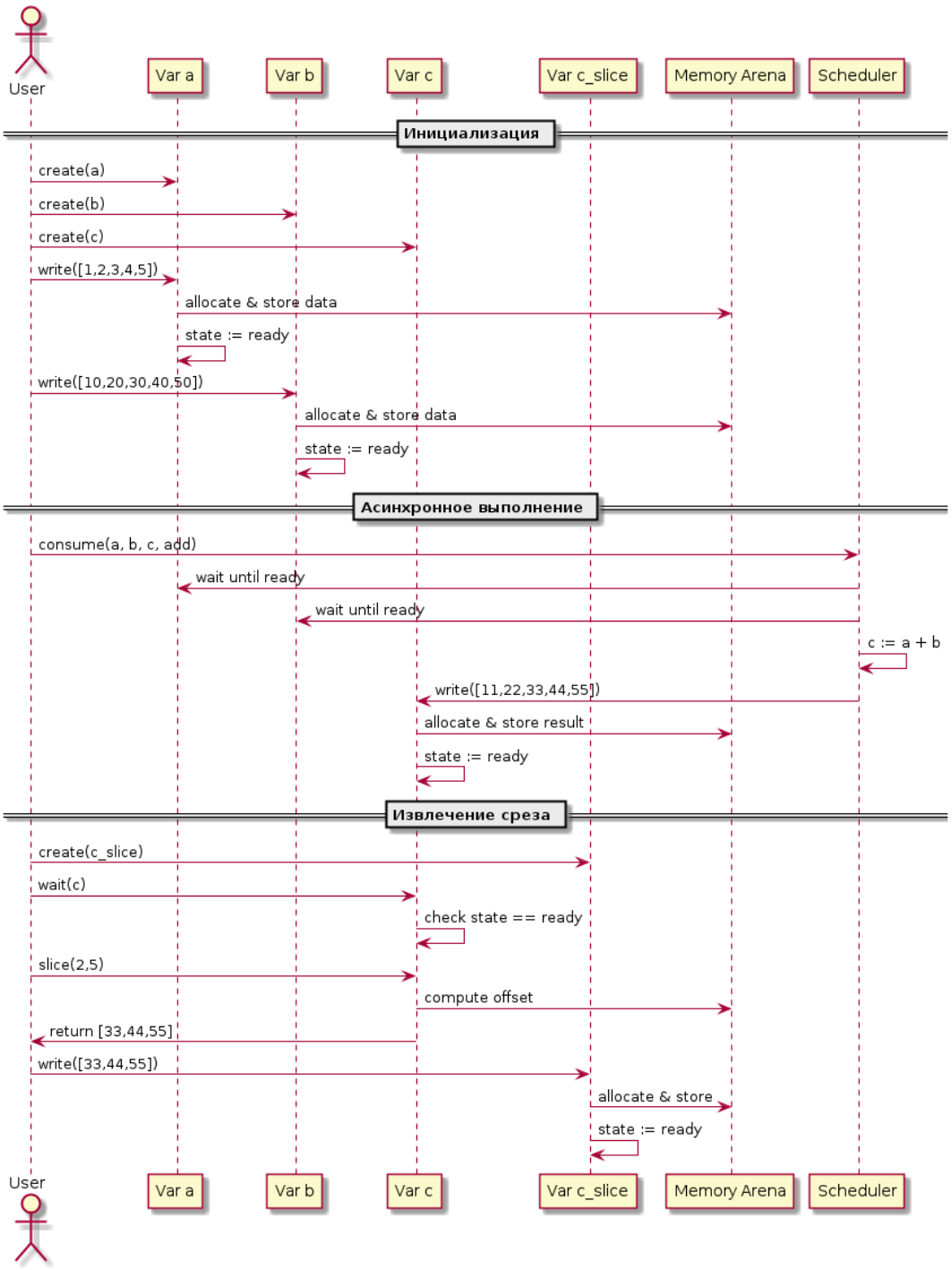

Рисунок 5 — Диаграмма последовательности обработки двух массивов через операции модели управления памятью

2.4 *Итоги*

В этом разделе были рассмотрены задачи управления памятью в системе активных знаний LuNA. Поскольку код, выполняющий вычисления, в этой системе генерируется автоматически, традиционные методы, такие как ручное управление или статическое размещение данных, не подходят. Это создаёт

необходимость в механизмах, способных учитывать динамический характер исполнения и адаптироваться к меняющимся условиям.

Для формализации представления данных используется аппарат описания переменных и операций над ними. Это позволяет явно задавать области жизни данных, их использование и зависимости, что важно для планирования размещения и освобождения ресурсов.

Предложена аренная модель управления памятью, в которой память выделяется в рамках изолированных контекстов — арен. Такой подход даёт возможность повторного использования участков памяти, упрощает локализацию ресурсов и может снижать фрагментацию, в том числе в распределенных и параллельных сценариях.

Модель задаёт ограничения и интерфейсы, отделяющие описание вычислений от решений по управлению ресурсами. Это даёт возможность применять одни и те же вычислительные схемы в различных средах исполнения без изменений на уровне пользовательского кода.

Ожидается, что использование этой модели позволит сократить общее количество используемой памяти за счёт более точного контроля времени жизни данных и возможности повторного использования участков памяти вычислителя. Эти предположения требуют экспериментального подтверждения и сравнительного анализа с альтернативными подходами.

3 Программная реализация

Выполненная работа представляет собой реализацию библиотеки на языке Си для управления переменными различных типов с использованием концепции арены — заранее выделенного непрерывного участка памяти. Выбор языка Си обусловлен тем, что большая часть операций для различных предметных областей в системе активных знаний LuNA уже реализованы на этом языке, кроме того, в целевых предметных областях, с которыми ещё предстоит работать и решать в них задачи автоматической генерации высокопроизводительных программ, также часто используют решения на языке Си.

Для хранения переменных и управления их связями в библиотеке используется односвязный список. Этот выбор обусловлен простотой реализации и низкими накладными расходами на вставку новых элементов.

Вместе с тем, предусмотрена возможность замены односвязного списка на ассоциативный массив, например, хеш-таблицу. Такая замена может существенно ускорить поиск переменных по имени, особенно при большом объеме данных. Однако, чтобы определить, какой подход лучше с точки зрения производительности, потребуется накопить практический опыт применения модуля на реальных задачах.

Одной из ключевых задач при реализации библиотеки управления переменными является расширение арены памяти при ее заполнении. В рамках разработки было рассмотрено несколько возможных решений, из которых два — использование realloc и чанковая модель (Рисунок 6) — были проанализированы более подробно, а еще два — индексная адресация и разделение данных и метаинформации — описаны концептуально как потенциальные направления развития.

```
typedef struct Var {
  char* name;
  VarType type;
  size_t dim1, dim2;
  VarState state;
  void* data;

  struct Var* parent;
  size_t offset1, offset2;
  pthread_cond_t cond;
  pthread_mutex_t lock;
} Var;

typedef struct VarNode {
  Var* var;
  struct VarNode* next;
} VarNode;

typedef struct ArenaChunk {
  unsigned char* data;
  size_t size;
  size_t used;
  struct ArenaChunk* next;
} ArenaChunk;

typedef struct MemManager {
  unsigned char* arena;
  size_t arena_size;        // сумма всех чанков
  size_t arena_used;        // суммарное использованное
  ArenaChunk* chunks;       // список чанков
  VarNode* vars;
  pthread_mutex_t global_lock;
} MemManager;
```

Рисунок 6 — Реализация чанковой модели управления памятью

В подходе с использованием realloc арена представляется единым непрерывным блоком памяти. При исчерпании свободного пространства вызывается realloc, который выделяет новый блок необходимого размера, копирует в него все текущие данные и возвращает новый указатель на начало области. Такой способ на первый взгляд прост в реализации, но имеет ряд существенных недостатков. Во-первых, копирование данных имеет линейную сложность по текущему объему, что делает операцию дорогостоящей при большом числе переменных. Во-вторых, все существующие указатели

становятся невалидными после перемещения памяти, что требует полного обхода всех связанных структур и обновления ссылок. Это особенно критично при использовании указателей в стороннем или пользовательском коде, а также в многопоточной среде, где может потребоваться синхронизация. Кроме того, частые расширения с перераспределением памяти увеличивают нагрузку на аллокатор и могут приводить к фрагментации.

В альтернативной чанковой модели арена реализуется как последовательность независимых блоков (чанков), каждый из которых представляет собой локальную арену фиксированного или возрастающего размера (Рисунок 7). При заполнении текущего чанка выделяется следующий, и новые данные размещаются уже в нём. Такой подход позволяет избежать полного копирования данных и, главное, сохранить валидность всех ранее выданных указателей, поскольку уже выделенная память не перемещается. Размер чанков может увеличиваться, например, удваиваться, что позволяет уменьшить частоту аллокаций по мере роста нагрузки.

```
MemManager* mem_manager_create(size_t
initial_chunk_size) {
  MemManager* mgr =
malloc(sizeof(MemManager));
  if (!mgr) return NULL;

  mgr->arena = NULL;
  mgr->arena_size = 0;
  mgr->arena_used = 0;
  mgr->vars = NULL;
  mgr->tasks = NULL;
  mgr->chunks = NULL;
  pthread_mutex_init(&mgr->global_lock,
NULL);

  unsigned char* dummy = arena_alloc(mgr,
initial_chunk_size);
  (void)dummy;
  mgr->chunks->used = 0;
  mgr->arena_used = 0;
  return mgr;
}
```

Рисунок 7 — Реализация инициализации арен при создании менеджера

Основным техническим вызовом при таком подходе становится организация навигации между чанками и управление локальной структурой размещения внутри каждого блока (Рисунок 8). Однако, по сравнению с затратами на realloc, эта модель оказывается более эффективной и масштабируемой.

```
static void* arena_alloc(MemManager* mgr, size_t size) {
  ArenaChunk* chunk = mgr->chunks;
  int chunk_index = 0;
  while (chunk) {
    size_t free_space = chunk->size - chunk->used;
    if (free_space >= size) {
      void* ptr = chunk->data + chunk->used;
      chunk->used += size;
      mgr->arena_used += size;
      return ptr;
    }
    chunk = chunk->next;
    chunk_index++;
  }
  size_t chunk_size = size;
  ArenaChunk* new_chunk = malloc(sizeof(ArenaChunk));
  if (!new_chunk) return NULL;
  new_chunk->data = malloc(chunk_size);
  if (!new_chunk->data) {
    free(new_chunk);
    return NULL;
  }
  new_chunk->size = chunk_size;
  new_chunk->used = size;
  new_chunk->next = mgr->chunks;
  mgr->chunks = new_chunk;
  mgr->arena_size += chunk_size;
  mgr->arena_used += size;
  if (!mgr->arena)
    mgr->arena = new_chunk->data;
  return new_chunk->data;
}
```

Рисунок 8 — Реализация управления чанками в менеджере памяти

Помимо этих двух подходов, рассматривались и другие архитектурные идеи. Один из них — индексная адресация, при которой переменные не ссылаются напрямую на участки памяти, а используют индексы, через которые

происходит косвенное обращение к таблице размещения. При необходимости перемещения данных достаточно обновить только таблицу, сохранив все внешние ссылки. Этот подход обеспечивает высокую гибкость, но требует косвенного доступа при каждом обращении, увеличивая накладные расходы и усложняя архитектуру.

Другая идея — разделение хранения данных и метаинформации: имена, типы и состояния переменных хранятся отдельно от значений. Такое разбиение позволяет по-разному оптимизировать доступ к метаданным и самим данным, а также уменьшить связность между компонентами. Однако оно влечет за собой необходимость поддержания связей между двумя независимыми пространствами, что может осложнить реализацию и синхронизацию.

В итоге для реализации текущей версии библиотеки была выбрана чанковая модель, как наиболее сбалансированное решение между производительностью и простотой реализации. Остальные варианты остаются потенциальными точками роста для последующих итераций и могут быть полезны в специфических сценариях.

Для обеспечения потокобезопасности в библиотеке используется глобальный мьютекс, который защищает критические участки кода при создании и поиске переменных (Рисунок 9). Однако стоит отметить, что не во всех задачах синхронизация требуется, и желательно иметь возможность отключать этот механизм для повышения производительности. В текущей реализации такой опции нет, но она может быть добавлена в дальнейшем. Кроме того, мьютекс — не единственный возможный примитив синхронизации; в зависимости от специфики задачи можно использовать другие механизмы, например, семафоры или блокировки на уровне отдельных переменных.

Разработанный модуль предоставляет возможности управления памятью с отображением данных на оперативную память вычислителя, но потенциальные возможности модуля намного шире — возможны реализации под GPU, SharedMemmory, обеспечение управления в рамках распределенной памяти,

взаимодействие с внешними сервисами по сети, взаимодействие с файловой системой.

```
int wait_var(MemManager* mgr, const char* name) {
  pthread_mutex_lock(&mgr->global_lock);

  Var* var = find_var(mgr, name);
  if (!var) {
    pthread_mutex_unlock(&mgr->global_lock);
    return -1;
  }

  pthread_mutex_lock(&var->lock);
  while (var->state != STATE_READY) {
    pthread_cond_wait(&var->cond, &var->lock);
  }
  pthread_mutex_unlock(&var->lock);
  pthread_mutex_unlock(&mgr->global_lock);
  return 0;
}
```

Рисунок 9 — Реализация функции блокировки переменной до ее готовности через мьютекс и условную переменную

Кроме того, общая точка входа, которую представляет собой разработанный модуль, создает возможности для журналирования и профилирования операций управления памятью, реализации механизмов рефлексии. Помимо этого, в текущей реализации менеджера памяти нет реализации метода consume для асинхронного исполнения функции по готовности входных данных.

Все вышеперечисленные векторы развития программной реализации модуля выходят за рамки данной работы и требуют дополнительных исследований в области автоматического управления памятью в системе активных знаний LuNA.

4 Тестирование

Цель данного этапа — убедиться в отсутствии явных ошибок реализованного менеджера памяти при различных сценариях использования. Тестирование проводится для подтверждения того, что модуль создает и управляет переменными различных типов, правильно работает с ареной памяти, поддерживает расширение объёма при необходимости, обеспечивает запись и чтение данных, а также синхронизирует доступ к переменным в многопоточном режиме. Кроме того, тестовые сценарии демонстрируют примеры взаимодействия с модулем, что упрощает его дальнейшее использование, поддержку и разработку. Это необходимо для уверенного использования разработанного менеджера в последующем сравнительном анализе с существующими решениями, включая LuNA v6 и ручную реализацию.

4.1 Проверка работоспособности модуля на тестовых сценариях

Для проверки корректности реализованного менеджера памяти был разработан и последовательно выполнен ряд автоматических тестов. Каждый из них проверяет отдельные функциональные аспекты системы. В первом тесте test_mem_manager_create_destroy проверяется корректность создания и освобождения менеджера памяти: выделяется арена фиксированного размера и после соответствующих проверок освобождается. Далее в тесте test_create_var проверяется логика создания переменных — успешное добавление новой переменной и невозможность повторного добавления переменной с тем же именем. В случае успеха гарантируется правильная инициализация всех полей структуры Var.

Тест test_arena_expand проверяет способность менеджера динамически расширять арену при нехватке памяти: создаётся большое количество переменных, и по завершении гарантируется, что размер арены увеличился по сравнению с начальным. В тесте test_read_write_var проверяется логика однократной записи данных в переменную и корректного чтения этих данных.

В случае повторной попытки записи система возвращает ошибку, как и ожидается.

Далее, test_create_slice проверяет возможность создания среза одномерного массива. После создания переменной и её инициализации данными формируется новый объект, представляющий срез заданного диапазона, и проверяется корректность его содержимого. Аналогично, test_create_submatrix проверяет работу с подматрицами: из двумерной матрицы вырезается подматрица, и сравниваются её значения с ожидаемыми.

Тест test_wait_var фокусируется на многопоточном взаимодействии. В нём основной поток создаёт переменную, а затем при помощи wait_var ожидает ее инициализации. Второй поток имитирует длительную обработку и позже записывает значение. После завершения работы потока основной поток читает значение переменной и проверяет его на корректность. Этот тест демонстрирует корректную работу блокировок, ожидания условий и отсутствие взаимных блокировок.

В заключение, все тесты запускаются последовательно в main, и при успешном прохождении выводятся соответствующие сообщения. Таким образом, реализованные тесты охватывают ключевые функции модуля и подтверждают его стабильную и предсказуемую работу как в однопоточном, так и в многопоточном окружении.

4.2 Сравнительный анализ расхода ресурсов вычислителя менеджера памяти с ручной реализацией и luna v6

В данном разделе изложены результаты сравнительного анализа расхода памяти на задаче блочного умножения матриц на процессоре. Представлены три параллельные реализации алгоритма — ручная реализация, реализация с использованием реализованного модуля управления и реализация посредством LuNA v6. Задача блочного управления матриц была выбрана по нескольким причинам:

- программное решение задачи требует обработки большого объема данных;
- алгоритм пригоден для распараллеливания, что позволяет в рамках задачи проверить работу модуля в рамках исполнения параллельной программы;
- задача является достаточно простой, чтобы суметь интерпретировать результаты тестов.

Исследуемой метрикой для анализа был расход оперативной памяти вычислителя.

Ручная реализация и реализация с использованием менеджера памяти представляют собой программы на языке Си, для анализа расхода памяти для исполняемых файлов использовалась утилита valgrind. Для анализа реализации посредством LuNA v6 необходимо было провести некоторые приготовления и эксперименты, которые описаны ниже.

Так как LuNA v6 представляет собой решение для автоматической генерации программ, то алгоритм умножения был описан на специальном языке, исходный код которого было необходимо преобразовать в код на языке Си, затем собрать в динамическую библиотеку и исполнить с использованием рантайм системы.

Кроме того, в процессе работы рантайм системы происходят системные вызовы, которые порождают дочерние процессы операционной системы, что усложняет анализ расхода памяти через valgrind. Использование флага --trace-children=yes позволило отследить расход памяти по дочерним процессам, но не позволило явно установить на каком этапе выполнения память потребляется — при генерации кода, исполнении, последующих вызовах для удаления сгенерированных файлов из окружения. Поэтому были предприняты следующие шаги:

- LuNA v6 была запущена с флагом –compile-only для получения исходного кода решения в виде динамической библиотеки;
- динамическая библиотека была исполнена через рантайм систему LuNA v6, а для отслеживания аллокаций были переопределены символы malloc, calloc, free, realloc и подгружены через переменную окружения LD_PRELOAD, что позволило обернуть библиотечные вызовы управления памятью в куче и отслеживать их с последующим журналированием;

Расходы памяти исследовались в динамике изменения размеров матриц — 512x512, 1024x1024, 2048x2048, 4096x4096 (Рисунок 10, Таблица 2).

Таблица 2 — Сводная таблица результатов расхода памяти вычислителя

Размер матрицы	Ручная реализация, байт	Менеджер памяти, байт	Реализация на LuNA v6, байт
512	6293568	6819710	11088771
1024	25167936	27266942	29963139
2048	100665408	102912642	105460322
4096	402655296	405854370	407450146

Тестирование проводилось на персональном ноутбуке Huawei MateBook 14, оснащенном следующими аппаратными и программными характеристиками:

- Процессор: AMD Ryzen 5 4600H (4 ядра, 8 потоков, базовая частота 2.7 ГГц);
- Оперативная память: 8 ГБ DDR4;
- Операционная система: Linux Ubuntu 22.04 LTS (64-битная).

По результатам экспериментального тестирования можно сделать следующие выводы:

- модуль управления памятью способен показывать приемлемый для практического использования расход по памяти в прикладных вычислительных задачах;

- в модуле возникают дополнительные расходы памяти для служебных целей — хранение дополнительной информации о переменных, объем которых также можно считать допустимым.

Рисунок 10 — Сравнительный анализ расхода памяти вычислителя в логарифмической шкале

4.3 Выводы

Проведенное тестирование и сравнительный эксперимент по задаче блочного умножения матриц продемонстрировали, что цели тестирования были достигнуты. Реализованный модуль управления памятью показал свою работоспособность при различных размерах данных.

В сравнении с ручной реализацией и реализацией в среде LuNA v6, модуль продемонстрировал сбалансированное соотношение между потреблением памяти и уровнем автоматизации. При этом он обеспечивал прозрачное управление памятью и правильное освобождение ресурсов в рамках параллельного исполнения программ.

Таким образом, на основе полученных экспериментальных данных можно сделать вывод о пригодности модуля управления памятью для практического применения в составе систем автоматической генерации кода и других

прикладных вычислительных задач, где важны контроль ресурсов и масштабируемость.

ЗАКЛЮЧЕНИЕ

В рамках выполнения выпускной квалификационной работы был разработан, реализован и протестирован модуль управления памятью, ориентированный на использование в системах автоматической генерации параллельных программ на основе декларативных описаний. Разработка направлена на сокращение избыточных аллокаций, повышение управляемости состояниями переменных и обеспечение совместимости с многопоточными вычислениями. Также были разработаны и составлены описание программы (приложение А) и руководство программиста (приложение Б).

Модуль реализует формальную модель управления переменными и их размещением в памяти вычислителя, а также включает алгоритмы, поддерживающие работу этой модели. Проведено экспериментальное тестирование на задаче блочного умножения матриц с использованием трех реализаций: ручной, с использованием разработанного модуля и на базе системы LuNA v6. По результатам сравнительного анализа показано, что модуль демонстрирует приемлемые характеристики расхода памяти, пригодные для использования в прикладных задачах.

Защищаемые положения:

1. Разработана формальная модель управления переменными в памяти вычислителя;
2. Реализован модуль управления переменными в памяти вычислителя, проведено экспериментальное исследование работоспособности алгоритма управления переменными.

Несмотря на то что интеграция модуля в систему LuNA пока не осуществлена, он может быть использован как в этой системе, так и в других аналогичных платформах. Практическая значимость работы подтверждена результатами тестирования и демонстрирует потенциал дальнейшего развития в направлении автоматизации управления памятью.

Планируется продолжение работы по следующим направлениям:

1. Интеграция разработанного модуля в LuNA и другие фреймворки;
2. Расширение модели для поддержки распределенных систем и гетерогенных вычислений;
3. Разработка механизмов статического анализа для автоматического синтеза правил управления памятью.

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».

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.
(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Малышкин В. Э. Технология фрагментированного программирования // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – № 46(305). – С. 45–55.
2. Erlingsson Ú. How to secure existing C and C++ software without memory safety // arXiv preprint arXiv:2503.21145. – 2025. – URL: https://arxiv.org/abs/2503.21145 (дата обращения: 03.05.2025).
3. de Vilhena P. E. et al. Extending the C/C++ memory model with inline assembly // Proceedings of the ACM on Programming Languages. – 2024. – Vol. 8, No. OOPSLA2. – P. 1081–1107.
4. Tranter J. Recent initiatives to improve C and C++ memory safety // EmbeddedRelated.com. – 2025. – 22 Jan. – URL: [https://www.embeddedrelated.com] (дата обращения: 03.04.2025).
5. Jones R., Hosking A., Moss E. The Garbage Collection Handbook: The Art of Automatic Memory Management. – 2nd ed. – Chapman and Hall/CRC, 2023. – 500 p.
6. Gao K. et al. Julia language in machine learning: Algorithms, applications, and open issues // Computer Science Review. – 2020. – Vol. 37. – P. 100254.
7. Roesch E. et al. Julia for biologists // Nature Methods. – 2023. – Vol. 20, No. 5. – P. 655–664.
8. Byrne S., Wilcox L. C., Churavy V. MPI.jl: Julia bindings for the Message Passing Interface // Proceedings of the JuliaCon Conferences. – 2021. – Vol. 1, No. 1. – P. 68.
9. Barros D. A. et al. Evaluating shared memory parallel computing mechanisms of the Julia language. – 2023.

10. Alomairy R. et al. Dynamic task scheduling with data dependency awareness using Julia // 2024 IEEE High Performance Extreme Computing Conference (HPEC). – IEEE, 2024. – P. 1–7.
11. Developers T. F. TensorFlow // Zenodo. – 2022. – URL: [https://zenodo.org/record/1234567] (дата обращения: 03.04.2025).
12. Imambi S., Prakash K. B., Kanagachidambaresan G. R. PyTorch // Programming with TensorFlow: solution for edge computing applications. – 2021. – P. 87–104.
13. Scuttari M. Design and implementation of a Modelica compiler with MLIR and LLVM. – 2020.
14. Williams B. Designs and Optimizations for Heterogeneous High Performance Computing: PhD thesis. – Texas Tech University, 2023.
15. Dai H. et al. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment // Science China Information Sciences. – 2022. – Vol. 65. – P. 1–17.
16. Lin M. Automatic functional differentiation in JAX // arXiv preprint arXiv:2311.18727. – 2023. – URL: [https://arxiv.org/abs/2311.18727] (https://arxiv.org/abs/2311.18727) (дата обращения: 16.04.2025).
17. Santoso O. K. A. et al. Rust's memory safety model: An evaluation of its effectiveness in preventing common vulnerabilities // Procedia Computer Science. – 2023. – Vol. 227. – P. 119–127.
18. Seifi N., Al-Mamun A. Optimizing memory access efficiency in CUDA kernel via data layout technique // Journal of Computer and Communications. – 2024. – Vol. 12, No. 5. – P. 124–139.
19. Chien S., Peng I., Markidis S. Performance evaluation of advanced features in CUDA unified memory // 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC). – IEEE, 2019. – P. 50–57.

20. Paul S. R. et al. A fine-grained asynchronous bulk synchronous parallelism model for PGAS applications // Journal of Computational Science. – 2023. – Vol. 69. – P. 102014.
21. Bodei C., Ferrari G. L., Priami C. Programming Languages with Applications to Biology and Security. – Springer, 2008. – 240 p.
22. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах: дис. ... канд. техн. наук: 05.13.11. – Новосибирск, 2022. – 150 с.

ПРИЛОЖЕНИЕ А

МОДУЛЬ УПРАВЛЕНИЯ ПЕРЕМЕННЫМИ В ПАМЯТИ ВЫЧИСЛИТЕЛЯ

ОПИСАНИЕ ПРОГРАММЫ

Листов 12

Новосибирск, 2025

СОДЕРЖАНИЕ

Аннотация

В данном документе приведено описание модуля управления переменными в памяти исполнительной системы LuNA. Модуль реализован в виде библиотеки на языке C и предназначен для автоматического управления размещением и временем жизни переменных в LuNA-программах, что позволяет сократить объем избыточных аллокаций и повысить эффективность использования оперативной памяти.

Пользователь получает в распоряжение механизм, обеспечивающий отслеживание состояния переменных, контроль их актуальности и упорядоченный доступ в условиях параллельного исполнения. Для компиляции и использования модуля необходим компилятор gcc.

Оформление программного документа «Описание программы» выполнено в соответствии с требованиями ГОСТ 19.402–78 «ЕСПД. Описание программы» и ГОСТ 19.105–78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением №1)».

.

Общие сведения

1.1 Обозначение и наименование программы

Полное наименование: модуль управления переменными в памяти вычислителя для системы автоматической генерации параллельных программ LuNA.

1.2 Необходимое программное обеспечение

Для функционирования модуля требуется:

- операционная система Linux (рекомендуется Ubuntu 22.04);
- компилятор: gcc;

1.3 Языки программирования

Модуль реализован на языке C. В процессе разработки использовались среда разработки VsCode .

Функциональное назначение

2.1 Назначение

Модуль предназначен для управления переменными и их размещением в оперативной памяти вычислителя в рамках исполнения программ системы LuNA. Он обеспечивает отслеживание состояния переменных, контроль доступа и управление их жизненным циклом.

Описание логической структуры

3.1 Алгоритм работы

Модуль принимает информацию о переменных и их связи с вычислениями в виде формализованных метаданных.

На этой основе:

- формируется модель состояний переменных;
- строится структура, отслеживающая связи между переменными, вычислениями и участками памяти;
- выполняется автоматическое обновление состояния переменных по ходу выполнения программы.

3.2 Используемые методы

- контейнеры для хранения и поиска информации о переменных;
- идентификация состояния переменных: «создана», «используется», «доступна для чтения;
- методы контроля синхронного доступа к переменным.

3.3 Структура программы

Модуль	Назначение
Variable	Представляет переменную с уникальным идентификатором, типом, текущим состоянием
MemoryManager	Отвечает за отслеживание размещения переменных в памяти, выделение и освобождение ресурсов

3.4 Взаимодействие компонентов

Компоненты взаимодействуют через внутренние API модуля. Обновления состояния переменных происходят автоматически.

Используемые технические средства

Модуль является частью консольного исполнения и требует следующих аппаратных ресурсов:

- Оперативная память: от 2 ГБ;
- Процессор: многоядерный;
- Операционная система: Linux (Ubuntu 22.04 или аналогичная).

Вызов и загрузка

Модуль загружается автоматически при запуске программы, сгенерированной системой LuNA.

Входные данные

Модуль не принимает входные данные в явном виде. Он реализован в виде библиотеки на C, работает с внутренними структурами данных во время исполнения программы.

Данные о переменных (их идентификаторы, типы, состояния и зависимости) формируются в процессе работы внешней системы и поступают в модуль через вызовы соответствующих функций и методов API модуля.

Входной информацией являются объекты и события, инициируемые во время исполнения, а не внешние файлы или потоки.

Выходные данные

Модуль не формирует отдельного выходного файла. Его работа отражается во внутреннем состоянии исполнительной системы и влияет на поведение памяти при выполнении программы.

Лист регистрации изменений

Таблица 3 – Лист регистрации изменении описания программы

Лист регистрации изменений									
Изм.	Номера листов (страниц)				Всего листов (страниц) в документе	Номер документа	Входящий номер сопроподи-тельного документа и дата	Подпись	Дата

ПРИЛОЖЕНИЕ Б

МОДУЛЬ УПРАВЛЕНИЯ ПЕРЕМЕННЫМИ В ПАМЯТИ ВЫЧИСЛИТЕЛЯ

РУКОВОДСТВО ПРОГРАММИСТА

Листов 12

Новосибирск, 2025

СОДЕРЖАНИЕ

Аннотация

В данном документе приведено руководство программиста для модуля управления переменными в памяти вычислителя, разработанного в рамках системы фрагментированного программирования LuNA. Модуль реализован на языке C. Разработка осуществлялась с использованием редакторов VsCode.

Основной функцией модуля является отслеживание и управление переменными в памяти вычислителя, обеспечивая эффективное освобождение ресурсов при завершении вычислений. Документ оформлен в соответствии с требованиями ГОСТ 19.504–79 «ЕСПД. Руководство программиста» и ГОСТ 19.105–78 «ЕСПД. Общие требования к программным документам».

Характеристика программы

1.1 Описание основных характеристик программы

Модуль расширяет функциональность системы LuNA механизмом контроля и освобождения переменных в оперативной памяти. Обработка переменных осуществляется в контексте фрагментированных вычислений, без участия пользователя.

1.2 Описание основных особенностей программы

Модуль работает на основе внутреннего представления о структуре переменных и данных, с ними связанных. Вызовы функций модуля может интегрировать в код генератор, либо непосредственно из динамической библиотеки вызывать внешняя рантайм система.

Обращение к программе

2.1 Описание процедур вызова программы

Модуль управления переменными в памяти вычислителя не требует прямого вызова со стороны пользователя. Все операции выполняются автоматически в процессе выполнения программы.

Внутреннее управление переменными основано на отслеживании состояния данных, ассоциированных с переменными, и их взаимных зависимостей.

Входные и выходные данные

3.1 Организация используемой входной информации

Модуль не получает внешних входных данных. Вся необходимая информация о переменных, их области видимости и связях с фрагментами вычислений формируется внешней системой в процессе компиляции и исполнения программы.

3.2 Организация используемой выходной информации

Модуль не производит выходных данных. Управление памятью осуществляется внутри исполнительной среды и прозрачно для пользователя.

Лист регистрации изменений

Таблица 4 – Лист регистрации изменении руководства программиста

Лист регистрации изменений									
Изм.	Номера листов (страниц)				Всего листов (страниц) в документе	Номер документа	Входящий номер сопроподи-тельного документа и дата	Подпись	Дата