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 компьютерных технологий

Направление подготовки 09.04.01 Информатика и вычислительная техника

Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Пермяшкина Дмитрия Андреевича

Тема работы:

ИССЛЕДОВАНИЕ МЕХАНИЗМОВ БАЛАНСИРОВКИ ЗАГРУЗКИ В ПРОЦЕСС-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

«К защите допущена»
Заведующий кафедрой,
д.т.н.
Зюбин В.Е. / ……
«31» мая 2022 г.

Руководитель ВКР
д.т.н, доцент,
заведующий кафедры КТ ФИТ НГУ
Зюбин В.Е. / ……
«31» мая 2022 г.

Соруководитель ВКР
к.т.н, доцент
старший преподаватель КТ ФИТ НГУ
Розов А.С. / ……
«31» мая 2022 г.

Новосибирск, 2022 г.

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…компьютерных технологий……
(название кафедры)

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

УТВЕРЖДАЮ
Зав. кафедрой…Зюбин В.Е.
(фамилия, И., О.)
…….
(подпись)
«17» декабря 2020г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту……Пермяшкину Дмитрию Андреевичу…… группы……20223……
(фамилия, имя, отчество, номер группы)

Тема…Исследование механизмов балансировки загрузки в процесс-ориентированных языках программирования...
(полное название темы выпускной квалификационной работы бакалавра)

утверждена распоряжением проректора по учебной работе от 17 декабря 2020г. №0446

Срок сдачи студентом готовой работы 31 мая 2022г.

Исходные данные (или цель работы) Изучение и адаптация различных алгоритмов балансировки загрузки по времени в процесс-ориентированных языках программирования

Структурные части работы

1) Введение (включая постановку задачи)
2) Описание алгоритмов, балансирующих нагрузку
3) Экспериментальные исследования
4) Вывод

Консультанты по разделам ВКР (при необходимости, с указанием разделов)

……
(раздел, ФИО)

Руководитель ВКР
ФИТ, профессор,
д-р техн. наук, доцент
Зюбин В.Е./……..
(ФИ О) / (подпись)

«17» декабря 2020 г.

Задание принял к исполнению
Пермяшкин Д.А./……..
(ФИО студента) / (подпись)

«18» декабря 2020г.

Соруководитель ВКР
ФИТ, старший преподаватель
б/с
Розов А.С../……..
(ФИ О) / (подпись)

«17» декабря 2020 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

WCET – Worst-Case Execution Time, наихудшее время исполнения. В данной работе используется как оценка сверху возможного времени исполнения программы или её части.

ЦП — центральный процессор.

Расписание — описание распределения нагрузки по различным циклам процесс-ориентированной программы.

Корректное расписание — расписание, не приводящее к сбоям в процессе исполнения программы.

Задача планирования — задача нахождения расписания для заданной программы.

Планировщик — алгоритм, решающий задачу планирования.

Задача балансировки — задача планирования, но с требованием оптимизации решения по некоторому критерию.

Балансировщик — алгоритм, решающий задачу балансировки.

ВВЕДЕНИЕ

В современном мире продолжается давно начавшийся процесс автоматизации промышленного производства. Всё большее число технологических процессов переводится на исполнение конвейерам, появляется всё больше процессов, которые выполнимы только при помощи автоматических станков и других устройств. При этом, одной из главной особенностей данных технологических процессов является делимость на кучу мелких процессов или действий. Например, простая операция «открыть клапан» описывается как: включить мотор на небольшой промежуток времени, через некоторое время проверить, открылся ли клапан, и если что — подать сигнал об неисправности. И выходит, что в рамках автоматизации производств надо решать уже классическую проблему об организации параллельных вычислений.

В Институте автоматики и электрометрии СО РАН была предложена, подтверждена практически и активно исследуется парадигма процесс-ориентированного программирования [1]. В рамках данной парадигмы предлагается рассматривать программу как гиперпроцесс, состоящей из процессов. Сам процесс — радикально модифицированный конечный автомат. Модификация состоит в том, что у процесса есть его состояние и набор функций, меж которыми он может переключатся по событиям (т. е. событийно-управляемый полиморфизм). Данная парадигма позволяет создавать программы и даже языки, ориентированные на программирование управляющих систем алгоритмов в промышленной автоматизации и робототехнике. Например, язык Рефлекс (или же «Си с процессами»).

Базовая реализация транслятора языка Рефлекс предполагает многопоточный логический параллелизм исполнения процессов в стиле round-robin: алгоритм исполняется циклически, в каждом цикле каждый процесс активизируется, исполняет свою текущую функцию и определяет свою реакцию на внешнее событие (включая текущую функцию процесса на следующем цикле). Т. е. можно с уверенностью говорить о корпоративной

многопоточности.

Но к сожалению в последнее время было замечена тенденция на усложнения управляющих алгоритмов. И это не было бы проблемой, если бы из-за этого не росли требования к вычислительной платформе. При этом стоит отметить то, что скорость роста требований к вычислительной мощности и требуемой электроэнергии превосходит скорость роста данных характеристик у вычислительных устройств.

И в таком случае встаёт вопрос — а насколько ранее описанное поведение транслятора языка Рефлекс (где все процессы активируются каждый цикл) является корректным с точки зрения требовательности алгоритма. Оказывается, что вполне можно исполнять некоторые процессы раз в несколько циклов. Вариант некоторого алгоритма, который выполняет такую разбивку, был представлен в [2]. Но к сожалению он требует выполнения «гипотезы о идеальном синхронизме» (т. е. без разбивки всё работает и укладывается в требования к времени реакции). Что не всегда выполнимо и требует гораздо более мощной платформы.

Целью работы является поиск алгоритмов балансировки нагрузки в процесс-ориентированных языках программирования. Такой алгоритм должен находить такое расписание для балансировки нагрузки, чтобы при любом ходе работы программы не нарушалось требование к времени обработки сигнала и не нарушалась цикличность программы.

Задачами дипломной работы в связи с указанной целью является:

1. Изучение предметной области на предмет уже существующих подходящих алгоритмов и моделей.

2. Адаптация алгоритмов для поиска любого правильного решения.

3. Адаптация алгоритмов для поиска более оптимального решения по какому-либо критерию.

1 МОДЕЛЬ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ И НОТАЦИЯ

1.1 Модель вычислительной системы

В данной работе рассматривается вычислительная система с одним процессором, которая исполняет всего одну процесс-ориентированную программу. Данная программа состоит из множества процессов $\boldsymbol{J}$ размером n и не изменяется после компиляции или трансляции программы. Процесс же сам можно описать как тройку чисел:

- i — порядковый номер процесса;
- T – WCET;
- P – максимальное допустимое время отклика на внешний сигнал.

Хоть у процесса можно выделить гораздо больше характеристик, но для целей данной работы данной тройки вполне будет достаточно. При этом стоит отметить, что хоть идея описания требуемой вычислительной мощности числом операций и выглядит более логично из-за возможности анализа без учёта конкретного вычислителя, но использование времени является общепринятой практикой в работах данной направленности. Для этого есть две причины:

1. Время исполнения процесса зависит не только от мощности процессора, но и также от различных других компонентов системы. Вне процесс-ориентированной парадигмы чаще всего главной проблемой является общение со внешней средой [3], в данной парадигме — обращение к медленной внутренней памяти [1].
2. Замер WCET для отдельного процесса — задача гораздо более простая и решаемая не только аналитическими, но и экспериментальными путями. В данной работе данные методы рассматриваться не будут.

Далее данная вычислительная система раз в $T_{акт}$ времени получает или производит для себя сигнал активации, по которому исполняет ранее описанный один цикл. Внутри цикла программа сначала считывает информацию из внешней среды, затем передаёт управление каждому процессу по порядку возрастания порядкового номера согласно расписанию, и в конце

записывает реакцию процессов на внешние события в окружающую среду. Если цикл завершился до начала следующего цикла — то программа просто ждёт начала следующего цикла. Учесть расходы на организацию ввода/вывода можно различными методами, в данной работе предполагается, что в наборе задач это уже учтено. Например, вводом нового процесса с наименьшим номером и $T_{акт}$ требуемым временем реакции.

1.2 Формальное определение расписания

Далее стоит ввести формальные определения решения исследуемых далее алгоритмов - расписания и корректного расписания. Возьмём некоторую функцию *shd*: $\mathbb{N} \rightarrow \boldsymbol{J}$, которая будет противопоставлять номерам циклов подмножество от множества процессов. Если программа будет запускать данное подмножество на цикле со соответствующем номере, то тогда данная функция — расписание.

Расписание *shd* будет являться корректным для набора процессов $\boldsymbol{J}$ и системы с периодом активации $T_{акт}$, если выполняются следующие требования:

1. Любой сигнал для любого процесса i будет обработан (записана реакция во внешнюю среду) не более, чем через P_i времени.
2. Любой цикл завершится не позднее, чем за $T_{акт}$ времени.

Достаточно логично, что нас будет интересовать только корректные расписания в качестве решения задачи балансировки.

2 МЕТОДИКИ РЕШЕНИЯ ЗАДАЧИ

2.1 Порядок решения задач

Для начала стоит определиться с порядком решения задач. Как не трудно заметить, любой алгоритм балансировки будет одновременно являться алгоритмом планирования. Поэтому с одной стороны логично рассматривать только алгоритмы планирования и строить оптимальные корректные расписания сразу. С другой стороны, некоторые алгоритмы решения задач оптимизации требуют некоторое стартовое значение. И хоть оно не обязательно должно быть корректным решением задачи, от выбора данного значения часто зависит как и время исполнения алгоритма, так и качество решения (близость к идеальному). Ну и некоторые алгоритмы оптимизации фактически работают по циклу: получить решение при некоторых условиях, проанализировать решение, поменять условия со сохранением корректности решений.

Работы в области оптимизации нагрузки не особо помогли с выбором [4, 5]. Самым популярным направлением в данной области является разработка планировщиков и оценка их эффективности на основе уменьшения требуемой памяти или вычислительной мощности. Но допустим для оптимизации потребляемой электроэнергии более выгодно будет распределить нагрузку равномерно по времени, даже если итогом будет более высокий процент утилизации ЦП.

Дополнительно при выборе порядка стоит учесть, что для работы с процесс-ориентированной программой почти все планировщики используют в явном или неявном виде «гипотезу об идеальном синхронизме». Учитывая это, в данной работе и был выбран такой порядок решения задачи:

1. Поиск и адаптация алгоритмов для решения задачи планирования.
2. Адаптация полученных алгоритмов планирования для решения задачи балансировки.
3. Поиск и адаптация новых алгоритмов для решения задачи планирования с учётом предыдущих алгоритмов.

2.2 Статические и динамические алгоритмы

Двумя основными классами алгоритмов для решения задач планирования и балансировки являются статические и динамические алгоритмы.

В случае статического алгоритма решение задачи известно до начала исполнения программы. Почти всегда данное решение не зависит от хода программы, единственное что зависит — это вставка паузы в случае, когда часть программы завершилась ранее, чем должна была в расписании.

Плюсы статического планировщика:

- Решение известно заранее — есть возможность верификации решения.
- Алгоритм может быть медленным, поскольку от скорости планирования не зависит скорость работы системы.

Недостатки статического планировщика:

- Невозможность учитывать изменения программы по ходу исполнения.
- Необходимость кодирования не только порядка, но и начала каждой части — проблема неожиданных пауз.
- Учёт только наихудшего случая.

В случае динамического алгоритма решение задачи составляется по ходу решения программы и зависит от хода исполнения программы.

Плюсы динамического планировщика:

- Программа может изменяться по ходу исполнения программы.
- Есть учёт завершения части программы ранее конца.
- Возможность использовать удачные моменты по ходу исполнения программы для дальнейшей оптимизации.

Недостатки динамического планировщика:

- Требование к малой ресурсоемкости алгоритма.
- Усложнение верификации расписания до почти невозможности произведения верификации.

По итогу, учитывая все плюсы и минусы, в данной работе будут

рассматриваться только статические алгоритмы. Причины на это три:

1. Требование к скорости работы. Поскольку рассматривается решение задачи в условиях промышленной автоматизации, то желательно уменьшить потребление вычислительных ресурсов на накладные расходы. Поэтому статическое решение, когда почти все расчёты происходят вне вычислительного устройства, позволяет экономить вычислительные ресурсы.
2. Возможность верификации полученного расписания. Что очень важно в условиях автоматизации фабрик, когда одна ошибка может повлечь за собой крупные убытки.
3. В процесс-ориентированном программировании не стоит активно проблема неожиданных пауз. Момент начала следующего цикла уже неявно закодирован через период активации, а завершение цикла ранее не приведёт к нарушению корректности расписания или неожиданным последствиям, ибо чтение новых сигналов происходит на момент начала цикла.

Таким образом, для целей данной работы пока не требуется подробное знание о возможных состояниях и матрице переходов между ними, поскольку переходы зависят от хода исполнения программы. А данной информации у алгоритма не будет. Поэтому процесс и можно описать всего тремя характеристиками.

2.3 Вытесняющие и не вытесняющие алгоритмы

В данном случае выбор не вытесняющих алгоритмов достаточно очевиден. Вытеснение появилось для того, чтобы позволить системе оставаться интерактивной даже при выполнении длительных задач [3]. Но в случае промышленной автоматизации вопрос об интерактивности полученной системы редко когда начинает влиять на дизайн системы. Даже наоборот, в случае фабрики вариант, когда все процессы не прерываются по умолчанию выгоднее — поскольку реализовать вытеснение для отдельных процессов возможно и для

не вытесняющей системы, а вот организация длинной процедуры без возможности прерывания, кроме как экстренного — зависит от схемы организации вытеснения. Вдобавок, организация вытеснения несёт дополнительные накладные расходы как на ресурсы ЦП, так и на требуемую память.

3 АЛГОРИТМЫ ПЛАНИРОВАНИЯ

3.1 Введение «делителя»

В большинстве работ по описанию алгоритмов планирования и балансировки используется немного отличная от описанной ранее в работе схема описания процесса (или задачи в терминах той схемы). Задача появляется раз с каким-то периодом и требуется исполнить её до прихода следующей подобной задачи. Достаточно легко заметить, что модель из этой работы вполне себе легко превращается в подобную модель. Для этого введём для каждого процесса i такое натуральное число D_i и назовём его «делителем». И поменяем первое требование для корректного расписания на следующее:

1. Для любого процесса i не должно существовать D_i циклов подряд, в течении которых процесс не запускался.

Потому и такое название — поскольку данное число делит тактовую частоту системы для конкретного процесса. А вопрос о том, будут ли расписания корректные при исполнении новой пары требований лежит в методе выбора делителя. Выберем такие делители, что для каждого процесса i верно, что $P_i \geq 2D_i \cdot T_{акт}$. При таком выборе делителей расписание будет корректно, поскольку для любого сигнала найдётся такой промежуток в D_i циклов, который полностью лежит в отрезке времени размером в P_i. И следовательно сигнал успеет обработаться. Доказательство на рисунке 1, где красной линией обозначен сигнал, пришедший вне момента начала цикла, а $T_a = D_i \cdot T_{акт}$.

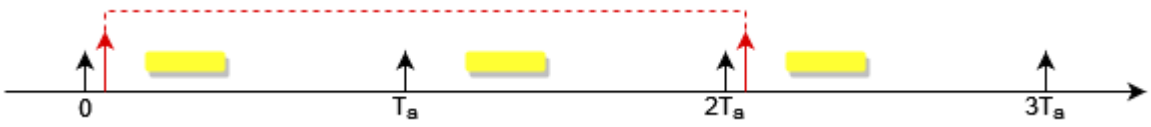

Рисунок 1 — Доказательство корректности перехода

Таким образом, можно сказать, что теперь в нашей модели тоже есть у каждого процесса период, с которым надо его выполнять. Остаётся открытым вопрос о существовании корректных расписаний.

3.2 Earliest Deadline First

Теперь, когда модель более подходит на широко используемую, можно начать описывать адаптации различных популярных алгоритмов для процесс-ориентированных программ. И начнём с одного из самых популярных и широко изученных алгоритмов — Earliest Deadline First (EDF) [6].

Идея данного алгоритма достаточно простая — планировать каждый раз задачу с ближайшим сроком исполнения. Данный алгоритм является динамическим не вытесняющим, что вроде как не подходит под наши цели. Но любой динамический алгоритм может быть превращён в статический путём имитации наихудшего случая в течении требуемого времени и запоминания вывода алгоритма.

А теперь встаёт вопрос — а сколько это «требуемого времени». И тут стоит заметить, что расписание в нашей модели является периодической функцией. Если до момента введения делителей это было не совсем очевидно, то теперь это должно быть очевидно. И даже период расписания (далее гиперпериод, чтобы не путать с периодом сигнала активации цикла) можно легко оценить как НОК(D_i). В таком случае адаптация алгоритма выглядит просто:

1. Выбрать наибольшие допустимые делители для минимизации требуемой ресурсоемкости.
2. Пройтись по первым НОК(D_i) циклам, симулируя ход работы программы в наихудшем случае:
    1. Каждый раз сигнал приходит в следующий момент времени, когда предыдущий сигнал начинает обрабатываться.
    2. Процесс всегда исполняется за WCET.
3. Каждый цикл добавлять неактивные процессы с ближайшим сроком исполнения, пока можно добавлять.
    1. Неактивным процессом считать такой процесс i, который ещё не исполнялся с предшествующего цикла, номер которого является наибольшим кратным D_i.

2. Добавлять процессы в цикл нельзя, если после добавления цикл будет исполняться больше, чем $T_{акт}$ времени.

Данная адаптация будет работать, но при реализации данного алгоритма у разработчика возникнет неожиданная проблема. Ранее было сказано, что число циклов, которые требуется перебрать — очевидно и можно явно посчитать. Так сколько это? В обычной процесс-ориентированной программе число процессов можно оценить как 1000. НОК от 1000 случайных чисел является огромным числом. Если быть точнее, то скорость роста НОК экспоненциально зависит от числа негармоничных чисел [7]. Как пример, набор из 10 чисел {1, 1, 5, 8, 9, 19, 160, 169, 297, 374} будет иметь НОК 2593974240, что много.

Для избегания данной проблемы в работе предлагается не брать наибольший делитель для каждого процесса, а воспользоваться популярной идеей о выборе гармоничных делителей [8]. Для этого сначала всё таки возьмём для каждого процесса наибольший допустимый делитель. Но потом понизим его по следующим правилам:

- Если делитель лежит в [1..10], то ничего не делаем.
- Если делитель лежит в [11..50], то понижаем его до ближайшего целого числа, кратного 5.
- Если делитель лежит в [51..500], то понижаем его до ближайшего целого числа, кратного 50.
- Если делитель лежит в [501..5000],..

Таким образом, делители будут лежать либо в [1..10], либо иметь вид $5a \cdot 10^b$, где a лежит в [1..10], b — небольшое целое число. И тогда верхняя граница на НОК будет $2520 \cdot 10^c$, где c — наибольшее из всех b. Для предыдущего набора из 10 чисел НОК после понижения будет всего лишь 252000. И данное число итераций выглядит уже адекватным и рабочим.

После понижения делителей данная адаптация планировщика будет уже вполне рабочим алгоритмом и будет находить корректные расписания для процесс-ориентированных программ.

3.3 Rate Monotonic

В качестве второго планировщика для адаптации в данной работе был выбран Rate Monotonic (RM) [9]. Данный планировщик также является популярным и широко известным и изученным — поэтому он и был выбран из всех различных вариантов. Вдобавок, как и EDF — данный планировщик является динамическим и не вытесняющим.

Идея данного планировщика простая — сортируем задачи по возрастанию периода появления задачи и каждый раз планируем доступный с наименьшим периодом. Адаптация очень простая:

1. Выбрать наибольшие допустимые делители для минимизации требуемой ресурсоемкости.
2. Понизить делители при необходимости.
3. Отсортировать процессы по возрастанию делителя.
4. Пройтись по первым НОК(D_i) циклам, симулируя ход работы программы в наихудшем случае:
    1. Каждый раз сигнал приходит в следующий момент времени, когда предыдущий сигнал начинает обрабатываться.
    2. Процесс всегда исполняется за WCET.
5. Каждый цикл добавлять неактивные процессы с наименьшими номерами, пока можно добавлять.
    1. Неактивным процессом считать такой процесс i, который ещё не исполнялся с предшествующего цикла, номер которого является наибольшим кратным D_i.
    2. Добавлять процессы в цикл нельзя, если после добавления цикл будет исполняться больше, чем $T_{акт}$ времени.

Как видно, адаптированные алгоритмы для EDF и RM примерно схожи. В теории данный алгоритм можно обобщить до общего алгоритма для адаптации динамических не вытесняющих планировщиков, где в последнем пункте добавление процесса происходит согласно выбранному алгоритму. На практике

данный вопрос требует дополнительного изучения на случай алгоритмов с памятью о предыдущих выборах.

4 АЛГОРИТМЫ БАЛАНСИРОВКИ

После того, как мы научились генерировать хотя бы какие-то корректные расписания, пришло время пытаться улучшить полученные расписания. Как было ранее описано, сначала будет предоставлена модификация алгоритмов из предыдущей главы с целью балансировки нагрузки между различными циклами процесс-ориентированной программы. Потом в работе будут рассмотрены два балансировщика, независимых от рассмотренных алгоритмов планирования, и открытые вопросы, связанные с ними.

4.1 Алгоритм балансировки на основе адаптированных алгоритмов

Оба ранее рассмотренных алгоритма добавляют процессы в цикл до того момента, когда они не смогут добавлять новые процессы из-за ограничения на время циклов. Данное свойство в теории должно привести к тому, что алгоритмы будут иметь один или несколько циклов, которые будут иметь почти максимальное время исполнения. Хорошая аналогия всему процессу балансировки — упаковка стопки книг различной толщины в несколько коробок, которые отличаются только номером. Естественно, одну и ту же книгу надо положить в несколько коробок, и есть ограничения на то, в какие коробки можно класть (делители), но на базовом уровне аналогия достаточно хорошая. И в рамках данной аналогии человек просто кладёт в каждую коробку книжки, пока коробка не заполнится, и потом берет следующую коробку. Но если текущая коробка полная, а в следующую данный человек кладёт всего одну книгу, не было бы более логично переложить часть книг из предыдущей коробки?

Но тут возникает вопрос — а как заставить вести так человека? Можно его научить перекладывать книжки и это будет аналогично новым алгоритмам. Но если мы не хотим менять сильно инструкции для человека, то что можно сделать? И тут появилась идея — а что, если запретить класть книги выше определённого процента высоты. Вне аналогии с коробками — а что, если запретить добавлять алгоритму процессы в цикл не при достижении процента

утилизации ЦП в цикле выше 100%, а раньше?

Теперь вопрос — а насколько раньше? Вполне логично, что выше 100% ставить ограничение нету. Дополнительно всегда можно выделить группу процессов, имеющих делитель равный одному. Процессы из этой группы будут исполняться на каждом цикле при любом корректном расписании, поэтому можно поставить нижнюю границу для ограничителя, равной WCET для всех процессов с таким делителем.

После установления начала и конца отрезка возможных решений остаётся выбрать алгоритм поиска решения на отрезке. Если бы можно было строго доказать, что если для ограничения T_1 на время исполнения цикла нет решения, то для любого другого ограничения $T_2 < T_1$ тоже нет решения, то можно было бы использовать двоичный поиск или любые другие методы оптимизации решения на отрезке. Но поскольку автору данной работы не удалось доказать данный факт строго, то было принято решение просто разделить отрезок на K частей и рассмотреть в качестве потенциальных решений все концы данных подотрезков.

Таким образом алгоритм балансировщика выглядит таким образом:

1. Выбрать наибольшие допустимые делители для минимизации требуемой ресурсоемкости.
2. Понизить делители при необходимости.
3. Найти процессы с делителем 1 и посчитать сумму их WCET, равную T_{d1}
4. Разделить отрезок $[T_{d1}..T_{акт}]$ на К подотрезков $\{[T_{d1}..T_1],[T_1..T_2],...,[T_{K-1}..T_K]\}$.
5. Для каждого конца подотрезка решать задачу планирования, пока не будет найдено корректное расписание, с учётом следующих изменений модели:
    1. Добавлять процессы в цикл нельзя, если после добавления цикл будет исполняться больше, чем T_i времени.
    2. Следующий цикл относительно текущего все равно начнётся только через $T_{акт}$ времени.

Таким образом можно преобразовать почти любой адаптированный алгоритм для балансировки нагрузки между циклами. Стоит отметить, что порядок обхода концов отрезка не важен для работоспособности алгоритма. Но достаточно очевидно, что если решение существует для некоторого ограничения T_i, то оно существует и для любого ограничения $T_j > T_i$. И таким образом, организация обхода подотрезков с начала отрезка позволяет остановить работу алгоритма при нахождении первого корректного расписания.

4.2 Балансировка на основе комбинаторной оптимизации

Следующим шагом в работе является поиск алгоритмов, зависящих от уже изученных только для предоставления начальной точки. И если внимательно присмотреться к постановке задачи, то можно заметить, что число возможных расписаний конечно. И следовательно — число решений конечно. В таком случае задача оптимизации вполне себе описывается в обозначениях комбинаторной оптимизации. Поэтому вполне логично попробовать использовать очень популярный метод ветвей и границ для решения задачи балансировки.

Хоть в теории данный метод и можно использовать прямиком на текущей модели, но одной из широко известных проблем всех комбинаторных методов является оценка сверху сложности алгоритма. Которая сводится к полному перебору всего множества решения. Даже после понижения делителей данное число может быть большим. На примере уже встречавшегося набора в 10 процессов беглая оценка даёт 2520000 возможных расписаний.

Поэтому было решено сузить число возможных расписаний. Для этого для каждого процесса i вводится ещё одно целое неотрицательное число S_i — смещение. И тогда процесс i запускается на цикле c тогда и тогда когда $c \bmod D_i = S_i$. Достаточно понятно, почему новое число называется смещением. Данная модель аналогична той, которая используется в языке Рефлекс. И достаточно очевидно, что данная модель не эквивалента основной модели в данной работе. Некоторые расписания, представимые в основной модели не представимы в

этой модели, поскольку промежуток в циклах между вызовом процесса статичен, а в основной модели он может меняться.

Теперь, когда мы уменьшили число перебираемых решений, время определить требуемые понятия для метода ветвей и границ. Пусть:

- Множество частичных решений $d = \{ S_i \mid i \in \{1..n\} \wedge S_i \in \{0..(D_i\text{-}1), o\}\}$. o - неопределённое смещение.
- $d(i) = S_i \mid S_i \in d$ - решение для процесса i из множества решений d.
- Атомарные множества решений - множество решений, в котором определены смещения для большей части процессов.
- Функция ветвления $b(d) = \{d_i \mid S_r = i \wedge d(r) = o \wedge d_i$ является корректным расписанием$\}$.
- Функцию выбора наилучшего решения — полный перебор всех неопределённых смещений.
- Начальный рекорд — расписание из входных данных.

Тогда можно применить классическую схему метода ветвей и границ [10], где алгоритм перебирает в цикле множество из множеств частичных решений, используя функцию ветвления для порождения новых частичных решений или отбрасывая решения в случае атомарности или если у множества нижняя граница решения уже выше, чем начальный рекорд.

Перед переходом к главной проблеме данного метода стоит отметить, что в данных определениях уже есть два важных момента, которые могут повлиять на скорость работы. Чем лучше начальный рекорд — тем больше множеств будет отброшено и тем быстрее алгоритм отработает. Выбор числа, когда множество частичных решений становится атомарным важен, поскольку полный перебор может быть достаточно долгим, но при этом накладные расходы на ветвление и потом перебор всех порождённых множеств могут оказаться больше, чем прямой полный перебор.

И теперь осталось определить последнюю часть алгоритма — функция

поиска нижней границы решения. Выбор данной функции очень важен, поскольку большую часть времени исполнения алгоритма балансировки занимает подсчёт нижней границы. В данной работе было опробовано две функции:

1. Распределение времени между всеми возможными смещениями для процессов с неопределённым смещениями.
2. Использование жадного алгоритма для временного выбора смещения и подсчёт суммы медианных нагрузок для каждого делителя.

И несмотря на то, что обе функции оказались рабочими в теории и на очень небольших программах с небольшим числом процессов алгоритм производил результат, но даже для 10 процессов данный балансировщик оказался не рабочим. Причина простая — производительность.

В случае первой функции поиска нижней границы для точного нахождения нижней границы требовалось пройтись по всем циклам. И хоть число циклов ограничено некоторым разумным числом, но учитывая, что данная функция вызывается очень часто, то даже если она будет работать менее секунды, это будет слишком долго. На практике это проявлялось как то, что для 10 процессов только через 8 часов было рассмотрено первое множество частичных решений, где было определено 5 различных смещения. При условии, что каждый раз рассматривалось самое свежее решение из множества множеств частичных решений.

В случае второй функции поиска сама функция была быстрой и позволяла перебирать достаточно быстро множества частичных решений. Но данная функция давала слишком слабую оценку нижней границы и по этому множество множеств частичных решений только росло и это привело к аналогичному результату по времени работы, как для предыдущей функции.

Таким образом, данный балансировщик хоть фактически и рабочий, но на практике он не может применяться с описанными функциями. Поэтому вопрос о применимости данного алгоритма на практике зависит от нахождения

достаточно строгой и быстрой функции поиска нижней границы. Именно поэтому далее данный балансировщик рассматриваться не будет.

4.3 Задача о разделении чисел

Ранее в данной работе упоминался язык Рефлекс и его идея балансировки нагрузки, основанная на делителях и смещениях. И в работе [2], описывающую данную идею был поставлен вопрос о том, можно ли решить задачу о распределении процессов по группам. И неожиданно, данная «задача о горошинах» достаточно хорошо изучена и называется «задача о разделении мультимножества чисел на подмультимножества» (multiway number partitioning).

Данная задача формулируется просто — есть мультимножество натуральных чисел и надо разделить его на несколько не пересекающихся подмультимножеств так, чтобы разница в сумме элементов подмультимножеств было минимально. Данная задача неплохо изучена, и несмотря на то, что она является NP-hard, есть достаточно методов получения приближенного к идеальному решений [11].

Единственная проблема, которая возникает при попытке наивно использовать любой алгоритм для решения задач о разделении внутри подмножеств процессов с одинаковым делителем по отдельности, заключается в похожем на интерференцию процессе. Если решать задачу для каждого подмножества процессов, то при комбинации данных решений в одно расписание оказывается, что итоговая нагрузка все равно имеет большие пики, просто потому что алгоритм решения принял схожие решения по распределению тяжёлых процессов. Вдобавок, не всегда равномерное распределение внутри группы может быть оптимальным. Например, если у нас есть два подмножества с делителями 5 и 10, и во втором подмножестве есть процесс, имеющий большое время работы, то логично распределить как можно меньше процессов в одно из смещений, процессы в котором будут запускаться в том же цикле, как и тяжёлый процесс.

Таким образом, необходимо сохранять информацию о распределении процессов и итоговой нагрузке для каждого делителя и смещения. И учитывать это при решении задачи на следующей итерации алгоритма.

В данной работе было опробовано два варианта сохранения информации. В первом варианте сначала решалась модификация задачи для каждого делителя отдельно. Модификация задачи заключалась в том, что вместо вычисления сумм подмультимножеств вычисляется сумма, умноженная на некоторый коэффициент — вес смещения для данного делителя. Далее после решения всех задач для каждого делителя каждый вес пересчитывался по следующему алгоритму:

1. Пройтись всем циклам в гиперпериоде и найти для каждой пары (D_j, S_l) $W_{j,l}$ — максимальную сумму всех WCET процессов, запускающихся в цикле.
2. Посчитать для каждого делителя и смещения новые коэффициенты w_j по формуле:

$$w_{j,k} = \frac{D_j * W_{j,k}}{\sum_{l=0}^{D_j - 1} W_{j,l}}$$

Второй вариант заключается в том, что обновление информации о распределении процессов с другими делителями происходило после каждого решения задачи о разделении. В данном варианте алгоритм проходил по всем существующим делителям по возрастанию (пропуская делитель 1, для которых смещение очевидно) и это считалось одной итерацией алгоритма. Сама же модификация задачи заключалась в том, что в подмультимножествах уже было некоторое начальное значение. Данное значение для каждой пары (D_j, S_l) было равно $W_{j,l}$ из первого варианта.

К сожалению оба варианта приводили к одному и тому же результату. На первой итерации в любом случае суммарная нагрузка на малые смещения была выше $T_{акт}$. Но на второй и далее итерациях распределение процессов по

мультимножествам либо повторялось с точностью до перестановки порядка мультимножеств, либо повторялось из достаточно небольшого набора возможных вариантов. При этом часто ни один из вариантов не приводил к корректному расписанию.

Изначальная идея была в том, что обычно есть один или два процесса, которые имеют слишком большие делители и поэтому было подмножество процессов с числом процессов гораздо меньше числа возможных делителей. Но даже после замены схемы понижения делителей на более агрессивную — понижение до минимального делителя, где уже есть хотя бы один процесс и число процессов с данным делителем не превосходит сам делитель.

К сожалению, данный эффект проявляется даже если есть всего два делителя, отличных от единицы, и оба образованных подмножества процессов имеют большую мощность. Поэтому данная идея балансировщика была признана не рабочей и далее рассматриваться не будет.

5 СХЕМА ДЛЯ СРАВНЕНИЯ АЛГОРИТМОВ

В ходе подготовки данной работы очень часто встречалась следующая проблема. Время, отведённое на подготовку, не бесконечно, поэтому для адаптации следовало выбирать только хорошие алгоритмы. Но тут возникла большая проблема — в большинстве работ производится только сравнительный анализ представленного алгоритма с некоторым идеальным алгоритмом на некотором наборе данных. При этом в качестве идеального алгоритма часто берут либо EDF, либо RM. Либо вообще какие-то модификации алгоритма, которые описаны в другой работе. Но для архитектора, который выбирает реализацию для системы, данная информация практически бесполезна. Частично можно сравнить некоторые свойства алгоритмов из-за транзитивности, но самая популярная метрика это «качество» расписания. Которая зависит от качества реализации эталонного алгоритма. Вдобавок для целей балансировки нагрузки простого «качества» не достаточно, поскольку в некоторых случаях большие пики могут быть нормальными, но в некоторых случаях желательно распределять нагрузку как можно равномернее.

Поэтому в данной работе будет предложена схема для экспериментального сравнения алгоритма, которая будет производить некоторые «абсолютные» величины, которые можно будет явно сравнивать меж разными работами. И далее все представленные алгоритмы, которые не были отвергнуты, будут подвергнуты экспериментальной оценке через эту схему с двумя целями:

1. Проверка работоспособности схемы. Поскольку RF и EDF хорошо изучены и основа алгоритма не была изменена, то если выводы из представленных критериях и выводы из других будут совпадать — то это будет означать, что схема рабочая.
2. Оценка эффективности изменений в адаптациях.

5.1 Критерии оценки

Как было сказано ранее — критерии должны быть «абсолютными». Это

означает, что критерии должны быть простым числом, которое можно напрямую сравнивать с аналогичным простым числом у другого алгоритма. При этом критерии должны как можно сильнее описывать именно алгоритм, а не конкретную реализацию. Естественно, что описывать только реализацию не возможно, но данное требование нужно для воспроизводимости результатов.

С учётом вышесказанного, данная работа предлагает следующий список критериев:

- Излишняя нагрузка расписания — разница между долями утилизации ЦП у расписания и в идеальном случае.
- Доли утилизации по циклам. В качестве критериев используется не сам ряд, а только следующие статистики — минимум, медиана, среднее, максимум.
- Размер расписания с указанием схемы кодирования или метода оценки.
- Время вычисления расписания.

Первый критерий показывает, сколько алгоритм добавляет к требуемой ресурсоемкости. Доли утилизации по циклам показывают распределение нагрузки по циклам (по времени). Размер расписания тоже важная метрика из-за того, что промышленные контроллеры ограничены не только в вычислительных ресурсах, но и в доступной памяти. А время вычисления позволит избежать проблем, аналогичных проблемам при адаптации комбинаторных методов оптимизации.

5.2 Схема эксперимента

Назовём набор из периода активации $T_{акт}$ и набора процессов $\boldsymbol{J}$ — задачей планирования. Хоть далее речь и будет говориться о схеме, применимой и к планировщикам, и к балансировщикам, особой разницы между ними делаться не будет. И поскольку балансировщик может выступать как планировщик, поэтому и было выбрано такое название. Сам эксперимент достаточно простой:

1. Генерация достаточно задач планирования.
2. Применить алгоритм к задачам планирования. В этот момент записывать

нужную информацию о ходе работы планировщика.

3. Проанализировать результат и записанную информацию для вычисления критериев.

Очевидно, что данный эксперимент может быть осуществляться в конвейерном стиле — каждый этап исполняется одновременно и каждый этап может осуществлять несколько исполнителей одновременно (на разных данных).

5.3 Генерация задач

Теперь осталось ответить на важный вопрос — как генерировать задачи планирования. Есть два основных подхода для генерации тестов — использовать некоторый заранее определённый набор тестов или генерировать его на лету. Первый набор обычно используется либо для слишком простых разбиений возможных входных данных на небольшое число классов или если выбранные элементы набора слишком важны. К сожалению, рано или поздно данные тесты начинают оценивать насколько хорошо алгоритмы проходят данные тесты. При этом это начинает происходить даже неосознанно, просто авторы начинают улучшать свои алгоритмы согласно результатам тестов...

Поэтому в данной работе предпочтение отдано генерации тестов на лету. Данный выбор тоже не идеален — требуется достаточно большое число тестов, чтобы покрыть все возможные классы входных наборов данных, всегда стоит вопрос о равномерности генерации среди возможных наборов. Начнём описывать схему генератора набора данных, при этом учитывая данные недостатки и пытаясь избежать их.

Для начала стоит ответить — сколько наборов генерировать. И на основе работы [5] рекомендуется использование минимум 1000 различных задач планирования.

Следующий вопрос — как генерировать сам набор задач. Сначала стоит решить, сколько процессов в наборе. В данной работе предлагается и используется случайный выбор из набора {10, 25, 50, 100}. Первый размер

набора позволяет проверить работоспособность на простых базовых программах и возможно проверить некоторые расписания вручную. Тогда как последующие всё ближе и ближе к реальным программам.

Следующий этап генерации — генерация максимальных допустимых времён реакции на внешние сигналы для процессов. Обычно для генерации периодов задач используется логарифмически равномерное распределение на интервале [1мс. 1000Мс] [5,12,13]. В данной работе предлагается генерация при помощи такого распределения, но из интервала [1мс, 2000мс]. Далее будет показано, как данный выбор позволит сохранить согласованность с другими работами.

Теперь надо выбрать WCET для процессов. Но перед этим надо выбрать $T_{акт}$. Которое зависит от процесса m с наименьшим P_m и обычно в процесс-ориентированных программах принимается за половину от P_m. В данной работе в главе 3.1 уже было приведено доказательство, что такой выбор позволит обработать любой сигнал для процесса m. Но встаёт вопрос — а можно ли выбрать иное. Достаточно очевидно, что можно выбрать меньший период и нижняя граница это максимальный WCET, который на данном этапе ещё не известен и забегая вперед — зависит от периода активации. Но можно ли выбрать больший период? И ответ на него — нет.

Докажем от противного. Пусть существует корректное расписание для $T_{акт} = 0{,}5P_m+\alpha$. И пусть для процесса m пришёл процесс и его обработка началась в цикле, начавшимся в T_0 момент времени. И пусть придёт второй сигнал для этого же процесса в момент времени $T_0+\beta$. И далее опишем, что будет происходить с этими сигналами по порядку:

1. В момент $T_0+T_{акт}$ закончится обработка первого сигнала и начнётся обработка второго сигнала.
2. В момент $T_0+2T_{акт}$ закончится обработка второго сигнала.
3. В момент $T_0+P_m+\beta$ наступит срок, до которого второй сигнал должен был обработаться.

Если расписание корректно, то $2T_{акт}$ должно быть меньше, чем P_m+β. Или же из-за нашего выбора $T_{акт}$, то $2\alpha \leq \beta$. И тогда очевидно, что для любого $\alpha > 0$ можно выбрать такое β, что расписание не корректно. И единственная возможность, когда расписание верно, это $\alpha = 0$. Что соответствует тому, что мы уже выбрали наибольший период активации.

И последним шагом генератор должен выбрать WCET для процессов. В данной работе не будет явно зафиксирован метод генерации, только требование того, что результирующий набор WCET должен быть равномерно распределён между всеми возможными значениями. Поэтому в данной работе будет использован алгоритм Dirichlet Rescale [14] для генерации долей утилизации ЦП u_i для каждого процесса i со следующими ограничениями:

- Сумма всех долей утилизации выбрана равновероятно из интервала [0,05, 0,95].
- Верхняя граница для процесса i равна $2T_{акт}/Pi$.

  И тогда WCET для процесса i будет равен $0{,}5Pi \cdot u_i$.

6 ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ

Как было ранее сказано, проведение эксперимента на исследуемых алгоритмах позволит достичь двух целей — проверки схемы эксперимента и анализ адаптаций. Именно поэтому сначала будут рассмотрены планировщики, потом балансировщики. И именно поэтому сначала в планировщиках будут описаны результаты, подтверждающие уже известные выводы про алгоритмы.

6.1 Анализ планировщиков

С графиками излишней нагрузки расписаний и статистиками долей утилизации можно ознакомится в Приложении А (вынесены из-за размера и числа). Анализ данных графиков для EDF показывает, что к сожалению данный планировщик склонен к большим пикам нагрузки. Почти для любой задачи планирования был найден цикл, который занимает почти 100% отведённого времени, и при этом минимум среди всех наборов максимальных долей все равно очень близок к 100%. При этом минимальная доля утилизации очень близка к нулю и соответствует почти горизонтальной линии. Если быть точнее, то минимальная доля скорее всего соответствует моменту, когда для планирования доступны только процессы с делителем один и очень малое число процессов с другим делителем. Это означает, что планировщик не пытается распределить нагрузку или сохранить некоторые процессы на потом — не сохраняющий нагрузку (non-preserving work) планировщик. При этом графики средней и медианной доли утилизации ЦП вполне себе ожидаемые и ведут себя вполне ожидаемо — на большом промежутке времени в среднем нагрузка будет выглядеть равномерно, но иногда будет выброс (либо пик, либо почти пустой цикл). Неравномерности линии можно отнести к двум причинам: наборы с малым числом процессов иногда не позволяют распределить нагрузку более равномерно, существование большой группы процессов с большим делителем.

Анализ с аналогичными результатами можно провести и для RM. Но при этом стоит выделить одно важное отличие — график медианы менее близок к

графику средней доли утилизации. Это связано с тем, что при нашей генерации процессы с меньшим делителем будут иметь чаще меньше WCET. Следовательно алгоритм сначала распределит более мелкие по ресурсоемкости процессы и потом будет распределять процессы с большей ресурсоемкостью. И как только процессы с меньшей ресурсоемкостью станут неактивными — то RM сразу их запланирует, а EDF может отложить их. Таким образом RM будет иметь больше пиков, а EDF будет иметь их реже, но больше по размеру.

Ранее описанные результаты совпадают с результатами анализов из работ [5, 12, 14], следовательно можно сделать вывод о применимости метода для анализа различных планировщиков и балансировщиков. Теперь можно продолжить анализировать специфичные для адаптаций моменты.

Начнём с алгоритма понижения делителей. Как видно из графиков величин излишнего планирования — внесённые поправки не привели к сильным излишним затратам. Из этого следует два вывода:

- Понижение делителя не происходит часто из-за использования логарифмически нормального распределения.
- Понижения делителя не приводит к большому излишнему расходу ресурсов ЦП.

Теперь осталось посмотреть на требуемые ресурсы по памяти и время расчёта расписания. Данные результаты приведены в таблицах 1 и 2. Для замера времени использовалась функция *clock()* из стандартной библиотеки C. По причине недостаточной точности используемого метода замер времени для задачи с 10 процессами не был произведён. Для экспериментов использовалась система с процессором AMD Ryzen 7 3700X с тактовой частотой 3.6 ГГц и с 16 ГБ оперативной памяти. Оценка размера расписания проводилась из расчёта 4 байта на делитель и 4 байта на каждый номер цикла, когда процесс должен активироваться.

И видно, что в среднем время расчёта неожиданно небольшое. Но вот иногда время расчёта увеличивается в несколько раз. Это связано с наборами,

которые имеют хотя бы один процесс, имеющий делитель более 500. Обычно наибольший делитель лежит около 50, что даёт разницу от 10 до 100 раз по гиперпериоду. А вот выбранная методика кодирования расписания требует сильных изменений и дальнейшего изучения других методов, ибо достаточно быстро размер полученного расписания превосходит разумные пределы.

Таблица 1 — Время работы планировщиков

Число процессов	Среднее время работы, EDF, мс	Максимальное время работы, EDF, мс	Среднее время работы, RM, мс	Среднее время работы, RM, мс
25	31	4187	15	3172
50	140	25764	84	21473
100	671	129522	393	103482

Таблица 2 — Размер полученных расписаний от планировщиков

Число процессов	Средний размер, EDF, байт	Максимальный размер, EDF, байт	Средний размер, RM, байт	Максимальный размер, RM, байт
10	315	4520	318	4516
25	10674	114562	10634	114580
50	606644	948232	606629	948200
100	1081292	113815944	1081291	113815944

По итогу данные алгоритмы уже могут использоваться в реальных программах на процесс-ориентированных языках, единственная проблема — размер расписания, что может пока что быть решена самим разработчиком системы.

6.2 Анализ балансировщиков

В данной работе для балансировщиков число отрезков было установлено в 10. С графиками излишней нагрузки расписаний и статистиками долей утилизации можно ознакомится в Приложении Б. Стоит отметить, что поскольку алгоритмы балансировки не сильно отличаются от алгоритмов планирования, то все сделанные выводы верны и для вариантов алгоритмов для балансировки. Стоит отметить, что графики минимальной, медианы и средней

долей утилизации не сильно поменялись. Но при этом заметно больше небольших отклонений в начальной части графика, что соответствует моментам, когда алгоритм балансировки смог успешно произвести заметную балансировку нагрузки. Данные пики совпадают с провалами на графике максимальной нагрузки, которые показывают моменты, когда применение алгоритма балансировки сработало очень удачно.

Хоть в схеме эксперимента и не было данного критерия, но для целей сравнения стоит посмотреть, какой процент задач удалось проанализировать. Процент таких задач приведён в таблице 3. И стоит отметить ещё один известный факт [15] — RM эффективнее EDF для такого рода задач.

Таблица 3 — Процент успешной оптимизации

Номер подотрезка	Процент задач, EDF	Процент задач, RM
7	1,00%	1,00%
8	4,00%	5,00%
9	41,00%	44,00%

При этом стоит отметить, что время исполнения алгоритма изменилось не в число отрезков раз. Это связано с тем, что в основном на ранних подотрезках алгоритм относительно рано замечает, что корректное расписание не получается построить и заканчивает итерацию рано.

Таблица 4 — Время работы балансировщиков

Число процессов	Среднее время работы, EDF, мс	Максимальное время работы, EDF, мс	Среднее время работы, RM, мс	Среднее время работы, RM, мс
25	93	20935	54	18472
50	432	103056	402	98328
100	4932	518088	4284	473829

По итогу стоит сделать выбор, что хоть оба алгоритма и рабочие как балансировщики, но лучше всего сделать выбор в сторону алгоритма на основе RM как более приспособленного к такого рода балансировки.

ЗАКЛЮЧЕНИЕ

В работе были исследована проблема балансировки нагрузки в процесс-ориентированной программе. Исследование заключалось в поиске и адаптации алгоритмов, решающих задачу планирования, а потом в адаптации данных и новых алгоритмов для решения задачи балансировки нагрузки. В начале работы был сделан обоснованный выбор в сторону статических не вытесняющих алгоритмов для решения задачи.

В данной работе было рассмотрены две адаптации популярных алгоритмов — Earliest Deadline First и Rate Monotonic. Полученная схема адаптации решает проблему роста гиперпериода полученного расписания и вдобавок может быть в теории использована для адаптации почти любого динамического не вытесняющего балансировщика.

На основе данных алгоритмов была разработана рабочая схема балансировки нагрузки между циклами процесс-ориентированной программы, использующая данные алгоритмы. Дополнительно была произведена попытка адаптации популярного метода комбинаторной оптимизации. Данная адаптация работает, но на практике не применима из-за долгого времени работы. В качестве последнего алгоритма была произведена адаптация уже известного алгоритма из языка Рефлекс с учётом изучения решений задачи о разделении мультимножества натуральных чисел на подмультимножества. К сожалению, данный алгоритм хоть и работал в теории, но на практике не смог произвести корректное расписание.

В последней части работы была представлена схема и критерии для сравнения алгоритмов планирования. Новизна заключается в том, что результаты данных экспериментов могут использоваться для сравнения алгоритмов меж работами без реализации самих работ. Работоспособность данной схемы была подтверждена путём вывода известных фактов об EDF и RM. Потом при помощи данной схемы было подтверждено, что выбранные планировщики и балансировщики хорошо справляются с поставленными перед

ними задачами.

В качестве планов дальнейшего развития предлагается адаптация большего числа алгоритмов планирования, попытки использования других методов комбинаторной оптимизации, а также решение большего числа проблем при реализации расписания на устройствах, включая оптимизацию кодирования готового расписания.

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

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

Пермяшкин Дмитрий Андреевич _____

«20» мая 2022г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Зюбин В. Е. Язык Рефлекс. Математическая модель алгоритмов управления //Датчики и системы. – 2006. – №. 5. – С. 24-30.
2. Зюбин В. Е. Статическая балансировка вычислительных ресурсов в процесс-ориентированном программировании // Вестник НГУ. Серия "Информационные технологии". 2012. Том 10. Выпуск 2. С. 44-54.
3. Arpaci-Dusseau R. H., Arpaci-Dusseau A. C. Operating systems: Three easy pieces. – Arpaci-Dusseau Books, LLC, 2018.
4. Buttazzo G. C. Hard real-time computing systems: predictable scheduling algorithms and applications. – Springer Science & Business Media, 2011. – Т. 24.
5. Nasri M., Brandenburg B. B. Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper) //2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). – IEEE, 2017. – С. 75-86.
6. Liu C. L., Layland J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment //Journal of the ACM (JACM). – 1973. – Т. 20. – №. 1. – С. 46-61.
7. Nikitin Y. Y., Abramovich S. On the Probability of Co-primality of two Natural Numbers Chosen at Random: From Euler identity to Haar Measure on the Ring of Adeles //Bernoulli News. – 2017. – Т. 24. – №. 1. – С. 7-13.
8. Nasri M., Fohler G. An efficient method for assigning harmonic periods to hard real-time tasks with period ranges //2015 27th Euromicro Conference on Real-Time Systems. – IEEE, 2015. – С. 149-159.
9. Garey M. R., Johnson D. S., Sethi R. The complexity of flowshop and jobshop scheduling //Mathematics of operations research. – 1976. – Т. 1. – №. 2. – С. 117-129.
10. Сухарев А. Г., Тимохов А. В., Федоров В. В. Курс методов оптимизации. – Физматлит, 2011.

11. Coffman, Jr E. G., Garey M. R., Johnson D. S. An application of bin-packing to multiprocessor scheduling //SIAM Journal on Computing. – 1978. – Т. 7. – №. 1. – С. 1-17.
12. Park M. Non-preemptive fixed priority scheduling of hard real-time periodic tasks //International Conference on Computational Science. – Springer, Berlin, Heidelberg, 2007. – С. 881-888.
13. Emberson P., Stafford R., Davis R. I. Techniques for the synthesis of multiprocessor tasksets //proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010). – 2010. – С. 6-11.
14. Griffin D., Bate I., Davis R. I. Generating utilization vectors for the systematic evaluation of schedulability tests //2020 IEEE Real-Time Systems Symposium (RTSS). – IEEE, 2020. – С. 76-88.
15. Bini E., Buttazzo G. C. Measuring the performance of schedulability tests //Real-Time Systems. – 2005. – Т. 30. – №. 1. – С. 129-154.

ПРИЛОЖЕНИЕ А

Графики для планировщиков EDF и RM

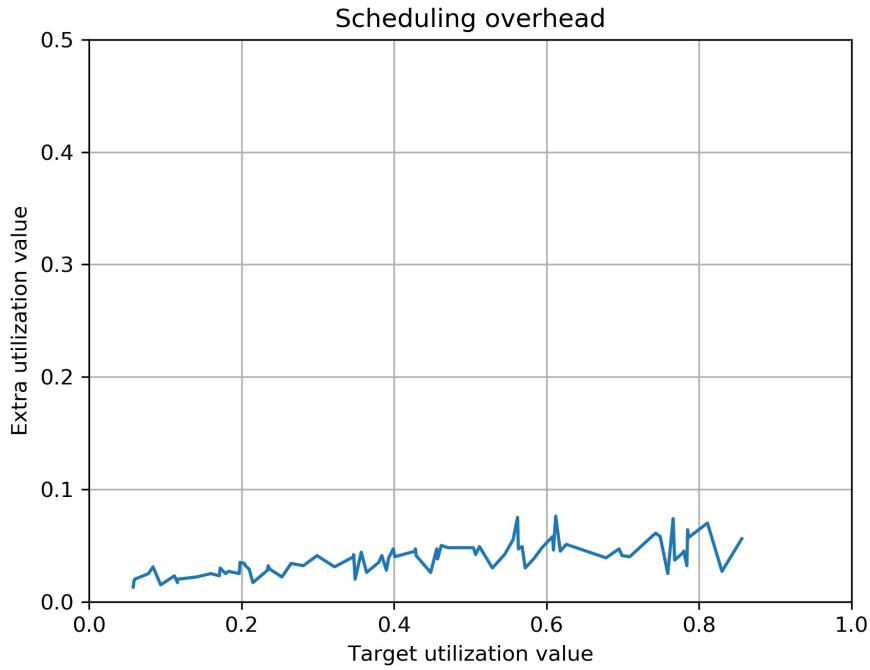

Рисунок А.1 — График величины излишнего планирования для EDF.

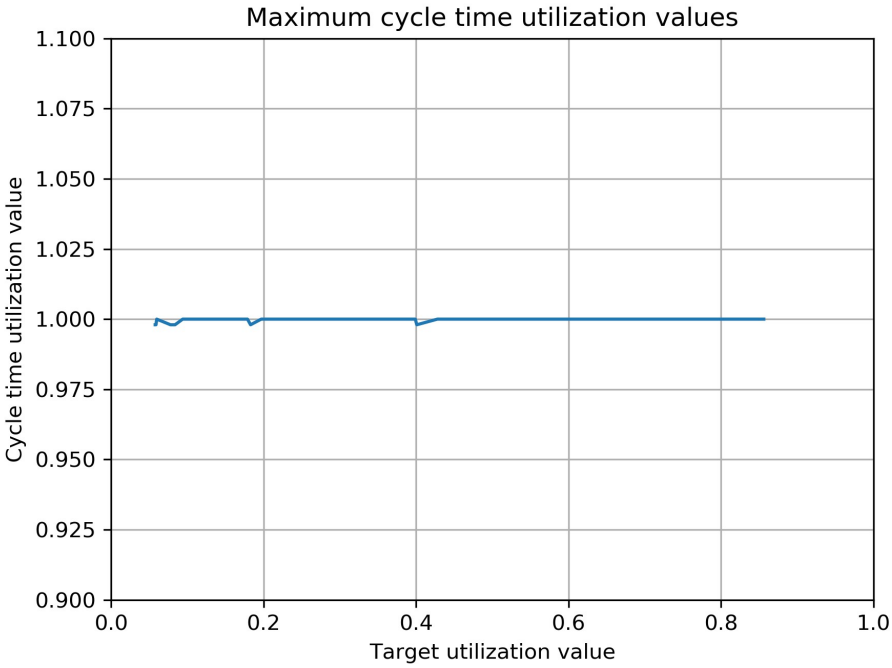

Рисунок А.2 — График максимальной доли утилизации ЦП за цикл для EDF.

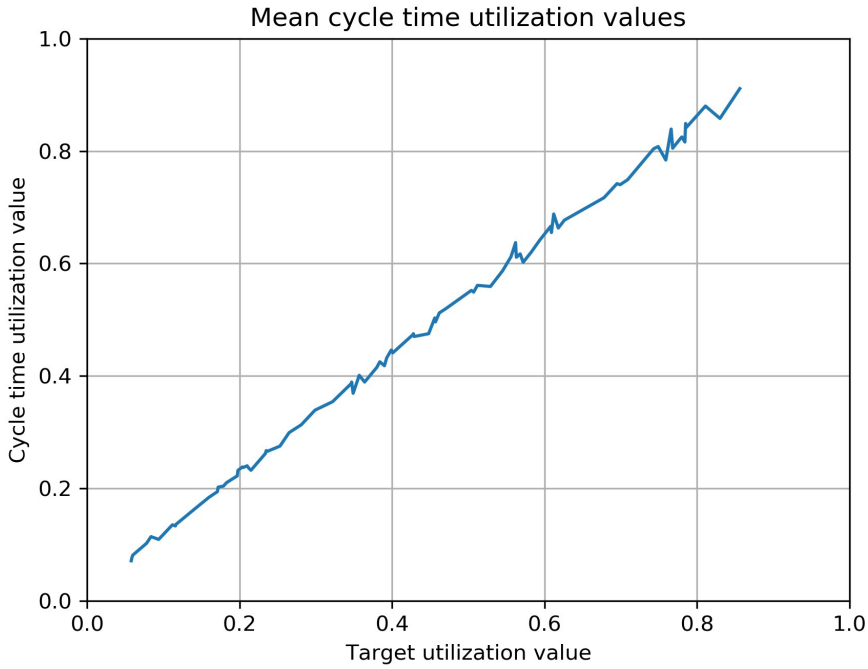

Рисунок А.3 — График средней доли утилизации ЦП за цикл для EDF.

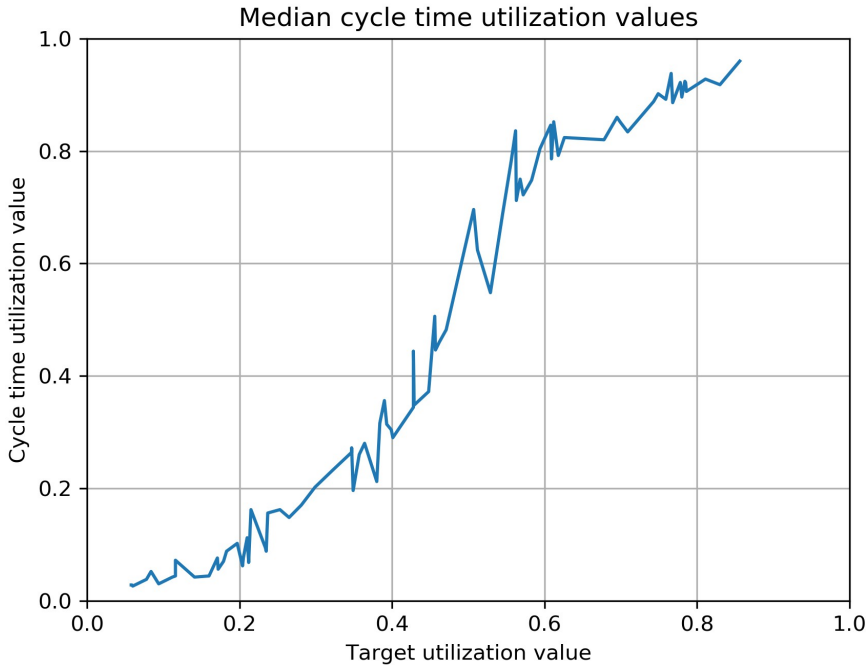

Рисунок А.4 — График медианной доли утилизации ЦП за цикл для EDF.

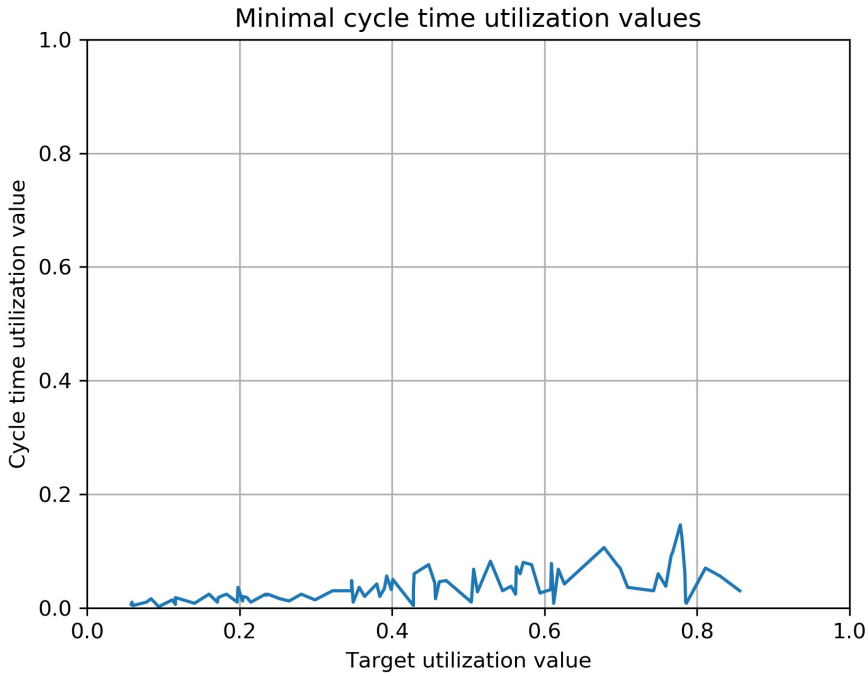

Рисунок А.5 — График минимальной доли утилизации ЦП за цикл для EDF.

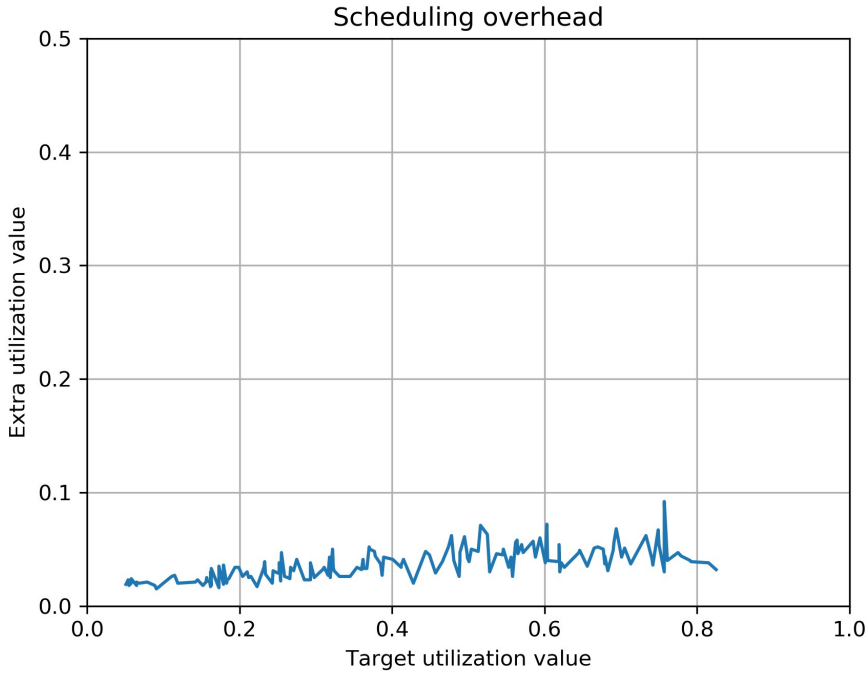

Рисунок А.6 — График величины излишнего планирования для RM.

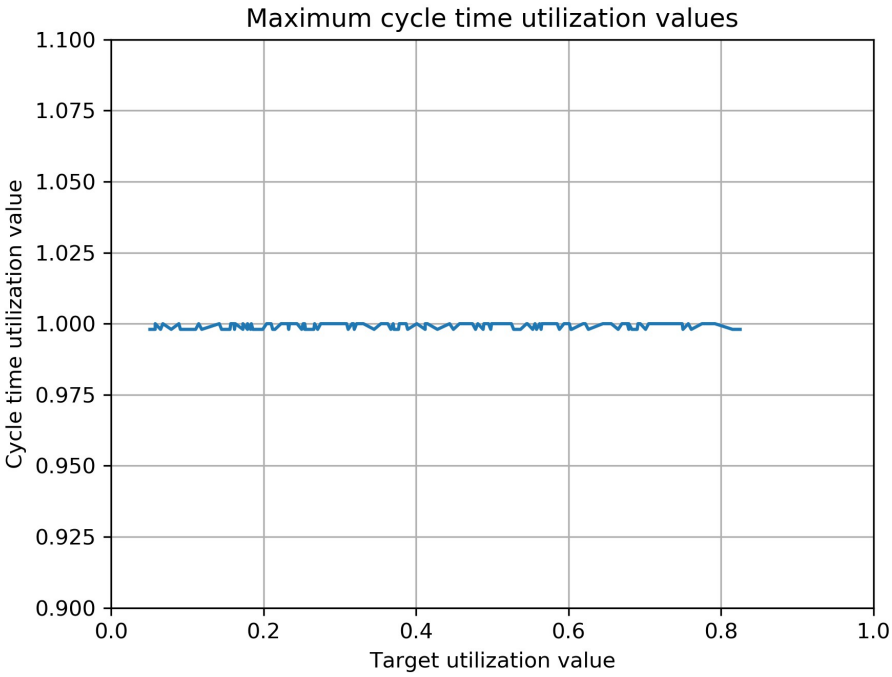

Рисунок А.7 — График максимальной доли утилизации ЦП за цикл для RM.

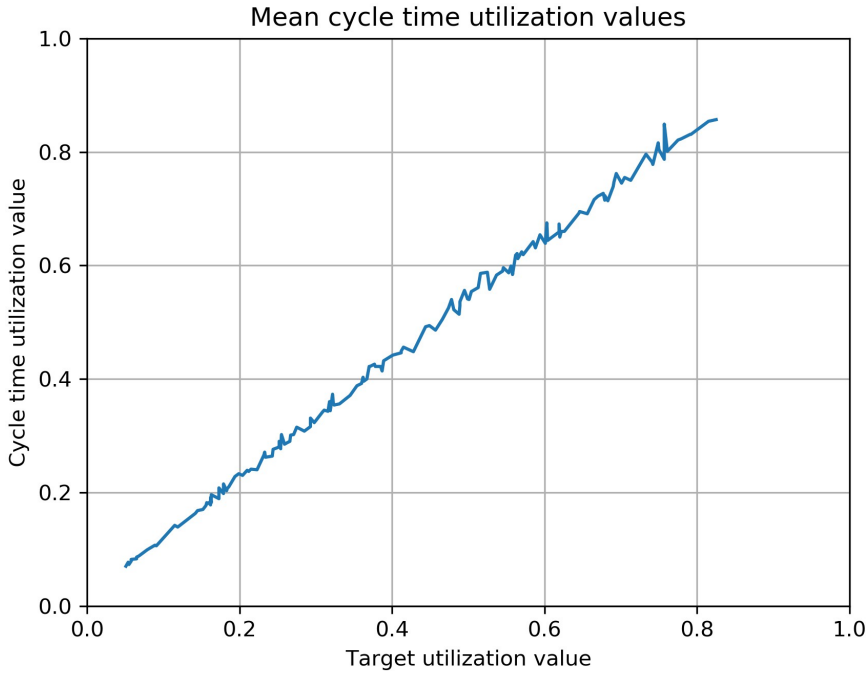

Рисунок А.8 — График средней доли утилизации ЦП за цикл для RM.

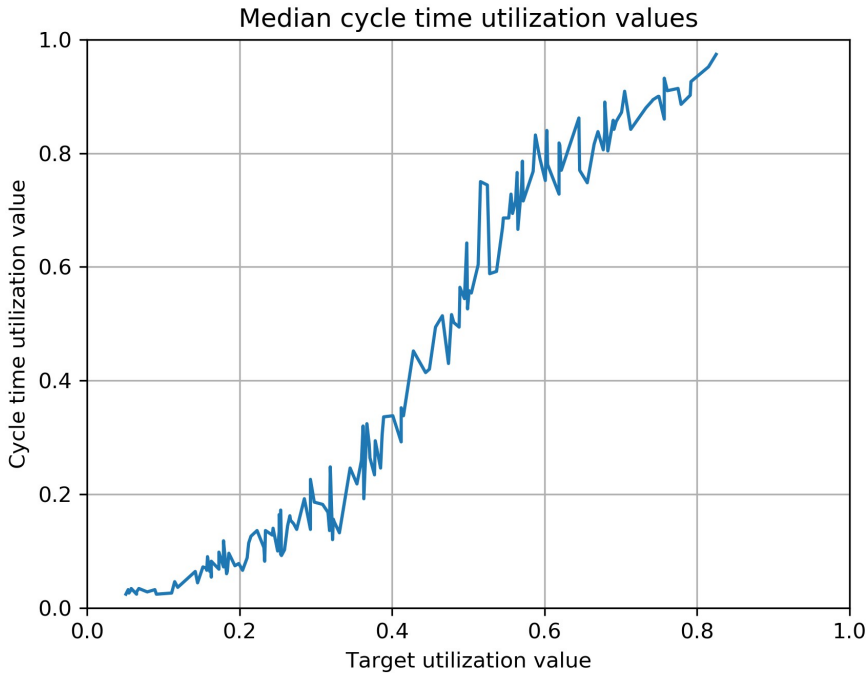

Рисунок А.9 — График медианной доли утилизации ЦП за цикл для RM.

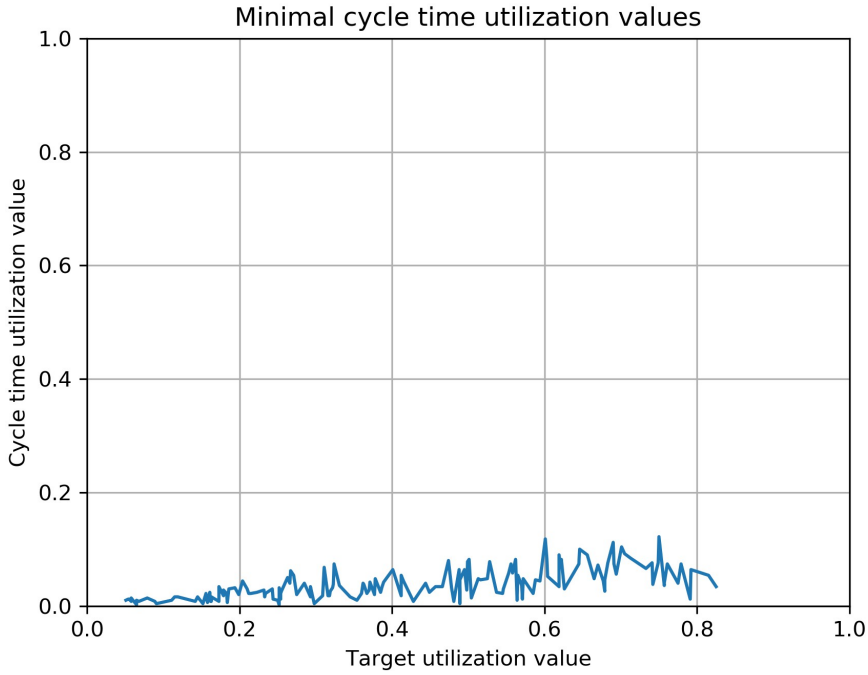

Рисунок А.10 — График минимальной доли утилизации ЦП за цикл для RM.

ПРИЛОЖЕНИЕ Б

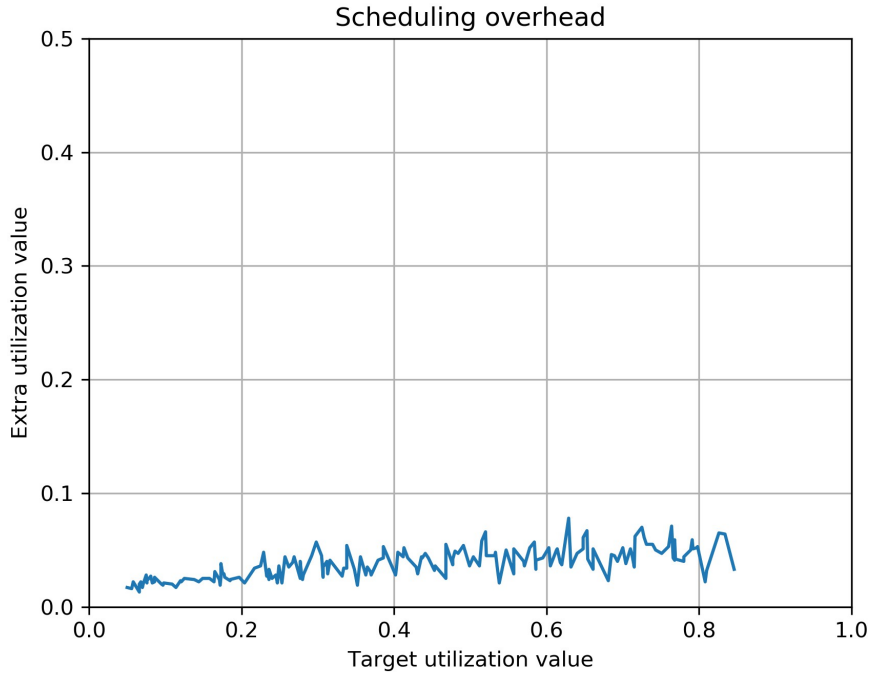

Рисунок Б.1 — График величины излишнего планирования для EDF.

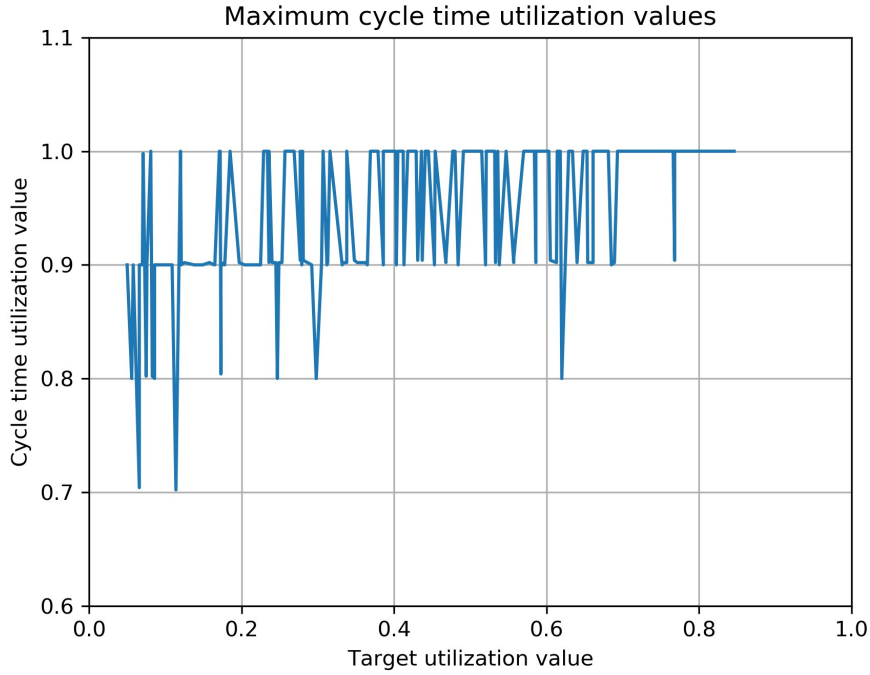

Рисунок Б.2 — График максимальной доли утилизации ЦП за цикл для EDF.

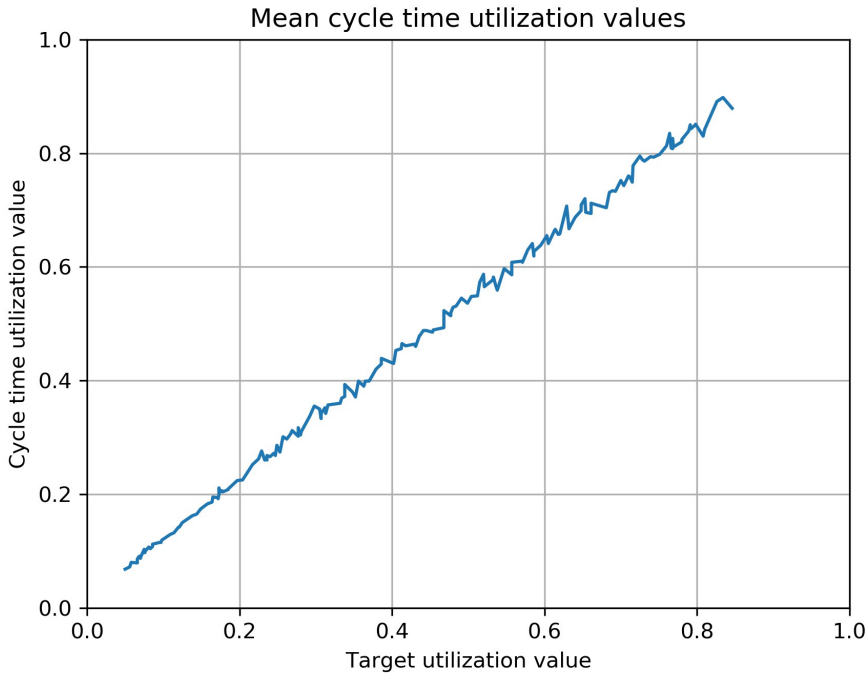

Рисунок Б.3 — График средней доли утилизации ЦП за цикл для EDF.

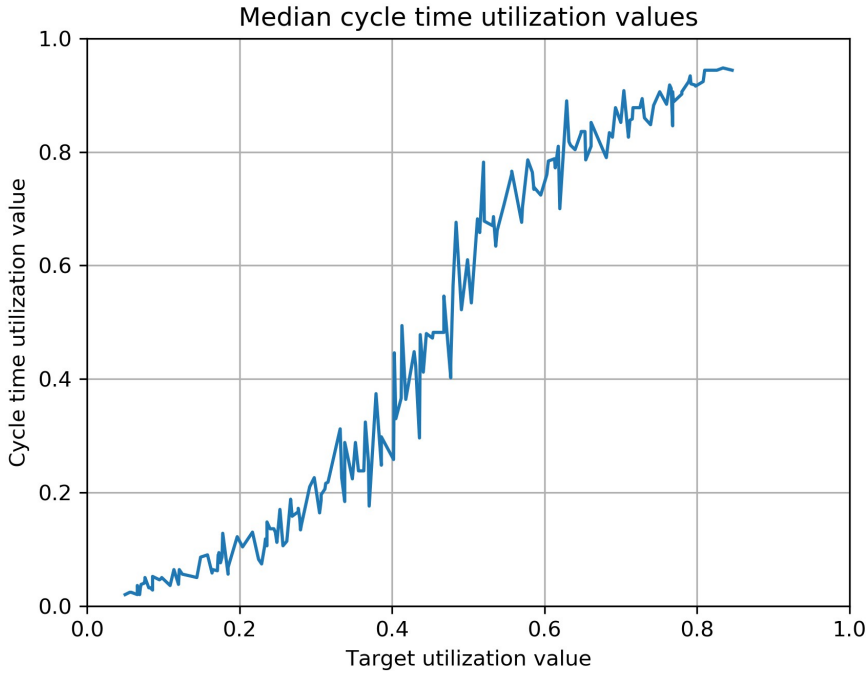

Рисунок Б.4 — График медианной доли утилизации ЦП за цикл для EDF.

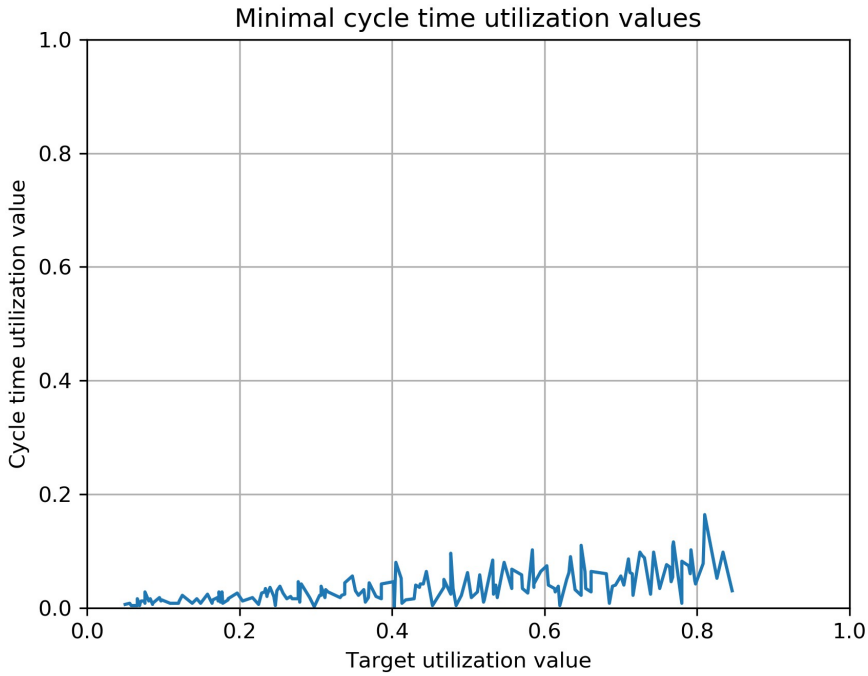

Рисунок Б.5 — График минимальной доли утилизации ЦП за цикл для EDF.

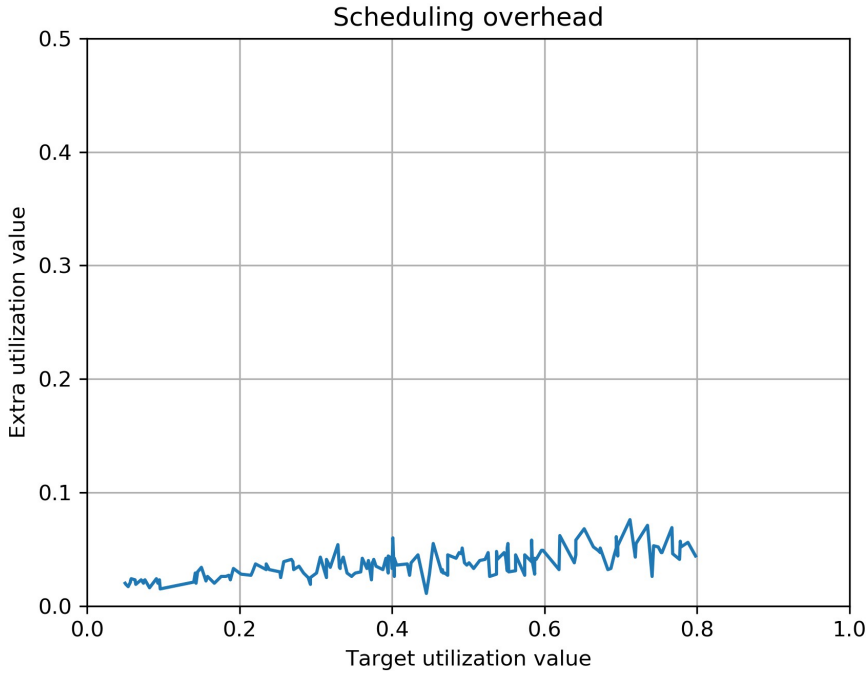

Рисунок Б.6 — График величины излишнего планирования для RM.

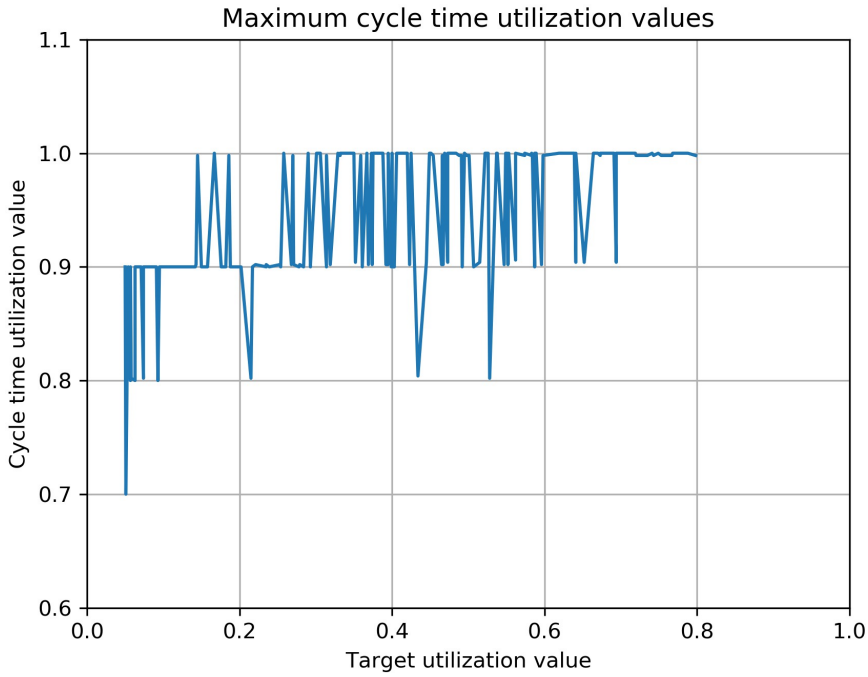

Рисунок Б.7 — График максимальной доли утилизации ЦП за цикл для RM.

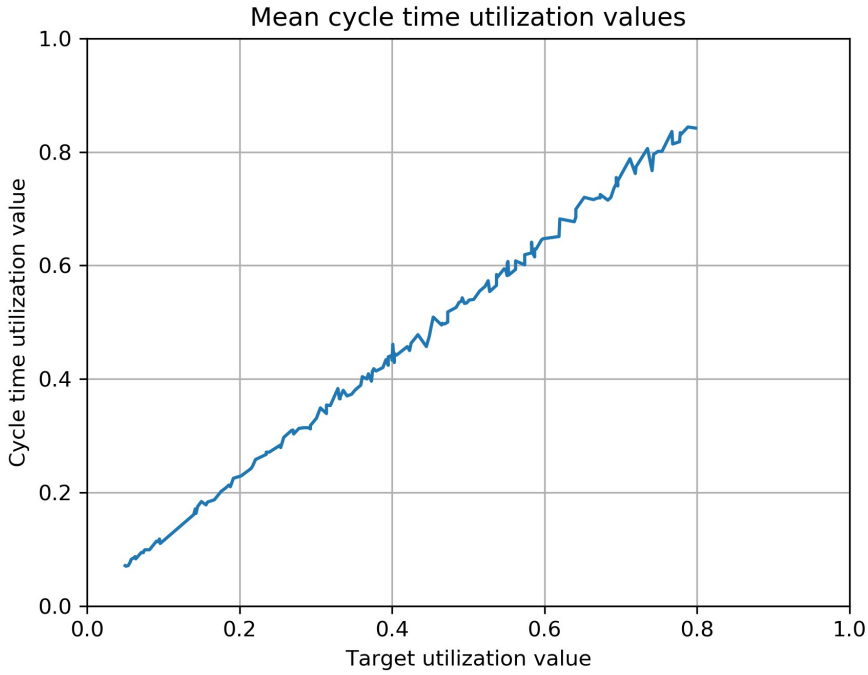

Рисунок Б.8 — График средней доли утилизации ЦП за цикл для RM.

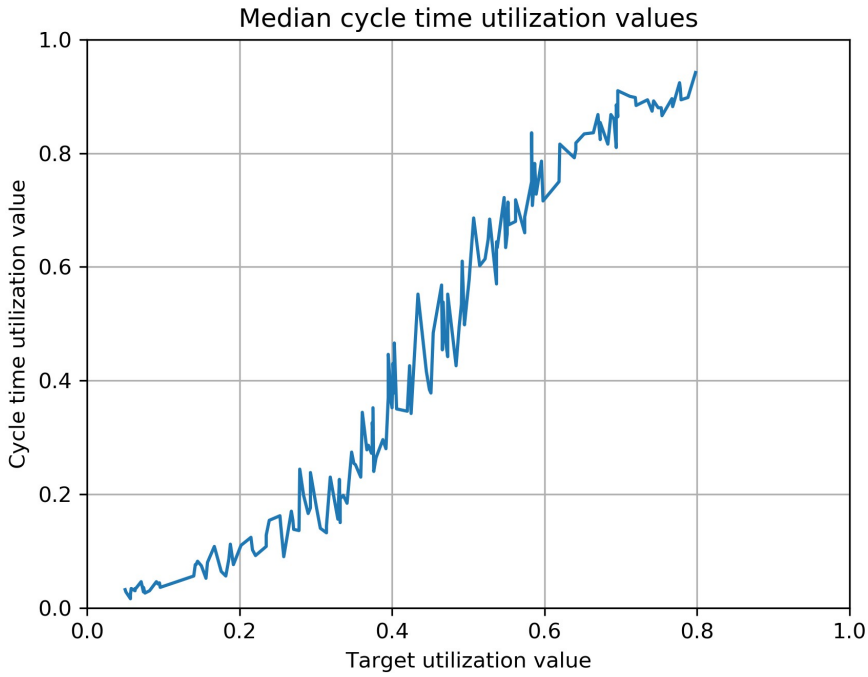

Рисунок Б.9 — График медианной доли утилизации ЦП за цикл для RM.

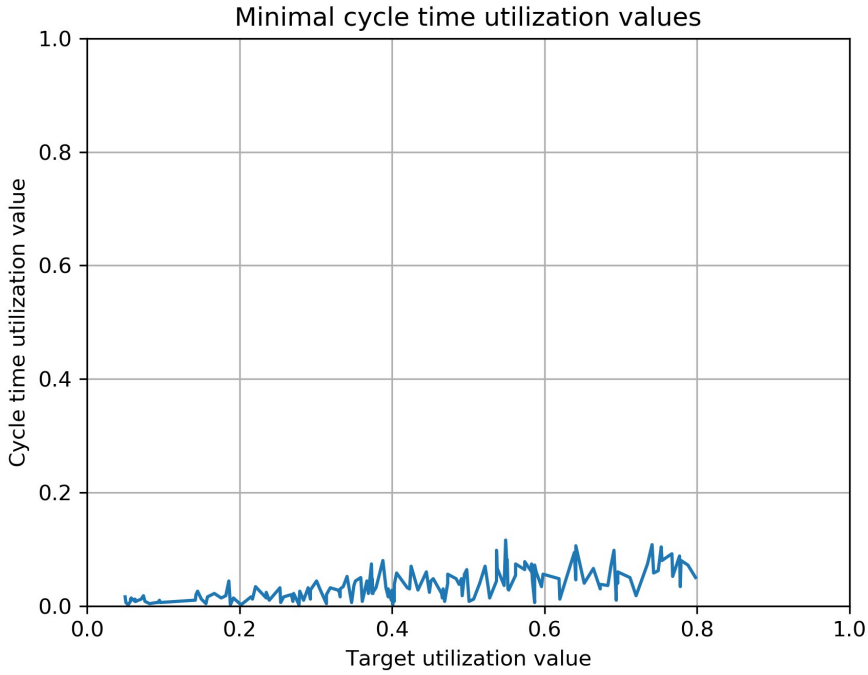

Рисунок Б.10 — График минимальной доли утилизации ЦП за цикл для RM.