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 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Симонова Владислава Игоревича

Тема работы:

РАЗРАБОТКА КОНФИГУРИРУЕМОЙ ПОДСИСТЕМЫ ПРОФИЛИРОВАНИЯ ДЛЯ СИСТЕМЫ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В. Э./……..
«30» мая 2025 г.

Руководитель ВКР
д.т.н., профессор
заведующий каф. ПВ ФИТ НГУ
Малышкин В. Э./……...
«30» мая 2025 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С. Е./……..
«30» мая 2025 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

Направление подготовки: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В. Э.
…….
«22» октября 2024 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студенту Симонову Владиславу Игоревичу, группы 23224

Тема Разработка конфигурируемой подсистемы профилирования для системы LuNA
утверждена распоряжением проректора по учебной работе от 21.10.2024 № 0378
Срок сдачи студентом готовой работы 20 мая 2025 г.
Исходные данные (или цель работы): разработать и реализовать конфигурируемую подсистему профилирования в системе фрагментированного программирования LuNA
Структурные части работы: обзор и анализ предметной области, постановка задачи, формулирование требований к подсистеме, разработка и реализация конфигурируемой подсистемы, тестирование

Руководитель ВКР
д.т.н., профессор
зав. каф. ПВ ФИТ НГУ
Малышкин В. Э./……..
«22» октября 2024 г.

Задание принял к исполнению

Симонов В. И./……..
«22» октября 2024 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С. Е./……..
«22» октября 2024 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Система фрагментированного программирования LuNA – это программная система, предназначенная для разработки и выполнения параллельных приложений на основе крупноблочного параллелизма [1]. Она реализует технологию фрагментированного программирования, при которой программа состоит из независимых фрагментов вычислений. Каждый фрагмент включает описание входных и выходных данных, а также код, выполняющий вычисления. Входные и выходные данные фрагментов представлены в виде неизменяемых фрагментов данных фиксированного размера.

Программа на языке LuNA задаёт параллельный алгоритм в виде компактного описания потенциально неограниченного множества фрагментов и поддерживает конструкции высокого уровня, такие как циклы, условные операторы и подпрограммы. Исполнение фрагментированной программы осуществляется исполнительной системой LuNA, которая управляет созданием, перемещением и выполнением фрагментов вычислений и данных. Выполнение происходит асинхронно и определяется только зависимостями между фрагментами: фрагмент может быть выполнен, как только доступны все его входные данные.

Профилирование – это процесс сбора и анализа характеристик исполнения программы, с целью исследования её поведения и производительности. Оно позволяет выявить участки кода, которые потребляют наибольшее количество ресурсов (процессорного времени, памяти и др.), а также определить наиболее частые операции.

Низкоуровневые события – это события, происходящие на уровне реализации программы и системных компонентов, такие как вызовы функций, обращения к памяти, изменения состояний потоков или выполнение инструкций процессора. Они характеризуются высокой детализацией и частотой возникновения и используются для отладки и анализа производительности.

Высокоуровневые события – это логические действия, отражающие поведение программы с точки зрения её бизнес-логики или взаимодействия с пользователем, такие как запуск приложения, вход в систему, обработка запроса или регистрация ошибки. Эти события фиксируются редко и служат для мониторинга состояния системы.

Трассировка – это процесс детальной регистрации низкоуровневых событий, происходящих в системе во время её работы. В ходе трассировки фиксируются такие действия, как вызовы функций, изменения состояний, обращения к памяти и другие технические детали исполнения программы. Трассировка генерирует большой объём информации и применяется преимущественно для глубокого анализа поведения системы, а также отладки и выявления ошибок на уровне реализации.

Логирование – это процесс записи информации о высокоуровневых событиях, происходящих в системе, в специальные файлы, называемые логами. Такие события включают, например, вход пользователя в систему, критические ошибки или важные изменения в состоянии приложения. Логирование используется в первую очередь для анализа работы программы и диагностики сбоев, а не для оценки её производительности. В отличие от трассировки, которая фиксирует большое количество низкоуровневых событий, логирование охватывает редкие, но значимые действия системы.

Обработчик – это функция, которая автоматически выполняется в момент происхождения события. Внутри обработчика может изменяться состояние модуля, накопление или анализ данных, а также передача информации другим компонентам системы. Обработчики позволяют гибко реагировать на события и реализовывать нужную логику поведения программы.

Объект-событие – это объект, хранящий в себе набор привязанных обработчиков, реагирующих на его вызов. Когда объект-событие вызывается, автоматически запускаются все связанные с ним обработчики, последовательно выполняя заданные действия. Таким образом, объект-событие служит связующим звеном между происходящими в системе изменениями и логикой их обработки.

Модуль – это логическая единица профилировщика LuNA, представляющая собой набор обработчиков событий, объединённых общим внутренним состоянием. Обработчики реагируют на определённые события во время исполнения программы, а общее состояние позволяет им совместно накапливать, анализировать и передавать данные. Для интеграции в систему модуль реализует специальный интерфейс, благодаря которому LuNA может при необходимости динамически его загрузить и зарегистрировать. Регистрация модуля означает привязку его обработчиков к соответствующим объектам-событиям.

Фрагменты вычислений – это независимые логические единицы фрагментированной программы, содержащие описание операций над данными. Каждый фрагмент вычислений включает в себя код (например, модуль или процедуру), а также список входных и выходных переменных, определяющих, с какими фрагментами данных он взаимодействует.

Фрагменты данных – это блоки информации фиксированного объёма, которые служат входными и выходными параметрами для фрагментов вычислений. Они представляют собой абстракции хранимых и передаваемых значений и обеспечивают взаимодействие между фрагментами вычислений.

Горячая точка – это участок кода, который выполняется существенно чаще остальных и оказывает значительное влияние на общее время выполнения программы. Выявление таких

точек позволяет сосредоточить усилия на их оптимизации с целью повышения производительности всего приложения.

Макрос – это конструкция препроцессора, которая позволяет заменить идентификаторы или шаблоны кода их текстовым представлением до этапа компиляции. Макросы могут использоваться для определения констант, упрощения повторяющегося кода, условной компиляции и других целей.

Блок параллельного выполнения – это атомарная структурная единица, представляющая собой некоторый участок исходного кода программы. Блоки используются для описания структуры процедуры и определения параллельной организации программы. В зависимости от выполняемых действий блок может быть представлен циклом, доступом к элементу массива (на чтение или запись), выделением памяти под массив, вызовом процедуры или последовательным участком кода, не относящимся ни к одной из перечисленных категорий.

Генератор – это компонент системы LuNA, предназначенный для создания LuNA-программы путём формирования определений фрагментов вычислений, данных и атомарных фрагментов кода на основе заданной схемы программы. Генератор преобразует последовательность блоков в соответствующие синтаксические конструкции, сохраняя их порядок и типологические особенности.

ВВЕДЕНИЕ

Современная наука и инженерия всё чаще используют численное моделирование в качестве ключевого инструмента для решения сложных задач. Моделирование позволяет получить прогнозы и результаты для процессов, экспериментальное исследование которых затруднено или невозможно из-за технических и экономических ограничений. Для проведения реалистичных и детальных вычислительных экспериментов требуются значительные ресурсы, и обеспечить их могут только высокопроизводительные вычислительные системы – параллельные и распределённые суперкомпьютеры. Таким образом, эффективное применение численного моделирования напрямую связано с возможностью задействовать возможности параллельных вычислений на полную мощность.

Разработка производительных параллельных программ остаётся нетривиальной задачей. Помимо реализации математического алгоритма, программисту необходимо разделить его на части, распараллелить вычисления, сбалансировать нагрузку между узлами и организовать обмен данными. Все эти аспекты существенно усложняют процесс разработки и требуют специальных знаний в области параллельного программирования. Ошибки в распределении нагрузки или организации коммуникаций могут привести к тому, что программа не будет масштабироваться с ростом числа процессоров, теряя преимущества высокопроизводительных систем. Поэтому вопрос обеспечения эффективности параллельных программ является чрезвычайно актуальным: даже небольшие неэффективности могут выливаться в значительные потери времени вычислений и ресурсов на современных суперкомпьютерах.

Для упрощения и частичной автоматизации создания параллельных программ в Институте вычислительной математики и математической геофизики СО РАН (ИВМиМГ СО РАН) была разработана технология фрагментированного программирования и её программная реализация – система LuNA. Данная система позволяет описывать программу в виде набора независимых фрагментов данных и вычислений, делегируя системе задачи распараллеливания. LuNA абстрагирует разработчика от низкоуровневых деталей, автоматически выполняя балансировку нагрузки, распределение фрагментов вычислений по узлам и адаптацию к архитектуре целевой платформы. При этом в системе предусмотрены механизмы, позволяющие программисту влиять на стратегию распределения ресурсов и балансировки нагрузки при необходимости. Благодаря этому программист может сосредоточиться на логике задачи, не заботясь напрямую о раскладке вычислений по потокам или узлам. Тем не менее, даже при использовании LuNA перед разработчиком остаётся важная проблема – анализ производительности созданной параллельной программы. Автоматизация распараллеливания не гарантирует оптимальной эффективности: необходимо

контролировать, насколько хорошо программа масштабируется и где возникают узкие места, чтобы при необходимости оптимизировать решение.

Для выявления причин недостаточной эффективности параллельных программ применяется профилирование – сбор информации о характеристиках выполнения программы. Профилировочные данные позволяют найти «узкие места» в коде и оценить, какие фрагменты потребляют наибольшее время или ресурсы. На основе этих сведений разработчик может целенаправленно оптимизировать наиболее проблемные участки, повышая общую производительность приложения. В контексте системы LuNA профилирование приобретает особое значение, так как программный код разбивается системой на фрагменты: эффективность всего вычислительного процесса определяется тем, как исполняются эти фрагменты и как они взаимодействуют друг с другом. Однако на данный момент в LuNA отсутствует единый универсальный инструмент профилирования, работающий на уровне абстракций самой системы (то есть в терминах фрагментов). Существующие решения носят разрозненный характер: разработаны отдельные утилиты и подпроекты, которые позволяют получать некоторые сведения о выполнении LuNA-программ, но они слабо интегрированы между собой. Например, в системе имеются макросы для простой трассировки и отдельные модули для логирования событий, однако эти средства не объединены общей инфраструктурой, мало документированы и не всегда совместимы. Такой разрозненный подход затрудняет дальнейшее развитие профилировочных средств и повторное использование их компонентов, что в итоге мешает всестороннему анализу и отладке производительности программ на LuNA.

В настоящее время ведутся исследования по созданию целостного решения для профилирования в LuNA. В частности, Абрамушкина Е. С. в своей выпускной работе [2] провела всесторонний обзор существующих средств профилирования LuNA-программ и подробно разобрала их возможности и ограничения. Этот анализ позволил сформулировать основные требования к будущей подсистеме профилирования и предложить первоначальную архитектуру, способную объединить разрозненные инструменты в единый подход. Таким образом, предпосылки для разработки профилировочной подсистемы были определены в предыдущих работах, однако само интегрированное решение до сих пор не реализовано. Необходима разработка новой подсистемы профилирования, которая бы соответствовала выдвинутым требованиям и устранила обозначенные недостатки.

Целью данной работы является разработка конфигурируемого компонента системы LuNA, который позволит собирать и анализировать сведения о выполнении фрагментированных программ. Для достижения этой цели в рамках исследования ставятся следующие задачи:

— спроектировать и реализовать событийную модель профилирования;

— спроектировать и реализовать механизм расширения через модули;

— спроектировать и реализовать средства конфигурирования.

Научная новизна работы заключается в изучении и разработке событийно-ориентированной архитектуры профилирования для системы фрагментированного программирования LuNA, а также в создании расширяемой модели взаимодействия модулей профилирования. Предложена и реализована модель конфигурирования профилировочных сценариев, обеспечивающая гибкую интеграцию новых методов анализа производительности без модификации основного кода системы. В ходе работы обоснованы основные принципы построения подсистемы профилирования для систем с подобной архитектурой, а также проведена оценка эффективности предложенного подхода на примерах реальных вычислительных задач. Практическая значимость работы состоит в том, что созданный инструмент позволяет разработчикам LuNA-программ гибко собирать и анализировать характеристики исполнения: через конфигурационный файл можно выбирать необходимые виды профилирования, а собранные метрики помогают выявлять «узкие места» и оптимизировать распределение вычислительной нагрузки.

1 Обзор средств профилирования параллельных программ

1.1 Средства профилирования параллельных MPI-программ

Прежде всего следует ознакомиться с существующими средствами профилирования параллельных MPI программ и оценить их функциональные возможности. Это позволит определить, насколько данные инструменты подходят для анализа фрагментированных программ, исполняемых в системе программирования LuNA.

1.1.1 Intel Trace Analyzer and Collector

Intel Trace Analyzer and Collector (ITAC) [3] – это инструмент для трассировки и анализа параллельных MPI-программ, разработанный Intel. Он позволяет собирать и визуализировать детальные данные о выполнении MPI-взаимодействий, помогая выявлять узкие места производительности, неэффективную коммуникацию между процессами и дисбаланс нагрузки. Сбор данных осуществляется с помощью компонента Collector, который интегрируется с MPI-библиотекой и сохраняет трассировочную информацию в файл. Затем с помощью Analyzer, изображённом на рисунке 1, можно открыть эти трассы и исследовать поведение программы во времени, а также строить временные диаграммы, граф вызовов и статистику. ITAC поддерживает масштабируемость до тысяч процессов и совместим с Intel MPI и другими реализациями MPI. Это мощное средство для профилирования высокопроизводительных приложений на кластерах и суперкомпьютерах.

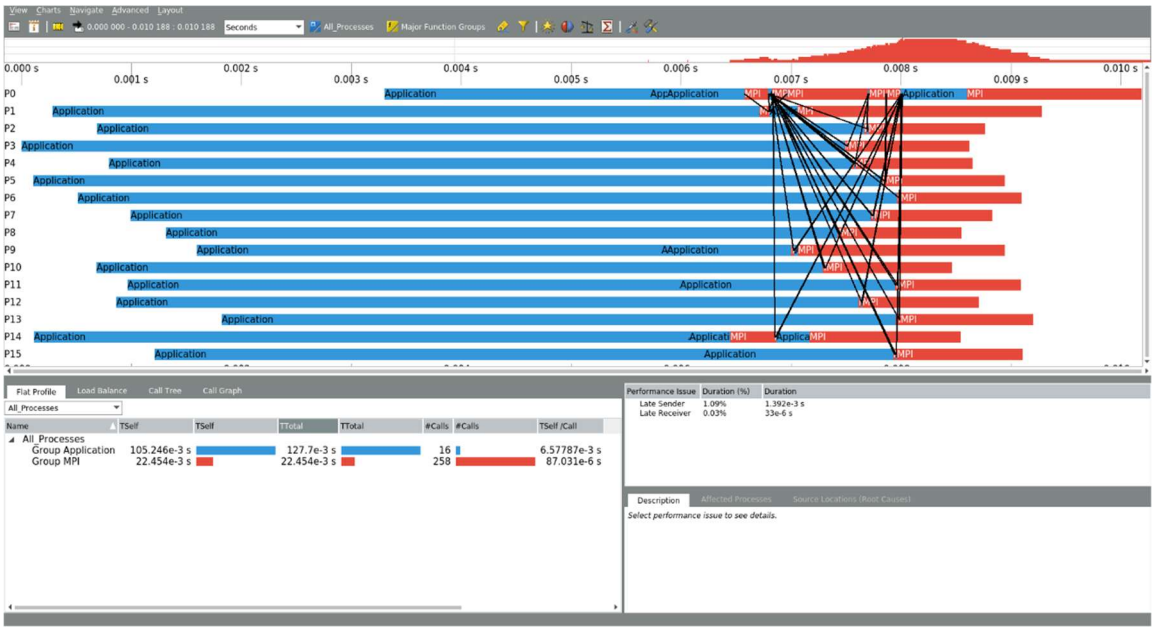

Рисунок 1 – Графический интерфейс утилиты ITAC

1.1.2 Scalasca

Scalasca [4] – это инструмент для анализа производительности параллельных программ, разработанный специально для систем с распределённой памятью и использующих

технологии MPI и OpenMP. Он автоматизирует сбор и анализ трасс, выявляя временные задержки и проблемы синхронизации между процессами. Основная особенность Scalasca – возможность выполнения пост-обработки трассировок с учетом причинно-следственных связей, что позволяет определить, какие именно вызовы и процессы привели к задержкам в исполнении. Поддерживает масштабируемый анализ на тысячах процессов и интегрируется с инструментами Score-P и Cube, изображённом на рисунке 2, для визуализации результатов.

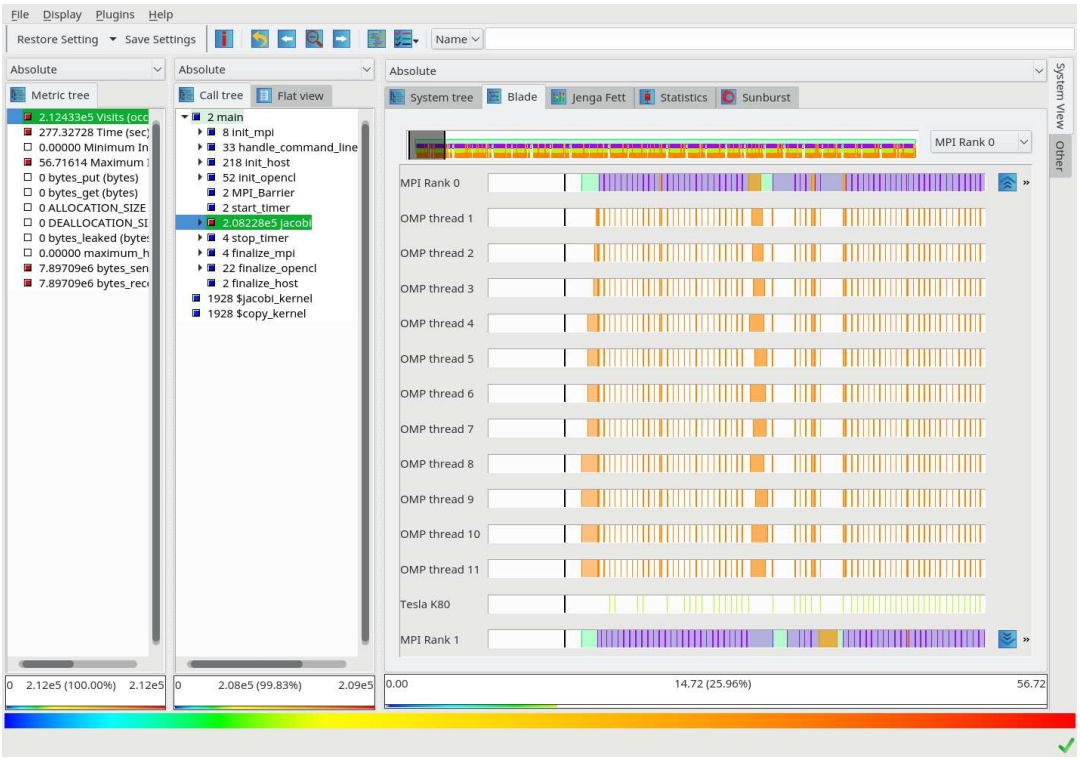

Рисунок 2 – Графический интерфейс утилиты Cube

1.1.3 Tuning and Analysis Utilities (TAU)

Tuning and Analysis Utilities (TAU) [5] – это мощный инструмент для профилирования и трассировки параллельных программ, поддерживающий MPI, OpenMP, CUDA, OpenCL и другие модели. TAU позволяет собирать детальную информацию о времени выполнения функций, использовании ресурсов и коммуникационных событиях. Он интегрируется с компиляторами и предоставляет разнообразные способы сбора данных: замеры времени, аппаратные счетчики, трассировка событий. TAU поддерживает визуализацию через инструменты ParaProf, изображённом на рисунке 3, и PerfExplorer и может работать на масштабируемых HPC-системах.

Рисунок 3 – Графический интерфейс утилиты ParaProf

1.1.4 HPC Toolkit

HPC Toolkit [6] – это мощный набор инструментов для анализа производительности параллельных программ, разработанный в Университете Райса. Он позволяет собирать и визуализировать детальную информацию о времени выполнения, распределении нагрузки, коммуникационных задержках и использовании ресурсов на уровне как CPU, так и GPU. HPC Toolkit поддерживает широкий спектр парадигм параллелизма, включая MPI, OpenMP, CUDA и HIP, и обеспечивает построение иерархических профилей, а также трассировку событий с привязкой к исходному коду. Благодаря визуальному интерфейсу hpctraceviewer, изображённом на рисунке 4, пользователи могут глубоко анализировать поведение программы, выявлять узкие места, дисбаланс нагрузки, избыточные синхронизации и другие критические проблемы производительности. Инструмент отличается низкими накладными расходами и хорошо масштабируется на больших вычислительных системах, что делает его особенно полезным при оптимизации производительности в HPC-среде.

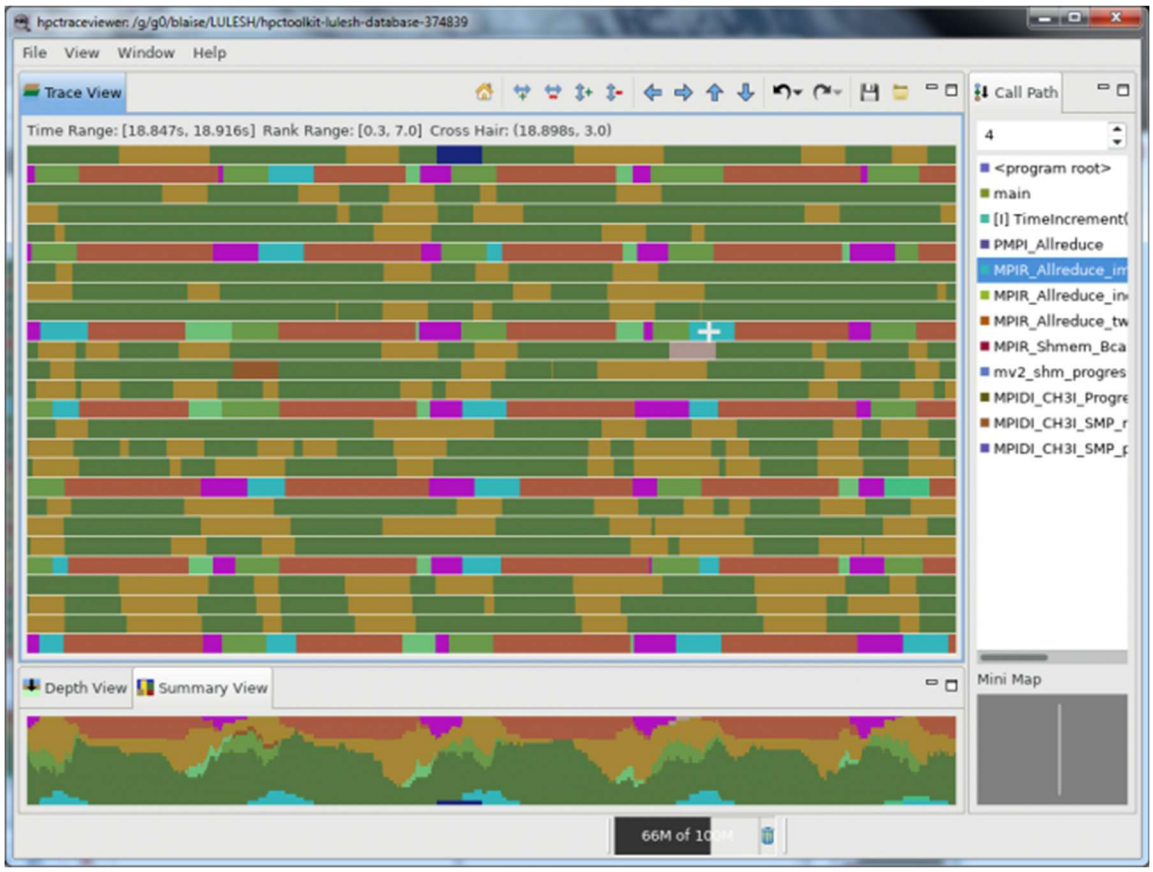

Рисунок 4 – Графический интерфейс утилиты hpctraceviewer

1.2 Средства профилирования LuNA-программ

В работе Абрамушкиной Е. С. [2] был проведён всесторонний анализ существующих средств профилирования программ, исполняемых в системе LuNA. Обзор охватывает множество подпроектов, использующих профиль фрагментированной программы, и позволяет чётко выделить ключевые достоинства и недостатки каждого из них. Особое внимание уделено набору макросов для трассировки, а также инструментам WorkflowTracer, LuNA Evlog и LuNA Evlog Analyzer.

1.2.1 Набор макросов для трассировки

Как отмечается в работе [2], система LuNA предоставляет простой, но эффективный набор макросов для трассировки исполнения фрагментированных программ:

— *TRACE* – выводит текстовое сообщение в стандартный поток вывода, облегчая отладку и анализ выполнения;

— *LOG* – сохраняет сообщение в файл с добавлением системной информации: номер процесса, временная метка, имя файла и строка кода.

Этот подход позволяет разработчику получать как быстрые диагностические сообщения, так и сохранять важные события для последующего анализа.

1.2.2 WorkflowTracer

Также в работе [2] рассматривается инструмент WorkflowTracer, предназначенный для трассировки жизненного цикла фрагментов. При инициализации требует номер узла и размеры буферов для отложенной записи. Используется для отладки семантических ошибок, связанных с управлением временем жизни фрагментов.

Основные методы:

— *cf_created* и *cf_destroyed* – отслеживают создание и удаление вычислительных фрагментов;

— *cfs_remaining* – помогает обнаружить зависшие фрагменты.

Для анализа данных WorkflowTracer применяется утилита luna_trace, предоставляющая:

— агрегацию трасс с разных узлов;

— отчёты о зависших вычислениях;

— обнаружение ошибок инициализации и повторной инициализации фрагментов данных.

1.2.3 LuNA Evlog и LuNA Evlog Analyzer

В более ранних версиях системы, как указано в [2], использовался модуль Evlog для логирования ключевых событий:

— запуск и завершение фрагмента вычислений;

— создание и удаление фрагментов данных;

— передача фрагментов по сети;

— состояние потоков исполнения.

Evlog внедряется через автоматическое инструментирование исходного кода, что позволяет встраивать вызовы логирования прямо в исполняемую программу.

Для последующего анализа логов применяется LuNA Evlog Analyzer, который предоставляет:

— данные об использовании оперативной памяти;

— статистику сетевых передач;

— информацию о параллелизме исполнения.

Характеристики могут быть получены как в виде интегральных величин, так и в динамике по времени – например, графики памяти или количества активных вычислений.

1.3 Примеры использования профилирования в подпроектах системы LuNA

1.3.1 Визуальный анализ исполнения (Evlog Profiler)

В рамках работы Голикова М. О. [7] была разработана система визуализации профиля фрагментированных программ, названная Evlog Profiler. Этот инструмент собирает детальную информацию о событиях при выполнении LuNA-программы (с помощью специального логирования) и предоставляет пользователю графическое отображение хода выполнения и краткую статистику программы. Для реализации такого профилирования исходный код LuNA-программы на этапе компиляции автоматически дополнялся вызовами методов класса Evlog, а структуры данных, представляющие фрагменты вычислений (CF) и данных (DF), были расширены, чтобы сохранять семантическую информацию о каждом фрагменте – например, имена фрагмента и выполняемой в нём пользовательской функции. Это позволило Evlog Profiler собирать профиль исполнения на уровне фрагментов и использовать его для визуального анализа производительности алгоритмов, включая оценку эффективности их фрагментации/

1.3.2 Распределение ресурсов на основе трасс

В проекте Лямина А. С. [8] профилирование использовано для автоматического улучшения балансировки нагрузки между узлами кластера. После выполнения фрагментированной программы собиралась трасса – хронологическая запись событий выполнения, – которая затем обрабатывалась специальной программой. Этот анализ определял нагрузку на вычислительные узлы и при обнаружении дисбаланса генерировал рекомендации по перераспределению фрагментов вычислений перед следующим запуском программы. По сути, трассировка исполнения позволяет алгоритму выявить узкие места и создать файл с указаниями для будущих запусков, оптимизируя размещение фрагментов по узлам. Для работы такого балансировщика требуются подробные данные о каждом событии исполнения (создание, начало и конец выполнения фрагмента и др.) с временными метками, то есть полноценный профиль выполнения программы.

1.3.3 Динамическая балансировка при воспроизведении трас

В работе Саяпина М. П. [9] разработаны алгоритмы динамической балансировки нагрузки с использованием воспроизведения трасс. Здесь профиль исполнения (трасса, собранная при запуске фрагментированной программы) применялся для имитации выполнения программы с разными стратегиями распределения фрагментов. Иными словами, профилирование позволяло повторно «проигрывать» выполнение программы на основе записанных событий и анализировать, как изменения в алгоритме балансировки влияют на

распределение нагрузки. Такой подход требовал специального инструментария для считывания трасс и применения к ним новых правил балансировки, что было реализовано как отдельный инструмент в рамках подсистемы воспроизведения трасс LuNA.

1.3.4 Централизованная динамическая балансировка нагрузки

Ещё один пример использования профилирования – модуль централизованной динамической балансировки, предложенный Мустафиным Д. Э. [10]. В этой подсистеме один выделенный узел во время выполнения программы собирает информацию о состоянии вычислительной системы (нагрузке на узлах, объёме очередей задач и т.п.) и на основе актуальных данных перераспределяет фрагменты вычислений между узлами в режиме реального времени. Для работы такого централизованного балансировщика необходимо получать оперативные сведения о происходящих событиях в системе, то есть выполнять динамическое профилирование прямо в ходе исполнения программы. В отсутствие готового решения, обеспечивающего сбор и агрегирование статистики в режиме реального времени, этот модуль также вынужден был включать собственные средства мониторинга фрагментов в исполнительной системе.

1.4 Выводы по результатам обзора

Существующие инструменты профилирования параллельных программ, рассмотренные в обзоре (например, средства для MPI-программ и профилировщики низкого уровня), обладают широкими возможностями, но ни один из них не способен полностью решить задачу профилирования в системе LuNA. Главная проблема заключается в несоответствии модели фрагментированного программирования тем уровням, на которых работают типовые профилировщики. Стандартные утилиты (такие как системные профилировщики и отладчики производительности для C/C++ и MPI) предоставляют информацию в терминах процессов, потоков и функций на языках программирования низкого уровня. Однако фрагментированная LuNA-программа оперирует более высокоуровневыми сущностями – фрагментами данных и вычислений, абстрагированными от процессов и потоков исполнения. Соответственно, профиль, полученный штатными средствами (например, время работы потоков, загрузка CPU или трассы MPI-вызовов), трудно соотнести с конкретными фрагментами LuNA-программы. Установление взаимосвязи между множеством системных потоков на каждом узле и множеством фрагментов вычислений является нетривиальной задачей, поскольку выполнение фрагментов распределяется динамически планировщиком LuNA и не привязано жестко к определённым потокам или узлам.

Кроме того, каждое из существующих средств профилирования ориентировано на некоторый класс задач и не предусматривало тех специфических метрик, которые требовались в вышеописанных проектах. Например, ни один из внешних профилировщиков не предоставлял «из коробки» информацию о количестве активных фрагментов вычислений или об объёме передаваемых между узлами фрагментов данных – эти показатели пришлось собирать самостоятельно внутри системы LuNA. Средства для трассировки MPI-взаимодействий тоже не решали проблему, поскольку во фрагментированной модели обмен данными и заданиями происходит неявно (через внутренние механизмы LuNA, а не прямые MPI-вызовы пользователя). Таким образом, участники каждого подпроекта пришли к выводу, что адаптация существующих профилировщиков к их задачам либо невозможна, либо сопоставима по сложности с созданием собственного инструмента. В результате в каждом случае реализация выполнялась с нуля, с учётом нужд конкретного проекта. Следует отметить, что даже внутри самой системы LuNA ранее созданные профилировочные модули не могли быть напрямую переиспользованы другими разработчиками. Они создавались разрозненно и без унифицированного подхода: реализованные инструменты дублировали функциональность друг друга, форматы собранных профилей оказались несовместимы, а документация практически отсутствовала. Эти обстоятельства затрудняют понимание и поддержку кода и не позволяют быстро встроить прежние наработки в новые проекты. В совокупности, как ограниченная применимость сторонних инструментов, так и фрагментарность собственных средств профилирования в LuNA обусловили необходимость каждый раз разрабатывать специализированное решение под задачи конкретного подпроекта. Такое положение дел подтверждает актуальность создания единой подсистемы профилирования, которая могла бы сохранить и расширить существующую функциональность, обеспечив совместимость и облегчая повторное использование в будущем.

На фоне выявленных недостатков, разрабатываемая профилировочная подсистема должна удовлетворять ряду важных требований, выполнение которых напрямую обеспечивает достижение поставленной цели. Во-первых, она должна предоставлять унифицированный подход к сбору данных о работе приложения на уровне фрагментов LuNA. Это означает, что независимо от вида анализируемых характеристик (время, память, коммуникации и т.п.) сбор информации происходит через общий механизм событий, понятных в терминах модели LuNA. Такой подход позволит интегрировать разные инструменты анализа в единую систему и сделать результаты профилирования сопоставимыми между собой. Во-вторых, подсистема должна быть легко расширяема: архитектура должна поддерживать подключение новых модулей профилирования без переработки существующего кода. Это обеспечит долгосрочное развитие инструмента – новые методы анализа производительности смогут добавляться по

мере необходимости, не нарушая работу уже реализованных функций. В-третьих, необходима гибкая настраиваемость профилирования. Пользователь (или разработчик) должен иметь возможность выбирать интересующие его аспекты производительности – например, активировать только сбор временных характеристик или дополнительно включить мониторинг памяти и сети. Благодаря такой гибкой конфигурации удастся подстроить профилирование под конкретную задачу и избежать избыточного сбора данных, что минимизирует накладные расходы. Соблюдение этих требований позволит создать мощный инструмент анализа производительности для LuNA-программ. В результате единая конфигурируемая подсистема профилирования упростит разработку новых профилировочных средств и повысит качество анализа параллельных программ в системе LuNA, способствуя более эффективному решению вычислительных задач на современных высокопроизводительных платформах. Таким образом, можно утверждать, что создание универсальной конфигурируемой подсистемы профилирования является логичным и необходимым этапом в развитии системы LuNA. Именно эта задача становится целью дальнейшей разработки.

2 Проектирование конфигурируемой подсистемы профилирования

2.1 Проектное решение

При разработке подсистемы профилирования для системы LuNA был выбран событийно-ориентированный подход с модульной архитектурой. Такой подход соответствует поставленным требованиям к единообразию, расширяемости и конфигурируемости инструмента. В центре архитектуры находится модель событий, которая обеспечивает отделение сбора профилировочных данных от основной логики выполнения фрагментированной программы. Основная идея состоит в том, что при наступлении определённых значимых событий во время выполнения программы (таких как начало или окончание выполнения фрагмента вычислений, передача фрагмента данных между узлами и др.), подсистема вызывает соответствующие объекты-события профилирования. При активации объекта-события автоматически вызываются все связанные с ним обработчики, что позволяет гибко реагировать на происходящее в системе.

Архитектура подсистемы основана на шаблоне наблюдатель [11] (от англ. observer): профилировочные модули выступают в роли наблюдателей, подписывающихся на интересующие их события исполнения программы. Ядро исполнительной системы LuNA при этом вызывает объекты-события, но не знает деталей того, как они обрабатываются – эту функцию выполняют внешние модули. Данный проектный выбор обоснован стремлением к слабой связанности компонентов: профилировочные возможности добавляются без изменения кода ядра системы, посредством подключения новых модулей. Это обеспечивает требуемую расширяемость – добавление нового вида профилирования сводится к созданию нового модуля, вместо модификации существующего кода.

Важным аспектом архитектуры является конфигурируемость. Было решено вынести управление подсистемой профилирования во внешнюю конфигурацию, чтобы пользователь мог задавать, какие именно события отслеживать и какие модули задействовать, не изменяя код приложения. Конфигурационный подход позволяет достичь унифицированности профилирования: все модули работают через общий механизм событий и на одном уровне абстракции (в терминах фрагментов LuNA), но при этом могут гибко включаться или отключаться. Например, для быстрой оценки времени выполнения фрагментов можно задействовать только модуль замера времени, отключив сбор иных метрик, а для детального анализа добавить модули отслеживания использования памяти или коммуникаций. Поддержка конфигурации облегчает адаптацию профилирования под различные задачи и минимизирует накладные расходы, активируя лишь необходимые события и обработчики.

Таким образом, спроектированная архитектура включает следующие ключевые элементы:

— система событий в исполнительной среде LuNA, вызывающая объекты-события профилирования при наступлении предопределённых ситуаций;
— механизм динамической регистрации обработчиков – привязки функций-обработчиков к объектам-событиям;
— набор подключаемых модулей профилирования, реализующих интерфейс обработчиков событий;
— средство конфигурации, определяющее, какие события активны и какие модули следует загрузить.

Такое решение обеспечивает модульность – профилировочные функции разделены по независимым модулям – и объединяет их общим интерфейсом, что соответствует цели создания единого и расширяемого инструмента профилирования для LuNA.

2.2 Структура подсистемы профилирования

Спроектированная подсистема профилирования, архитектура которой показана на рисунке 5, имеет определённую структуру взаимодействующих компонентов:

— модули;
— обработчики;
— объекты-события;
— диспетчер модулей;
— диспетчер событий.

На уровне исполнительной системы LuNA внедряется компонент диспетчера событий, ответственный за регистрацию модулей и уведомление привязанных к объектам-событиям обработчиков. Исполнительная система в ключевых точках своего алгоритма выполняет вызовы диспетчера событий, вызывая объект-событие определенного типа. Например, при запуске нового фрагмента вычислений вызывается объект-событие типа «Начало выполнения фрагмента», при завершении вычисления – объект-событие «Окончание выполнения фрагмента», а при пересылке фрагмента данных между узлами – объект-событие «Отправка/получение фрагмента данных». Каждое событие сопровождается контекстной информацией (данными события), необходимой обработчикам: например, указателем на идентификатор фрагмента или размер отправляемых данных, временной меткой и т.п.

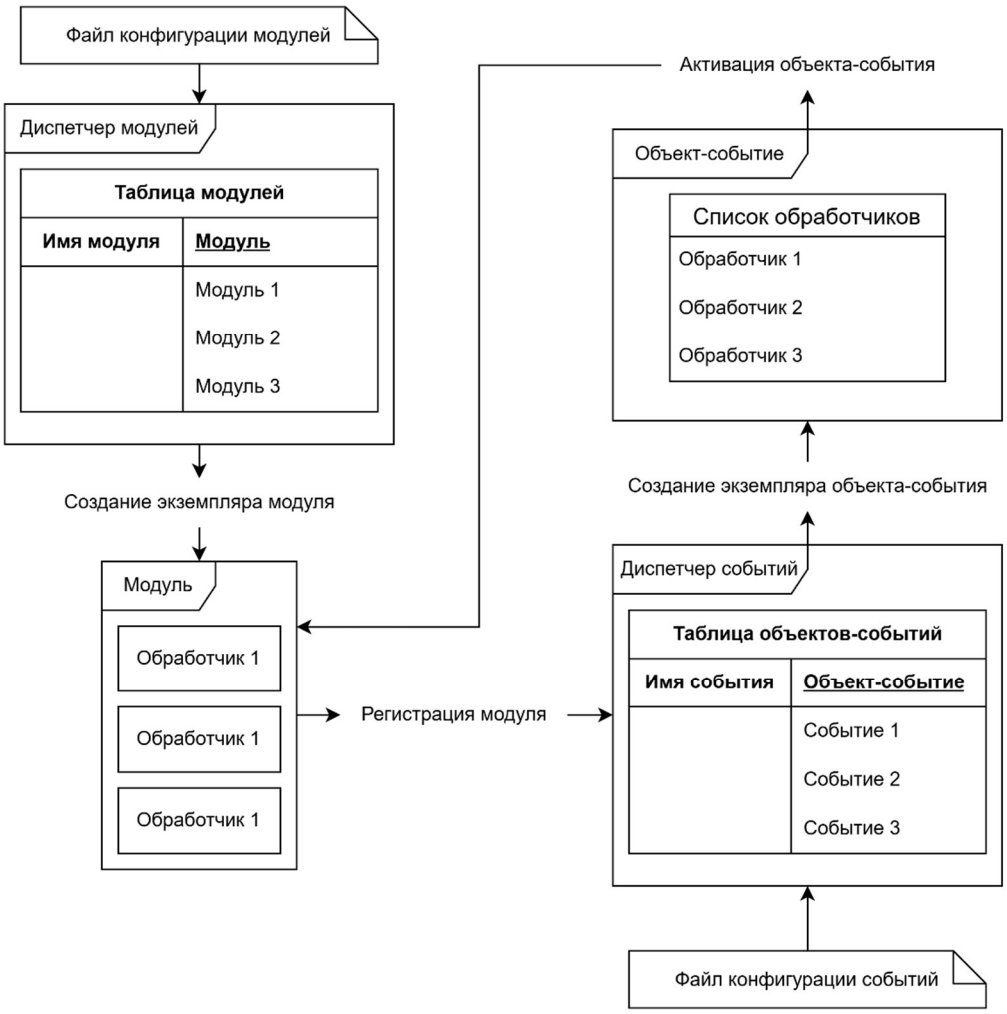

Рисунок 5 – Архитектура подсистемы профилирования

Модули профилирования реализованы в виде внешних динамически загружаемых библиотек, отвечающих специальному интерфейсу. Каждый модуль содержит один или несколько обработчиков, привязанных к определённым типам событий. Структурно модуль может хранить внутреннее состояние для накопления статистики (например, суммарное время работы фрагментов, счетчики событий, списки результатов) и предоставлять методы для регистрации своих обработчиков. Для того чтобы связать модуль с системой, используется процедура инициализации и регистрации модуля. При старте приложения подсистема загружает указанные в конфигурации модули и регистрирует их: для этого каждый модуль должен предоставить функцию, в которой вызывает API диспетчера событий для привязки своих обработчиков к нужным событиям. Регистрация модуля представляет собой процесс перебора всех поддерживаемых модулем событий и вызова для каждого метода подписки, предоставляемого диспетчером событий. В результате во внутренней структуре каждого объекта-события появляется ссылка на функцию-обработчик из данного модуля. После регистрации всех модулей диспетчер событий знает, какие обработчики необходимо вызывать при наступлении каждого типа события.

Внешний конфигурационный файл описывает состав и параметры профилирования. В конфигурации явно перечисляются профилировочные модули, которые нужно загрузить, и (при необходимости) события, подлежащие активации или деактивации. Группировка событий также поддерживается для удобства: можно задать логическую группу, например «Основное профилирование», которая включает в себя сразу набор событий (начало/конец фрагмента) и модуль для замера времени, либо «Расширенное профилирование», добавляющее события трассировки. Конфигурация, таким образом, служит связующим звеном между пользователем и подсистемой: она определяет сценарий профилирования перед запуском программы.

Процесс инициализации подсистемы профилирования проходит в несколько этапов. Сначала загружается и разбирается файл конфигурации: устанавливается, какие события будут активны и какие модули необходимо подключить. На основе этой информации диспетчер событий создаёт внутренние структуры для управления указанными событиями. Неактивные события будут пропущены, чтобы вызовы к ним не генерировались во время выполнения (это позволяет снизить накладные расходы, устраняя лишние проверки во время выполнения). Далее подсистема динамически загружает профилировочные модули и вызывает их инициализационные функции. Каждый модуль при инициализации регистрирует свои обработчики через интерфейс диспетчера событий. После успешной регистрации формируется окончательная таблица объектов-событий и связанных с ними обработчиков. В этот момент настройка подсистемы завершена, и управление возвращается основной программе. Таким образом, к началу исполнения пользовательского кода все необходимые профилировочные события и модули готовы к работе, и структура подсистемы представляет собой совокупность активных объектов-событий (в виде объектов с коллекцией обработчиков) и загруженных модулей (со своим состоянием), объединённых через диспетчер событий.

2.3 Алгоритмы, реализующие профилирование

2.3.1 Генерация и вызов объектов-событий во время исполнения

Когда в исполнительной системе LuNA происходит событие, требующее профилирования, основной код программы инициирует профилировочное объект-событие через диспетчер. Вызов включает передачу контекстных данных (например, указатель на структуру с информацией о фрагменте) диспетчеру. Диспетчер, получив запрос на возникновение события, находит в своей таблице соответствующий объект-событие и последовательно вызывает все привязанные к нему обработчики. Обработчики вызываются, как правило, в определённом порядке (например, в порядке регистрации модулей или по приоритетам, если они задаются). Внутри каждого обработчика выполняется логика сбора

данных: один модуль может зафиксировать текущее время начала выполнения фрагмента, другой – увеличить счётчик активных фрагментов, третий – сохранить размер переданных данных и т.д. В многопоточной среде вызовы объектов-событий могут происходить параллельно на разных потоках (например, несколько фрагментов стартуют одновременно на разных узлах). Поэтому алгоритм вызова объектов-событий реализован с учётом потокобезопасности: структуры данных диспетчера защищены от одновременной модификации, а сами обработчики либо не взаимодействуют друг с другом напрямую, либо используют механизмы синхронизации при доступе к разделяемым ресурсам. В результате вызов объекта-события накладывает минимальную дополнительную нагрузку – в основном это перебор списка обработчиков и вызов функций модулей.

2.3.2 Подключение обработчиков (регистрация модулей)

Этот алгоритм происходит во время инициализации, но является ключевой частью обеспечения гибкости системы. Каждый модуль, реализующий интерфейс профилирования, содержит код регистрации своих обработчиков. Обобщённо алгоритм выглядит так: для каждого поддерживаемого модулем типа события вызывается функция диспетчера. Диспетчер в ответ добавляет указатель на функцию-обработчик в список обработчиков соответствующего объекта-события. Если объект-событие до этого не был создан, то диспетчер может создать новый объект-событие в своей таблице (или, если событие отключено настройками, игнорировать попытку подписки). Таким образом, алгоритм подключения обработчиков строит отображение «объект-событие → список обработчиков». Стоит отметить, что модуль может подписываться на несколько разных событий, а один и тот же обработчик может быть привязан к разным событиям, если логика модуля это предусматривает. После завершения регистрации всех модулей диспетчер событий содержит полную информацию о том, какие действия предпринимать при каждом событии.

2.3.3 Взаимодействие модулей и обработка результатов

В ходе выполнения программы различные модули профилирования работают параллельно и независимо друг от друга, но их совместное использование предоставляет целостную картину исполнения. Например, модуль A может подсчитывать число выполненных фрагментов и хранить результаты в своей внутренней структуре, в то время как модуль B замеряет время начала и конца каждого фрагмента, а модуль C отслеживает объём переданных данных. Через общий механизм событий их деятельность синхронизирована со стадиями выполнения программы: все получают уведомление о начале того или иного фрагмента почти одновременно (разница лишь в порядке вызова обработчиков), затем о его завершении и т.д. Хотя прямого обмена данными между модулями не происходит в общем

случае (каждый модуль сохраняет свои результаты отдельно), косвенное взаимодействие осуществляется благодаря единому контексту. Например, если несколько модулей реагируют на одно событие, они получают один и тот же идентификатор фрагмента и могут сопоставлять свои метрики по этому идентификатору при последующем анализе. Кроме того, возможна организация дополнительного обмена: один модуль при возникновении определённых условий может вызывать свой собственный объект-событие (например, событие высокого уровня "Горячая точка обнаружена"), на которое подпишется другой модуль (например, модуль логирования, фиксирующий такие случаи). Данный механизм расширяет возможности системы, позволяя модулям опосредованно координировать работу через события без жёсткой связности.

Сгенерированные профилировочные данные сохраняются модулями либо непосредственно во время обработки событий, либо в конце выполнения программы. Алгоритм завершения работы профилировочного модуля обычно предусматривает вывод или передачу накопленной информации (например, запись в файл, отправка в визуализатор или консоль). Диспетчер событий может по окончании программы сгенерировать финальное событие “Завершение программы”, на которое модули также могут быть подписаны – это удобный момент для финализации: вычисления суммарных статистик, освобождения ресурсов и т.д. После этого подсистема профилирования выгружает модули (при необходимости) и освобождает свои структуры. Все собранные данные готовы для дальнейшего анализа пользователем с помощью предусмотренных инструментов (графического интерфейса, анализаторов логов и пр., что выходит за рамки данной главы). Таким образом, совокупность алгоритмов подсистемы профилирования обеспечивает полный цикл работы: от динамического подключения необходимых обработчиков в начале выполнения, до активации объектов-событий в процессе работы и получения модулем необходимых данных, и завершая сохранением результатов. Эти алгоритмы реализуют требуемое профилирование с минимальным вмешательством в основную программу и с высокой гибкостью в части подключаемых метрик.

2.4 Принципы расширяемости и модульности подсистемы

Разработанная подсистема профилирования изначально спроектирована как модульная и расширяемая. Принцип модульности означает, что каждая функциональность профилирования изолирована в отдельном модуле, имеющем чётко определённый интерфейс взаимодействия с остальной системой. Это позволяет независимо развивать и отлаживать разные аспекты профилирования. Например, модуль, отвечающий за измерение временных характеристик, не зависит от модуля, анализирующего память или коммуникации, и наоборот.

Добавление или удаление модуля не влечёт за собой изменений в коде других модулей или в коде ядра LuNA – за счёт этого достигается свойство слабой связности. Взаимодействие происходит только через добавленные объекты-события, что служит своего рода контрактом между ядром и модулями.

Принцип расширяемости проявляется в возможности безболезненно наращивать функциональность подсистемы. Если возникает необходимость профилировать новый аспект исполнения (например, появится метрика, изначально не предусмотренная: кэш-промахи, загрузка GPU или иное), архитектура позволяет это сделать двумя способами. Во-первых, можно определить новый тип события в системе (расширив перечень стандартных событий) и внести минимальные изменения в исполнительную систему, активируя этот объект-событие в нужных точках. Поскольку подсистема изначально проектировалась с учётом унифицированной системы событий, добавление нового события не нарушит работу существующих модулей – они просто не будут на него реагировать, если не подписаны. Зато новые модули смогут использовать этот событие для сбора дополнительных данных. Во-вторых, расширение возможно на уровне модулей без изменения ядра: новые модули могут подписываться на существующие события, предлагая новые способы анализа тех же данных. Таким образом, система следует принципу открытости-закрытости (от англ. Open-Closed Principle) в контексте архитектуры: она открыта для расширения (через добавление новых модулей и событий), но закрыта для модификации базовых механизмов (существующий код диспетчера и базовых событий не требует изменений при расширении функциональности).

Модульность решения также упрощает поддержку и сопровождение кода. Каждый модуль имеет ограниченную зону ответственности и может разрабатываться своим автором или командой, соблюдая общий интерфейс. При обнаружении ошибки или необходимости улучшения достаточно внести правки только в соответствующий модуль. Кроме того, модульный подход облегчает повторное использование компонентов: профилировочные модули можно применять в различных проектах или обмениваться ими в сообществе пользователей LuNA, а также отключать ненужные в конкретном запуске компоненты для снижения нагрузки.

Следует отметить, что для обеспечения модульности были введены интерфейсы и соглашения. Интерфейс модуля определяет набор функций, которые должны быть реализованы. Интерфейс объекта-события определяет, какие данные передаются обработчикам и как происходит подписка. Благодаря этому добавление нового модуля не требует знания внутренней реализации других частей системы – достаточно следовать установленному шаблону. Такая унификация взаимодействия повышает совместимость

модулей: все они могут совместно работать в рамках одной программы, не конфликтуя, поскольку разделяют единый механизм подключения.

Наконец, модульность и расширяемость поддерживаются группировкой и конфигурированием, описанными ранее. Пользователь может комбинировать наборы событий, составляя профиль профилирования, подходящий под его задачу. Появление новых модулей или событий легко отражается в конфигурации, не требуя изменений в уже отлаженных частях системы. Эти принципы, заложенные на этапе проектирования, обеспечивают долговременную пригодность подсистемы: она может эволюционировать вместе с системой LuNA и адаптироваться к новым требованиям без кардинальной переработки архитектуры.

2.5 Соответствие требованиям

Предложенная архитектура конфигурируемой подсистемы профилирования удовлетворяет ключевым требованиям, сформулированным во Введении и обоснованным в главе 1.

Унифицированность (единый подход) профилирования достигается за счёт использования общей модели событий, непосредственно отражающей абстракции системы LuNA. Все профилировочные данные собираются и обрабатываются в терминах фрагментов вычислений и фрагментов данных, что обеспечивает согласованность анализа производительности с логикой самой программы. В отличие от ранее существовавших разрозненных средств, новая подсистема предоставляет единообразный интерфейс: независимо от того, измеряется ли время выполнения, объем памяти или коммуникационные задержки, всё представлено через единый механизм событий и модулей. Это позволяет комбинировать результаты разных модулей и сопоставлять их между собой без дополнительных преобразований, а также облегчает интерпретацию результатов пользователем – профилировочные отчёты оперируют понятными категориями (например, фрагмент №X выполнялся T секунд, передал Y байт данных и т.д. в рамках единого отчёта).

Расширяемость решения заложена на уровне архитектуры модулей. Как было сказано, добавление новых видов анализа не требует переписывания системы – достаточно реализовать новый модуль или ввести новое событие. Требование расширяемости выполняется и в отношении развития самой системы LuNA: по мере усложнения моделей вычислений или добавления новых возможностей (например, поддержки новых типов фрагментов или ресурсов) профилировочная подсистема может быть соответствующим образом расширена. Благодаря модульности и строгому разделению обязанностей, риск нарушить работоспособность существующих функций при расширении минимален. Кроме того,

наличие набора стандартных событий и модулей (разработанных на этапе создания подсистемы) служит основой, которую можно дополнять.

Конфигурируемость подсистемы непосредственно вытекает из включения механизма внешней настройки. Пользовательские требования к профилированию могут значительно различаться: от минимального влияния на выполнение (сбор лишь самых общих метрик) до максимально подробной трассировки всех событий. Разработанное решение предоставляет гибкую настройку через конфигурационный файл, тем самым полностью выполняя требование конфигурируемости. Единый формат конфигурации и поддержка групповых профилей (предустановленных наборов событий/модулей) делают использование подсистемы удобным: переключение между разными режимами профилирования не требует перекомпиляции или внесения изменений в код программы, а достигается изменением параметров. Это существенно упрощает проведение экспериментов по оптимизации: можно оперативно собирать разные виды статистики, сравнивать результаты, не тратя время на модификацию исходного кода приложения.

Дополнительно, предложенное решение удовлетворяет и сопутствующим требованиям, важным для качества инструмента. За счёт отключения неиспользуемых событий и модулей достигается минимизация накладных расходов на профилирование – система генерирует только те события, которые действительно нужны, и вызывает ограниченный набор обработчиков. Это означает, что влияние профилирования на время выполнения параллельной программы сведено к разумному минимуму, что является необходимым условием корректного анализа производительности. Также, единый подход к сбору данных повышает точность и сопоставимость результатов: все метрики собираются синхронно и в одной среде, исключая разнобой методов и разницу во временных основах.

Таким образом, разработанная конфигурируемая подсистема профилирования для LuNA в полной мере отвечает заявленным требованиям. Она обеспечивает унифицированный способ получения сведений о работе фрагментированных программ, легко расширяется под новые задачи и тонко настраивается под нужды пользователя. Эти свойства достигнуты благодаря продуманной архитектуре и заложенным принципам модульности, что подтверждает правильность выбранных проектных решений.

3 Реализация базовой функциональности подсистемы профилирования

3.1 Реализация системы событий

Система профилирования программы, реализованной в контексте LuNA основана на событийно-ориентированной архитектуре и разработана на языке C++. Ключевыми элементами этой архитектуры являются объекты-события, обработчики и модули. Руководство программиста с подробным описанием создания и использования элементов системы профилирования приведено в приложении А.

Объект-событие в контексте LuNA программы представляет собой объект шаблонного класса, предназначенный для хранения списка указателей на привязанные обработчики. Обработчики событий привязываются заранее и ассоциируются с соответствующими объектами-событиями ещё на этапе инициализации программы.

Обработчик события – это функция, реализующая конкретную логику, которая должна быть выполнена при возникновении соответствующего события. Логика обработчика может включать накопление статистических данных, запись информации в файл, передачу данных по сети или иные действия, необходимые для профилирования.

Модуль является классом, реализующим специализированный интерфейс. С помощью этого интерфейса LuNA-программа привязывает обработчики событий. Реализация интерфейса внутри модуля представляет собой набор конкретных обработчиков, привязанных к событиям.

Общая схема функционирования системы событий выглядит следующим образом:

— инициализация и регистрация модулей: на старте работы программы запускается регистрация модулей, в процессе которой происходит привязка обработчиков событий, реализованных в модулях;

— возникновение события: когда в коде программы происходит событие (например, начало вычислений, выделение памяти или завершение задачи), вызывается соответствующий объект-событие;

— вызов обработчиков: объект-событие содержит контейнер указателей на обработчиков; при возникновении события происходит итерация по этому контейнеру с последовательным вызовом всех привязанных обработчиков;

— приоритет обработки: порядок вызова обработчиков регулируется заранее, на этапе регистрации модулей; перед добавлением обработчиков в контейнер, они сортируются в соответствии с заданными приоритетами. Таким образом, наиболее приоритетные обработчики вызываются первыми.

Данная архитектура обеспечивает гибкость и расширяемость системы профилирования LuNA. Опираясь на описание программы, представленное в приложении Б, пользователь получает возможность разрабатывать собственные обработчики, расширять существующие модули и добавлять новые компоненты, обеспечивая точное соответствие подсистемы профилирования специфическим задачам конкретного приложения.

3.2 Реализация механизма генерации и вызова событий

Пользователь программы может вставлять события в код двумя способами: вручную или автоматически.

3.2.1 Ручная вставка вызовов события

Для вставки события вручную пользователю достаточно использовать специальный макрос *EMIT_EVENT*. При использовании макроса необходимо указать пространство имён объекта-события, его название и параметры, которые должны передаваться обработчику события. Пример использования макроса:

EMIT_EVENT(CFEvents, onCreated, "CF created");

Данная запись означает, что объект-событие принадлежит пространству имён *CFEvents*, имеет название *onCreated*, и при его возникновении в обработчик события передаётся строка *"CF created"*.

Типы параметров, передаваемых в объект-событие, определяются первым обработчиком, привязанным к данному событию. Если последующие обработчики попытаются подключиться к этому же объекту-событию с другими типами параметров, программа выдаст ошибку времени выполнения с соответствующим сообщением. Также важно отметить, что событие должно быть предварительно включено в конфигурационном файле.

3.2.2 Автоматическая вставка вызовов события

Автоматическая вставка вызовов событий осуществляется с помощью конфигурационного файла. Добавление нового события сводится лишь к описанию правила вставки события и указанию аргументов для передачи обработчику, а не к написанию повторяющегося шаблонного кода вручную, что значительно облегчает поддержку большого числа событий и уменьшает вероятность ошибок.

Пример, указанный в листинге 1, означает, что генератор автоматически вставит вызов объекта-события перед каждым оператором *return EXIT*.

```
"onExited": {
    "enabled": true,
    "rule": "before return EXIT",
    "args": [ "__func__", "std::chrono::high_resolution_clock::now()" ]
},
```

Листинг 1 – Правило вставки события

Таким образом, генерируемый вызов объекта-события будет выглядеть следующим образом:

EMIT_EVENT(GlobalEvents, onExited, __func__, std::chrono::high_resolution_clock::now());

return EXIT;

В конфигурации можно использовать различные правила для автоматической вставки событий, например *"after BlockRetStatus"* или другие условия и позиции кода.

3.2.3 Реакция на вызов объекта-события

При вызове объекта-события система выполняет поиск соответствующего объекта-события в ассоциативном неупорядоченном контейнере, имеющем среднюю сложность поиска O(1). Если объект-событие не найден, никакого действия не происходит. Если вызванный объект-событие найден, но переданные параметры не соответствуют ожидаемым типам, программа выдаст ошибку времени выполнения с соответствующим сообщением об ошибке.

Если объект-событие найден и параметры корректны, вызывается специальная функция объекта-события, которая последовательно итерирует по всем привязанным обработчикам и выполняет их. Таким образом, реализуется логика, предусмотренная пользователем для обработки событий.

3.3 Реализация модулей профилирования

Модуль в контексте системы профилирования LuNA представляет собой пользовательский класс, наследующий специальный интерфейс *IEventModule*. Интерфейс позволяет программе LuNA динамически взаимодействовать с модулями, обеспечивая их гибкую и независимую разработку и компиляцию. Такой подход даёт возможность легко переносить модули между различными средами и командами разработки.

Обязательной частью интерфейса *IEventModule* является функция *registerEvents*. В этой функции разработчик модуля привязывает обработчики к событиям с использованием макроса *CONNECT_EVENT*. Макрос принимает три аргумента: пространство имён события, название события и обработчик в виде лямбда-функции, описывающей конкретную логику обработки события. Пример реализации функции registerEvents показан в листинге 2.

```
void registerEvents() override {
    CONNECT_EVENT(CFEvents, onCreated, [](const char* message) {
        std::cout << message << std::endl;
    });
}
```

Листинг 2 – Пример реализации функции registerEvents

В данном примере к событию *onCreated* из пространства имён *CFEvents* привязан обработчик, который выводит передаваемое сообщение в поток стандартного вывода.

В текущей реализации системы модули и их обработчики существуют в памяти с момента динамической загрузки программой LuNA до её завершения, что позволяет сохранять состояние обработчиков и модуля на протяжении всей работы программы.

4 Конфигурационное управление событиями и модулями

Конфигурация событий и модулей в системе LuNA осуществляется посредством двух отдельных JSON-файлов, которые загружаются однократно при старте программы. Такой механизм конфигурационного управления отражает широко применяемую практику, о чём свидетельствует обзор, приведённый в разделе 1.1. Он реализует принцип селективного профилирования – позволяет собирать только действительно необходимые данные, избегая избыточной нагрузки на систему и влияния на производительность. Благодаря этому подходу разработчик получает возможность гибко управлять поведением подсистемы профилирования без необходимости перекомпиляции программы при каждом изменении настроек.

4.1 Конфигурирование событий

Настройки событий описываются в файле *events_config.json*, который разделён на две основные части:

— *"eventsSettings"* – здесь указываются конкретные настройки отдельных событий;

— *"groups"* – здесь описываются настройки групп событий (подробнее рассматриваются в разделе 3.3).

В части *"eventsSettings"* необходимо указать пространство имён события, его название и соответствующие настройки. Настройки могут быть простыми или расширенными.

Пример простой настройки:

```
"CFEvents": {
   "onCreated": true
}
```

Эта запись означает, что событие *onCreated* из пространства имён *CFEvents* активно и будет обрабатываться системой. Если событие не указано или явно выставлено значение *false*, оно обрабатываться не будет.

Расширенные настройки предназначены в основном для генератора кода, автоматически вставляющего события, и описаны подробно в разделе 3.2.2.

4.2 Конфигурирование модулей

Конфигурация модулей осуществляется через файл modules_settings.json. В нём содержится глобальная группа настроек "globalSettings", позволяющая изменять глобальные параметры модулей, а также перечисляются индивидуальные настройки модулей.

Пример конфигурации модулей показан в листинге 3.

```
"globalSettings": {
    "enabled": false
},
"libdf_sizer_events_module": {
    "enabled": true,
    "priority": 1
},
"libforeign_block_timer_module": {
    "overrideEnabled": true,
    "priority": 2
}
```

Листинг 3 – Конфигурация модулей

В представленном примере глобальная настройка отключает все модули по умолчанию (*"enabled": false*). Однако модуль *libforeign_block_timer_module* использует параметр *"overrideEnabled": true*, чтобы переопределить глобальную настройку и оставаться включённым. Дополнительно можно указывать приоритеты модулей, что влияет на порядок обработки событий.

В дальнейшем можно расширять конфигурацию модулей, добавляя параметры, которые будут передаваться в сами модули, такие как уровень детализации логирования, адрес веб-интерфейса или путь к входным/выходным файлам.

4.3 Группировка событий

Как описано в разделе 4.1, файл *events_config.json* содержит разделы *"eventsSettings"* и *"groups"*. В разделе *"groups"* перечисляются все группы событий, их состояния и события, входящие в эти группы. Группы предназначены для упрощения конфигурации и удобного переключения между различными сценариями профилирования.

Например, разработчик, заинтересованный только в отслеживании потребления памяти, может отключить все группы, кроме *"Memory"*. Если же необходимо дополнительно измерить производительность и время выполнения, можно включить группу *"Performance"*. Это позволяет быстро и удобно переключать профилируемые аспекты работы программы.

Пример конфигурации группы показан в листинге 4.

```
"Memory": {
    "enabled": true,
    "events": [
        "DFEvents::onCreate",
        "DFEvents::onDestroy"
    ]
}
```

Листинг 4 – Группа событий

Данная запись обозначает, что группа *"Memory"* является активной *("enabled": true*) и содержит события *onCreate* и *onDestroy* из пространства имён *DFEvents*.

Важно отметить следующие ограничения и правила:

— имя каждой группы должно быть уникальным;

— одно событие может одновременно находиться в нескольких группах;

— событие считается активным, если оно входит хотя бы в одну активную группу и явно указано как активное в разделе *"eventsSettings"*. Если событие присутствует в нескольких группах и одна из них отключена, но при этом событие явно указано как активное, оно всё равно будет обрабатываться системой.

5 Предустановленные события и модули

В данном разделе описаны предустановленные (стандартные) события и модули, включённые в состав разработанной подсистемы профилирования. Под предустановленными событиями и модулями понимаются такие компоненты подсистемы, которые доступны пользователю «из коробки» и покрывают типовые потребности анализа производительности LuNA-программ. Эти встроенные средства профилирования позволяют сразу приступить к сбору ключевых метрик (времени выполнения, использования памяти и пр.) без разработки собственных модулей. Термин «стандартные модули» при этом указывает не на соответствие внешнему стандарту, а на принадлежность модулей к базовому набору подсистемы. Иными словами, стандартные модули – это модули, изначально входящие в типовую конфигурацию системы (по аналогии с тем, как во многих профилировщиках заранее присутствуют встроенные инструменты для анализа использования процессора и памяти).

5.1 Набор стандартных событий

Предустановленные события в системе LuNA подразделяются на две основные группы по методу их размещения в коде:

— автоматически расставляемые генератором кода события: данные события автоматически вставляются генератором в определённые места программы согласно заранее заданным правилам;

— ручное размещение событий: эти события были вручную встроены в код в местах, наиболее подходящих для их вызова; например, событие *DFEvents::onCreate*, сообщающее о создании нового фрагмента данных, вручную вставлено непосредственно в конструктор фрагмента.

Дополнительно все предустановленные события классифицируются по трём категориям:

— события жизненного цикла блока параллельного выполнения:
  — начало;
  — окончание;
  — ожидание;
  — перемещение;
  — продолжение;
  — начало пользовательской функции;
  — конец пользовательской функции;

— события фрагментов данных:
  — создание;

  - уничтожение;
- события фрагментов вычислений:
  - создание;
  - начало;
  - окончание;
  - ожидание.

Эти категории также отражены в разделении событий на пространства имён и группы.

При вызове объектов-событий жизненного цикла передаются имя функции, из которой был вызван объект-событие (предопределённый идентификатор *__func__*), и временная метка. События фрагментов вычислений включают в себя пользовательское сообщение о произошедшем событии, а события фрагментов данных, помимо пользовательского сообщения, дополнительно передают информацию о размере созданного или уничтоженного фрагмента данных.

Предустановленные события интегрированы таким образом, чтобы пользовательские события могли добавляться аналогично. То есть архитектура единообразна: не важно, встроенное событие или пользовательское – обработка одинакова.

Выбор именно таких предустановленных событий обусловлен потребностями в сборе информации о памяти и времени выполнения задач, что подтверждается обзором инструментов профилирования, представленным в разделе 1.1. Подобные события часто встречаются в существующих профилировщиках, что подтверждает важность их наличия в системе LuNA.

5.2 Набор стандартных модулей

Для анализа LuNA-программ были разработаны несколько стандартных модулей, которые охватывают основные аспекты профилирования, в частности время и память, являющиеся наиболее критичными ресурсами. Как отмечалось в главе 1, анализ времени исполнения и потребления памяти играет центральную роль в профилировании программ. Стандартные модули LuNA были специально ориентированы на решение данных задач.

Перечень стандартных модулей включает:

- *logger_module*: записывает все поступающие события в текстовый файл или на консоль, включая время события, тип и дополнительные данные;
- *cf_counter_events_module*: подсчитывает количество всех созданных, начатых, ожидающих и завершённых фрагментов вычислений;
- *df_sizer_events_module*: анализирует объём памяти, выделенный и освобождённый при работе с фрагментами данных;

— *foreign_block_timer_module*: измеряет время выполнения пользовательских функций по потокам, помогая быстро обнаружить горячие точки в программе;

— *prefetto_module*: сохраняет поступающие события в текстовый файл в специальном формате, который можно визуализировать с помощью веб-интерфейса Perfetto UI [12].

В качестве примера в работе подробно разбирается модуль *foreign_block_timer_module*, предназначенный для точного измерения пользовательских функций, запущенных на различных потоках. В основе работы модуля лежит структура данных, которая хранит ключи (*BlockKey*) с именем функции, идентификатором процесса и идентификатором потока, а также время начала и общее время выполнения функции (*ForeignEventTime*).

При возникновении события начала пользовательской функции (*onForeignStarted*) модуль сохраняет текущее время и помечает блок как выполняемый. Соответственно, событие завершения функции (*onForeignEnded*) рассчитывает общее время работы, добавляя его к накопленному времени выполнения для данного блока.

Пример регистрации обработчиков в модуле показан в листинге 5.

```
void registerEvents() override {
    CONNECT_EVENT(GlobalEvents, onForeignStarted, [this](const char* func, myTime startT) {
        startBlock(func, startT);
    });
    CONNECT_EVENT(GlobalEvents, onForeignEnded, [this](const char* func, myTime endT) {
        endBlock(func, endT);
    });
}
```

Листинг 5 – Регистрация обработчиков в модуле

Механизм работы функций *startBlock* и *endBlock* обеспечивает корректную обработку параллельных вызовов благодаря использованию мьютекса для синхронизации доступа к общей структуре данных.

При завершении работы программы (во время вызова деструктора *~ForeignBlockTimerModule*) модуль выводит отсортированную информацию по времени выполнения каждой пользовательской функции с указанием идентификаторов процесса и потока, что существенно облегчает последующий анализ.

Использование данного модуля позволяет эффективно выявлять наиболее ресурсоёмкие участки кода (горячие точки), тем самым помогая оптимизировать производительность программы.

Как отмечалось в главе 1, анализ времени исполнения и памяти – ключевые аспекты профилирования. Стандартные модули LuNA ориентированы именно на эти ресурсы,

обеспечивая покрытие основных потребностей разработчиков и позволяя оперативно выявлять и решать проблемы производительности и расхода памяти.

5.3 Формирование модулей и расширяемость

В текущей реализации системы LuNA конфигурирование модулей осуществляется через переменные-настройки, определённые внутри самих модулей, с последующей перекомпиляцией. Однако архитектура системы заранее спроектирована таким образом, что в дальнейшем не составит труда реализовать чтение параметров модулей из внешнего конфигурационного файла, описанного в разделе 4.2.

Механизм разработки и подключения модулей в LuNA предельно гибок. Помимо создания модулей «с нуля», пользователь может легко переиспользовать существующую реализацию. Например, можно унаследоваться от класса *logger_module* и изменить формат вывода: например, реализовать логирование событий в формате CSV для дальнейшего анализа табличными инструментами.

Модули также могут эффективно работать совместно, используя один из двух подходов:

— Обмен пользовательскими событиями: один модуль может внутри себя или в рамках одного из обработчиков вызвать объект-событие, на которое будет подписан другой модуль; таким образом, модули могут взаимодействовать через механизм событий, не зная о реализации друг друга;

— прямое обращение к другим модулям через хранилище модулей: модуль может получить доступ к другому модулю по имени, используя *ModulesManager::instance()->get_module("имя_модуля");* например, модуль визуализации может получить указатель на модуль логирования и использовать его для доступа к накопленным данным.

Пример такого взаимодействия – пара модулей, где один сохраняет данные в файл в низкоуровневом формате, а второй затем читает этот файл и преобразует данные для визуализации. При этом для корректной работы приоритет второго модуля может быть выставлен ниже, чтобы он начал работу только после завершения работы первого.

Важно отметить, что добавление большого количества модулей не оказывает значительного влияния на производительность программы, так как регистрация модулей происходит однократно при запуске. Однако эффективность работы системы в дальнейшем целиком зависит от качества реализации самих модулей. Именно разработчик определяет, насколько сильно конкретный модуль будет влиять на производительность программы:

использование тяжёлых операций, чрезмерное логирование или блокировки могут замедлить выполнение приложения.

Предоставление набора стандартных модулей "из коробки" значительно упрощает начальный этап работы с системой LuNA. Пользователь сразу получает возможность профилировать основные характеристики приложения – такие как объём потребляемой памяти, количество операций и временные затраты – без необходимости разрабатывать собственные инструменты.

Как отмечалось в главе 1, многие современные профилировщики включают стандартные модули анализа CPU и памяти. Подобно таким инструментам, разработанная система предоставляет готовые модули для анализа жизненного цикла задач, использования памяти и поиска горячих точек, что делает её конкурентоспособной и адаптированной под практические потребности разработчиков.

5.4 Критерии полноты набора событий

Завершая описание предустановленных событий и модулей, важно обосновать, почему выбранный набор считается достаточным для эффективного профилирования программ в системе LuNA.

5.4.1 Критерии полноты набора

При формировании набора событий и модулей были использованы следующие основные критерии:

- покрытие ключевых ресурсов: события охватывают такие критически важные характеристики, как:
  - время выполнения (через события начала и окончания задач);
  - использование памяти (через события выделения и освобождения);
  - вычислительная активность и состояние фрагментов (через события начала, ожидания и завершения вычислений);
  - потоковая и процессная активность (учёт PID и TID в модулях);
- отражение структуры LuNA-программ: события точно соответствуют архитектурной модели LuNA, в которой основными единицами являются фрагменты данных и фрагменты вычислений; также предусмотрены события, отслеживающие жизненный цикл блоков параллельного выполнения;
- реализация задач анализа: события и модули позволяют выявлять наиболее распространённые проблемы в параллельных приложениях:
  - медленные участки кода;

— утечки или неэффективное использование памяти;

— дисбаланс вычислительной нагрузки по потокам или фрагментам;

— чрезмерное ожидание синхронизаций.

5.4.2 Сравнение с существующими решениями

Как показано в главе 1, аналогичные подходы реализованы и в других профилировщиках. Например, в системе Scalasca или Intel ITAC особое внимание уделяется событиям начала и окончания вычислений, а также анализу использования памяти. Такие события позволяют проводить фундаментальный анализ поведения параллельной программы и были успешно внедрены в разрабатываемую систему профилирования.

В рамках рассматриваемой реализации системы LuNA не охватываются такие аспекты, как профилирование потребления энергии и использование аппаратных ускорителей. Однако это обусловлено ограниченностью объёма выпускной квалификационной работы и сосредоточенностью на задачах многопоточной и распределённой обработки на CPU. Следует отметить, что система LuNA в целом поддерживает работу с ускорителями, и вопросы оценки энергопотребления представляют собой перспективное направление для дальнейших исследований.

6 Пример практического применения подсистемы профилирования

6.1 Описание задачи профилирования

В качестве примера практического применения разработанной подсистемы профилирования рассмотрим задачу перемножения матриц, реализованную на системе программирования LuNA. В данном эксперименте производится перемножение двух матриц, каждая из которых разбита на 40x40 фрагментов размером по 100x100 элементов. Такой подход обеспечивает декомпозицию задачи на множество параллельно исполняемых вычислительных задач, что хорошо соответствует архитектуре LuNA.

Целью профилирования в данном случае является получение информации о:

— временных затратах на выполнение пользовательских функций, отвечающих за отдельные вычислительные задачи;

— динамике использования оперативной памяти в процессе выполнения вычислений.

Для реализации профилирования были активированы два типа событий:

— *onForeignStarted* и *onForeignEnded* — эти объекты-события генерируются в начале и в конце исполнения каждой пользовательской функции соответственно; благодаря этому становится возможным замерять продолжительность выполнения каждой задачи, что позволяет анализировать балансировку нагрузки и выявлять потенциальные узкие места в вычислениях;

— *onCreateSize* и *onDestroySize* — эти объекты-события генерируются при выделении и уничтожении памяти, ассоциированной с фрагментами данных; это позволяет отслеживать динамику использования памяти во времени и выявлять пиковые значения, которые могут свидетельствовать, например, о неэффективном управлении памятью или потенциальных утечках.

В процессе выполнения эксперимента все соответствующие события фиксируются профилировочными модулями. В деструкторах данных модулей производится вывод всех собранных данных в терминал.

Таким образом, выбранный пример полностью демонстрирует возможности конфигурируемой подсистемы профилирования для решения типичных задач анализа производительности и использования ресурсов в параллельных приложениях на LuNA.

6.2 Настройка профилирования через конфигурацию

Для реализации рассматриваемого сценария профилирования был подготовлен конфигурационный файл, определяющий, какие группы событий и модули профилирования необходимо активировать.

6.2.1 Конфигурация событий

В конфигурационном файле событий были включены только те группы, которые необходимы для решения поставленных задач: отслеживание начала и конца пользовательских функций, а также мониторинг использования памяти. При этом группы, связанные с отслеживанием событий фрагментов вычислений (*CFEvents*) и событий жизненного цикла (*Lifecycle*), были отключены как нерелевантные для текущего эксперимента.

Фрагмент соответствующего раздела конфигурации приведён в листинге 6.

```
"groups": {
    "CFEvents": {
        "enabled": false,
        "events": [
            "CFEvents::onStarted", "CFEvents::onFinished"
        ]
    },
    "Memory": {
        "enabled": true,
        "events": [
            "DFEvents::onCreateSize", "DFEvents::onDestroySize"
        ]
    },
    "ForeignBlocks": {
        "enabled": true,
        "events": [
            "GlobalEvents::onForeignStarted", "GlobalEvents::onForeignEnded"
        ]
    },
    "Lifecycle": {
        "enabled": false,
        "events": [
            "GlobalEvents::onStarted", "GlobalEvents::onExited", "GlobalEvents::onMigrated"
        ]
    }
}
```

Листинг 6 – Группы событий

В данной конфигурации:

— включены группы *Memory* и *ForeignBlocks*, отвечающие за сбор событий, связанных с использованием памяти и запуском/завершением пользовательских функций;

— отключены группы *CFEvents* и *Lifecycle*, так как анализ событий жизненного цикла и фрагментов вычислений, в данном эксперименте не требуется.

Если бы целью профилирования был анализ времени выполнения не только пользовательских функций, но и вычислительных блоков в целом, можно было бы активировать соответствующие группы (*Lifecycle*) и расширить набор анализируемых метрик.

6.2.2 Конфигурация модулей профилирования

Аналогично для модулей профилирования: глобальная настройка профилировщика предполагает отключение всех модулей по умолчанию. Однако для данного эксперимента необходимо было активировать только те модули, которые отвечают за обработку выбранных метрик. Это было реализовано с помощью флага *overrideEnabled*, позволяющего переопределить глобальную настройку и явно включить нужные модули. В листинге 7 представлена конфигурация модулей.

```
{
    "globalSettings": {
        "enabled": false
    },
    "libdf_events_module": {
        "enabled": true
    },
    "libdf_sizer_events_module": {
        "overrideEnabled": true
    },
    "libforeign_block_timer_module": {
        "overrideEnabled": true
    },
    "libglobal_events_module": {
        "enabled": false
    },
    "libperfetto_module": {
        "enabled": false
    }
}
```

Листинг 7 – Конфигурационный файл модулей

В данном эксперименте были активированы модули:

— *libforeign_block_timer_module* — отвечает за измерение времени выполнения пользовательских функций по событиям *onForeignStarted* и *onForeignEnded*.

— *libdf_sizer_events_module* — отслеживает динамику выделения и освобождения памяти по событиям *onCreateSize* и *onDestroySize*.

Остальные модули были оставлены отключёнными, чтобы минимизировать накладные расходы и сфокусироваться исключительно на необходимых для эксперимента метриках.

Таким образом, настройка профилирования была произведена избирательно – с помощью конфигурационных файлов были активированы только те события и модули,

которые отвечают целям данного анализа. Такой подход демонстрирует гибкость и конфигурируемость разработанной подсистемы профилирования: при изменении целей эксперимента можно легко скорректировать перечень собираемых метрик без модификации исходного кода приложения.

6.3 Сбор данных профилирования

Профилирование LuNA-программы перемножения матриц осуществлялось путём запуска приложения с заранее подготовленными конфигурационными файлами событий и модулей. Для запуска использовалась следующая команда:

luna examples/mxm/mxm.fa

где *examples/mxm/mxm.fa* – путь к тестовой LuNA-программе, реализующей перемножение матриц в блоках.

В процессе выполнения приложения подсистема профилирования фиксировала все необходимые события, связанные с запуском и завершением пользовательских функций, а также с выделением и уничтожением памяти для фрагментов данных. После завершения работы программы агрегированные данные в заданном формате были выведены модулями в консоль. Пример консольного вывода профилировочных данных приведён на рисунке 6:

```
Func: block_45, PID: 1405, TID: 140577953797888 => 1 ms total
Func: block_45, PID: 1405, TID: 140577962190592 => 1 ms total
Func: block_45, PID: 1405, TID: 140577970583296 => 1 ms total
Func: block_45, PID: 1405, TID: 140578053592832 => 1 ms total
Func: block_5, PID: 1405, TID: 140577953797888 => 10388 ms total
Func: block_5, PID: 1405, TID: 140577962190592 => 10393 ms total
Func: block_5, PID: 1405, TID: 140577970583296 => 10362 ms total
Func: block_5, PID: 1405, TID: 140578053592832 => 10390 ms total
Func: block_9, PID: 1405, TID: 140577953797888 => 43 ms total
Func: block_9, PID: 1405, TID: 140577962190592 => 42 ms total
Func: block_9, PID: 1405, TID: 140577970583296 => 42 ms total
Func: block_9, PID: 1405, TID: 140578053592832 => 43 ms total
Total DF size created: 10371200168 bytes
Total DF size destroyed: 10371200168 bytes
```

Рисунок 6 – пример вывода результатов профилирования в консоль

Из полученного вывода можно сделать несколько ключевых наблюдений:

— суммарный объём выделенной памяти (*Total DF size created*) совпадает с объёмом уничтоженной памяти (*Total DF size destroyed*); это свидетельствует об отсутствии явных утечек памяти в программе, что является важным индикатором корректности работы с памятью;

— профилировщик зафиксировал выполнение пользовательских функций на четырёх различных потоках (по разным идентификаторам TID), что подтверждает корректное распределение задач по потокам процессора;

— по результатам профилирования можно видеть, что функция *block_5* существенно отличается по времени выполнения от других блоков; если обратиться к реализации кода, становится понятно, что именно в *block_5* производится непосредственно операция перемножения матриц, обладающая кубической сложностью по времени – это объясняет значительные затраты времени на выполнение данного блока.

— подобная детализация позволяет оперативно обнаруживать тяжёлые участки кода, которые требуют оптимизации или дополнительного внимания со стороны разработчика.

Таким образом, собранные в ходе эксперимента данные подтвердили эффективность и удобство применения разработанной подсистемы профилирования для анализа параллельных приложений на платформе LuNA.

6.4 Визуализация и анализ результатов

Для наглядного анализа собранных данных профилирования был использован инструмент Perfetto [12] – современная система трассировки, изначально разработанная компанией Google для анализа производительности Android и Linux, а ныне активно применяемая для широкого круга задач мониторинга и визуализации временных событий. Perfetto предоставляет удобный веб-интерфейс (Perfetto UI), позволяющий загружать профилировочные логи в формате JSON и отображать их в виде временных диаграмм, графиков и таблиц, что существенно облегчает анализ сложных параллельных программ.

Для интеграции с Perfetto был разработан специальный модуль *perfetto_module*. Данный модуль собирает события профилирования во время работы программы, а в деструкторе формирует итоговый файл в формате JSON, полностью совместимом с Perfetto UI.

Формат событий, записываемых модулем, соответствует общепринятым категориям Perfetto:

— для профилирования объёмов памяти используется структура записи с полями: *{"name":"DFSizer","cat":"DF","ph":"C","ts":1992147,"dur":0,"pid":6506,"tid":6524,"args":{"created":256252196,"destroyed":61008}}*

где:

— *name*: имя события (например, "*DFSizer*" для событий, связанных с памятью),

— *cat*: категория (например, "*DF*");

— *ph*: тип события (*C* – counters, используется для отображения изменения значения во времени);

— *ts*: момент времени события (в микросекундах с момента старта профилировщика);

— *pid, tid*: идентификаторы процесса и потока;

— *args*: дополнительные параметры, например объём выделенной (*created*) и уничтоженной (*destroyed*) памяти к данному моменту;

— для временных интервалов выполнения пользовательских функций: *{"name":"block_5","cat":"ForeignBlock","ph":"X","ts":5468567,"dur":521,"pid":6506,"tid":6523,"args":{}}*

где *"ph": "X"* обозначает интервал между двумя событиями (начало и продолжительность выполнения).

Пример фрагмента итогового файла трассировки показан в листинге 8.

```
{"name":"bl5","cat":"ForeignBlock","ph":"X","ts":3287227,"dur":404,"pid":6506,"tid":6526,"args":{}}
{"name":"DF","cat":"DF","ph":"C","ts":3287640,"dur":0,"pid":6506,"tid":6526,"args":{"created":600,"destroyed":400}}
{"name":"DF","cat":"DF","ph":"C","ts":3287641,"dur":0,"pid":6506,"tid":6526,"args":{"created":650,"destroyed":410}}
{"name":"bl5","cat":"ForeignBlock","ph":"X","ts":3287506,"dur":368,"pid":6506,"tid":6523,"args":{}}
{"name":"DF","cat":"DF","ph":"C","ts":3287884,"dur":0,"pid":6506,"tid":6523,"args":{"created":660,"destroyed":420}}
{"name":"bl5","cat":"ForeignBlock","ph":"X","ts":3286887,"dur":995,"pid":6506,"tid":6525,"args":{}}
{"name":"bl5","cat":"ForeignBlock","ph":"X","ts":3287526,"dur":373,"pid":6506,"tid":6524,"args":{}}
```

Листинг 8 – фрагмент файла трассировки

Полученный JSON-файл загружается в Perfetto UI, где автоматически строится временная шкала всех событий, а также графики изменения выделенной и освобождённой памяти, и временные интервалы выполнения функций. Результат показан на рисунке 7.

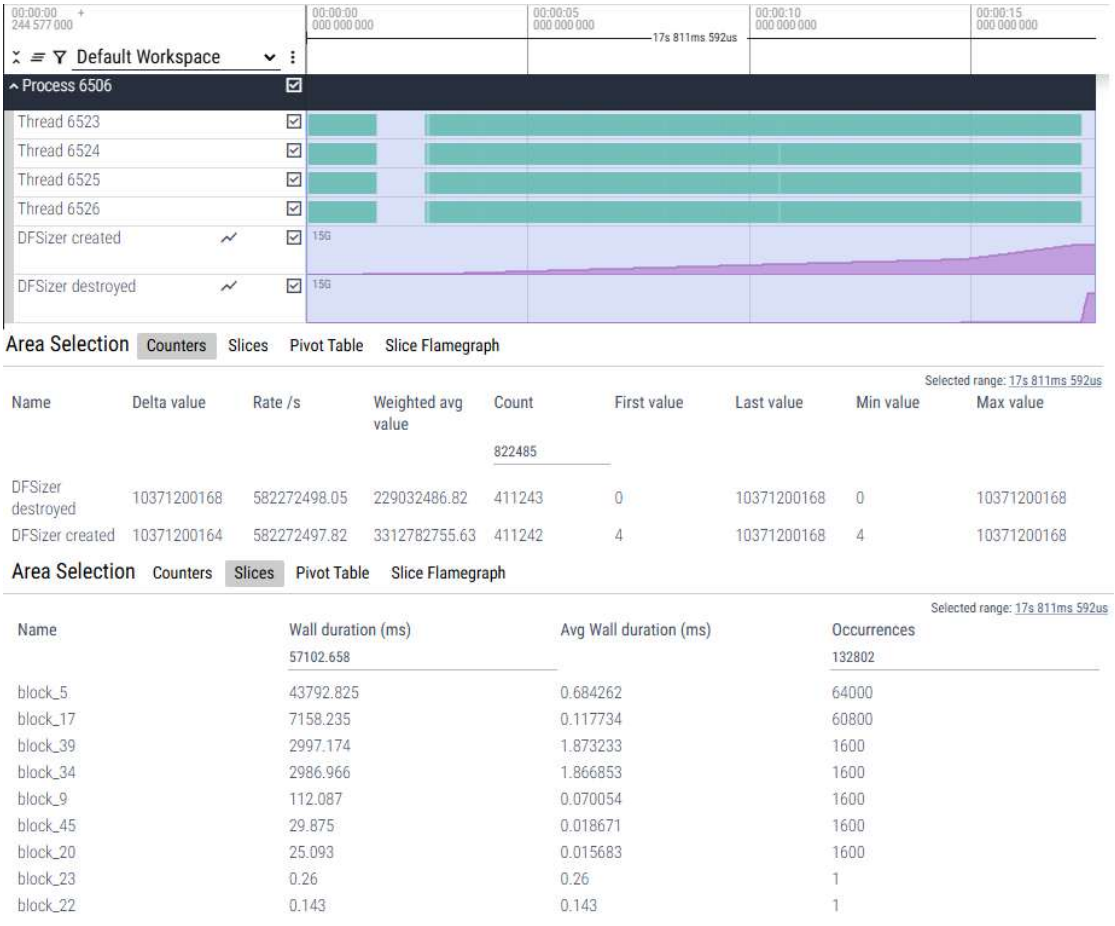

Рисунок 7 – результат визуализации инструментом Perfetto

Визуализация профиля в Perfetto наглядно подтверждает наблюдения, полученные при анализе консольного вывода (см. раздел 6.3):

— ясно видно, что функция *block_5* занимает существенно больше времени по сравнению с остальными блоками;

— график памяти демонстрирует, что суммарный объём выделенной памяти равен объёму освобождённой, что говорит об отсутствии утечек;

— на временной диаграмме заметно, что интенсивное выделение памяти начинается только во второй части выполнения программы, а окончательное освобождение происходит уже в самом конце; это указывает на специфику работы приложения: память удерживается до завершения вычислений, и только после выполнения основной задачи происходит её полное освобождение; подобное поведение может быть неэффективным для задач с ограниченными ресурсами и сигнализирует о потенциальных направлениях оптимизации, например, о необходимости более раннего освобождения памяти по мере завершения отдельных блоков.

Таким образом, использование модуля Perfetto и визуализации в Perfetto UI позволяет не только собирать и анализировать данные профилирования, но и быстро выявлять узкие места в производительности и особенностях использования ресурсов, что делает данный подход эффективным инструментом для отладки и оптимизации параллельных приложений на платформе LuNA.

6.5 Влияние профилирования на производительность

Одним из важных аспектов внедрения подсистемы профилирования является анализ накладных расходов, которые возникают при сборе метрик во время выполнения параллельной программы. Для количественной оценки накладных расходов профилирования была проведена серия измерений времени выполнения задачи умножения матриц на двух разных конфигурациях фрагментации данных: высокой (матрица разбивалась на 40x40 фрагментов размером 100x100) и низкой (матрица разбивалась на 20x20 фрагментов размером 200x200). Эксперимент выполнялся на ноутбуке с процессором Intel Core i7-1355U (10 ядер 12 потоков). В каждой конфигурации сравнивались четыре режима профилирования: «события отключены» (базовый случай), «события пользовательских функций», «события памяти» и «события памяти и пользовательских функций». Время выполнения базового случая при низкой степени фрагментированности составило 19,64 секунды, тогда как при высокой степени фрагментированности данный показатель увеличился до 26,54 секунды. Относительные приросты времени выполнения для различных режимов профилирования приведены на рисунке 8.

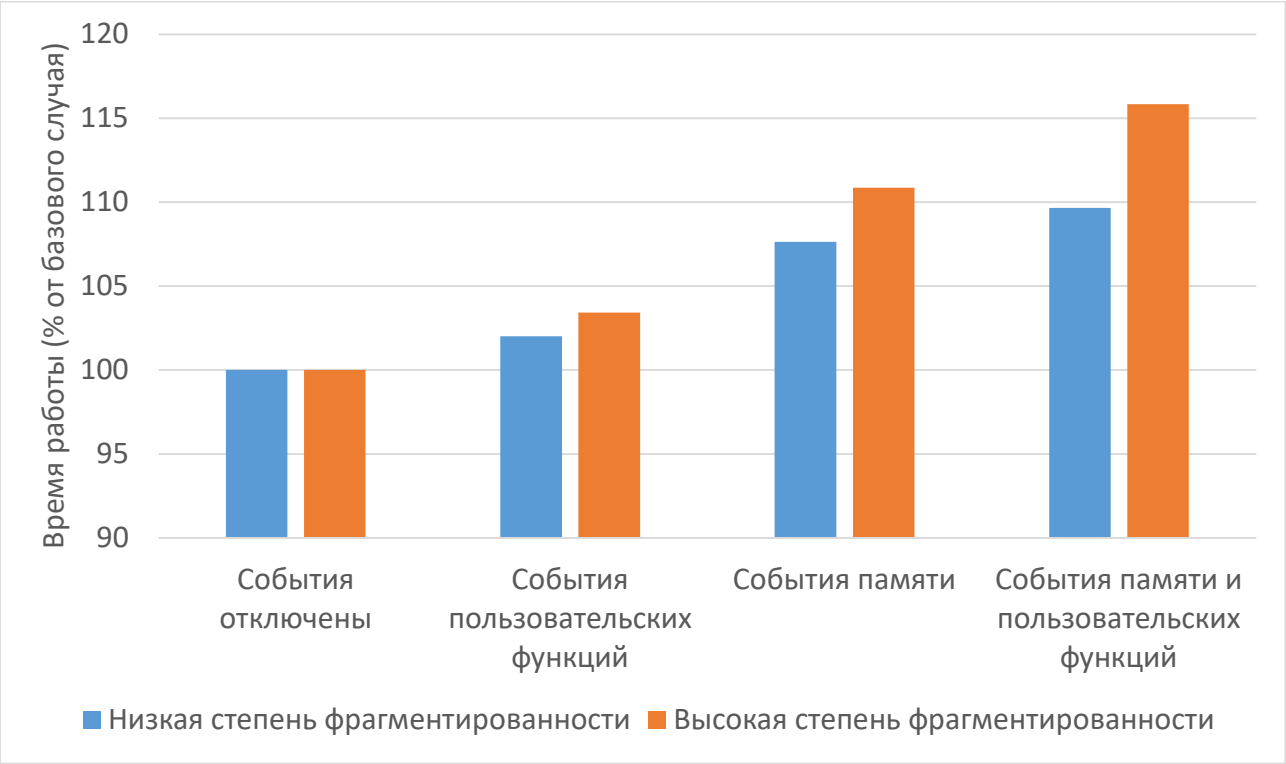

Рисунок 8 – относительные приросты времени

Из приведённых данных видно следующее:

— минимальное время выполнения достигается при полном отключении профилирования, то есть, когда не происходит фиксации ни одного события;

— включение только событий пользовательских функций (*onForeignStarted*, *onForeignEnded*) увеличивает время выполнения в среднем на 2-3%. В данном случае обработка событий происходит гораздо реже (только при запуске и завершении функций), и накладные расходы на синхронизацию оказываются значительно ниже, чем при интенсивном отслеживании операций с памятью;

— включение только событий памяти (*onCreateSize*, *onDestroySize*) приводит к увеличению времени выполнения на 8-11%; это объясняется тем, что операции выделения и освобождения памяти происходят очень часто (в связи с активным созданием фрагментов данных), и каждый вызов события сопровождается блокировкой мьютекса-счётчика, что приводит к дополнительным задержкам.

— совместное включение обеих групп событий приводит к наибольшим накладным расходам – время выполнения увеличивается на 9-13%; здесь наблюдается кумулятивный эффект от частых блокировок мьютекса при отслеживании операций с памятью и дополнительной обработки событий пользовательских функций.

Представленные на графике значения наглядно отражают закономерность: с увеличением числа фрагментов – а значит, и числа фиксируемых событий – накладные расходы на профилирование возрастают. Следует подчеркнуть, что в рассматриваемом

эксперименте основной вклад в увеличение времени выполнения вносит отслеживание событий, связанных с памятью, что обусловлено их высокой частотой и необходимостью синхронизации между потоками. В то же время профилирование только пользовательских функций оказывает сравнительно незначительное влияние на производительность. При проектировании и настройке подсистемы профилирования рекомендуется активировать лишь те события, которые действительно необходимы для анализа, с целью минимизации влияния профилирования на время выполнения программы.

Таким образом, гибкая система конфигурирования подсистемы профилирования LuNA позволяет пользователю самостоятельно выбирать баланс между полнотой сбора информации и влиянием на производительность приложения.

6.6 Выводы по результатам эксперимента

В ходе эксперимента с примером умножения матриц на платформе LuNA была продемонстрирована работоспособность разработанной событийной подсистемы профилирования. В профилируемом коде записывались события начала и конца пользовательских функций (*onForeignStarted* и *onForeignEnded*) и выделения/освобождения памяти (*onCreateSize* и *onDestroySize*). С помощью визуализатора Perfetto получены временные диаграммы работы потоков и динамики использования памяти. Анализ данных показал, что основной объём времени занимает код вычисления умножения матриц (тройной вложенный цикл), что соответствует типичным "горячим" участкам программ. Одновременно было видно, что профилирование вносит дополнительную нагрузку: измеренное время выполнения при включённых событиях оказалось выше, чем при обычном запуске. Как известно, инструментальное профилирование может существенно увеличивать затраты времени (например, метод профилирования путей даёт 30–40% замедления). При этом грамотный подход позволяет снизить накладные расходы – при выборочном включении событий их можно ограничить несколькими процентами от времени исполнения. В данном эксперименте дополнительное время составило порядка десятка процентов от базового, что соразмерно масштабам исследуемой задачи. Фрагментация и поведение менеджера памяти остались в пределах ожидаемого: после завершения цикла «освободившаяся» память практически сразу возвращалась в общий пул, утечек памяти (несогласованных пар *create*/*destroy*) не выявлено. Нагрузка на потоки распределена достаточно равномерно – каждый поток выполнял сопоставимый объём работы и не простаивал длительное время. Диаграммы Perfetto показали периодические синхронизации потоков между этапами умножения, но явных «узких мест» по параллельности обнаружено не было.

Практическая ценность полученных результатов состоит в демонстрации возможностей подсистемы профилирования: она позволяет выявлять критические участки кода, где концентрируется основное время работы программы, а также контролировать поведение менеджера памяти и обнаруживать потенциальные утечки памяти [13]. Если бы при профилировании показалось, что суммарный объём освобождённых объектов меньше объёма выделенных, это сразу сигнализировало бы об утечке. В данном случае гармоничное совпадение событий создания и разрушения памяти подтверждает корректную работу менеджера памяти LuNA. Кроме того, инструмент выдал наглядную сводку загрузки каждого потока и диаграмму конкуренции, что важно для оценки эффективности параллельных вычислений. Таким образом, собранные сведения полезны как для оптимизации «тяжёлых» вычислительных узлов, так и для отладки и улучшения подсистемы управления памятью платформы.

На основании анализа можно сформулировать рекомендации по настройке профилирования в зависимости от целей эксперимента. Если главная цель – измерить производительность кода, целесообразно активировать только события времени выполнения функций (*onForeignStarted*/*onForeignEnded*) и избегать детального профилирования памяти, чтобы не создавать избыточных накладных расходов. При необходимости сконцентрироваться на динамике памяти, следует включать события *onCreateSize*/*onDestroySize*. Современные трассировщики, например Perfetto, рекомендуют собирать «только то, что нужно». Настройка должна учитывать компромисс между полнотой информации и точностью исходного выполнения: сбор всех возможных событий (функций, системных вызовов, аллокаций и т.д.) даст богатую картину, но значительно замедлит программу. Следовательно, для эксплуатационного мониторинга рекомендуется узкоспециализированное профилирование на нужных модулях, а для детальной отладки – обширное с последующим анализом.

Наконец, результаты данного примера подкрепляют утверждение о достижении общей цели работы – создании конфигурируемого компонента для сбора и анализа сведений о фрагментированных LuNA-программах. На практическом примере показано, что событийная модель реализована: события *onForeignStarted*/*onForeignEnded* и *onCreateSize*/*onDestroySize* успешно генерируются и обрабатываются независимо друг от друга. Была продумана модульная архитектура – при необходимости можно добавлять новые типы событий или источники данных без перекомпиляции кода. Механизм конфигурации также подтвердил свою работоспособность: через файлы настроек задаются именно те события и модули, которые требуются пользователю, и игнорируются все остальные. Таким образом, эксперимент завершён успешно: собранные профили позволяют анализировать

производительность приложений LuNA, а применённые в нём решения (настроенные события и их визуализация) демонстрируют выполнение задач ВКР по разработке расширяемого инструмента профилирования.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы была разработана конфигурируемая подсистема профилирования для системы LuNA. Основная цель разработанного программного компонента — сбор и анализ характеристик исполнения параллельных приложений, генерируемых в рамках системы LuNA. Подсистема позволяет гибко настраивать параметры профилирования через конфигурационные файлы, что обеспечивает адаптацию к различным сценариям работы. В разработанной подсистеме реализованы механизмы сбора данных о времени выполнения модулей, загрузке процессора и объёме используемой памяти. Архитектурные решения подсистемы интегрированы в общую структуру LuNA, что обеспечило корректную передачу и обработку собранных метрик.

Проведён анализ эффективности работы профилировщика: результаты экспериментальных исследований показывают, что внедрение подсистемы не оказывает критического влияния на общую производительность системы. Накладные расходы на сбор статистики остаются в допустимых пределах, что подтверждает целесообразность выбранных решений. Конфигурационные возможности позволяют задавать уровень детализации и объёмы собираемых данных без существенной реконфигурации системы. В целом поставленные задачи решены: реализован функциональный инструмент для мониторинга производительности параллельных приложений в LuNA. Использование профилировщика открывает перспективы для оптимизации распределения вычислительной нагрузки на основе собранных данных. Таким образом, достигнуты заявленные технические результаты, подтверждающие обоснованность предложенной архитектуры и методов профилирования.

Следует отметить, что основные результаты работы были представлены на 63-й международной научной студенческой конференции МНСК-2025 (16 апреля 2025 г. – 22 апреля 2025 г.) [14], где работа была отмечена дипломом третьей степени. Кроме того, полученные результаты и функционирование разработанной подсистемы были продемонстрированы на студенческом рабочем научно-организационном семинаре «Активные знания и система LuNA» в Лаборатории синтеза параллельных программ ИВМиМГ СО РАН 6 мая 2025 г.

Основные защищаемые положения:

— разработана конфигурируемая подсистема профилирования для системы LuNA, обеспечивающая сбор и анализ характеристик выполнения её модулей;

— предложена архитектура подсистемы, включающая механизм чтения конфигурационных параметров, динамическое профилирование и сохранение результатов в настраиваемые выходные файлы;

— реализован механизм настройки профилировщика через конфигурационные файлы LuNA, позволяющий гибко управлять процессом сбора данных;
— проведены экспериментальные исследования, подтвердившие корректность работы подсистемы и приемлемый уровень нагрузки на ресурсы системы при её использовании.

Направления дальнейшей работы:

— добавление возможности указания переменных конфигурации (например, названий выходных файлов профилировщика) для модулей через конфигурационные файлы, что позволит повысить гибкость настройки подсистемы;
— интеграция подсистемы профилирования с механизмом балансировки нагрузки, при которой собранные профилировщиком данные будут использоваться для динамического перераспределения задач между узлами системы;
— разработка инструментов автоматизированного анализа и визуализации профилировочных данных для более эффективной интерпретации результатов измерений.

Таким образом, реализация указанных направлений позволит существенно расширить функциональные возможности системы LuNA и повысить эффективность распределения вычислительных ресурсов.

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».

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах [Текст] / В. А. Перепелкин // Проблемы информатики. – 2020. – №1. – С. 66–74.
2. Абрамушкина Е. С. Разработка и реализация подсистемы профилирования для системы фрагментированного программирования LUNA // Выпускная работа бакалавра, НГУ Факультет информационных технологий. – 2023.
3. Intel Trace Analyzer and Collector [Электронный ресурс] // Корпорация Intel : официальный сайт. Режим доступа: https://www.intel.com/content/www/us/en/developer/tools/oneapi/trace-analyzer.html (дата обращения: 24.04.2025).
4. Scalasca [Электронный ресурс] // Scalasca : официальный сайт. Режим доступа: https://www.scalasca.org (дата обращения: 24.04.2025).
5. TAU (Tuning and Analysis Utilities) [Электронный ресурс] // Школа компьютерных наук и обработки данных Орегонского университета. Режим доступа: https://www.cs.uoregon.edu/research/tau/home.php (дата обращения: 24.04.2025).
6. HPCToolkit [Электронный ресурс] // HPCToolkit : официальный сайт. Режим доступа: https://hpctoolkit.org (дата обращения: 24.04.2025).
7. Голиков М. О. Разработка и реализация средств для визуального анализа исполнения фрагментированных программ // Выпускная работа бакалавра, НГУ, Факультет информационных технологий. – 2021.
8. Лямин А. С. Разработка и реализация алгоритма распределения ресурсов фрагментированных программ на основе профилирования // Выпускная работа бакалавра, НГУ, Факультет информационных технологий. – 2021.
9. Саяпин М. П. Разработка и реализация алгоритмов динамической балансировки вычислительной нагрузки для подсистемы воспроизведения трасс в системе LuNA. Выпускная работа бакалавра, НГТУ, Факультет прикладной математики и информатики. – 2022.
10. Мустафин Д. Э. Реализация централизованного подхода к динамической балансировке вычислительной нагрузки в системе фрагментированного программирования LuNA и его сравнение с децентрализованным подходом. Выпускная работа бакалавра, НГУ, Факультет информационных технологий. – 2022.
11. Объяснение паттерна Наблюдатель на примере Redux [Электронный ресурс] // Habr. Режим доступа: https://habr.com/ru/companies/nordclan/articles/697026/ (дата обращения: 24.04.2025).

12. Perfetto UI [Электронный ресурс]. Режим доступа: https://perfetto.dev/ (дата обращения: 24.04.2025).
13. Профайлер памяти. Зачем он нужен и как использовать [Электронный ресурс] // Habr. Режим доступа: https://habr.com/ru/companies/ruvds/articles/827644/ (дата обращения: 24.04.2025).
14. Симонов В. И. Разработка конфигурируемой подсистемы профилирования для системы LuNA // Параллельные вычисления : Материалы 63-й Междунар. науч. студ. конф. 16–22 апреля 2025 г. / Новосиб. гос. ун-т. – Новосибирск : ИПЦ НГУ (принято в печать).

ПРИЛОЖЕНИЕ А

КОНФИГУРИРУЕМАЯ ПОДСИСТЕМА ПРОФИЛИРОВАНИЯ ДЛЯ СИСТЕМЫ «LUNA»

Руководство программиста

Листов 7

Новосибирск 2025

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено руководство программиста для конфигурируемой подсистемы профилирования для системы LuNA. Программа написана на языке C++.

Оформление программного документа «Руководство программиста» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением № 1)».

1 Назначение программы

1.1 Функциональное назначение программы

Конфигурируемая подсистема профилирования LuNA предназначена для сбора информации об исполняемых фрагментированных параллельных программах в системе программирования LuNA.

1.2 Эксплуатационное назначение программы

Подсистема предназначена для сбора и анализа характеристик исполнения параллельных приложений, созданных в системе фрагментированного программирования LuNA. Она используется при разработке и оптимизации LuNA-программ, позволяя исследовать поведение программы и выявлять «горячие» участки кода, наиболее ресурсоёмкие фрагменты и узкие места в распределении вычислительной нагрузки. Подсистема позволяет гибко настраивать профилирование через конфигурационные файлы, адаптируя сбор данных под различные сценарии выполнения.

1.3 Состав функций

— Измерение времени выполнения функций, учёт загрузки процессора и использования памяти при работе фрагментированных программ.

— Загрузка списка профилировочных модулей и наборов событий согласно конфигурациям, поддержка групп событий для разных сценариев (например, «Базовое» и «Расширенное» профилирование).

— Динамическая регистрация модулей профилирования при старте приложения; привязка их обработчиков к объектам-событиям (например, «начало/окончание фрагмента», «отправка/приём данных»); при возникновении события диспетчер вызывает все соответствующие обработчики, которые накапливают метрики (запоминают время запуска фрагмента, увеличивают счётчики, измеряют объём данных и т. д.).

— Отправка собранных данных по сети, в настраиваемые выходные файлы или терминал.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

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

— оперативную память объемом 4 Гб или выше;

— процессор с тактовой частотой 1 ГГц или выше;

— жесткий диск объемом 32 Гб или выше;

— монитор;

— клавиатуру.

2.2 Минимальный состав программных средств

Системные программные средства, используемые подсистемой профилирования фрагментированных программ, должны быть представлены:

— операционной системой, поддерживающей интерпретатор языка Python версии 3.8 и выше и компилятор GNU C++;

— интерпретатор языка Python версии 3.8 и выше;

— компилятор языка GNU C++.

2.3 Требования к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки и опытом использования системы LuNA.

3 Выполнение программы

3.1 Загрузка и запуск программы

Для активации профилирования необходимо:

— убедиться, что динамические библиотеки профилировочных модулей и конфигурационные файлы (events_config.json, modules_settings.json) расположены в доступных для системы каталогах;

— указать в конфигурационных файлах необходимые модули, события, включить группы, требуемые для текущего сценария профилирования.

При запуске LuNA-приложения модуль профилирования загружается автоматически: ядро системы инициализирует диспетчер событий, читает конфигурационные файлы, загружает необходимые модули, вызывает их функции инициализации и регистрации обработчиков событий.

Загрузка модулей осуществляется динамически, посредством механизмов загрузки библиотек, предусмотренных в ОС (например, dlopen в Linux). Все подготовительные действия производятся до начала исполнения основного алгоритма программы.

3.2 Выполнение программы

Выполнение программы с активированной подсистемой профилирования происходит следующим образом:

— в ходе работы LuNA-программы в ключевых точках (например, при начале и завершении выполнения фрагментов, передаче данных и других событиях) вызываются объекты-события профилировщика;

— диспетчер событий определяет, какие обработчики должны быть вызваны для данного события, и последовательно инициирует выполнение соответствующих функций, реализованных в модулях;

— каждый обработчик реализует свою логику: измеряет время, подсчитывает количество событий, анализирует использование памяти, записывает данные в файл и т.д.

Собранные данные накапливаются во внутреннем состоянии модулей до завершения выполнения программы.

3.3 Завершение работы программы

Завершение работы подсистемы профилирования происходит синхронно с завершением LuNA-программы:

— все модули, выполняют процедуру финализации: анализируют собранные метрики, формируют итоговые отчёты и производят запись выходных данных (обычно в формате JSON, совместимом с Perfetto);
— в деструкторах модулей выполняется освобождение внутренних ресурсов и вывод финальных статистик (при необходимости – на консоль, на веб-сервер или в отдельные лог-файлы);
— после завершения всех процедур подсистема выгружает из памяти динамические библиотеки модулей, очищает внутренние структуры и освобождает используемые ресурсы.

ПРИЛОЖЕНИЕ Б

КОНФИГУРИРУЕМАЯ ПОДСИСТЕМА ПРОФИЛИРОВАНИЯ ДЛЯ СИСТЕМЫ «LUNA»

Описание программы

Листов 12

Новосибирск 2025

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено описание программы конфигурируемой подсистемы профилирования для системы LuNA. Программа написана на языке C++. Средство разработки – интегрированная среда разработки Visual Studio Code от компании Microsoft.

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 № 1)».

1 Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программного модуля – «Конфигурируемая подсистема профилирования для системы LuNA». Краткое название – «Конфигурируемая подсистема профилирования LuNA».

1.2 Программное обеспечение, необходимое для функционирования программы

— Интерпретатор языка Python версии 3.8 и выше.

— Компилятор языка C++ GCC.

1.3 Языки программирования

Программа написана на языке C++. Средство разработки – интегрированная среда разработки Visual Studio Code от компании Microsoft.

2 Функциональное назначение

2.1 Назначение программы

Программный модуль предназначен для сбора и анализа характеристик исполнения параллельных приложений, созданных в системе фрагментированного программирования LuNA. Она используется при разработке и оптимизации LuNA-программ, позволяя исследовать поведение программы и выявлять «горячие» участки кода, наиболее ресурсоёмкие фрагменты и узкие места в распределении вычислительной нагрузки. Подсистема позволяет гибко настраивать профилирование через конфигурационные файлы, адаптируя сбор данных под различные сценарии выполнения.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для запуска на платформах, не поддерживающих C++ 11 или Python 3.8.

3 Описание логической структуры

3.1 Алгоритм программы

Основные алгоритмы подсистемы профилирования следующие:

— чтение конфигурационных файлов: при запуске программы диспетчер событий осуществляет загрузку и разбор конфигурационного файла событий, в то время как диспетчер модулей читает и обрабатывает конфигурационный файл модулей;

— регистрация обработчиков (инициализация подсистемы): каждый загруженный модуль последовательно вызывается для регистрации; для каждого поддерживаемого модулем типа события вызывается макрос *CONNECT_EVENT(пространство_имен, событие, обработчик)*; макрос принимает три аргумента: пространство имён объекта-события, имя объекта-события и лямбда-функцию для обработки; если объект-событие ещё не создан, диспетчер создает его или игнорирует вызов, если событие отключено; в результате строится отображение «объект-событие → список обработчиков»;

— обработка событий во время исполнения: когда в LuNA происходит событие (например, «начало выполнения фрагмента»), исполнительная система вызывает метод диспетчера событий, передавая контекст (например идентификатор фрагмента, время, размер данных и т. д.); диспетчер ищет соответствующий объект-событие и последовательно вызывает все его обработчики; каждый обработчик выполняет свою часть логики: один может записать текущий момент времени, другой – увеличить счётчик, третий – добавить запись о памяти и т. д.; нагрузка на систему минимальна – преимущественно затраты времени занимают обход списка обработчиков и их вызов;

— обработка завершения работы: в конце работы LuNA программы, каждый модуль вычисляет итоговые статистики и выводит результаты (например, сериализует накопленные данные в файл); например, *ForeignBlockTimerModule* в деструкторе выводит в терминал статистику со временем всех функций; после этого диспетчер событий выполняет очистку, а диспетчер модулей выгружает библиотеки модулей; таким образом, обеспечивается полный цикл: от подключения обработчиков в начале до сохранения результатов и выгрузки модулей по завершении.

3.2 Структура программы

Подсистема состоит из нескольких логических компонентов:

— диспетчер событий (*EventsManager*): входит в ядро LuNA, получает сигналы из основного кода и оповещает модули.

— профилировочные модули (*IEventModule*): реализованы как отдельные динамические библиотеки внутри которых класс, реализующий интерфейс *IEventModule*; каждый модуль содержит свою логику сбора метрик и регистрирует обработчики событий;
— менеджер модулей (*ModulesManager*): класс для загрузки и хранения указателей на загруженные модули; обеспечивает поиск модуля по имени (например, *ModulesManager::instance()->get_module("logger_module"))*;
— файлы конфигурации: *events_config.json* (настройки событий и групп) и *modules_settings.json* (настройки модулей), располагаются в каталоге конфигурации проекта.

3.3 Связи между составными частями программы

Модули профилировщика спроектированы как слабосвязанные компоненты. Каждый модуль независимо собирает свою часть метрик и никак не влияет на внутренности других модулей или ядра LuNA. Взаимодействие происходит лишь через события: при регистрации каждый модуль подписывается на нужные события (например, запуск или завершение фрагмента), но не знает о наличии других модулей.

Таким образом, в процессе работы несколько модулей могут срабатывать «параллельно», каждый на своём наборе событий. Например, модуль A может считать количество обработанных фрагментов, модуль B — измерять время их выполнения, модуль C — отслеживать объём переданных данных. Все они получают уведомления об одних и тех же событиях почти одновременно (синхронизированно через диспетчер).

При этом модуль A может обратиться к модулю B косвенно: через *ModulesManager* один модуль может получить указатель на другой (например, модуль визуализации может запросить *logger_module*). Это позволяет разделять ответственность: например, *logger_module* пишет в файл всё, что приходит на вход, а другие модули используют его накопленные данные для генерации итоговых отчётов.

3.4 Связи программы с другими программами

Для запуска программы необходимы интерпретатор языка Python версии 3.8 и выше, а также компилятор языка GNU C++.

4 Используемые технические средства

Программа эксплуатируется на персональном компьютере или вычислительном кластере. Работа выполняется в консоли. Устройство, на котором запускается программа, должно иметь:

— оперативную память объемом 4 Гб или выше;

— процессор с тактовой частотой 1 ГГц или выше;

— жесткий диск объемом 32 Гб или выше;

— монитор;

— клавиатура.

5 Вызов и загрузка

Подсистема профилирования стартует вместе с LuNA-приложением при его запуске с активированной профилировкой. Во время инициализации LuNA считывает конфигурацию профилировщика и загружает указанные модули. Каждый модуль выполняет регистрацию своих обработчиков, вызывая API диспетчера событий. После инициализации все подготовлено к сбору событий: состояние подсистемы состоит из набора объектов-событий и связанных с ними обработчиков.

Дальнейшая работа происходит автоматически: при выполнении программы LuNA в ключевых местах (начало/окончание фрагмента, передача данных и т. д.) вызываются объекты-события через диспетчер. Подсистема вносит минимальное вмешательство – почти сразу после таких вызовов модули осуществляют сбор и накопление данных, после чего исполнение программы продолжается.

6 Входные данные

Настройки событий и групп задаются в файле *events_config.json* (секторы "*eventsSettings*" и "*groups*"). Настройки модулей указываются в файле *modules_settings.json*, содержащем глобальные параметры ("*globalSettings*") и разделы с индивидуальными настройками каждого модуля. Файлы конфигурации должны содержать ключи, соответствующие известным типам событий и названиям модулей; при отсутствии или некорректности ключей подсистема пропускает такие ключи.

7 Выходные данные

Основной результат профилирования – это лог событий и собранных метрик. В стандартной настройке используется формат JSON, совместимый с инструментарием просмотра Perfetto UI. Каждый обработчик модуля, как правило, формирует свои записи (например, временные метки начала/окончания фрагментов, объём выделенной памяти и т. д.), которые в конце программы сериализуются в один итоговый JSON-файл. Этот файл содержит хронологию событий, данную о ресурсах, и обеспечивает последующую визуализацию и анализ. Возможна также настройка на вывод в другие текстовые форматы, консоль или веб-сервер (например, модуль *logger_module* выводит простое текстовое логирование).

8 Лист регистрационных изменений

Таблица 1 – Лист регистрационных изменений

Лист регистрационных изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ документа	Входящий № сопроводит. докум. и дата	Подп.	Дата
Изм.	измененных	заменённых	новых	аннулированных					