МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова

На правах рукописи

ЗОРИН Даниил Александрович

СИНТЕЗ АРХИТЕКТУР ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ С УЧЕТОМ ОГРАНИЧЕНИЙ НА ВРЕМЯ ВЫПОЛНЕНИЯ И ТРЕБОВАНИЙ К НАДЕЖНОСТИ

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ

на соискание ученой степени кандидата физико-математических наук

Научный руководитель:
кандидат технических наук,
В.А. Костенко

МОСКВА – 2014

Оглавление

Введение ... 5

1 Постановка задачи ... 10

1.1 Содержательное описание задачи ... 10

1.2 Модель входных данных ... 12

1.3 Механизмы обеспечения надежности ... 12

1.4 Модель расписания ... 14

1.5 Вычисление времени выполнения расписания ... 17

1.6 Вычисление надежности ... 19

1.7 Математическая постановка задачи ... 20

1.8 Выводы ... 22

2 Обзор возможных подходов к построению алгоритмов решения задачи и алгоритмов решения близких задач ... 23

2.1 Цель обзора ... 23

2.2 Полный перебор ... 23

2.3 Метод ветвей и границ ... 24

2.4 Жадные алгоритмы ... 25

2.5 Алгоритм имитации отжига ... 27

2.6 Генетические алгоритмы ... 29

2.7 Выводы ... 31

3 Алгоритм построения расписания ... 33

3.1 Общая схема алгоритма ... 33

3.2 Операции преобразования расписания ... 34

3.3 Стратегии применения операций. ... 39

3.4 Условие перехода к новому расписанию и критерий останова. 43

3.5 Вычислительная сложность одной итерации алгоритма 44
3.6 Выводы 46
4 Исследование свойств алгоритма 47
4.1 Асимптотическая сходимость алгоритма 47
4.2 Метрика в пространстве расписаний 54
4.2.1 Метрика L(A, B) 54
4.2.2 Метрика H(A, B) 55
4.2.3 Связь между метриками L(A, B) и H(A, B) 59
4.2.4 Оценка для L(A, B) 62
4.3 Экспериментальное исследование алгоритма 62
4.3.1 Оценка точности на модельных данных 62
4.3.2 Оценка точности на совместимых исходных данных 72
4.3.3 Сравнение законов понижения температуры 75
4.4 Выводы 79
5 Инструментальная система 80
5.1 Требования к системе 80
5.2 Описание системы 80
5.2.1 Описание архитектуры системы 80
5.2.2 Графический пользовательский интерфейс 86
5.2.3 Подсистема для построения вычислительных систем для обработки данных от фазированных антенных решеток 88
5.3 Выводы 90
Заключение 92
Литература 93

Приложение А. Способы оценки надежности 104
А.1 Надежность процессоров. Резервирование 104
А.2 Надежность программ. N-версионное программирование 106
Приложение Б. Модели среды передачи данных 119
Б.1 Вычисление времени для системы без конфликтов на портах .. 119
Б.2 Вычисление времени для системы с общей шиной или коммутатором 120
Приложение В. Вычисление времени выполнения с помощью имитационного моделирования 122
Приложение Г. Описание задачи обработки данных от фазированных антенных решеток 129

Введение

В данной работе рассматривается задача построения вычислительных систем, использующих наименьшее возможное число аппаратных ресурсов для выполнения прикладной программы за время, не превышающее заданного, и удовлетворяющих требованиям к надежности. Предполагается, что программа задана графом потока данных, вершины которого далее называются «заданиями». Распределение заданий по процессорам с указанием очередности выполнения называется расписанием. Вычисление времени выполнения расписания в общем случае зависит от особенностей архитектуры вычислительной системы, для которой строится расписание. Однако рассматриваемое представление расписания не включает в себя значения времени запуска и выполнения отдельных заданий, тем самым оно позволяет абстрагироваться от особенностей аппаратуры и дает возможность использовать при вычислении времени выполнения расписания модели вычислительных систем с различным уровнем детализации [9][103]. Таким образом, результаты работы смогут быть применимыми для широкого класса вычислительных систем.

Вычислительная система работает в жестком реальном времени, то есть программа должна завершиться до наступления директивного срока. Также к системе предъявляются требования по надежности. Под надежностью вычислительной системы понимается вероятность того, что она будет работать по спецификации в течение заданного времени. Вычислительные системы с высокой надежностью применяются в тех областях, где безотказная работа особенно важна – авиакосмической промышленности, ядерной энергетике, медицинской промышленности. Повышать надежность системы можно с помощью введения резервных элементов для аппаратных средств и копий программных модулей. В данной работе для повышения надежности вычислительной системы используются горячее резервирова-

ние процессоров и многоверсионное программирование. Необходимо построить расписание, использующее наименьшее возможное число процессоров, при соблюдении заданных ограничений на время выполнения расписания и требований к надежности вычислительной системы.

К бортовым вычислительным система реального времени на стадии проектирования кроме жестких ограничений на время выполнения программ и требований надежности также предъявляются повышенные требования к массогабаритным и мощностным характеристикам, поэтому актуально построение системы с минимальным необходимым числом процессоров.

Актуальность разработки алгоритмов структурного синтеза подтверждается особенностями рассматриваемой задачи, отличающими ее от других задач построения расписаний, а именно:

- Наличие одновременно двух ограничений (время выполнения программы и надежность вычислительной системы);
- Необходимость обеспечить применимость алгоритма для построения вычислительных систем с различной архитектурой;
- Необходима возможность использовать для вычисления времени выполнения расписания имитационные модели различного уровня детализации.

Целью данной работы является разработка и исследование свойств алгоритма структурного синтеза вычислительных систем реального времени с учетом ограничений на время выполнения программ и требований к надежности вычислительной системы.

Для достижения поставленной цели необходимо решить следующие задачи:

- Сформулировать задачу структурного синтеза, в том числе предложить математическую модель вычислительных систем реального

времени и расписаний, позволяющую использовать модели программных и аппаратных средств с различной степенью детализации;

- Разработать алгоритм структурного синтеза, позволяющий построить вычислительную систему с минимальным числом процессоров, учитывая ограничения на время выполнения программы и требования к надежности системы;
- На основе предложенных алгоритмов реализовать программное инструментальное средство структурного синтеза, позволяющее поддерживать процесс проектирования вычислительных систем реального времени.

При получении основных результатов диссертации использовались методы математического программирования, теории графов, теории случайных процессов, а также математической статистики.

Работа состоит из введения, пяти глав, заключения и четырех приложений.

В первой главе описана организация работы вычислительной системы реального времени. Введены математическая модель рассматриваемой системы и исходных данных задачи построения расписаний, определение расписания и условия его корректности. Предложена математическая постановка задачи выбора минимального необходимого числа процессоров и доказана ее NP-трудность.

Во второй главе проведен обзор известных методов решения задач построения расписаний. Проведен анализ возможных подходов к построению алгоритмов решения поставленной задачи и обоснован выбор метода имитации отжига.

В третьей главе представлен разработанный алгоритм структурного синтеза, основанный на выбранных в предыдущей главе методах и подхо-

дах. Доказаны его завершимость и корректность; дана оценка вычислительной сложности.

В четвертой главе приведено доказательство асимптотической сходимости алгоритма, описанного в третьей главе. Также описана методика экспериментального исследования разработанных алгоритмов структурного синтеза, приведены численные результаты исследования и их анализ. Предметом исследования являлись точность и скорость работы алгоритма на различных классах исходных данных. Исходные данные для экспериментов использовались как модельные исходные данные, создаваемые случайным образом по шаблонам, так и данные на основе реальных задач структурного синтеза вычислительных систем реального времени. При обработке результатов экспериментов применены методы математической статистики.

В пятой главе представлено инструментальное программное средство структурного синтеза. Описано применение инструментальной системы для решения практической задачи структурного синтеза вычислительных систем для обработки данных от фазированных антенных решеток.

В заключении сформулированы основные результаты диссертационной работы.

В Приложении А описаны различные схемы оценки надежности вычислительных систем.

В Приложении Б приведено детальное описание моделей среды передачи данных, которые могут быть использованы в сочетании с предложенными алгоритмами: полносвязной модели, модели с общей шиной, модели коммутируемой сети Fibre Channel.

В Приложении В описан механизм интеграции разработанных алгоритмов и инструментальных средств со средой моделирования, позволяющей получать оценки времени выполнения расписания с помощью имитационного моделирования.

В Приложении Г приведено подробное описание задачи определения координат источника сигнала с помощью фазированных антенных решеток. Для построения вычислительной системы для решения данной задачи применялись созданные в рамках данной работы алгоритм и инструментальное средство.

1 Постановка задачи

1.1 Содержательное описание задачи

Задача структурного синтеза в самом общем виде состоит в поиске оптимальной конфигурации вычислительных систем реального времени. В данной работе рассматривается класс многопроцессорных вычислительных систем реального времени [89]. Такая система состоит из *вычислителей* (процессоров) и *среды передачи данных,* объединяющей эти вычислители. Также в системе могут присутствовать устройства для ввода или вывода данных, но в дальнейшем рассматривается задача выбора минимально необходимого числа процессоров.

В данной работе рассматриваются только однородные системы [33][87][88], то есть системы, в которых процессоры одинаковые по производительности и надежности. Программа, выполняемая в системе, представляется ориентированным ациклическим графом потока данных [41][42][78]. Программа состоит из конечного множества взаимодействующих заданий, каждое из которых представляет собой подпрограмму, принимающую и передающую данные от других подпрограмм. Для каждого из заданий известно, какие вычисления оно производит и каков объем результата, получаемого на выходе и передаваемого другим заданиям. Таким образом, вершины графа соответствуют заданиям, а ребра – зависимостям по данным между заданиями. Формально модель системы введена в подразделе 1.2.

Для систем реального времени важно выполнение двух ограничений. Во-первых, программа должна полностью выполниться до наступления *директивного срока*. Во-вторых, система должна удовлетворять требованиям к *надежности,* подробно рассмотренным в подразделе 1.3. В работе вводятся понятия *функции интерпретации*, представляющей собой математическую абстракцию способа вычисления времени выполнения про-

граммы и *функции оценки надежности*. Эти функции зависят от характеристик заданий и архитектуры системы в решаемой задаче структурного синтеза и входит в число исходных данных.

Задача синтеза, рассматриваемая в данной работе, предполагает определение минимально необходимого *числа процессоров* и построение *расписания* выполнения заданий на этих процессорах. Само расписание, введенное в подразделе 1.4, определяет лишь *очередность* выполнения заданий на каждом процессоре (привязку заданий к процессорам и порядок выполнения заданий на каждом процессоре), а конкретное время выполнения каждого задания и, соответственно, время окончания работы всей программы зависят от используемой модели вычислений и среды передачи данных (подраздел 1.5).

Резюмируя, рассматриваемая далее задача структурного синтеза вычислительной системы реального времени имеет следующие входные данные:

- Граф потока данных выполняемой программы,
- Способ вычисления времени и надежности (соответствующие функции интерпретации и оценки надежности),
- Ограничения на время выполнения программы,
- Требования к надежности системы.

Требуется определить минимально необходимое число процессоров и построить расписание выполнения заданий на этих процессорах, удовлетворяющее ограничениям на время выполнения и требованиям к надежности. Математическая формулировка данной задачи будет дана в подразделе 1.8.

1.2 Модель входных данных

Формально модель системы состоит из следующих объектов.

- M – множество процессоров. Рассматривается однородная система, то есть все процессоры имеют одинаковые технические характеристики, такие как производительность и надежность;
- $G = \{V, E\}$ – граф потока данных программы, ориентированный граф без циклов;
- V – множество вершин, соответствующих заданиям.;
- E – множество ребер. В графе есть дуга (v_1, v_2), если задание v_2 принимает на вход результаты работы задания v_1;
- $H: V \to \mathbb{R}^n$ – функция, задающая характеристики заданий. По этим характеристикам возможно оценить время выполнения задания;
- $F: E \to \mathbb{R}$ – функция, задающая объем передаваемых данных для каждой взаимодействующей пары заданий. Зная объем, можно в рамках используемой модели среды передачи данных однозначно определить, сколько времени требуется на передачу;
- $ft(S), fR(S, v)$ – функция интерпретации и функция оценки надежности задания, позволяющие рассчитать соответственно время выполнения программы и надежность системы. Формальные определения будут введены в подразделах 1.5 и 1.6.

1.3 Механизмы обеспечения надежности

Надежность есть свойство системы работать без отказов на некотором промежутке времени [20]. Подробно вопросы формализации понятия надежности рассмотрены в приложении А; в данном разделе считается, что время работы системы ограничено, и надежность от времени не зависит. Поэтому в дальнейшем будет использоваться следующее определение.

Определение 1.1. Надежность – это вероятность (число от 0 до 1) того, что вычислительная система проработает безотказно.

В данной работе будут использоваться два метода обеспечения требуемой надежности: резервирование процессоров и многоверсионное программирование.

Резервирование процессоров заключается в том, что к некоторому процессору в системе добавляется дублирующий процессор, на котором выполняются те же задания. В этом случае система отказывает, только если отказывают оба процессора. Дублирующий процессор работает в режиме горячего резерва, то есть принимает все те же данные и выполняет те же вычисления, что и основной, но передает данные только в случае отказа основного.

Утверждение 1.1. Пусть на процессоре m_i выполняются задания $v_1, \ldots, v_n$. Тогда добавление резервного процессора m_r и дублирование на нем всех заданий $v_1, \ldots, v_n$ обеспечивает надежность системы не меньшую, чем добавление резервных процессоров $m_r, \ldots, m_{r+k}$ и дублирование на каждом из них некоторого подмножества заданий $v_1, \ldots, v_n$.

Доказательство. Если на некотором процессоре m выполняется задание v_i, которое не продублировано ни на каком другом процессоре, то отказ m будет приводить к отказу всей системы, таким образом, надежность не повысится. Поэтому пусть задания $v_1, \ldots, v_n$ с процессора m продублированы на процессорах $m_1, \ldots, m_n$ соответственно. На один процессор может быть назначено более одного дублирующего задания. Отказ m тогда уже не приведет к отказу всей системы, но только при условии, что среди $m_1, \ldots, m_n$ все процессоры не откажут. Пусть надежность процессора m равна p, надежность процессора m_i равна p_i. Надежность системы для заданий $v_1, \ldots, v_n$ (каждое из заданий выполнится хотя бы на одном процессоре) равна $p + (1-p)\prod_{i=1}^{n} p_i$. Поскольку все вероятности меньше 1, с добавлением новых множителей в произведение общая надежность становится меньше. Значит, для любого набора дублирующих процессоров его

надежность будет не больше, чем надежность системы, где все дублирующие задания были назначены на один процессор из этого набора.

При *N-версионном программировании* (NVP) создается N версий реализации какого-либо задания [49]. Число версий всегда нечетно (обычно 3 либо 5), и результаты подвергаются простому сравнению, итоговым результатом объявляется тот, который выдали больше половины версий. Таким образом, при отказе не более чем $(N+1)/2$ версий отказа не происходит. Разные версии обычно разрабатываются разными группами программистов для того, чтобы по возможности неисправности были различны в разных версиях. Предполагается, что если неисправности различны, то версии отказывают на разных входных данных.

Для определения надежности системы должны быть заданы следующие исходные данные.

- $P(m_i)$ – надежность процессора m_i;
- $Vers(v_i)$ – множество доступных версий задания v_i;
- $P(k): Vers(v_i) \to \mathbb{R}^n$ – функция, задающая количественные характеристики надежности каждой версии.

Формулы для вычисления надежности системы приведены в разделе 1.6.

1.4 Модель расписания

Будем считать, что для программы задано расписание, если для каждого из заданий определена привязка – однозначно определено, на каком процессоре оно выполняется, и задан порядок – для каждого процессора известно, в какой очередности выполняются задания.

Если используется многоверсионное программирование, то помимо указания задания, необходимо указывать номер его версии, то есть привязка и порядок задаются не для задания, а для пары «задание-версия».

Определение 1.2. Расписание (для системы с резервированием и многоверсионным программированием) определяется как пара (S, D), где S – множество четверок (v, k, m, n), где $v \in V, k \in Vers(v), m \in M, n \in \mathbb{N}$, такое, что

$$\forall v \in V \; \exists k \in Vers(v) \colon \exists s = (v_i, k_i, m_i, n_i) \in S \colon v_i = v, k_i = k;$$

$$\forall s_i = (v_i, k_i, m_i, n_i) \in S, \forall s_j = (v_j, k_j, m_j, n_j) \in S \colon (v_i = v_j \wedge k_i = k_j) \Rightarrow s_i = s_j;$$

$$\forall s_i = (v_i, k_i, m_i, n_i) \in S, \forall s_j = (v_j, k_j, m_j, n_j) \in S \colon (s_i \neq s_j \wedge m_i = m_j) \Rightarrow n_i \neq n_j.$$

D – мультимножество, состоящее из элементов множества процессоров M. Общее число процессоров в системе $M(S) = |D|$. $Dup(m_i)$ – кратность процессора m_i в мультимножестве D.

Содержательно m и n задают соответственно привязку к процессору и порядок выполнения для каждой версии каждого задания. Ограничения, записанные выше логическими условиями, означают, что 1) у каждого задания хотя бы одна версия должна присутствовать в расписании, 2) каждая версия каждого задания может присутствовать в расписании только один раз, 3) у всех заданий, назначенных на один процессор, разные номера. Мультимножество D задает резервируемые процессоры.

Расписание можно представить в виде графа. Вершинами этого графа являются элементы множества S. Если между соответствующими заданиями есть дуга в графе G, то она добавляется в граф расписания, также в граф расписания добавляются дуги между вершинами, назначенными на один процессор и имеющими соседние номера.

Определение 1.3. Элемент расписания s_2 будем называть зависимым от s_1, если либо $(v_1, v_2) \in E$, либо $m_1 = m_2 \wedge n_1 < n_2$. Иначе говоря, s_2 зависит от s_1, если s_2 не может начать выполняться раньше s_1 из-за зависимости по данным или из-за порядка выполнения на процессоре.

Из определения следует, что каждая версия каждого задания может присутствовать в расписании в единственном экземпляре, а также на каждом процессоре у всех заданий различные номера и как минимум одна версия каждого задания должна быть включена в расписание. Помимо этих ограничений, необходимо ввести еще одно, чтобы гарантировать завершимость программы.

Определение 1.4. Расписание S является корректным, если его граф является ациклическим.

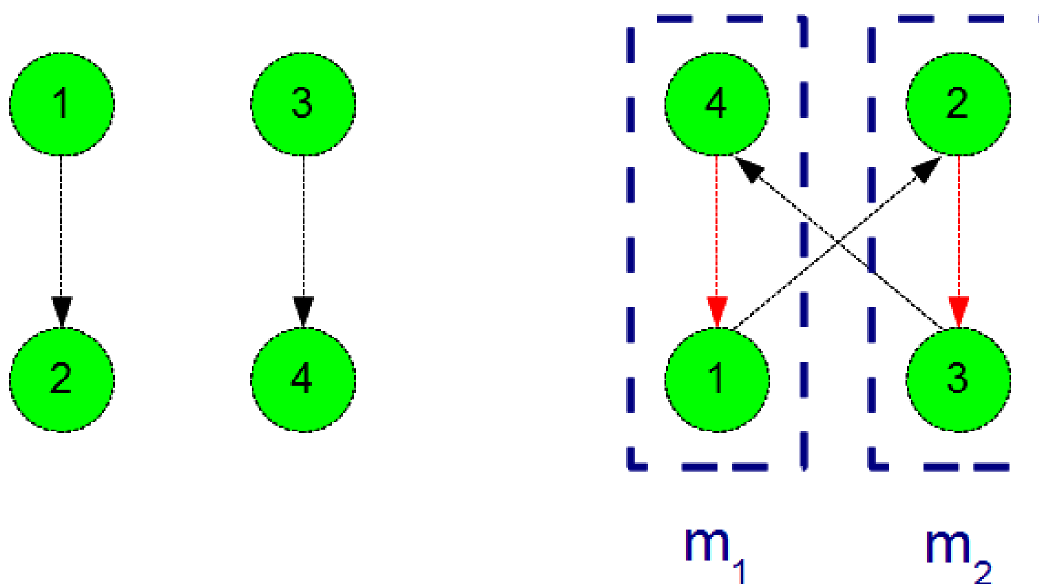


Рисунок 1.1. Пример некорректного расписания.

В дальнейшем будем рассматривать только корректные расписания. Множество всех корректных расписаний для заданного графа программы будем обозначать $\overline{S}$.

Пример некорректного расписания приведен на рисунке 1.1. Задание 1 не зависит на графе программы от задания 4, а задание 3 – от задания 2, но

из-за неправильного порядка на процессорах четыре задания зависят друг от друга циклически.

1.5 Вычисление времени выполнения расписания

Пусть заданы программа G и расписание (S, D) для нее.

Определение 1.5. Будем говорить, что для корректного расписания S определена временная диаграмма, если определены две функции:

$$T_v: S \rightarrow \mathbb{R} \times \mathbb{R},$$

$$T_e: E \rightarrow \mathbb{R} \times \mathbb{R},$$

удовлетворяющие следующим условиям:

$\forall s \in S; (a, b) = T_v(s): a < b;$

$\forall e \in E; (a, b) = T_e(e): a \leq b;$

$\forall s_1, s_2 \in S; (a_1, b_1) = T_v(s_1);\ (a_2, b_2) = T_v(s_2):\ m_1 = m_2 \wedge n_1 < n_2 \Rightarrow b_1 \leq a_2;$

$\forall s_i = (v_i, k_i, m_i, n_i) \in S; e = \left(v_j, v_i\right) \in E; (a_1, b_1) = T_v(s_i);\ (a_2, b_2) = T_e(e):\ b_2 \leq a_1.$

Поясним смысл используемых обозначений. Функция T_v для каждого задания определяет момент начала (a) и окончания (b) его выполнения. Функция T_e для каждой связи между заданиями определяет время начала и окончания соответствующей передачи данных. Первые два требования определения тривиальны и означают, что время окончания больше времени начала. Для функции T_e также возможно равенство, если передача данных не требуется (оба задания на одном процессоре). Третье условие означает, что если два задания назначены на один процессор, то порядок их выполнения должен соответствовать расписанию: время окончания первого меньше или равно времени начала второго. Последнее условие требует сохранения во временной диаграмме частичного порядка, задаваемого

графом программы: если задание ждет передачи данных, то оно может начать выполняться не раньше окончания передачи данных.

Функции T_v и T_e неявно зависят от функций H и F, введенных в разделе 1.2. Эта связь задается функцией интерпретации.

Определение 1.6. Функцией интерпретации времени называется вычислимая функция, по произвольному корректному расписанию однозначно строящая временную диаграмму:

$$ft(S): S \rightarrow (T_v, T_e)$$

Определение 1.7. Время выполнения *t(S)* – это время завершения последнего задания во временной диаграмме:

$$t(S) = \max_{s \in S}(T_v(s))_2$$

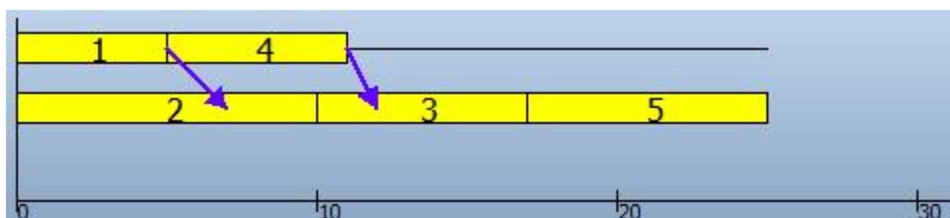


Рисунок 1.2. Визуальное представление функции интерпретации.

На рисунке 1.2 приведено наглядное представление функции интерпретации, которое будет использоваться на иллюстрациях в дальнейшем. Горизонтальные линии соответствуют процессорам, по оси ОХ обозначено время. Прямоугольники соответствуют заданиям, расположение их по оси ОХ зависит от значений функции T_v. Стрелки обозначают передачи данных, координаты их начала и конца по оси ОХ соответствуют значениям функции T_e.

Вид функции интерпретации зависит от используемой физической среды передачи данных. Во многих практических задачах функция интерпретации может не задаваться аналитически, а вычисляться по определенному алгоритму или с помощью имитационной модели. В данной работе реали-

зованы несколько функций интерпретации, соответствующих различным архитектурам вычислительных систем:

- Полносвязная модель [23], когда любая передача данных может быть начата в любой момент времени, т.е. не существует конфликтов в среде передачи данных (приложение Б.1);
- Модель шины [26], когда в любой момент времени во всей системе может быть только одна передача данных (приложение Б.2);
- Модель коммутируемой сети Fibre Channel [21], когда любой процессор может одновременно участвовать только в одной передачу данных (приложение Б.2);
- Интерпретация с помощью имитационного моделирования [17] (приложение В).

1.6 Вычисление надежности

Определение 1.8. Функция оценки надежности задания – функция, определяющая надежность задания с учетом всех его версий:

$$fR(S, v): S, v \ \to \mathbb{R}$$

Функция оценки надежности учитывает, какие из $Vers(v)$ версий задания используются в расписании S, и их характеристики $P(k)$.

Определение 1.9. Надежность системы $R(S)$ – это следующая величина [100]:

$$R(S) = \prod_{m_i \in M} (1 - (1 - P(m_i))^{Dup(m_i)}) \cdot \prod_{(v_i, k_i, m_i, n_i) \in S} fR(S, v_i).$$

Первый множитель соответствует совокупной надежности процессоров. Если $P(m_i)$ – вероятность безотказной работы, то $1 - P(m_i)$ – вероятность отказа. Вероятность отказа всех $Dup(m_i)$ дублирующих процессоров есть $(1 - P(m_i))^{Dup(m_i)}$. Соответственно, надежность группы дублирующих друг друга процессоров равна $1 - (1 - P(m_i))^{Dup(m_i)}$. Второй множитель соответствует совокупной надежности всех заданий с учетом использую-

щихся версий. Более подробно формулы вычисления надежности и виды функций $fR(S,v)$ рассмотрены в приложении А.

1.7 Математическая постановка задачи

Пусть заданы программа G, t^{dir} – срок, к которому программа должна быть выполнена, и R^{dir} – надежность, которой должна обладать система.

Пусть также фиксированы функция интерпретации $ft(S)$ и функция оценки надежности задания $fR(S,v)$.

Необходимо построить расписание S, для которого требуется минимальное количество процессоров, но при этом выполняются ограничения на время выполнения программы и требования к надежности системы:

$$\begin{gathered} \min_{S\in\overline{S}} M(S) \\ t(S) \leq t^{dir} \\ R(S) \geq R^{dir} \end{gathered} \qquad (1)$$

Требования к алгоритму решения данной задачи следующие:

- Алгоритм должен позволять использовать различные функции для вычисления времени выполнения расписания и надежности системы, таким образом, алгоритмы, использующие формы представления расписания со временем, заведомо не применимы;
- Требуется введение операций для построения решений, связанных с необходимостью обеспечения требуемой надежности;
- Необходимость на каждом шаге строить корректные расписания для возможности вычисления времени выполнения расписания и надежности.

Предложенные в литературе подходы к решению задач построения расписаний и решению вышеуказанных проблем будут рассмотрены в следующей главе.

Утверждение 1.2. Задача (1) является NP-трудной.

Доказательство. Сформулируем задачу определения свойства, соответствующую задаче (1):

Существует ли расписание S, удовлетворяющее ограничениям на $t(S)$ и $R(S)$, такое что $M(S) = m_0$?

Сводимость по Тьюрингу задачи (1) к этой задаче очевидна, поэтому достаточно доказать NP-полноту полученной задачи определения свойства.

Во-первых, необходимо доказать, что задача о построении расписания входит в класс NP, но это действительно так, так как проверка соблюдения ограничений на заданном расписании возможна за полиномиальное время. Во-вторых, нужно доказать, что любая задача из класса NP сводится к нашей задаче [13].

Рассмотрим задачу о разбиении: даны числа $a_1, \dots, a_n$, требуется ответить на вопрос, существует ли разбиение этих n чисел на две группы, так что сумма чисел в каждой из групп одинакова? Задача о разбиении является NP-полной, так как сводится к задаче выполнимости [13].

Сведем задачу о разбиении к нашей задаче. Пусть есть задача о разбиении для некоторых чисел $a_1, \dots, a_n$. Обозначим через B число $\sum_{i=1}^{n} a_i$. Пусть $R^{dir} = 0, t^{dir} = B/2$. Граф G будет состоять из n вершин, $E = \emptyset$, то есть зависимостей нет, задания можно располагать на процессорах в любом порядке. Функция $ft(S)$ задает фиксированное время выполнения для каждого задания: для задания v_i время равно a_i. Так как $E = \emptyset$, достаточно задать функцию T_v. Пусть задание s_0 следует на некотором процессоре за заданиями $s_1, \dots, s_n$. Тогда $T_v(s_0) = (\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} a_i + a_0)$. Получили некоторую индивидуальную задачу построения расписания.

Пусть задача о разбиении имеет решение в виде двух множеств X и Y. Тогда можно назначить вершины графа, соответствующие числам из X, на первый процессор, а вершины, соответствующие числам из Y, – на второй. Получится расписание, для которого время выполнения равно ровно $B/2$,

ограничение на надежность соблюдается, и используется два процессора, то есть задача построения расписания имеет решение.

Аналогично, если задача о разбиении не имеет решения, значит при любом разбиении $a_1, \ldots, a_n$ на два множества сумма элементов одного из них будет больше $B/2$, а значит, ограничение на время в соответствующем расписании соблюсти невозможно.

Поскольку при сведении рассчитывалась только сумма n чисел, сведение имеет полиномиальную сложность.

Таким образом, задача о построении расписания входит в класс NP и NP-полная задача о разбиении сводится к ней, а значит, задача о построении расписания входит в класс NPC, что и требовалось доказать.

1.8 Выводы

В данной главе сформулирована математическая постановка задачи структурного синтеза вычислительных систем реального времени с учетом ограничений на время выполнения и требований к надежности. Введены математические модели исходных данных, механизмов обеспечения надежности, расписания и ограничений корректности, накладываемых на него. Рассматриваемая задача сформулирована как задача условной оптимизации, и доказана ее NP-трудность, сформулированы требования к алгоритму ее решения.

2 Обзор возможных подходов к построению алгоритмов решения задачи и алгоритмов решения близких задач

2.1 *Цель обзора*

В данном разделе будет проведен аналитический обзор подходов к разработке алгоритмов построения расписаний, которые могут быть использованы для построения алгоритмов сформулированной в разделе 1 задачи. Достаточно полная классификация задач построения расписаний приводится в [34]. Между тем, значительная часть упомянутых там задач не являются NP-трудными, а значит, алгоритмы их решения неприменимы к рассматриваемой задаче.

Задача построения расписания в постановке, описанной в разделе 1, в известных автору работах не рассматривалась, поэтому будет рассмотрена возможность применения алгоритмов решения близких задач. А именно, в дальнейшем рассматриваются методы, применимые к задачам построения расписаний для программы, представленной ориентированным графом без циклов (графом потока данных), когда задания выполняются без прерываний, с единым директивным сроком.

Далее рассматриваются основные методы решения близких задач: полный перебор, метод ветвей и границ, жадные алгоритмы, алгоритмы имитации отжига и генетические алгоритмы.

2.2 *Полный перебор*

Наиболее простой метод решения любой комбинаторной задачи, в том числе и задачи построения расписания – перебор всех возможных вариантов. Поскольку в рассматриваемой постановке число дублирующих процессоров не ограничено, то формально число различных расписаний бесконечно и перебрать его нельзя. Если ограничить сверху общее число про-

цессоров некоторой достаточно большой величиной, то теоретически все возможные расписания можно упорядочить. Например, можно упорядочивать сначала по числу используемых процессоров, а расписания для одинакового числа процессоров сравнивать как наборы сочетаний заданий. Данный метод выдает гарантированно оптимальное решение. Однако пространство, на котором осуществляется перебор, содержит значительно больше $n!$ расписаний (с учетом всех возможных способов распределить задания по процессорам и упорядочить их внутри процессоров). При этом перебор всех возможных перестановок заданий не позволяет отсеивать некорректные расписания (с циклами) без непосредственной проверки отсутствия циклов, которая также требует вычислительных ресурсов. По этим причинам полный перебор нельзя рассматривать как эффективный способ решения поставленной задачи с точки зрения вычислительной сложности.

2.3 Метод ветвей и границ

Суть метода ветвей и границ [58] в ограничении полного перебора с помощью разбиения всего множества решений на подмножества и отсечения заведомо неперспективных подмножеств. Если в подмножестве A нижняя граница значений целевой функции больше, чем верхняя граница в подмножестве B, то подмножество A можно не рассматривать, так как минимум на нем заведомо не достигается. Для некоторых задач эта процедура может существенно сузить область поиска оптимального решения.

Однако для оптимизации числа процессоров метод ветвей и границ не подходит, так как по значению целевой функции нельзя проводить отсечение из-за наличия ограничений. Пусть в результате некоторого разбиения удалось получить условие, задающее подмножество расписаний, для которых $M(S) < c$ для некоторой константы c. Нет гарантии, что это подмножество непустое, то есть содержит хотя бы одно корректное расписание. Перебор же всех расписаний даже для фиксированного числа процессоров

имеет факториальную сложность, то есть метод ветвей и границ не даст никакого существенного ускорения перебора.

2.4 Жадные алгоритмы

Жадные алгоритмы [34] решения NP-трудных задач в общем случае дают приближенное решение. В начале работы алгоритма текущее расписание пустое. Итоговое расписание строится путем последовательного добавления заданий к текущему в соответствии с некоторым критерием. Например, можно на каждом шаге ставить очередное задание на такую позицию, чтобы общее время выполнения текущего расписания было минимальным. Итоговый результат получается за полиномиальное время.

В литературе достаточно часто рассматриваются алгоритмы динамического построения расписаний с учетом надежности (для фиксированного числа процессоров). Наиболее простые жадные алгоритмы DASAP (do as soon as possible) и DALAP (do as late as possible) описаны в [83]. Первый алгоритм ставит очередное задание как можно раньше, второй – как можно ближе к директивному сроку, но не превосходя его. В той же работе предложен алгоритм DRCD (Dynamic reliability cost driven), учитывающий при построении расписания надежность каждого задания. В работе [82] описан еще более сложный алгоритм eFRCD (efficient fault-tolerant reliability cost driven).

Существуют также жадные алгоритмы другого типа, основная идея которых заключается в следующем: изначально каждое задание назначается на отдельный процессор, а затем начинается поочередное объединение заданий с пары процессоров на один, до тех пор, пока ни одну из пар процессоров нельзя будет объединить. Эти методы рассмотрены в работах [8], [13]. Такие алгоритмы также в общем случае не гарантируют фиксированной точности полученного результата: полученное решение может использовать большее число процессоров, чем оптимальное.

Жадные алгоритмы такого типа имеют ряд недостатков. Так как требуется найти расписание для минимального возможного числа процессоров, расстановка заданий в соответствии со временем выполнения и надежностью потенциально может приводить к решениям, на которых значение целевой функции очень далеко от оптимального. Если же на каждом шаге расставлять задания так, чтобы не увеличивать число процессоров (если это возможно), то тогда неудачный выбор задания на первых шагах может привести к получению плохого результата, поэтому потребуются нетривиальные процедуры выбора очередного задания для добавления его в расписание. Приведем два примера, которые показывают потенциальную неоптимальность жадных алгоритмов.

Пример, когда жадный алгоритм, минимизирующий число процессоров, работает некорректно. Пусть программа состоит из трех заданий с фиксированной длиной 10, 10 и 5 единиц времени соответственно, причем третье задание требует результатов первого и второго. Пусть $t^{dir} = 20$. Надежность всех процессоров и заданий равна 1. Тогда на первом шаге жадный алгоритм поставит первое задание на первый процессор, затем второе задание – также на первый процессор, после чего третье задание нельзя будет никуда поставить, не нарушив ограничения на время выполнения.

Алгоритм, минимизирующий время, как несложно проверить, на этом примере работает корректно.

Пример, когда жадный алгоритм, минимизирующий время, работает некорректно. Пусть программа состоит из трех заданий, каждое фиксированной длиной 1 единицу времени, связей между заданиями нет. Резервных версий нет, надежность каждого процессора $P(m)$ равна 0.9, надежность каждого задания равна 1. $t^{dir} = 3. R^{dir} = 0.9.$ Жадный алгоритм расставит каждое из заданий на отдельный процессор, после чего потребуется использовать резервные процессоры, чтобы поднять надежность до уровня

0.9. Для этого нужно к каждому из трех процессоров добавить резерв, то есть всего процессоров будет 6. Общая надежность составит 0.99^3. Если рассмотреть семейство аналогичных задач с $R^{dir} = P(m) = p$, то число резервов каждого процессора, необходимое для достижения требуемой надежности будет возрастающей функцией от p, стремящейся к бесконечности при p стремящемся к 1 слева. Таким образом, можно привести пример, когда результат работы жадного алгоритма будет содержать сколь угодно большое число процессоров. Однако оптимальное решение использует только один процессор, на который надо поместить все задания.

2.5 Алгоритм имитации отжига

Термин «имитация отжига» происходит от названия способа выжигания дефектов в кристаллической решетке металла. Атомы, занимающие в ней неправильное место, при низкой температуре не могут сместиться в нужное положение, – им не хватает кинетической энергии для преодоления потенциального барьера. При этом система в целом может находиться в состоянии локального энергетического минимума. Для выхода из него металл нагревают до высокой температуры, а затем медленно охлаждают, позволяя атомам занять правильное положение в решетке (соответствующее энергетическому минимуму).

Общий принцип работы алгоритма имитации отжига следующий [51]. На каждом шаге есть текущее приближение решения. Некоторым образом его изменяют, так чтобы полученное решение-кандидат было не очень далеко от текущего приближения. Если характеристики кандидата лучше характеристик текущего приближения (в наиболее простом случае это значение скалярной целевой функции), то кандидата берут в качестве приближения на следующем шаге. Иначе, его берут с некоторой вероятностью p, где p убывает с ростом номера итерации. Таким образом, на начальных итерациях велика вероятность перемещения по пространству решений, а

на более поздних итерациях происходит спуск в точку текущего локального минимума. Алгоритм имитации отжига не гарантирует нахождение лучшего решения, однако всегда есть ненулевая вероятность, что это решение будет найдено.

Алгоритм имитации отжига создавался для задач безусловной непрерывной оптимизации, однако успешно применяется для широкого круга задач, в том числе для нелинейной условной оптимизации [98]. Имеются примеры использования имитации отжига для решения задач планирования (job shop scheduling). В [15] формулируются общие принципы применения алгоритма имитации отжига для планирования: необходимо определить *пространство* решений, на котором идет поиск; задать структуру окрестности каждого решения, то есть, *ввести операции* преобразования решения; наконец, определить *целевую функцию* алгоритма. При соблюдении этих требований алгоритм не только будет работать на практике, но можно будет также и теоретически доказать его асимптотическую сходимость.

В работе [76] приведены экспериментальные подтверждения эффективности алгоритма имитации отжига для задачи построения расписаний (не идентичной рассматриваемой, но также связанной с построением расписания для программы, заданной ориентированным ациклическим графом потока данных). В более подробной работе [77] предлагается усовершенствованный подход к построению алгоритма, основанный на применении эвристик. В классическом алгоритме имитации отжига новое приближение выбирается в окрестности текущего случайным образом, однако, зная, что поиск идет в пространстве многопроцессорных расписаний, можно использовать эвристики, направляющие поиск. Примерами таких эвристик служит построение расписаний по критическому пути [102], топологическое перемещение заданий (задания, связанные друг с другом, по воз-

можности перемещаются на один процессор), балансировка нагрузки на процессоры, оптимизация отдельных обособленных подграфов [76].

В работах [8][50] описано применения алгоритма имитации отжига к решению схожей задачи оптимизации времени выполнения при ограничениях на аппаратные ресурсы, но без ограничений на надежность. Добавление характеристик надежности существенно усложняет задачу, однако общую идею, предложенную в данных работах, можно использовать и при решении рассматриваемой задачи: используя операции преобразования расписания, учитывая оба имеющихся в задаче ограничения, можно построить итерационный алгоритм поиска решения.

2.6 Генетические алгоритмы

Генетический алгоритм моделирует природный процесс естественного отбора. Общая схема работы генетического алгоритма следующая [44]:

1. Сначала возможные решения кодируются (обычно в виде бинарных строк или строк из чисел). Далее делаются следующие шаги.
2. Создание начальной популяции из некоторого фиксированного количества решений.
3. Скрещивание – лучшие решения обмениваются отдельными частями строк, которые их кодируют, друг с другом.
4. Мутация – в некоторых решениях делается небольшое случайное изменение.
5. Селекция – отбор лучших решений по определенному критерию
6. Переход на пункт 3 или завершение работы, если выполнен критерий останова (нет существенных изменений в течение нескольких шагов подряд либо достигнуто максимальное количество шагов).

Существует гипотеза о возможности сходимости генетического алгоритма к оптимуму [39].

Конкретная реализация генетического алгоритма зависит от задачи и тех объектов, с которыми она работает. Кодирование многопроцессорных расписаний представляет собой нетривиальную задачу. В случае, когда задания в расписании независимы, достаточно указать привязку к процессорам, и тогда расписание можно закодировать [67][68] списком $(m_1, m_2, \dots, m_n)$, где m_i – процессор, на котором выполняется задание s_i. Если же между заданиями есть зависимости, кодировка должна быть гораздо более сложной, при этом необходимо также, чтобы операции скрещивания и мутации не порождали в результате некорректные решения [40][30]. Операции скрещивания и мутации в этом случае могут быть совершенно не похожи на традиционные операции над битовыми строками, вместо этого расписания обмениваются фрагментами, перемещение которых заведомо не нарушает корректность [45][46].

В работе [55] рассмотрена задача определения минимального необходимого числа процессоров и построения расписания для системы с единым директивным сроком. Для решения задачи предлагаются генетические алгоритмы с целочисленной и бинарной кодировкой. Для алгоритма определена область эффективного применения и разработана методика автоматической адаптации к различным вариантам постановки задачи структурного синтеза вычислительной системы.

В работах [31], [47], [48] описаны эволюционные алгоритмы, применяемые для построения расписаний. Их нельзя определенно назвать генетическими, так как в них нет операции скрещивания в чистом виде, и решения не кодируются, а работа происходит непосредственно с расписаниями. В этих работах рассмотрены несколько стратегий применения операции, являющейся аналогом мутации. Отбор в новую популяцию происходит в зависимости от соблюдения ограничений на время, надежность не рассматривается. Новые особи в популяции создаются путем случайной мутации существующих.

При применении генетических алгоритмов к оптимизационным задачам с несколькими ограничениями необходимо также предложить метод сравнения особей на приспособленность, учитывающий все ограничения. В рассматриваемой в данной работе задаче это означает, что такая функция должна зависеть от числа процессоров в расписании, времени и надежности. Некоторые исследователи предлагают функции, схожие с барьерными функциями [72].

Наиболее существенным препятствием для применения генетических алгоритмов к рассматриваемой задаче является потенциально низкая скорость работы. Алгоритм должен позволять использовать различные функции интерпретации для оценки времени выполнения расписания, при этом оценка времени может требовать больших вычислительных затрат, особенно если для нее используется имитационное моделирование. На каждой итерации генетического алгоритма никак не избежать оценки времени для всех расписаний из популяции, тогда как в алгоритме имитации отжига оценка времени происходит на каждой итерации один раз. Следовательно, чем больше популяции, тем медленнее будет работать алгоритм. Также в ходе работы генетического алгоритма требуется проводить проверку корректности вновь создаваемого в процессе скрещивания и мутации расписания, тогда как при имитации отжига такая проверка также делается один раз за итерацию.

2.7 Выводы

В данном разделе были рассмотрены подходы к разработке алгоритмов и алгоритмы построения многопроцессорных расписаний для программ, задаваемых графом потока данных. Поскольку ни один из существующих алгоритмов не может быть применен к рассматриваемой задаче напрямую без модификации, необходимо выбрать подход для построения такого алгоритма. Так как методы полного перебора и ветвей и границ непримени-

мы из-за чрезмерно высокой вычислительной сложности, остается выбор из трех методов: жадные алгоритмы, алгоритмы имитации отжига и генетические алгоритмы. Жадные алгоритмы работают быстро, но их результат может быть очень далек от оптимального. Генетические алгоритмы потенциально могут получать более точные результаты, но в условиях необходимости вычислять время выполнения расписания многократно на каждой итерации требуют очень много времени на работу. Метод имитации отжига лишен этих недостатков, поэтому было решено использовать в качестве основы его, используя все основные достижения в данной области:

- Ввести систему операций на пространстве расписаний, которая обладает свойствами полноты и замкнутости,
- Использовать эвристики для выбора нового приближения,
- Рассмотреть вопрос асимптотической сходимости алгоритма к оптимальному решению,
- Оценить результаты работы алгоритма экспериментально.

В следующем разделе будет описан алгоритм имитации отжига для решения рассматриваемой в работе задачи.

3 Алгоритм построения расписания

3.1 Общая схема алгоритма

Для решения сформулированной задачи предлагается алгоритм, основанный на схеме имитации отжига. Общая схема работы одной итерации алгоритма следующая.

Шаг 1. Задать начальное корректное решение $(S_0, D_0) \in \overline{S}$ и считать его текущим вариантом решения ($(S, D) = (S_0, D_0)$).

Шаг 2. Установить начальную температуру T_0, приняв её текущей ($T = T_0$).

Шаг 3. Применить операции преобразования решения к текущему решению (S, D) и получить новый корректный вариант решения $(S', D') \in \overline{S}$, если это решение является лучшим из ранее найденных решений, то запомнить его.

Шаг 4. Найти изменение функционала оценки качества решения $\Delta F = F(S', D') - F(S, D)$.Если $\Delta F \leq 0$ (решение улучшилось), то новый вариант решения считать текущим ($(S, D) = (S', D')$). Если $\Delta F > 0$ (решение ухудшилось), то принять с вероятностью $p = e^{-\Delta F/T}$ в качестве текущего решения новый вариант решения (S', D').

Шаг 5. Повторить заданное число раз шаги 3 и 4 без изменения текущей температуры.

Шаг 6. Если критерий останова выполнен, то завершение работы алгоритма.

Шаг 7. Понизить текущую температуру в соответствии с выбранным законом и перейти к шагу 3.

Для построения алгоритма имитации отжига для решения конкретной задачи условной оптимизации требуется решить следующие задачи [109]:

1. Разработать способ представления решения (S, D) и операций преобразования текущего решения на шаге 3.
2. Разработать стратегию применения операций преобразования текущего решения на шаге 3: какую операцию применять, к какому элементу (S, D), как его изменять.
3. Выбрать закон понижения температуры на шаге 7.
4. Определить функционал $F(S, D)$ используемый для оценки качества текущего решения на шаге 4. В него входят целевая функция и могут входить все или часть функций-ограничений.
5. Выбрать критерий останова алгоритма, используемый на шаге 6.

Далее рассмотрим каждый из шагов подробно.

3.2 Операции преобразования расписания

Введем следующие обозначения.

$Dep(s)$ – множество таких элементов s_i, что в графе G есть ребро (v_i, v), т.е. множество вершин – непосредственных предшественников s;

$Trans(s)$ – множество таких элементов s_i, что в графе G есть цепь (v_i, v), то есть множество вершин – непосредственных и транзитивных последователей s;

$Succ(s)$ – множество заданий, косвенно зависящих от s. $s_1 \in Succ(s_0)$, если в графе расписания есть путь из s_0 в s_1, не содержащий дугу от s_0 к следующему за ним заданию на процессоре m_0, если такой дуги нет в графе программы. Формально, пусть . $N_0 = Trans(s)$, $N_i = \{s_1, s_2, \dots, <_n\}$, и $s_{n+1}, \dots, s_{n+k}$ таковы, что $\forall l \in [1..k]: \exists i \in [1..n]: m_i \neq m \wedge m_i = m_{n+l} \wedge$

$n_i < n_l$. Тогда $N_i = N_{i+1} \cup Trans(s_{n+1}) \cup Trans(s_{n+2}) \cup \ldots \cup Trans(s_{n+k})$. Если $N_{i-1} = N_i$, то $N_i = Succ(s)$.

Для преобразования расписаний введены следующие операции.

Операция добавления резервного процессора. В исходном расписании (S, D) к мультимножеству D добавляется новый элемент.

Операция удаления резервного процессора. В исходном расписании (S, D) из мультимножества D удаляется элемент m, для которого $Dup(m) > 1$.

Операция переноса задания. В исходном расписании (S, D) выбирается элемент $s_1 = (v_1, k_1, m_1, n_1)$ процессор m_2 и номер n_2, такой что

$\forall s_i : m_i = m_2 : \left(n_i < n_2 \Rightarrow s_i \notin Succ(s_1)\right) \wedge (n_i \geq n_2 \Rightarrow s_1 \notin Succ(s_i))$,

и происходит следующая замена:

$s_1' = (v_1, k_1, m_2, n_2), \forall s_i : m_i = m_2 : n_i \geq n_2 \Rightarrow s_i' = (v_i, k_i, m_i, n_i + 1)$.

Эта операция позволяет или изменить порядковый номер выполнения задания на процессоре, или перенести задание на другой процессор.

Операция добавления версий. Версии можно добавлять только парами в силу принципа работы NVP. Добавление одной версии (v, k) эквивалентно последовательности операций: 1) добавить новый процессор m_0; 2) назначить версию первым заданием на этот процессор $s = (v, k, m_0, 1)$; 3) перенести s на другой процессор в соответствии с определением операции переноса задания; 4) удалить m_0.

Операция удаления версий. Версии удаляются также парами. Из расписания удаляются два элемента, соответствующие удаляемым версиям.

Утверждение 3.1. Замкнутость системы операций. Если (S, D) – корректное расписание, то после применения любой из операций также получается корректное расписание.

Доказательство. Операции добавления и удаления процессоров приводят к корректному расписанию, так как число резервных процессоров не влияет на циклы в графе расписания.

Рассмотрим операцию перестановки задания. В графе расписания при выполнении этой операции удаляется ребро от s_1 к следующему после него заданию на том же процессоре и добавляются ребра от s_2 к s_1 и от s_1 к следующему за s_2 заданию s'. Удаление ребер к появлению циклов привести не может. Добавление ребра от s_2 к s_1 может привести к появлению цикла только в том случае, если есть путь от s_1 к s_2, не включающий удаляемое ребро, однако это невозможно в силу требования $s_2 \notin Succ(s_1)$. Добавление ребра от s_1 к s' также не может создать цикл: в исходном графе не могло быть пути из s' в s_1, так как в нем заведомо был путь из s_1 в s' (они находятся на одном процессоре), но не было циклов.

Рассмотрим операцию добавления версии. Операции, используемые на шагах 1, 3, 4, как уже доказано, корректны. Шаг 2 также корректный, так как в исходном графе нет циклов по условию, а других ребер, инцидентных новой вершине, нет, так как она на процессоре одна. Операция удаления версии также корректна, так как удаление заданий не может привести к появлению циклов.

Утверждение 3.2. Для каждой операции, переводящей расписание A в расписание B, существует обратная операция, переводящая расписание B в расписание A.

Доказательство.

Для добавления процессора m обратной будет операция удаления этого же процессора m, и наоборот. Эти операции всегда приводят к корректным расписаниям.

Для операции переноса задания $s = (v_1, k_1, m_1, n_1)$ на процессор m_2 и позицию n_2 обратной операцией будет перенос задания (v_1, k_1, m_2, n_2) на

процессор m_1 и позицию n_1. Корректность этой операции следует из того, что если в исходном расписании не было циклов, то при возвращении к нему они также не появятся.

Для добавления версии обратной будет операция удаления, и наоборот. Так как эти операции содержат в себе операции переноса, их корректность следует из корректности операции переноса.

Утверждение 3.3. Полнота системы операций. Если $(S_1, D_1), (S_2, D_2)$ – корректные расписания, то существует последовательность операций, приводящая (S_1, D_1) к (S_2, D_2), такая, что все промежуточные расписания корректны.

Доказательство. В силу утверждения 3.1, все операции над расписаниями, введенные ранее, приводят только к корректным расписаниям. В силу утверждения 3.2, для любой операции существует обратная, которая также приводит к корректному расписанию. Таким образом, если доказать, что есть расписание (S_0, D_0), такое, что (S_1, D_1) и (S_2, D_2) можно привести к нему, то из этого будет следовать доказываемое утверждение.

Упорядочим задания на графе программы. Для каждого задания v можно определить $Level(v)$ – ярус, на котором оно расположено. $Level(v) = 1$, если в вершину v на графе программы не входит ни одно ребро. $Level(v) = n$, если в вершину v на графе программы входят ребра только из вершин с ярусом меньше n. В силу ацикличности графа программы для каждого задания его минимальный возможный ярус определен единственным образом. Пусть на первом ярусе p_1 заданий. Пронумеруем их числами от 1 до p_1. Аналогично, если на втором ярусе p_2 заданий, то их можно пронумеровать числами от $p_1 + 1$ до $p_1 + p_2$. Продолжая эту процедуру, можно поставить каждому заданию в соответствие число от 1 до N (N – общее число заданий).

Рассмотрим расписание, состоящее из четверок вида $s_i = (v_i, k_i, m_i, n_i)$, где i – номера, полученные описанным выше способом. В этом расписании все задания стоят на одном процессоре, причем каждому заданию предшествуют только задания с того же яруса или ярусов ниже. В силу определения яруса в этом расписании нет циклов. Назовем его канонической формой расписания для заданной программы.

Покажем, что любое расписание для заданной программы можно привести к канонической форме. Сначала удалим все дублирующие процессоры и дополнительные версии программы. В результате получится расписание, в котором столько элементов, сколько заданий в программе. После этого можно взять пустой процессор m_0 и переносить на него задания в порядке номеров, ставя каждое последним. В результате получится каноническая форма. Осталось доказать, что каждая операция переноса задания является корректной.

Доказательство проведем методом индукции. Перенос первого задания на пустой процессор корректен, так как по определению нумерации первое задание не зависит ни от какого другого, а раз оно стоит на новом процессоре первым в очереди, то на графе расписания в него не входит ни одна дуга, а значит, циклов не возникает.

Пусть теперь задания $1 \dots p$ перенесены, и происходит перенос задания под номером $p + 1$. По определению операции переноса должно выполняться условие

$$\forall s_i : m_i = m_2 : \big(n_i < n_2 \Rightarrow s_i \notin Succ(s_1)\big) \wedge (n_i \geq n_2 \Rightarrow s_1 \notin Succ(s_i)).$$

Поскольку задание v_{p+1} ставится последним, то вторую половину условия проверять не нужно, тогда условие корректности можно записать как $\forall i : i \leq p \Rightarrow s_i \notin Succ(s_{p+1})$.

Построим множество $Succ(s_{p+1})$. На первой итерации построения в него войдут четверки из $Trans(s_{p+1})$. В силу определения нумерации, ни

один из элементов $Trans(s_{p+1})$ не имеет номера меньше $p+1$, так как они находятся на более высоких ярусах. Следовательно, ни один из них не лежит на процессоре m_0, а значит и следующие за ними на нем не лежат. Таким образом, на второй итерации построения $Succ(s_{p+1})$ к текущему множеству добавляются множества $Trans(s_j), j > p$. Продолжая эти рассуждения, получаем, что ни один из элементов $Succ(s_{p+1})$ не имеет номера меньше или равного p, а это означает, что условие корректности выполнено.

Таким образом, любые два расписания (S_1, D_1) и (S_2, D_2) можно привести к канонической форме указанной последовательностью операций. Так как для каждой операции есть обратная, то можно получить цепочку операций, приводящую (S_1, D_1) к (S_2, D_2).

3.3 Стратегии применения операций.

Стратегия выбора операции на текущем шаге следующая: если не выполняются ограничения на надежность, то выполняется либо добавление процессора, либо добавление версии (с равными вероятностями). В противном случае необходимо уменьшить время выполнения программы или число используемых процессоров, поэтому выполняется либо удаление версий и процессоров (если такая операция вообще возможна, т.е. хотя бы какие-то резервные процессоры или версии используются), либо перенос заданий. Применение некоторых операций на определенной итерации алгоритма может быть невозможно. Например, если ни у одного процессора нет резервных копий, то удалять процессоры нельзя, а если все имеющиеся версии использованы, то нельзя добавлять версии. Такую ситуацию всегда можно определить, поэтому на этом шаге невозможные операции не рассматриваются.

Далее описана процедура выбора параметров для каждой операции.

Добавление версии. Случайно выбирается задание, версии которого добавляются (среди заданий, для которых имеется нужное количество доступных версий). Вероятность выбора задания обратно пропорциональна количеству уже использующихся его версий.

Удаление версии. Случайно выбирается задание, версии которого удаляются. Вероятность выбора задания пропорциональна количеству уже использующихся его версий.

Добавление резервного процессора. Аналогично добавлению версии задания, процессоры с меньшим числом резервов имеют большую вероятность добавления.

Удаление резервного процессора. Удаляется резерв для случайно выбранного процессора. Чем больше резервов процессоров, тем больше вероятность его удаления.

Вероятности удаления и добавления процессоров и версий заданы таким образом, чтобы стремиться соблюдать баланс между надежностью различных компонентов.

Перенос задания.

Если $t < t^{dir}$, то следует пытаться уменьшить число процессоров. Происходит следующая операция: выбирается процессор, на котором меньше всего заданий, и все задания с него переносят на другие процессоры.

Если $t > t^{dir}$, то необходимо улучшить время выполнения расписания. Это можно сделать либо перенеся часть заданий на пустой процессор, либо поменяв порядок или привязку на существующих. Предлагаются четыре стратегии выбора параметров для этой операции.

Стратегия уменьшения задержек (сокращенно стратегия S1). Эта стратегия основана на следующем утверждении. Если время начала выполнения каждого задания равно длине критического пути в графе G от истоков до задания, то расписание будет оптимальным. Длина критического

пути является минимально возможным временем начала выполнения задания и равна сумме значений времени выполнения заданий, соответствующих вершинам критического пути.

Для каждого элемента s можно определить минимально возможное время, когда s может начать выполняться, то есть все задания из $Dep(s)$ завершены. Разница между этим временем и временем, когда задание s начало выполняться в расписании, является задержкой задания s.

Для переноса выбирается задание, стоящее перед заданием с максимальной задержкой. Если операция не принимается, так как полученное расписание хуже текущего, то на следующей итерации рассматривается задание со второй по величине задержкой и так далее. Если все задания с ненулевой задержкой исчерпаны, задание выбирается случайно. Далее выбирается позиция (пара (m, n) из четверки, задающей элемент расписания), куда можно перенести выбранное задание. Это задание переносится на случайно выбранную позицию, в которую перенос возможен без нарушения условий корректности.

Пример работы стратегии приведен на рисунке 3.1. Задание 3 не зависит от задания 4, поэтому перенос задания 4 на первый процессор уменьшает задержку задания 3, и общее время выполнения также уменьшается.

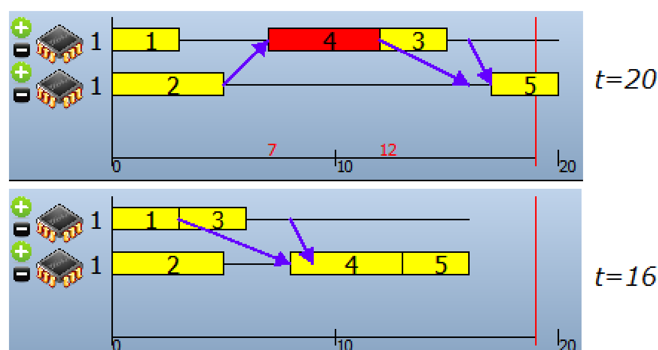


Рисунок 3.1. Пример стратегии уменьшения задержек.

Стратегия заполнения простоев (сокращенно стратегия S2). Эта стратегия основана на эмпирической гипотезе: чем меньше времени в сумме простаивают процессоры, тем лучше расписание.

Для каждой позиции (m, n) можно определить время простоя. Если $n = 1$, то простой – время от начала работы до начала выполнения задания на позиции $(m, 1)$. Если позиция (m, n) обозначает место после завершения последнего задания на процессоре m, то простой – время от конца работы последнего задания на m до конца выполнения всего расписания. Иначе, простой позиции (m, n) – это время между завершением работы задания в $(m, n - 1)$ и началом работы задания в (m, n).

Из всех позиций выбирается позиция с максимальным простоем. Если операция не принимается, так как полученное расписание хуже текущего, то на следующей итерации рассматривается позиция со вторым по величине простоем, и так далее. Если все позиции ненулевой длины исчерпаны, то позиция выбирается случайно. Задание для переноса выбирается случайно с учетом соблюдения условий корректности.

На рисунке 3.2 приведен пример применения данной стратегии. Между заданиями 1 и 4 большой простой, поместив туда задание 3, можно добиться уменьшения времени выполнения всей программы.

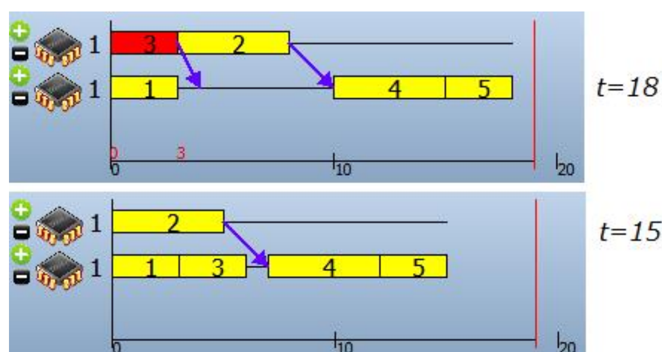


Рисунок 3.2. Пример стратегии заполнения простоев.

Смешанная стратегия (сокращенно стратегия S3). Смешанная стратегия объединяет две предыдущих. На каждой итерации случайным образом применяется либо одна стратегия, либо другая. Таким образом, в этой стратегии попеременно заполняются области в расписании, когда один из процессоров бездействует в ожидании завершения работ на других процессорах, либо делается попытка поставить задание с большой задержкой как можно раньше, чтобы уменьшить время задержки. Эта стратегия позволяет использовать преимущества двух предыдущих стратегий.

Случайная стратегия (сокращенно стратегия S0). В этой стратегии параметры для операции выбираются случайным образом.

3.4 Условие перехода к новому расписанию и критерий останова.

После применения выбранной операции получается новое расписание, для которого можно рассчитать время выполнения, надежность и число процессоров. В зависимости от отношений характеристик нового и предыдущего расписаний новое расписание может стать текущим приближением на следующей итерации алгоритма. Как и в стандартном алгоритме имитации отжига, присутствует параметр T, моделирующий температуру, значение которого в ходе работы алгоритма уменьшается в соответствии с выбранным законом понижения температуры.

Быстрое понижение температуры уменьшает вычислительную сложность алгоритма, но может привести к быстрой сходимости алгоритма к «плохому» локальному оптимуму. Медленное понижение температуры уменьшает вероятность сходимости алгоритма к «плохому» локальному оптимуму, но при этом вычислительная сложность алгоритма может оказаться неприемлемой. Поэтому при выборе закона понижения температуры требуется находить баланс между вычислительной сложностью и точно-

стью алгоритма [99]. Следует заметить, что при любом законе понижения температуры есть вероятность потери наилучшего решения, найденного в ходе работы алгоритма. Поэтому во всех алгоритмах имитации отжига, которые были разработаны автором для решения задач построения расписаний и структурного синтеза вычислительных систем реального времени, на 3-м шаге алгоритма была добавлена операция сохранения наилучшего решения, которое было найдено в ходе работы алгоритма. Это решение и выдавалось в качестве результата работы алгоритма при его завершении.

При построении алгоритмов имитации отжига наиболее часто используются следующие законы понижения температуры [92]:

- Закон Больцмана: $T = T_0/ln(1+x)$,
- Закон Коши: $T = T_0/(1+x)$,
- Смешанный закон: $T = T_0 \frac{ln(1+x)}{1+x}$.

Здесь T – текущая температура алгоритма, T_0 – начальная температура, x – номер итерации алгоритма. Начальная температура является параметром алгоритма и задается в качестве исходных данных.

Преимущества и недостатки различных законов понижения температуры анализируются в [100]. В данной работе будут реализованы все три основных закона (закон Больцмана, закон Коши, смешанный закон). Будет проведено сравнение законов понижения температуры применительно к задаче определения минимального необходимого числа процессоров.

Условие останова выбрано следующее: если число совершенных итераций равно $10 \cdot |V|$, алгоритм завершает работу. Здесь V – множество заданий, Такое условие останова гарантирует *завершимость* алгоритма.

3.5 Вычислительная сложность одной итерации алгоритма

Входными данными задачи являются множества вершин и ребер исходного графа G, функции интерпретации и оценки надежности, а также

константные ограничения на время выполнения программы и надежность. Обозначим число вершин графа программы как N, а число ребер – как E. По условию $E \leq N^2$ в силу отсутствия циклов в графе. Далее приведены оценки сложности разных этапов алгоритма в зависимости от N и E.

Выбор операции имеет константную сложность.

Операции добавления и удаления процессоров имеют сложность не более N.

Операции добавления и удаления версий имеют сложность не более $N \cdot W$, где W – максимальное число версий у одного задания.

Сложность проверки корректности операции переноса имеет порядок $O(N \cdot E)$. Для проверки корректности нужно вычислить функцию $Succ(s)$ столько раз, сколько заданий на процессоре, куда осуществляется перенос. Для вычисления функции $Succ(s)$ нужно объединить значения функции $Trans(s)$, но не больше раз, чем ребер в графе. Значения функции $Trans(s)$ вычисляются один раз до начала работы алгоритма.

Значения величин задержек и простоев вычисляются одновременно с вычислением времени выполнения. Сам выбор параметров имеет константную сложность, поэтому сложность операции переноса равна сложности проверки корректности $O(N \cdot E)$.

Вычисление надежности для текущего расписания требует не более $N \cdot W$ операций. $N \cdot W$ есть оценка сверху для количества процессоров в системе (без учета резервных). Сложность функции оценки надежности линейно зависит от W.

Сложность операции вычисления времени зависит от используемой функции интерпретации. Обозначим эту сложность как $O(f\quad)$. Для модели Fibre Channel сложность порядка $O(N \cdot (N + E))$, для полносвязной модели – $O(N + E)$.

Суммируя все оценки, получаем сложность одной итерации алгоритма $O(N \cdot E) + O(ft)$.

3.6 Выводы

В данном разделе был предложен алгоритм имитации отжига для решения задачи структурного синтеза многопроцессорных вычислительных систем реального времени с учетом ограничений на время выполнения программы и требований к надежности системы. Введена система операций преобразования расписаний и предложены эвристические стратегии применения данных операций. Была обоснована корректность алгоритма через доказательство полноты и замкнутости системы операций преобразования расписаний, используемых алгоритмом. Приведена оценка вычислительной сложности алгоритма.

В следующем разделе будут рассмотрены вопросы асимптотической сходимости предложенного алгоритма, а также проведено экспериментально исследование точности и вычислительной сложности алгоритма.

4 Исследование свойств алгоритма

4.1 Асимптотическая сходимость алгоритма

Несмотря на недетерминированный характер алгоритма имитации отжига, для него возможно строгое обоснование сходимости в терминах теории вероятностей. Для этого используется теория цепей Маркова [10].

Последовательность дискретных случайных величин $\{X_n\}$ называется **цепью Маркова**, если

$$P(X_{n+1} = i_{n+1}|X_n = i_n,\, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = P(X_{n+1} = i_{n+1}|X_n = i_n)$$

Иначе говоря, цепь Маркова – это последовательность случайных величин, в которой значение каждой случайной величины зависит только от значения предыдущей [11].

Если указанная выше вероятность не зависит от номера шага n, то цепь Маркова является **однородной**.

Множество всех значений случайных величин $\{i_k\} = I$ также называют состояниями. Цепь Маркова описывает систему со счетным числом состояний, переходы между которыми осуществляются случайным образом с известными вероятностями. Так как множество состояний счетно (по определению дискретной случайной величины), цепь Маркова можно представить конечной либо бесконечной матрицей, где элемент на пересечении -й строки и j-го столбца есть вероятность перейти из состояния i в состояние j.

Схему работы предложенного в разделе 3 алгоритма имитации отжига можно рассматривать как цепь Маркова. Состояния соответствуют всем возможным расписаниям. Множество состояний счетно, а если количество резервных процессоров ограничено, то конечно. Вероятность перехода из одного состояния в другое определяется в соответствии со стратегиями применения операций, описанными в разделе 3.3. По построению алгорит-

ма вероятность выполнить ту или иную операцию зависит только от текущего расписания, что соответствует определению цепи Маркова [43].

Построенная цепь Маркова обладает двумя свойствами: неразложимостью и ацикличностью [73].

Неразложимость означает, что $\forall i, j \in I: \exists n \in \mathbb{N}: (P^n)_{ij} > 0$, то есть из любого состояния можно перейти в любое за конечное число шагов с ненулевой вероятностью

Ацикличность означает, что $\forall i \in I: \gcd(n \in \mathbb{N} | (P^n)_{ii} > 0) = 1$. Для неразложимой цепи достаточно [61], чтобы это условия выполнялось хотя бы на одном решении.

У цепи Маркова $\{X_n\}$ существует **стационарное распределение**, если предел $\{q_i\} = \lim_{k \to \infty} P(X_k = i | X_0 = j)$ не зависит от j.

Пусть цепь Маркова неразложима и ациклична. Тогда [61] у нее существует единственное стационарное распределение, определяемое формулами:

$$q_i = \sum_{j \in I} q_j \cdot P_{ji}$$

Следствием данной теоремы является то, что вектор стационарного распределения можно определить из системы *уравнений детального баланса*:

$$q_i \cdot P_{ij} = q_j \cdot P_{ji} \; \forall i, j$$

Для стандартного алгоритма имитации отжига, представленного цепью Маркова, вероятность перехода из состояния A в состояние B есть произведение вероятности генерации решения B, если текущее решение – A, и вероятности принять B в качестве нового приближения [29].

$$P_{ij}(t) = G_{ij} \cdot A_{ij}(t)$$

Если температура не меняется, то эти вероятности всегда одинаковые, то есть цепь Маркова однородна. Если вероятность генерации любого из решений в окрестности текущего одинакова, а вероятность принятия ново-

го решения определяется стандартными формулами, то вероятности перехода в цепи Маркова при заданной температуре определяются следующей формулой.

$$P_{ij}(t) = \frac{1}{N} \cdot min(1, e^{\frac{f(i)-f(j)}{t}})$$

Здесь N – размер окрестности решения, $f(i)$ – значение целевой функции на решении i.

При фиксированном значении температуры стационарное распределение для заданной таким образом цепи Маркова будет иметь следующий вид, определяемый из уравнений детального баланса [62].

$$q_i(t) = \frac{exp(-\frac{f(i)}{t})}{\sum_j exp(-\frac{f(j)}{t})}$$

Устремляя температуру к нулю и переходя к пределу, получим, что вероятность после бесконечного числа шагов получить неоптимальное решение равна 0, а вероятности получения каждого из оптимальных решений одинаковы. Следовательно, имеет место следующая теорема.

Теорема. При стремлении температуры к нулю стационарное распределение приближается к распределению, допускающему только оптимальные решения.

На практике данное утверждение не согласуется с реальным применением алгоритма имитации отжига: теорема содержит два предельных перехода, что означает, что нужно при каждом значении температуры делать бесконечное число итераций, и затем переходить к пределу по значениям температуры. В реальности число итераций с фиксированным значением температуры ограничено, поэтому требуется более точная модель.

Пусть значение температуры на -й итерации алгоритма равно t_n, а матрица переходов цепи Маркова непостоянна и зависит от текущего значения температуры. Такая цепь называется **неоднородной**.

Определим вид элементов матрицы переходов для рассматриваемого в данной работе алгоритма имитации отжига. Пусть $\varepsilon(i)$ – множество всех расписаний, которые можно получить из текущего расписания i с учетом предыдущих операций алгоритма, влияющих на применение операции переноса задания (расширенная окрестность [105]). Тогда с учетом вероятности принять полученное в результате применения операции расписание в качестве нового приближения определяется как

$$P_{ij}(t_n) = \frac{1}{|\varepsilon(i)|} \cdot min(1, e^{\frac{f(i)-f(j)}{t_n}})$$

Пусть $U_{ij}(m,k) = P(X(k) = j | X(m-1) = i)$, то есть матрица U определяется как $U(m,\ k) = \prod_{n=m}^{k} P(n)$.

Неоднородная цепь Маркова называется **слабо эргодической**, если $\forall i,j,l \in I, \forall m > 0: \lim_{k\to\infty}(U_{il}(m,k) - U_{jl}(m,k)) = 0$. Неформально это означает, что $X(k)$ перестает зависеть от $X(m)$ с ростом k, независимо от номера итерации m.

Неоднородная цепь Маркова называется **сильно эргодической**, если $\forall i,j \in I, \forall m > 0: \lim_{k\to\infty}(U_{ij}(m,k)) = q_j^*$. Неформально это означает сходимость по распределению к вектору q^*.

Для однородных цепей Маркова слабая и сильная эргодичность тождественны, но для неоднородных в общем случае это не так.

Дальнейшие рассуждения опираются на следующие две теоремы [62].

Теорема. Цепь Маркова слабо эргодична, если следующий ряд расходится для некоторых значений $\{N_k\}$: $\sum_{k=1}^{\infty}(1 - \tau_1(P(k)^{N_k}))$, где коэффициент эргодичности $\tau_1(P) = 1 - \min_{i,j\in I} \sum_{l=1}^{|I|} \min(P_{il}, P_{jl})$.

Теорема. Цепь Маркова сильно эргодична, если: 1) она слабо эргодична, 2) на каждой итерации у матрицы $P(k)^T$ есть собственное значение, равное 1 и собственный вектор $q(k)$, 3) для этих собственных векторов сходится числовой ряд $\sum_{k=1}^{\infty}\|q(k)-q(k+1)\|_1$

Сформулируем основную теорему об асимптотической сходимости алгоритма имитации отжига по аналогии с классическим алгоритмом [5][107] и докажем ее применительно к рассматриваемому алгоритму. Для этого надо учесть особенности алгоритма, выраженные во введенной выше формуле, определяющей вероятности перехода от одного состояния цепи Маркова к другому.

Теорема 4.1. Пусть значения температуры таковы, что $t_k \geq \frac{\Gamma}{\log(k+k_0)}$, $\Gamma > 0$, $k_0 > 2$. Тогда цепь Маркова, соответствующая алгоритму имитации отжига, сходится к стационарному распределению вида $q_i = \frac{1}{|\Im|}\chi_{\Im}(i)$, где $\Im$ – множество оптимальных решений.

Доказательство. Сначала докажем сильную эргодичность цепи Маркова. Как несложно видеть, условие (2) из теоремы означает существование собственного вектора q, удовлетворяющего уравнению $\mathrm{P(k)}^{\mathrm{T}} \cdot q = q$, или построчно, $q_i = \sum_{j\in I} q_j \cdot P_{ji}$, что равносильно уравнениям детального баланса. Проверим, что значения

$$q_i = \frac{|\varepsilon(i)|}{\sum_j \left(|\varepsilon(j)| \cdot \frac{min\left(1, e^{\frac{f(i)-f(j)}{t_n}}\right)}{min\left(1, e^{\frac{f(j)-f(i)}{t_n}}\right)}\right)}$$

удовлетворяют этим уравнениям.

Обозначим $min\left(1, e^{\frac{f(i)-f(j)}{t_n}}\right) = A_{ij}$. Заметим, что для этой функции верно тождество $A_{ij} \cdot A_{jk} = A_{ik}$.

Запишем уравнение детального баланса для q и проверим, что они выполняются.

$$\frac{|\varepsilon(i)|}{\sum_j(|\varepsilon(j)|\cdot\frac{A_{ij}}{A_{ji}})}G_{ij}\cdot A_{ij}=\frac{|\varepsilon(j)|}{\sum_k\left(|\varepsilon(k)|\cdot\frac{A_{jk}}{A_{kj}}\right)}G_{ji}\cdot A_{ji}$$

Значения G_{ij} и G_{ji} можно сократить, так как $|\varepsilon(i)|\cdot G_{ij}=1$.

$$\frac{1}{\sum_j(|\varepsilon(j)|\cdot\frac{A_{ij}}{A_{ji}})}=\frac{1}{\sum_k\left(|\varepsilon(k)|\cdot\frac{A_{jk}}{A_{kj}}\right)}\cdot\frac{A_{ji}}{A_{ij}}$$

$$\frac{1}{\sum_j(|\varepsilon(j)|\cdot\frac{A_{ij}}{A_{ji}})}=\frac{1}{\sum_k\left(|\varepsilon(k)|\cdot\frac{A_{jk}}{A_{kj}}\cdot\frac{A_{ij}}{A_{ji}}\right)}$$

$$\frac{1}{\sum_j(|\varepsilon(j)|\cdot\frac{A_{ij}}{A_{ji}})}=\frac{1}{\sum_k\left(|\varepsilon(k)|\cdot\frac{A_{ik}}{A_{ki}}\right)}$$

Получили тождество.

Проверим выполнение условия (3).

$$\sum_{k=1}^{\infty}\|q(k)-q(k+1)\|_1=\sum_{k=1}^{\infty}\sum_i|q_i(k)-q_i(k+1)|=$$

$$=\sum_{k=1}^{\infty}\sum_i\left|\frac{|\varepsilon(i)|}{\sum_j\left(|\varepsilon(j)|\cdot\frac{A_{ij}(k)}{A_{ji}(k)}\right)}-\frac{|\varepsilon(i)|}{\sum_j\left(|\varepsilon(j)|\cdot\frac{A_{ij}(k+1)}{A_{ji}(k+1)}\right)}\right|\leq$$

$$\leq\ C\cdot\sum_{k=1}^{\infty}\sum_i\left|\frac{1}{\sum_j(A_{ij}(k)/A_{ji}(k))}-\frac{1}{\sum_j(A_{ij}(k+1)/A_{ji}(k+1))}\right|=$$

$$=\ C\cdot\sum_i\left|\frac{1}{\sum_j(A_{ij}(1)/A_{ji}(1))}-\frac{1}{\sum_j(A_{ij}(k)/A_{ji}(k))}\right|.$$

Выражение $\sum_j(A_{ij}(k)/A_{ji}(k))$ строго больше 0, так как в нем по крайней мере есть ненулевое слагаемое, соответствующее $i=j$, поэтому вся сумма ряда конечна.

Для проверки условия (1) воспользуемся достаточным условием слабой эргодичности. Нужно доказать, что ряд $\sum_{k=1}^{\infty}(1-\tau_1(P(k)^{N_k}))$ расходится. Оценим $P(k)$ снизу.

$$P(k) \geq \left(\min_{i,j} G_{ij}\right) \cdot \exp\left(-\frac{\min\left(1, f(i)-f(j)\right)}{t_k}\right) = C_1 e^{-C_2/t_k}$$

$$\sum_{k=1}^{\infty}\left(1-\tau_1(P(k)^{N_k})\right) \geq \sum_{k=1}^{\infty} C_1^{N_k} e^{-C_2 N_k/\mathrm{t}_k}$$

Учитывая, что $t_k \geq \frac{\Gamma}{\log(k+k_0)}$, получаем ряд вида $C \cdot \sum_{k=1}^{\infty}\frac{1}{k}$, который расходится.

Осталось определить предел вектора q при стремлении температуры к нулю. Для этого рассмотрим предел

$$\lim_{\mathrm{n}\to\infty} \sum_{j} \left(\frac{min\left(1, e^{\frac{f(i)-f(j)}{t_n}}\right)}{min\left(1, e^{\frac{f(j)-f(i)}{t_n}}\right)}\right)$$

Если решение i – оптимальное, то знаменатель дроби всегда будет равен 1. Числитель будет равен 1, если j – тоже оптимальное решение, то есть $f(i)=f(j)$, если же $f(i)<f(j)$, то $e^{\frac{f(i)-f(j)}{t_n}} \to 0$. Получается, что слагаемое в сумме стремится к 1, если оно соответствует оптимальному решению, и к 0 в противном случае. То есть весь предел равен $|\mathfrak{I}|$, и соответственно $q_i = \frac{1}{|\mathfrak{I}|}$.

Если решение i – не оптимальное, то в знаменателе одной из дробей $f(j)-f(i)>0$, то есть последовательность $e^{\frac{f(i)-f(j)}{t_n}} \to \infty$, а соответствующее $q_i \to 0$.

Объединяя все значения q_i, получаем $q_i = \frac{1}{|\mathfrak{I}|}\chi_{\mathfrak{I}}(i)$, что и требовалось доказать.

Что касается скорости сходимости, приведем основные результаты из работ [94][97].

Пусть $a(k)$ – распределение вероятностей оказаться в том или ином состоянии на шаге k. Сравнив это распределение со стационарным $q(c)$ по норме, получим следующую оценку:

$$\|a(k) - q(c)\|_1 = O(k^{m_2 - 1} |\lambda_2(c)|^k)$$

Здесь λ_2 – второе по модулю собственное значение матрицы $P(c)$ кратности m_2.

$\|a(k) - q(c)\|_1 < \varepsilon$, если

$$k > K \cdot \left(1 + \frac{\ln(\varepsilon/2)}{\ln(1 - (\min_{i,j} P_{ij})^{K})}\right)$$

$$K = |I|^2 - 3|I| + 3.$$

Из данной оценки на k следует, что чтобы достичь заданной наперед точности, нужно совершить число итераций, пропорциональное квадрату от размера пространства поиска, то есть для достижения абсолютной точности алгоритм имитации отжига подходит хуже, чем полный перебор.

4.2 Метрика в пространстве расписаний

4.2.1 Метрика L(A, B)

Пусть S_1, S_2 – расписания, $O = \{O_1, \dots, O_n\}$ – набор операций, применение которых переводит S_1 в S_2. Введем над расписаниями следующую функцию $L(S_1, S_2) = \min|O|$.

Утверждение 4.2. $\overline{S}$, $L(A, B)$ – метрическое пространство.

Доказательство. Нужно доказать три свойства метрики [18]:

$\forall A, B: L(A, B) \geq 0, L(A, B) = 0 \leftrightarrow A = B$;

$\forall A, B: L(A, B) = L(B, A)$;

$\forall A, B, C: L(A, B) + L(B, C) \geq L(A, C)$.

Докажем первое свойство. Действительно, по определению L – длина цепочки операций, значит, она не может быть отрицательной. Для одинаковых расписаний цепочка минимальной длины пустая. Если два расписания соединены пустой цепочкой, то они совпадают.

Докажем второе свойство. Пусть $K = L(A, B)$. Из теоремы об обратной операции следует, что существует цепочка O_1, переводящая B в A, длины K. Эта цепочка состоит из операций, обратных к операциям из цепочки, задающей $L(A, B)$. Следовательно, $L(B, A) \leq K$. Предположим, что верно строгое неравенство. Тогда существует цепочка O_2 длины $M < K$, переводящая B в A. Но тогда цепочка из операций, обратных операциям O_2, переводит A в B и имеет длину меньше K, что противоречит тому, что $K = L(A, B)$. Следовательно, $L(B, A) = K = L(A, B)$.

Докажем третье свойство. Предположим, что для некоторых A, B, C: $L(A, B) + L(B, C) < L(A, C)$. Пусть O_1 – цепочка, соответствующая $L(A, B)$, O_2 – цепочка, соответствующая $L(B, C)$. Тогда объединение O_1 и O_2 имеет длину $L(A, B) + L(B, C)$ и переводит A в B. Но по предположению минимальная цепочка из A в C имеет длину большую, чем $L(A, B) + L(B, C)$. Получили противоречие.

4.2.2 Метрика H(A, B)

Метрика L определена теоретически, но вычисление ее очень сложно: вычисление этой метрики эквивалентно решению исходной задачи, то есть имеет экспоненциальную сложность. Введем другую метрику $H(A, B)$, для вычисления которой известен полиномиальный алгоритм, и которая будет аппроксимировать метрику L.

Пусть заданы два расписания A и B, использующие равное число процессоров без учета резервных. Пусть между процессорами в первом и втором расписании установлено биективное соответствие $f(m)$.

Определение 4.1. Если в расписании A задание (v, k) расположено на процессоре m_{A1}, а в расписании B – на процессоре m_{B2}, таком что $m_{B2} \neq f(m_{A1})$, то будем называть это *перестановкой* для (v, k).

Определение 4.2. Если в расписании A задание (v, k) расположено на процессоре m_{A1}, а в расписании B – на процессоре m_{B2}, таком что $m_{B2} = f(m_{A1})$, но порядок (v, k) относительно других заданий не совпадает, то будем также называть *перестановкой* для (v, k).

Определение 4.3. Если m_{A1} и $f(m_{A1})$ имеют число резервов, отличающееся на i, то будем говорить, что задано i *перестановок* для m_{A1}.

Определение 4.4. Если в одном из расписаний A, B есть пара версий $(v, k_1), (v, k_2)$, а в другом ее нет, то будем говорить, что задана *перестановка* для версий $(v, k_1), (v, k_2)$.

Содержательно каждая перестановка соответствует операции переноса задания или добавления/удаления процессора, которую нужно сделать, чтобы перевести расписание A в расписание B.

Обозначим через $H_f(A, B)$ общее число перестановок для двух расписаний при условии, что соответствие между процессорами задается функцией f.

Определение 4.5. $H(A, B) = \mathrm{argmin}_f H_f(A, B)$.

Утверждение 4.3. $\overline{S}$, $H(A, B)$ – метрическое пространство.

Доказательство. Нужно доказать три свойства метрики:

$\forall A, B: H(A, B) \geq 0, H(A, B) = 0 \leftrightarrow A = B$;

$\forall A, B: H(A, B) = H(B, A)$;

$\forall A, B, C: H(A, B) + H(B, C) \geq H(A, C)$.

Докажем первое свойство. Действительно, по определению H – число перестановок, значит, оно не может быть отрицательным. Для одинаковых расписаний число перестановок равно нулю по определению перестановки.

Если два расписания получаются друг из друга без перестановок, то они совпадают.

Докажем второе свойство. Пусть $K = H(A,B)$. Аналогично операциям, для каждой перестановки существует обратная. Следовательно, $H(B,A) \leq K$. Предположим, что верно строгое неравенство. Тогда существуют функция f и набор перестановок длины $M < K$, переводящие B в A. Но тогда набор обратных перестановок, переводит A в B и имеет длину меньше K, что противоречит тому, что $K = H(A,B)$. Следовательно, $H(B,A) = K = H(A,B)$.

Докажем третье свойство. Предположим, что для некоторых A, B, C: $H(A,B) + H(B,C) < H(A,C)$. Пусть H_1 – набор перестановок, соответствующий $H(A,B)$, H_2 – набор перестановок, соответствующий $H(B,C)$. Тогда последовательное применение перестановок H_1 и H_2 имеет длину $H(A,B) + H(B,C)$ и переводит A в B. Но по предположению минимальная цепочка из A в C имеет длину большую, чем $H(A,B) + H(B,C)$. Получили противоречие. Реально $H(A,C)$ может быть и меньше, если среди перестановок из H_1 и H_2 есть взаимно обратные.

Число возможных биективных отображений одного конечного множества процессоров на другое конечно. Поиск минимального из возможных значений числа перестановок может быть осуществлен следующим полиномиальным алгоритмом.

Достаточно рассматривать расписания без версий, так как число перестановок, связанных с версиями, не зависит от соответствия процессоров друг другу. Если в одном из расписаний есть дополнительные версии, то для добавления их достаточно сделать только нужное количество операций добавления версии после приведения всех остальных заданий в нужный порядок.

Пусть в первом расписании n процессоров, во втором – m. Добавим к тому, в котором их меньше, столько процессоров, сколько надо, чтобы уравнять число.

Для каждой пары процессоров можно определить значение числа перестановок, которые возникнут, если сопоставить эти процессоры друг другу: число перестановок заданий на процессоре + число недостающих заданий + число резервных процессоров, которые нужно удалить.

Например, пусть необходимо перевести расписания, изображенные на рисунке 4.1, одно в другое.

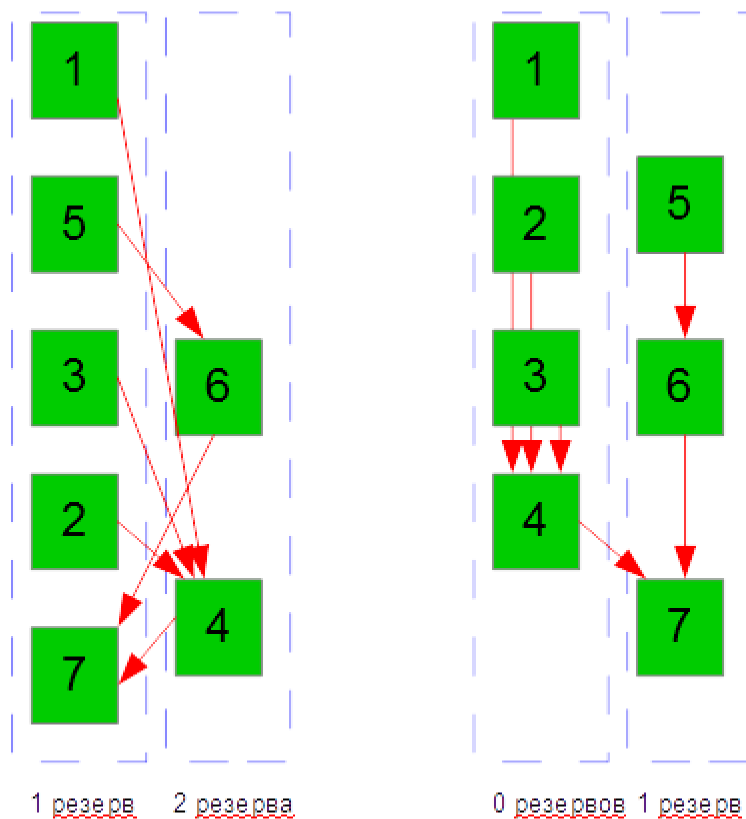


Рисунок 4.1. Пример двух расписаний для подсчета значения метрики $H(A,B)$.

Так, если сопоставить первый процессор в расписании слева (m_{A1}) первому процессору в расписании справа (m_{B1}), на нем надо переставить задания 3 и 2, и недостает задания 4. Тогда на втором процессоре не хвата-

ет заданий 5 и 7. Также надо удалить по одному резервному процессору. Итоговое число перестановок при всевозможных сопоставлениях процессоров друг другу приведено в таблице 4.1.

Таблица 4.1. Число перестановок для расписаний на рисунке 4.1.

	m_{B1}	m_{B2}
m_{A1}	**3**	**1**
m_{A2}	**5**	**3**

Поиск $H(A,B)$ эквивалентен поиску такого соответствия, чтобы сумма числа перестановок всех пар процессоров была минимальной. Данную задачу можно переформулировать следующим образом.

Пусть дана матрица размеров $n \times n$. Необходимо выбрать n ее элементов так, чтобы все они находились в различных строках и столбцах, а сумма была минимальной.

Данная задача решается алгоритмом Мункреса (также называемым венгерским алгоритмом) [56][69], имеющим сложность $O(n^3)$.

4.2.3 Связь между метриками L(A, B) и H(A, B)

Хотя перестановки соответствуют операциям преобразования расписания, определение перестановок не гарантирует, что соответствующая операция является корректной. По этой причине реальной цепочки длины $H(A,B)$, соединяющей два расписания, может не существовать.

Утверждение 4.4. $H(A,B) \leq L(A,B)$.

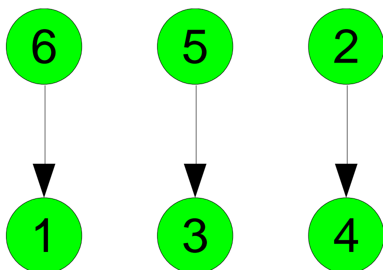
Доказательство. Предположим, что это не так. Но тогда для цепочки, соответствующей $L(A,B)$, соответствующий ей набор перестановок будет меньше, чем $H(A,B)$, что противоречит определению $H(A,B)$.

Утверждение 4.5. $H(A,B) \geq L(A,B)/2$.

Доказательство. Пусть все перестановки связаны с перемещением заданий (другие перестановки соответствуют операциям, которые можно делать всегда, не нарушая корректности). Перенесем сначала все задания, находящиеся не на своих местах, на отдельный процессор, не нарушая корректность. Затем, начиная с последнего, перенесем их на требуемые места. Так как по условию конечное расписание B корректно, эти операции также будут корректны. Итого получили цепочку из $2H(A, B)$ операций. Минимальная цепочка имеет длину не меньшую, чем эта. Утверждение доказано.

Примечание. Оценку из утверждения 2 в общем случае нельзя улучшить. Пусть в расписании A n перестановок, и каждая из них такова, что пока она не будет сделана, все остальные перестановки являются некорректными. Иными словами, n перестановок блокируют друг друга. Тогда нужно предварительно $n-1$ заданий перенести на отдельный процессор, и всего операций переноса получится $2n-1$. Устремляя n к бесконечности, получаем, что коэффициент в неравенстве из утверждения 2 может быть сколь угодно близок к 0.5.

Приведем пример для $n = 2$. Примеры для больших n могут быть построены по аналогии, но будут достаточно громоздкими. Граф программы и два расписания приведены на рисунке 4.2 (вершины, находящиеся на одной вертикали, соответствуют одному процессору).



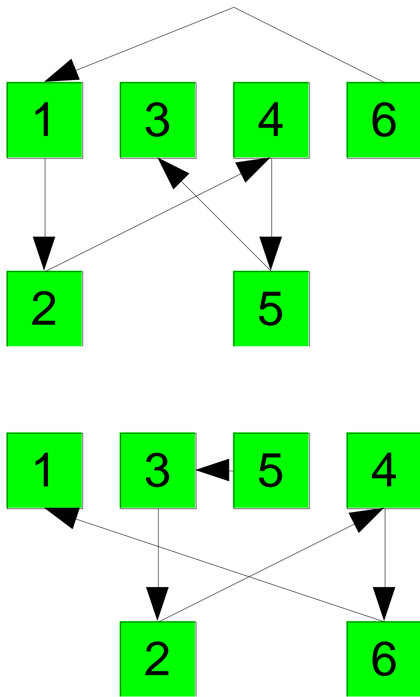


Рисунок 4.2. Пример, когда перестановки не соответствуют корректным операциям.

Некорректные расписания, получающиеся после устранения перестановок, приведены на рисунке 4.3.

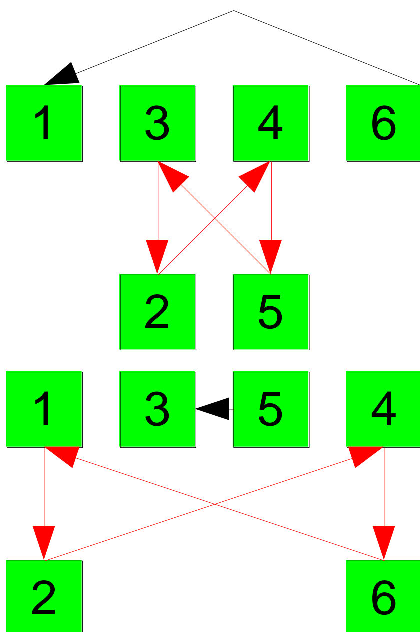


Рисунок 4.3. Пример, когда перестановки не соответствуют корректным операциям.

4.2.4 Оценка для L(A, B)

Пусть известно $H(A,B)$. Построим цепочку операций по следующему принципу. Сначала перебираются все перестановки, которые надо сделать. Если есть возможность сделать корректную операцию, то она выполняется. Иначе, задание переносится на отдельный процессор и устанавливается на свое место в самом конце. Длину полученной цепочки обозначим через $L'(A,B)$.

По построению $L'(A,B) \leq 2H(A,B)$. Из утверждения 2 следует, что $L(A,B) \leq L'(A,B)$. Итого получаем следующую цепочку неравенств: $H(A,B) \leq L(A,B) \leq L'(A,B) \leq 2H(A,B)$.

Следствие. Если $L'(A,B) = H(A,B)$, то $L(A,B) = H(A,B)$.

Это следствие можно использовать для оценки $L(A,B)$, так как на практике равенство может часто достигаться.

4.3 Экспериментальное исследование алгоритма

4.3.1 Оценка точности на модельных данных

4.3.1.1 Классификация исходных данных

Возможные частные задачи рассматриваемой задачи построения расписания можно классифицировать по следующим параметрам: N – число заданий, Q – соотношение между количеством ребер и количеством вершин в графе программы. Рассматриваются значения Q равное 1 (стандартное значение) и $Q \gg 1$ (экстремальное значение). Значение $Q = 0$ не будет рассматриваться, так как в этом случае задача фактически сводится к известной задаче о рюкзаке [27][64]. Рассмотрим еще два параметра классификации частных задач, которые определяются отношением директивного срока и оценками нижней границы времени выполнения программы и отношением требуемой надежности и оценками верхней границы надежности [4][6][106].

Нижнюю границу времени выполнения программы можно оценить, найдя в графе программы критический путь (путь, имеющий максимальное суммарное время выполнения входящих в него заданий). Эта граница не гарантирует существование решения, так как возможны задержки при выполнении заданий из этого пути из-за передачи данных. Если прибавить к суммарному времени выполнения критического пути суммарное время передачи данных для всех его заданий, то получится более реалистичная оценка нижней границы времени выполнения программы. Из-за возможных конфликтов на портах существование решения не гарантируется и в этом случае. Выделим следующие отношения нижних оценок времени выполнения программы к заданному директивному интервалу.

Невозможное: $0 < t^{dir} < L$, где L – длина критического пути. Данный вариант интересен только для исследования поведения алгоритма на заведомо неразрешимых задачах.

Жесткое: $t^{dir} \approx L + K$, где K – время передачи данных для всех заданий критического пути.

Нормальное: $t^{dir} \gg L + K$.

Заведомо выполнимое: $t^{dir} = +\infty$.

Верхнюю границу надежности системы можно оценить как произведение надежностей всех заданий при использовании максимального доступного числа версий в NVP. Будем считать ограничение на требуемую надежность жестким, если оно равно этой верхней границе. Нормальным ограничением будем считать ограничение, достижимое при наличии у каждого из процессоров не больше одного резерва и использования не более трех версий каждого задания.

Практический интерес представляют только ограничения, определенные выше как «нормальные» и, в меньшей степени, «жесткие». При заведомо выполнимых ограничениях алгоритм в силу своего построения найдет оптимальное расписание. Если же ограничения заведомо невыпол-

нимые, то частная задача заведомо поставлена некорректно, так как оптимального решения (как и вообще какого-либо, укладывающегося в ограничения) не существует, то есть анализ работы алгоритма не имеет смысла.

4.3.1.2 Проверка статистических гипотез

Для исследования алгоритма используется метод проверки статистических гипотез. В последующих разделах будут сформулированы гипотезы о работе алгоритма на выделенных классах исходных данных, и эти гипотезы будут проверены с заданным уровнем значимости [7][14].

Все рассматриваемые гипотезы являются гипотезами о значении параметров простых распределений, гипотез о виде распределений не выдвигается. Так как все эксперименты предполагаются независимыми, сумму исследуемых случайных величин можно по центральной предельной теореме аппроксимировать нормальным распределением. Следовательно, в данном случае правомерно использовать нерандомизированную критическую функцию, приближенно равную функции нормального распределения. Проверку такого типа называют -тестом.

Пусть известен тип распределения исследуемой случайной величины, но неизвестен конкретный параметр θ. Например, для распределения Бернулли, которое рассматривается далее, параметр есть вероятность выпадения 1. Проводится серия экспериментов, в результате каждого из которых получается некое наблюдаемое значение случайной величины. Если провести достаточно много экспериментов, то среднее арифметическое всех наблюдаемых величин должно приближаться к математическому ожиданию (закон больших чисел). Имеется некоторый уровень значимости b, для которого проверяется гипотеза. В результате, если провести много экспериментов и найти вероятность получить имеющийся результат, исходя из гипотезы, тогда если эта вероятность окажется меньше b, то это будет зна-

чить, что мы наблюдаем событие, которое имеет очень низкую вероятность в рамках гипотезы, то есть гипотеза признается неверной.

Формально, описанный выше алгоритм выглядит так:

1) Строится гипотеза вида «параметр $\theta = \theta_0$». Выбирается количество экспериментов и уровень значимости (в данной работе используются стандартные [3] для такого рода исследований значения $n = 100, b = 0.05$).
2) Производятся эксперименты и получается выборка X. Вычисляется выборочное среднее $\mathrm{p} = \sum_{\mathrm{i=1}}^{\mathrm{n}} X_i/n$.
3) Так как испытания независимы и проводятся в одинаковых условиях, применяется центральная предельная теорема, таким образом от неизвестного распределения $X(\theta)$ с $n = 100$ делается переход к стандартному нормальному распределению: $\frac{p-\mu}{\sqrt{\sigma-n)}} \sim N(0,1)$, где $\mu = \mathrm{EX}(\theta_0), \sigma = \mathrm{DX}(\theta_0)$.
4) Строится альтернативная гипотеза, которая задает значения параметра θ, которые нас не устраивают. В результате критическая область K имеет вид либо $(-\infty, -X) \cup (X, +\infty)$ либо $(-\infty, -X)$, где X – это такое число, что $P(N(0,1) \notin K) = b$. Непосредственное значение вычисляется напрямую с использованием функции Гаусса: $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.
5) Вычисляется значение статистики $\frac{p-\mu}{\sqrt{\sigma-n)}}$. Если оно не попадает в критическую область, то гипотеза считается доказанной.

4.3.1.3 Оценка точности различных стратегий применения операций

Для оценки точности различных стратегий было проведено около 10000 экспериментов с различными исходными данными. Число заданий варьировалось от 25 до 200 с шагом 5, также варьировались значения Q и

t^{dir} для каждого значения числа заданий создавался граф программы, на нем алгоритм запускался поочередно с каждой из стратегий $n = 100$ раз. Число итераций алгоритма каждый раз задавалось одинаковым.

Пусть Res_i – значение целевой функции (количество процессоров), получаемое алгоритмом с использованием стратегии Si. Под результатом работы алгоритма в дальнейшем понимается расписание, на котором достигается наилучшее значение целевой функции. Для значений $n = 100, b = 0.05$ проверялись статистические гипотезы следующего вида:

- Результат стратегии Si лучше результатов стратегии Sj при любых t^{dir}, R^{dir} и Q: $Res_i < Res_j$;
- При любых t^{dir}, R^{dir} и Q выполняется некоторое соотношение на разницу результатов: $Res_i - Res_j \leq c$;
- Локальная оптимальность: если A – расписание, построенное алгоритмом, то $\forall B: L(A,B) = 1: M(B) \geq M(A)$, то есть результат, найденный алгоритмом, не может быть улучшен применением одной операции. Теоретически такая ситуация возможна, но так как в реальности она не возникает, это означает, что количество итераций алгоритма достаточно по крайней мере для поиска локального оптимума.

На графиках (Рисунок 4.4-4.8) приведены результаты сравнения четырех стратегий: стратегия уменьшения задержек (S1), стратегия заполнения простоев (S2), смешанная стратегия (S3), случайная стратегия (S0). Первые три графика (Рисунок 4.4-4.6) показывает среднее значение целевой функции в зависимости от числа заданий. Результаты для смешанной стратегии и стратегии уменьшения задержек почти одинаковые, стратегия заполнения простоев и случайный поиск заметно хуже.

Зависимость результата работы алгоритма от числа заданий не является монотонной функций из-за случайности генерируемых исходных данных алгоритма: при случайном создании графов программ нельзя гарантировать, что существует решение с заданным числом процессоров, поэтому возможна ситуация, когда программа из n заданий требует больше процессоров, чем программа из $n + 5$ заданий.

Графики на рисунках 4.7-4.8 показывают попарное сравнение стратегий. Для каждой пары стратегий вычислялась разница значений целевых функций (числа процессоров). Графики показывают число экспериментов, на которых наблюдается то или иное значение этой разницы. Данные графики являются наглядным подтверждением статистической значимости результатов, проиллюстрированных на рисунках 4.4-4.6, так как разница значений целевых функций имеет распределение, близкое к нормальному.

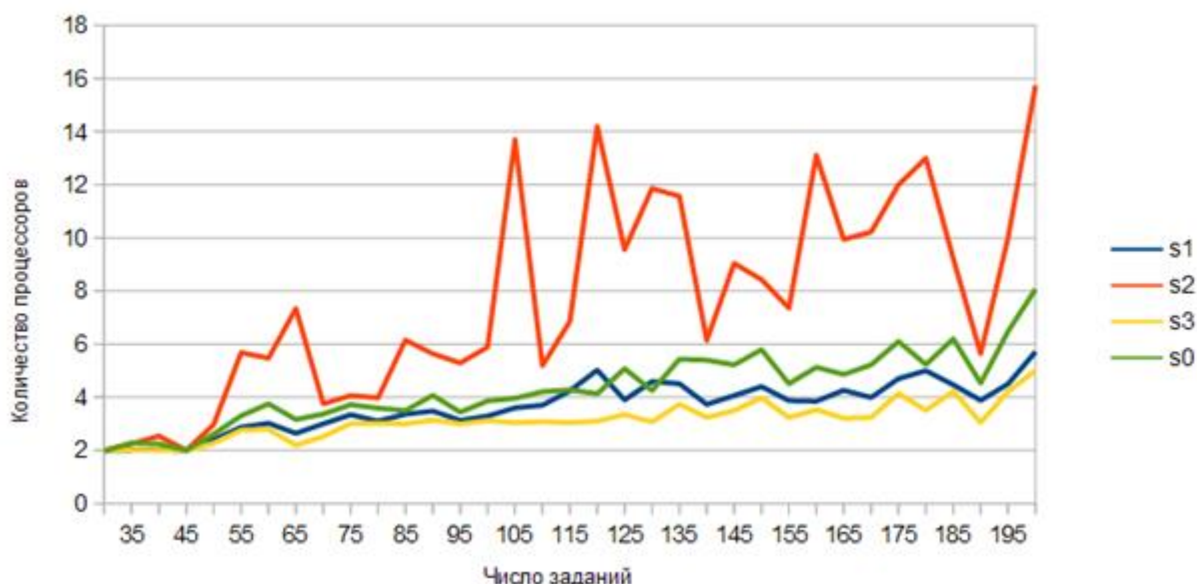


Рисунок 4.4. Среднее значение целевой функции в зависимости от числа заданий для четырех стратегий, $Q \gg 1$.

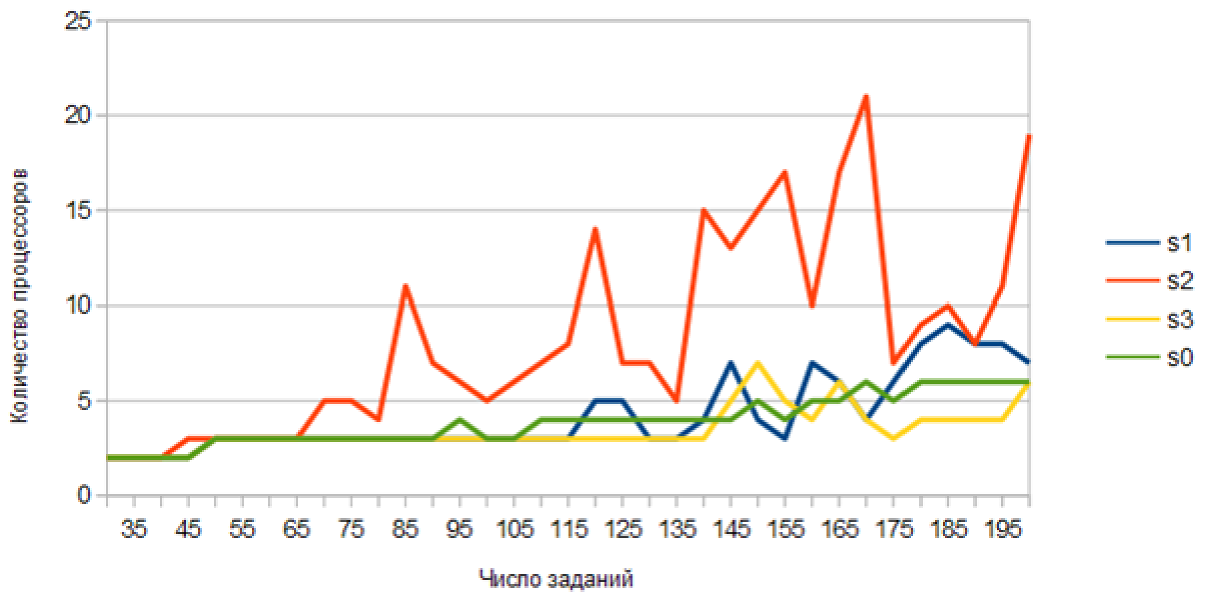


Рисунок 4.5. Среднее значение целевой функции в зависимости от числа заданий для четырех стратегий, $Q = 1$.

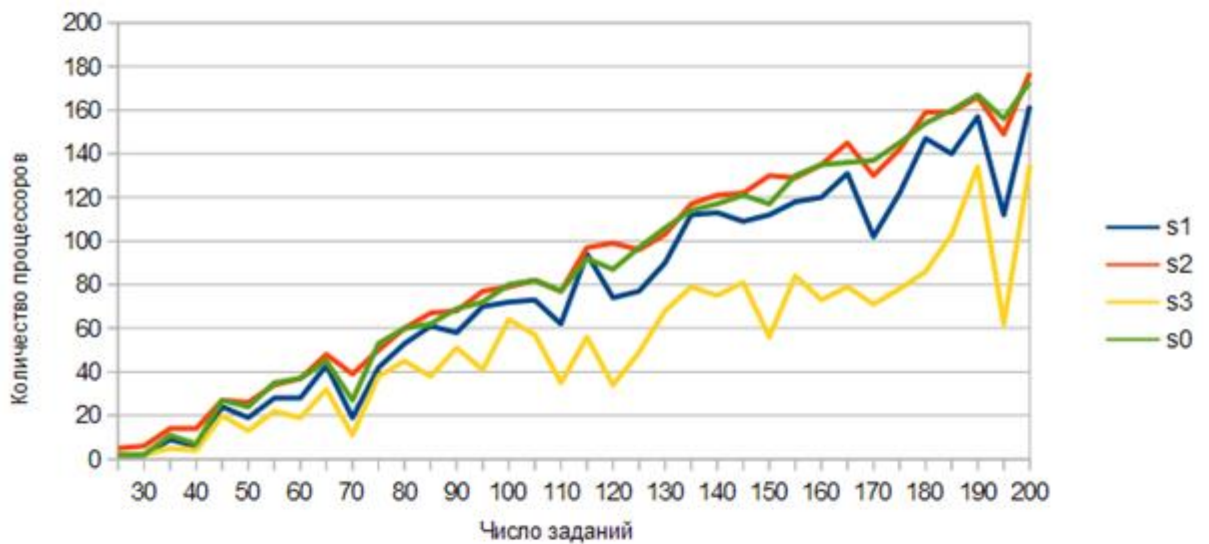


Рисунок 4.6. Среднее значение целевой функции в зависимости от числа заданий для четырех стратегий, $Q = 1$, ограничение на время жесткое.

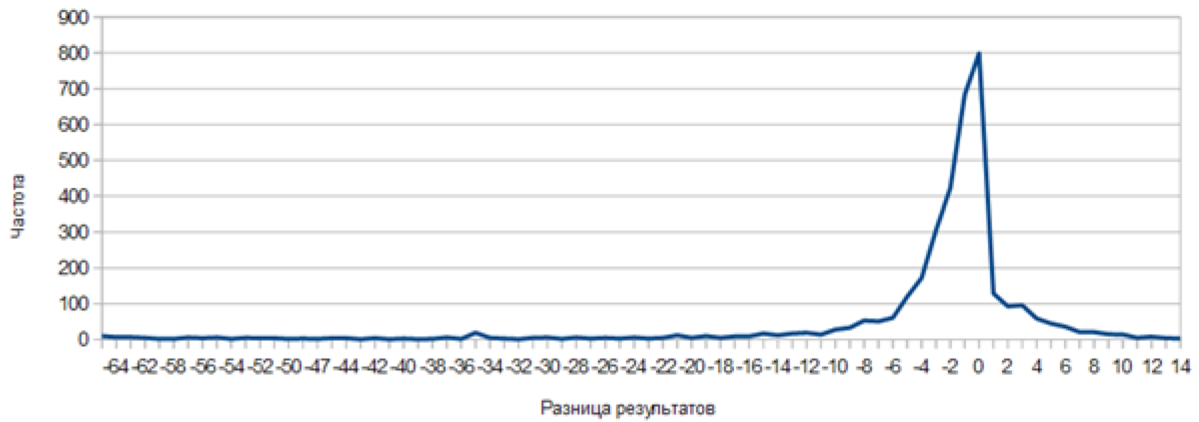


Рисунок 4.7. Сравнение стратегии уменьшения задержек и стратегии заполнения простоев (по оси OX значение $Res_1 - Res_2$).

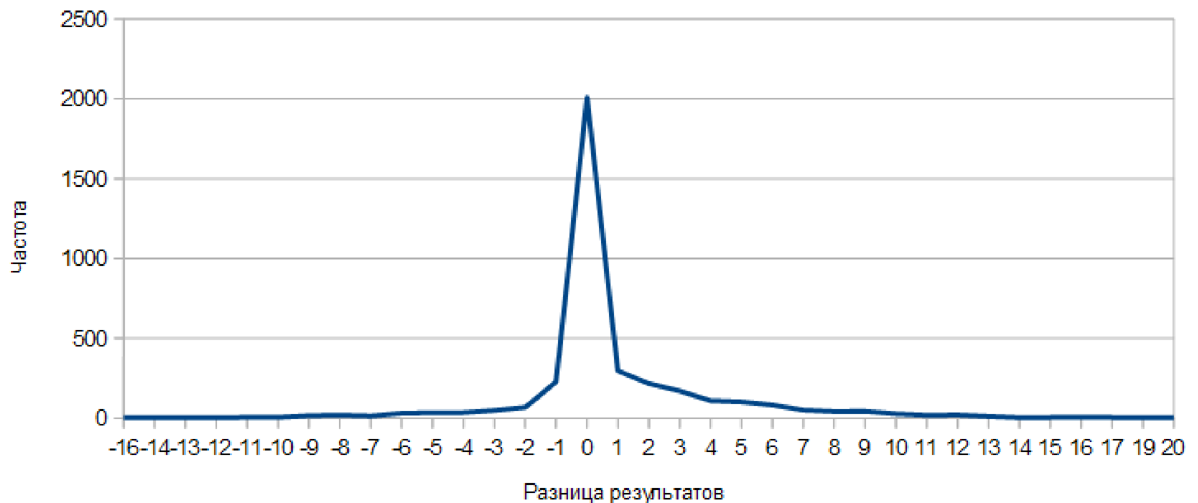


Рисунок 4.8. Сравнение стратегий уменьшения задержек и смешанной стратегии (по оси OX значение $Res_1 - Res_3$).

Следующие гипотезы имеют место при размере выборки $n = 100$ и уровне значимости $b = 0.05$:

- Результат стратегий S1 и S3 лучше результатов стратегий S2 и S0 при любых t^{dir}, R^{dir} и Q : $Res_1 > Res_2, Res_1 > Res_0, Res_3 > Res_2, Res_3 > Res_0$;
- При любых t^{dir}, R^{dir} и Q: $Res_3 - Res_1 \leq 1$;

- Локальная оптимальность алгоритма (имеет место в 100% случаев на рассмотренных выборках).

В таблице 4.2 приведены значения дисперсии (среднеквадратичного отклонения от среднего значения) результатов работы алгоритма при разных стратегиях. Высокая дисперсия у стратегии заполнения простоев означает, что она также может давать качественные результаты, как стратегии S1 и S3, но работает нестабильно.

Таблица 4.2. Среднеквадратичное отклонение для каждой из четырех стратегий.

Число заданий	S1	S2	S3	S0
30	0.00	0.00	0.00	0.30
35	0.00	0.49	0.10	0.91
40	0.00	1.19	0.14	0.68
45	0.00	0.00	0.00	0.00
50	0.45	0.14	0.49	0.49
55	0.45	2.67	0.46	0.92
60	0.43	3.03	0.51	2.10
65	0.39	3.14	0.67	0.93
70	0.50	1.37	1.03	1.24
75	0.10	0.45	0.54	0.99
80	0.20	0.72	0.30	0.65
85	0.00	1.42	0.66	0.63
90	0.38	2.18	0.58	0.81
95	0.00	0.84	0.37	0.56
100	0.33	3.95	0.82	0.75
105	0.20	5.75	1.60	0.78
110	0.27	1.07	0.77	0.70

115	0.20	0.69	1.13	0.77
120	0.30	2.40	2.49	0.76
125	0.58	6.62	1.09	2.70
130	0.26	0.98	1.72	0.82
135	1.12	8.22	2.29	1.49
140	0.49	2.42	0.92	1.56
145	0.61	4.95	1.00	1.05
150	0.99	3.44	2.92	1.28
155	0.43	5.39	0.76	0.79
160	0.64	7.37	0.74	1.15
165	0.41	4.60	1.42	0.75
170	0.43	8.19	0.83	1.34
175	1.02	10.34	1.79	1.57
180	0.78	11.25	2.03	0.93
185	1.45	4.69	2.14	1.61
190	0.22	0.58	0.64	0.59
195	0.80	21.19	2.47	1.27
200	4.16	12.41	4.88	2.43

4.3.1.4 Оценка скорости работы алгоритма

График на рисунке 4.9 показывает число итераций алгоритма, которое потребовалось для достижения наилучшего решения. В каждом эксперименте выполнялось $10N$ итераций (N – число заданий), но с определенного момента текущее приближение не улучшалось. Номер итерации, на котором было достигнуто наилучшее решение, показан по оси OY на рисунке 4.9. Эксперименты показывают, что скорость стратегий S1 и S3 практически одинаковая, S3 лишь незначительно быстрее. Стратегия S2 значительно быстрее, но это объясняется тем, что точность ее результатов гораздо ниже, значит, и число итераций, необходимое, чтобы их достичь, ниже.

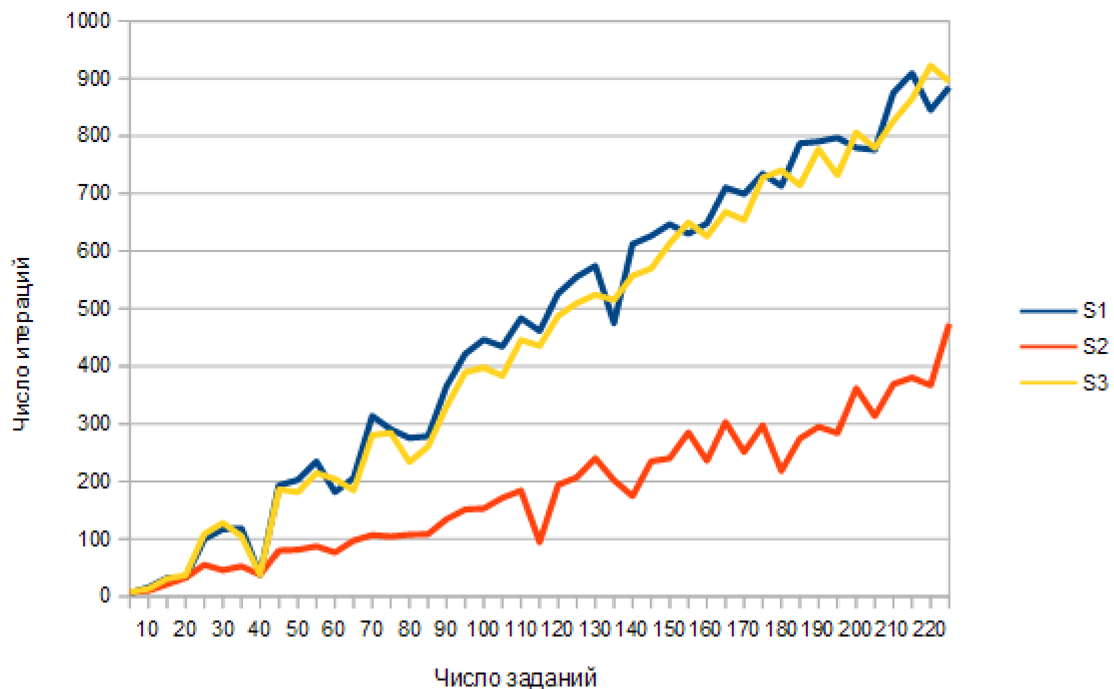


Рисунок 4.9. Сравнение скорости трех стратегий.

4.3.2 Оценка точности на совместимых исходных данных

4.3.2.1 Алгоритм генерации совместимых исходных данных

Помимо случайным образом созданных исходных данных, алгоритм тестировался на специальных совместимых исходных данных, то есть таких, для которых заранее известно оптимальное решение задачи определения минимального числа процессоров. Результат работы алгоритма сравнивался с оптимальным решением с помощью метрики $H(A, B)$, описанной в разделе 4.2.

Совместимые исходные создавались по следующей схеме.

1. Фиксируются число заданий N, число ребер E, число процессоров M и директивный срок t^{dir}. Функция интерпретации соответствует модели коммутатора.
2. Случайным образом выбираются N чисел, которые можно разбить на M групп, так, что сумма чисел в каждой группе будет равна t^{dir}. Число чисел во всех группах равно $[N/M]$, в остав-

шейся группе – $N - \left[\frac{N}{M}\right] \cdot M$. В каждой группе все числа, кроме одного, выбираются случайно, оставшееся подбирается так, чтобы сумма была равна t^{dir}. Эти числа соответствуют времени выполнения каждого из N заданий, каждая группа чисел соответствует заданиям, выполняющимся на одном процессоре.

3. Далее создаются ребра. Ребра добавляются случайным образом либо между заданиями, расположенными на одном процессоре (в таком случае объем передаваемых данных может быть выбран произвольно), либо между любыми двумя другими заданиями, при условии, что объем передаваемых данных не вызовет задержек в выполнении расписания. Число ребер и, соответственно, значение отношения числа ребер к числу заданий Q могут варьироваться.

Значение R^{dir} устанавливается так, чтобы для его соблюдения было достаточно M процессоров.

Построенное таким способом расписание будет оптимальным для соответствующих исходных данных, так как при меньшем числе процессоров директивный срок не может быть соблюден.

4.3.2.2 Результаты работы алгоритма на совместимых исходных данных

Было проведено 15000 экспериментов с различными исходными данными (число заданий варьировалось от 10 до 150, Q варьировалось от 0 до 2), на каждом алгоритм запускался поочередно с каждой из стратегий. Результат работы алгоритма сравнивался с оптимальным расписанием по метрике $H(A, B)$.

На рисунках 4.10 и 4.11 приведены результаты.

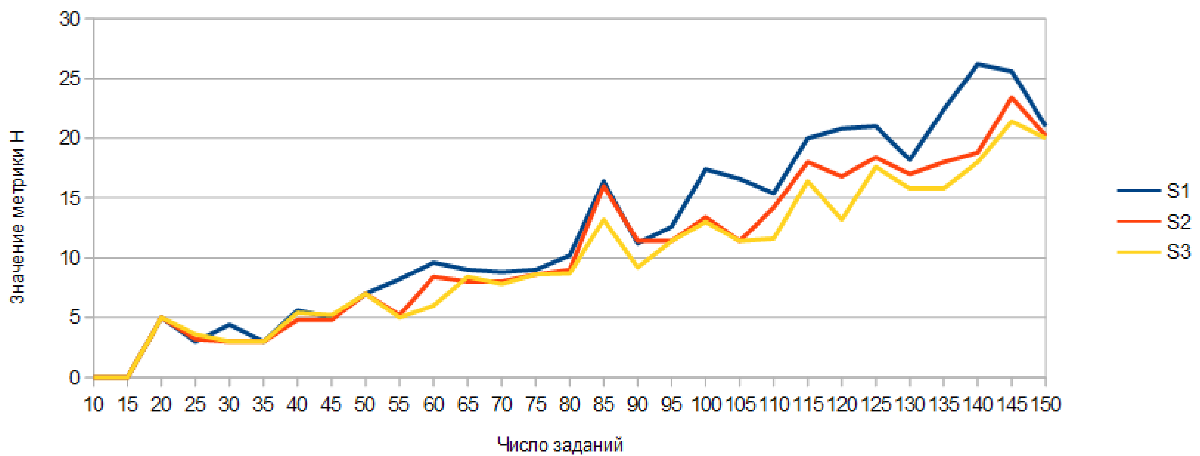


Рисунок 4.10. Значение метрики H между оптимальным решением и решением, полученным алгоритмом, при высоком Q.

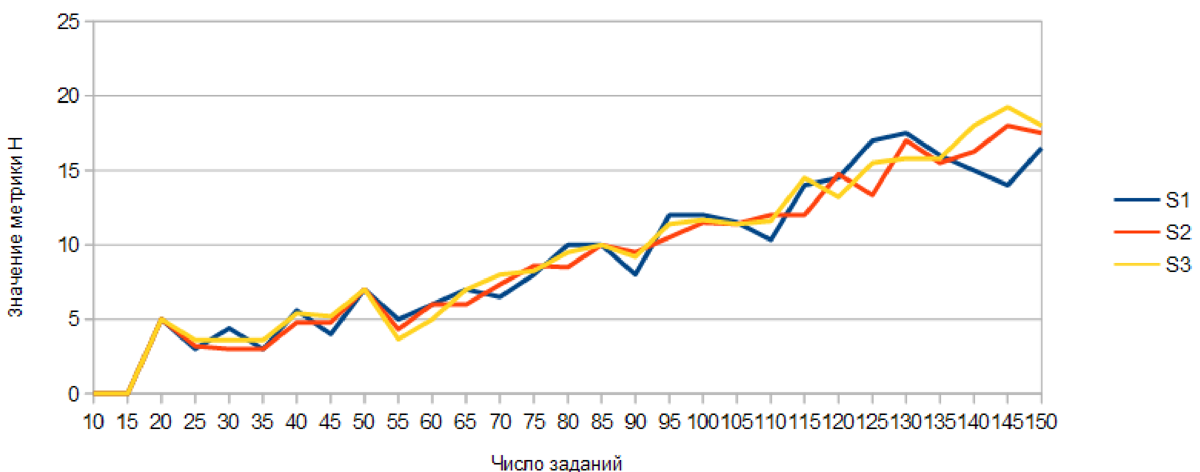


Рисунок 4.11. Значение метрики H между оптимальным решением и решением, полученным алгоритмом, при низком Q.

Различия в работе стратегий проявляются с ростом значения Q. При малых Q значение метрики H не прямо пропорционально качеству работы алгоритма, так как в этом случае существуют множество оптимальных решений, которые могут сильно отличаться друг от друга. Так, если $Q = 0$, то в оптимальном расписании, созданном по алгоритму из подраздела 4.3.2.1, можно переставить местами любые два задания, и получить корректное расписание с таким же временем выполнения, то есть всего опти-

мальных расписаний для такой задачи будет $(n/2)!^2$. Чем больше в графе программы ребер, тем меньше возможных перестановок, и, соответственно различных оптимальных решений. Соответственно, при малых Q различия в работе стратегий не проявляются, тогда как с ростом Q стратегия S3 показывает лучшие результаты (Рисунок 4.10).

Кроме того, следует отметить, что на рассматриваемых примерах стратегия S2 работает лучше, чем на случайно созданных исходных данных. Если в предыдущем подразделе было выяснено, что значения Res_2 имеют высокую дисперсию, то значения метрики $H(A, B)$ для стратегии заполнения простоев на совместимых исходных данных имеют близкие значения для всех трех стратегий.

4.3.3 Сравнение законов понижения температуры

В данном подразделе приведены результаты экспериментальных исследований точности и вычислительной сложности алгоритмов при использовании различных законов понижения температуры. Было проведено 1500 экспериментов с различными исходными данными (число заданий варьировалось от 10 до 150), на каждом алгоритм запускался поочередно с каждым из законов понижения температуры.

Для каждой пары законов понижения температуры значения целевой функции (число процессоров), полученные на одних и тех же исходных данных, вычитались одно из другого. Число итераций алгоритма для каждого закона понижения температуры задавалось одинаковым. Диаграммы (рисунки 4.12-4.14) показывают количество экспериментов, на которых наблюдалось то или иное значение этой разности среди тех экспериментов, где число заданий было равно 150.

Рисунок 4.12. Разница в точности: Больцман – смешанный.

Рисунок 4.13. Разница в точности: Больцман – Коши.

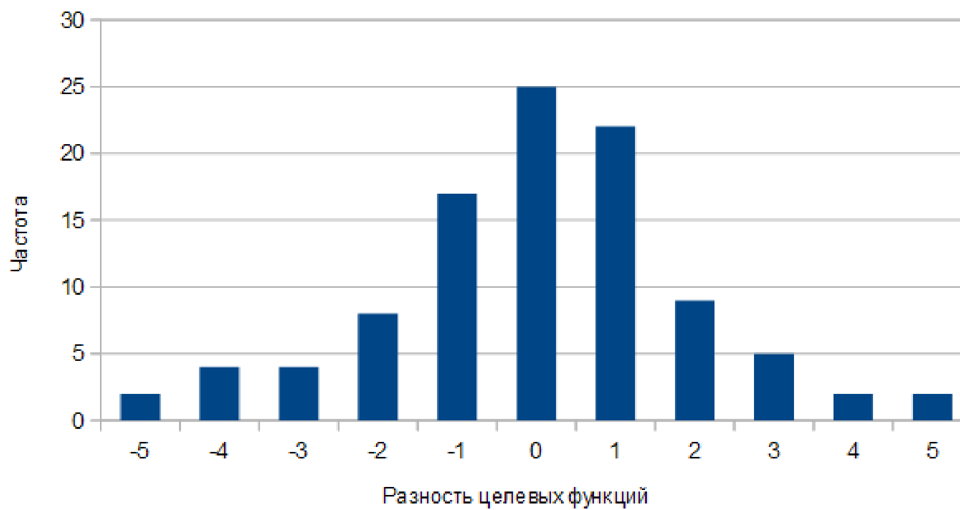


Рисунок 4.14. Разница в точности: Коши – смешанный.

Смешанный закон имеет небольшое преимущество перед законами Больцмана и Коши.

На рисунке 4.15 приведено поведение алгоритма при различных начальных значениях температуры. Исследование проводилось для фиксированного числа заданий, равного 150.

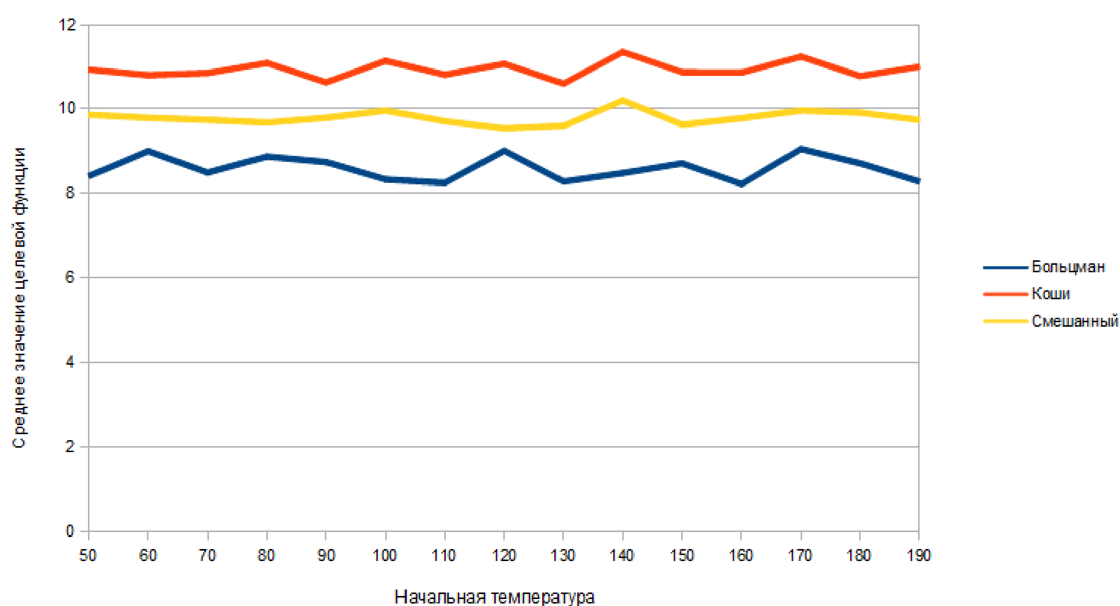


Рисунок 4.15. Зависимость точности от начальной температуры.

Видно, что изменения начального значения температуры в пределах от 50 до 200 мало влияет на качество получаемых решений.

На диаграммах (рисунки 4.16-4.17) показаны результаты сравнения скорости работы алгоритма при разных режимах понижения температуры. По оси X на графике приведена скорость работы алгоритма, определяемая как разница (в процентах) числа итераций, которые потребовались алгоритму для достижения одного и того же значения целевой функции. Например, если с законом Больцмана было найдено решение с 10 процессорами за 100 итераций, а с законом Коши решение с 10 процессорами было достигнуто за 110 итераций, это означает, что второй закон дает резуль-

тат на 10 % медленнее. По вертикальной оси указано число экспериментов, на которых был достигнут результат.

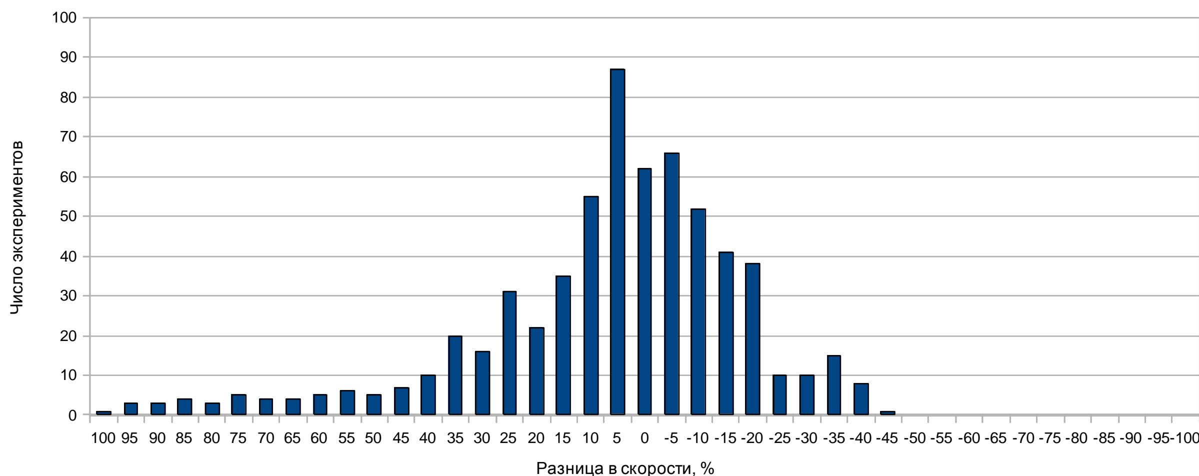


Рисунок 4.16. Сравнение скорости законов Больцмана и смешанного для случая, когда смешанный дает лучший результат.

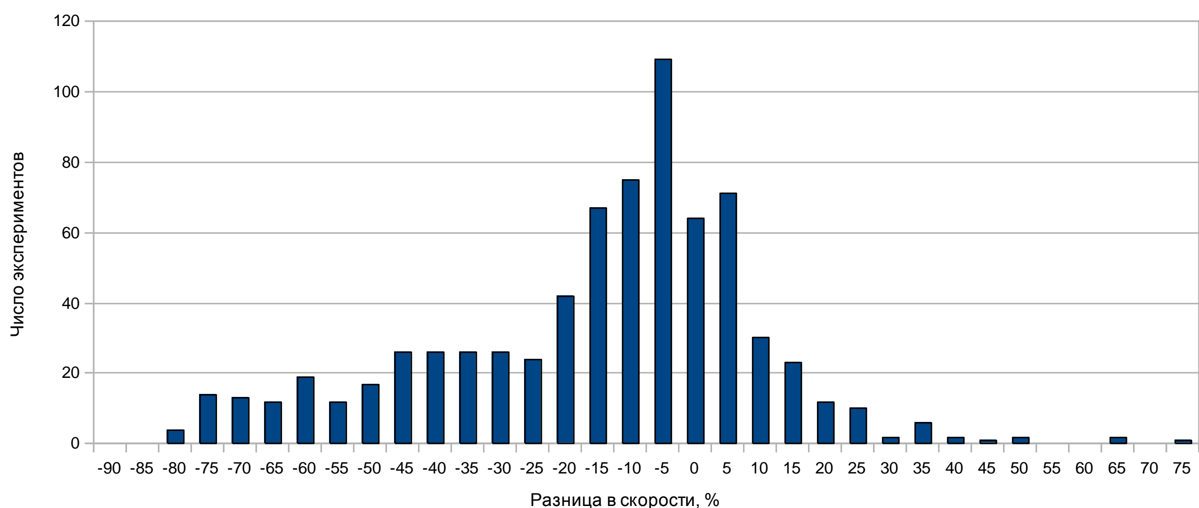


Рисунок 4.17. Сравнение скорости законов Больцмана и смешанного для случая, когда Больцман дает лучший результат.

Напомним, что исходя из результатов, полученных выше, смешанный закон дает несколько лучшие результаты, чем закон Больцмана. Однако в значительном количестве экспериментов результат при смешанном законе

получился лучше, поэтому отдельно приведены диаграммы для обоих случаев.

Итак, когда результат, получаемый алгоритмом с законом Больцмана, точнее, он получается быстрее в среднем на 18,23 % (с дисперсией 24,79). Когда же смешанный закон дает более точные результаты, они получаются быстрее в среднем на 11,28 % (с дисперсией 44,66). Под дисперсией понимается среднеквадратичное отклонение разницы в скорости работы алгоритма при двух разных режимах понижения температуры от её среднего значения.

4.4 Выводы

В данном разделе приведены результаты теоретического и экспериментального исследования свойств алгоритма имитации отжига для решения задачи структурного синтеза вычислительных систем, предложенного в разделе 3.

Доказана асимптотическая сходимость алгоритма. Предложены метрики для сравнения расписаний. Проведены эксперименты с алгоритмом, которые показали: эффективность использования направленных стратегий по сравнению со случайным поиском, превосходство стратегии уменьшения задержек и смешанной стратегии, более низкую дисперсию значений целевой функции у стратегии уменьшения задержек, преимущество использования закона Больцмана для понижения температуры.

5 Инструментальная система

5.1 Требования к системе

На основе разработанных алгоритмов построено программное средство, позволяющее решать задачу структурного синтеза в режиме диалога с пользователем. Приведем основные требования, которым должна удовлетворять такая система:

- Наличие графического интерфейса пользователя.
- Возможность ввода исходных данных как из заранее созданного файла, так и с использованием графического интерфейса.
- Возможность запускать алгоритм, менять его настройки, визуализировать на экране результаты.
- Возможность править результаты алгоритма в ручном режиме. При изменении решения автоматически должна проверяться его корректность.
- Возможность задавать различные модели вычислительной системы для оценки времени и различные методики расчета надежности.
- Возможность создавать специализированные подсистемы для построения вычислительных систем для конкретных практических задач.
- Возможность генерации кода процедур обмена.

5.2 Описание системы

5.2.1 Описание архитектуры системы

Библиотека структур данных и алгоритмов решения задачи структурного синтеза написана на языке Python и совместима как с версией 2.X [80], так и с версией 3 (Python 3000) [81]. Исходный код разделен на несколько

пакетов, соответствующих различным группам классов: структурам данных и методам работы над ними.

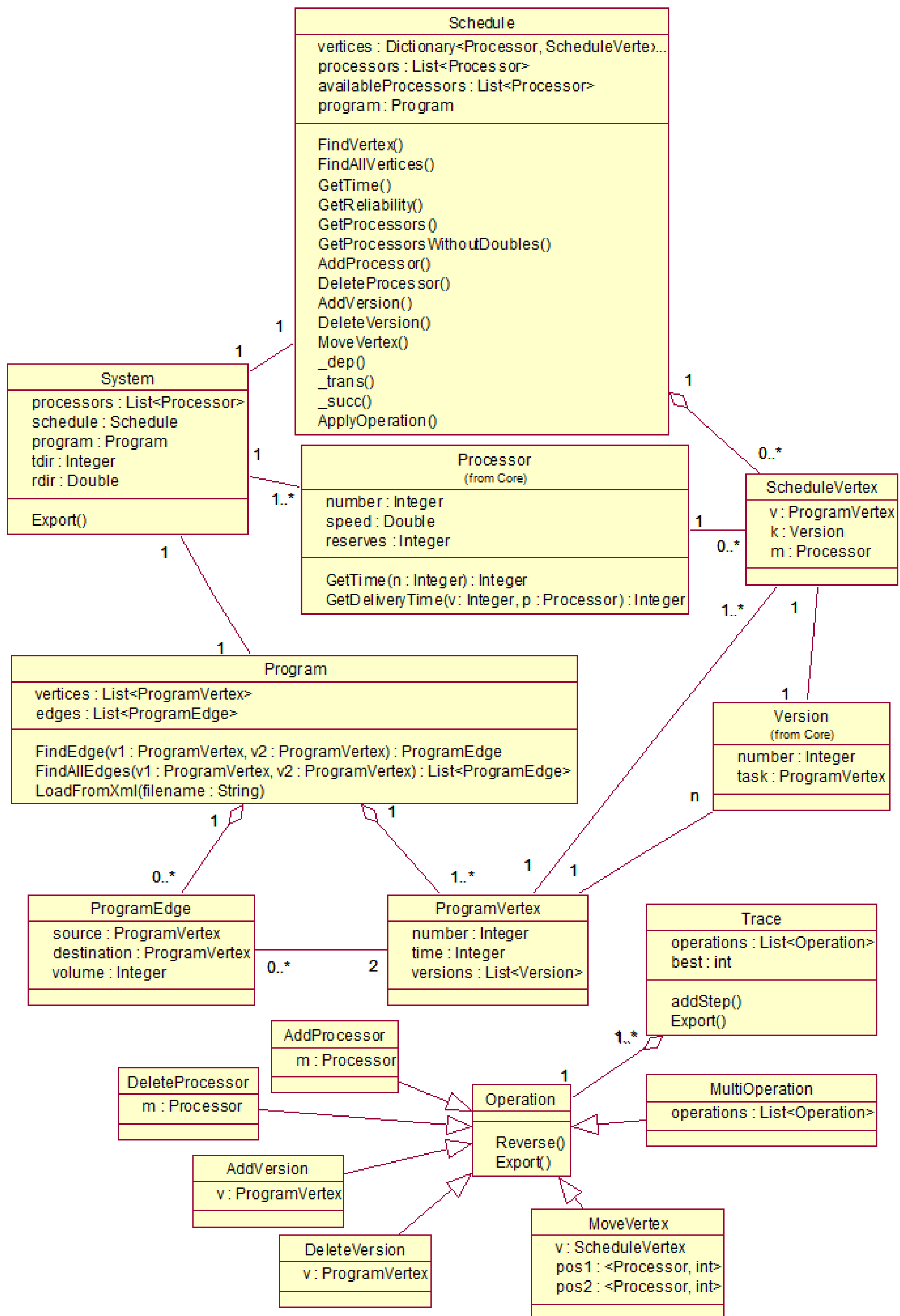


Рисунок 5.1. Диаграмма классов пакета Schedules.

На рисунке 5.1 приведена UML-диаграмма классов [3][75][85] пакета Schedules. В этом пакете реализованы структуры данных, описанные в постановке задачи: графы программ, расписания, наборы процессоров, операции над расписаниями. Поясним отношения между классами, указанные на диаграмме.

Программа (Program) состоит из списка вершин (ProgramVertex) и списка ребер (ProgramEdge). Каждое ребро содержит ссылки на две вершины и хранит объем данных, передающихся между соответствующими заданиями. Каждая вершина имеет номер и хранит данные о времени выполнения и списке доступных версий (Version). Каждая версия содержит ссылку на задание, которое она реализует, номер и показатели надежности.

Расписание (Schedule) создается для заданной программы и списка доступных процессоров (availableProcessors), при этом реально использующиеся процессоры (возможно, однотипные), хранятся в списке processors. Расписание состоит из списка вершин (ScheduleVertex), соответствующих четверкам , как в постановке задачи. Класс процессора (Processor) инкапсулирует данные о надежности и количестве резервных копий процессора.

Программно-аппаратная система (System) – это заданная программа, расписание для нее, набор используемых процессоров (в решаемой задаче это процессор одного типа, для которого доступно неограниченное число копий, но потенциально возможно задание и нескольких типов) и константные ограничения на время выполнения и надежность.

Класс Program предоставляет методы для загрузки исходных данных из конфигурационного файла в формате XML [24] (описание используемого формата приведено далее) и поиска ребер, содержащих некоторые вершины.

Класс Processor содержит методы для расчета времени выполнения заданий и передачи данных. Как указано в постановке задачи, эти величины

являются константами, но потенциально можно реализовать расчет времени выполнения или передачи в зависимости от процессора.

Класс Schedule реализует все описанные в постановке задачи операции над расписаниями, расчет времени, надежности и числа процессоров, а также содержит служебные функции, некоторые из которых также были описаны в постановке, использующиеся для проверки корректности операций.

Класс Operation представляет собой абстракцию операции над расписанием. От него наследуются классы-реализации конкретных операций. В каждом из них хранятся параметры соответствующей операции. Класс MultiOperation представляет собой контейнер для последовательности операций.

В пакете Core (рисунок 5.2) реализованы основные абстракции, с которыми работают программы в данной работе: программа, устройство, метод обеспечения надежности.

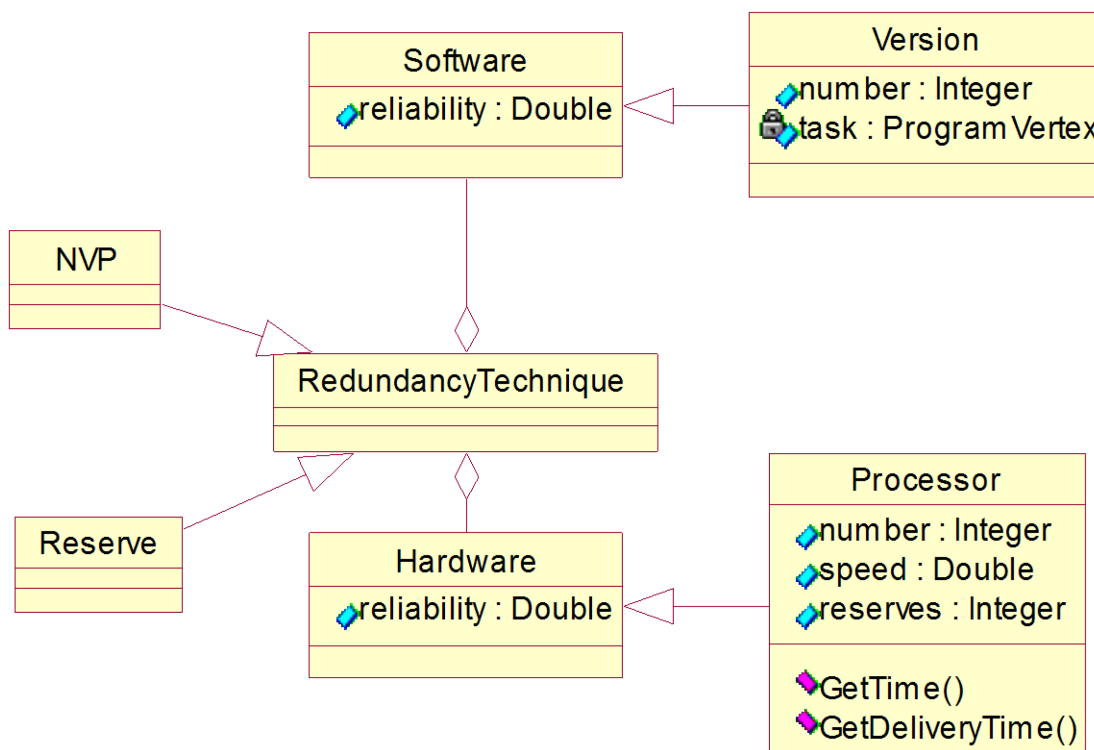


Рисунок 5.2. Диаграмма классов пакета Core.

Программа (Software) и аппаратура (Hardware) имеют некоторую надежность (поле reliability). Версия программы (Version) – частный случай программы, процессор (Processor) – частный случай аппаратуры. У них есть свои специальные характеристики, используемые в пакете Schedules. Метод обеспечения надежности (RedundancyTechnique) использует несколько объектов типа Software и Hardware и на основании их вычисляет надежность. Конкретные реализации методов обеспечения надежности: резервирование (Reserve) и многоверсионное программирование (NVP). NVP работает по-разному в зависимости от количества резервных версий программы.

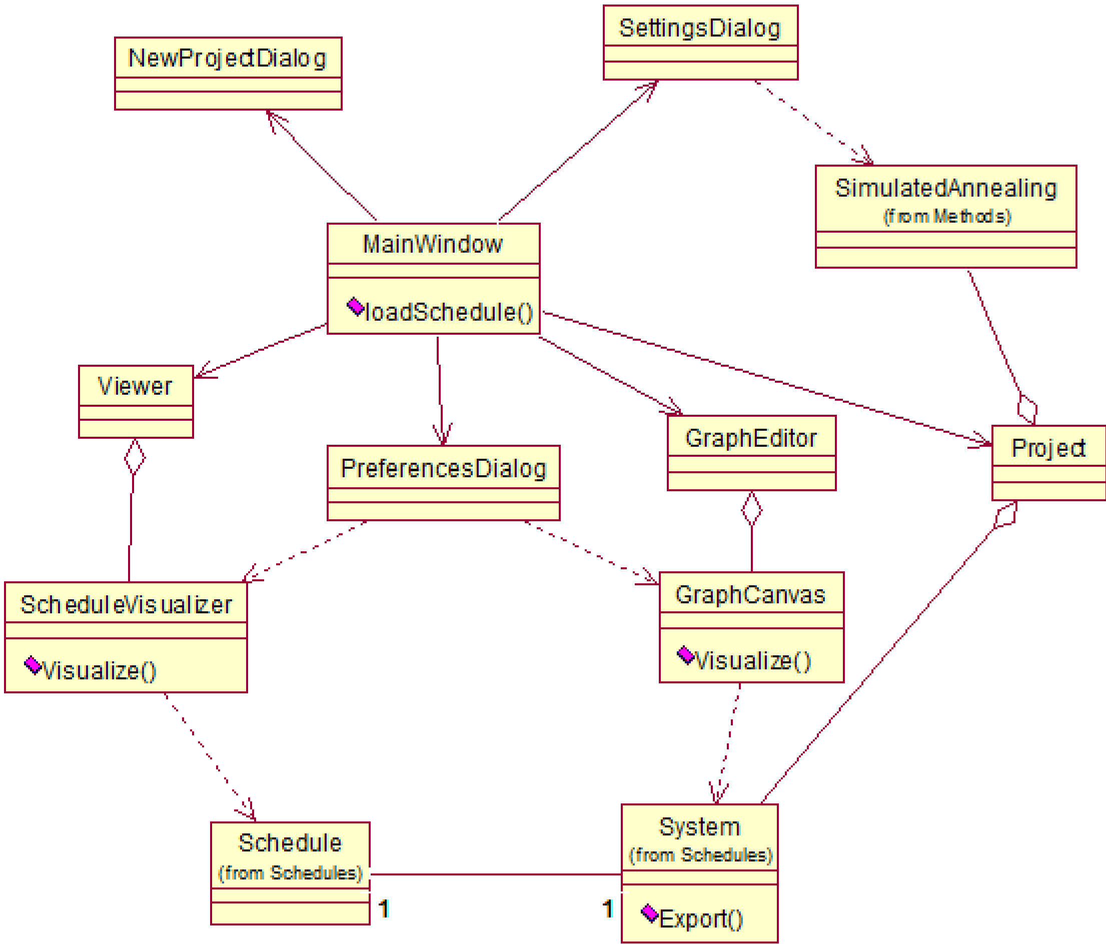


Рисунок 5.3. Диаграмма классов графического пользовательского интерфейса.

Исходный код созданного графического пользовательского интерфейса (ГПИ) написан на языке программирования Python 3.1 с использованием библиотеки Qt версии 4.4 [84]. Широкая распространенность и поддержка указанного языка и библиотек позволяют обеспечить кроссплатформенность созданного интерфейса.

Кроме функции main, в которой вызывается создание основного окна, в состав исходного кода ГПИ входят классы, изображенные на рисунке 5.3.

Основной класс в пакете Methods – имитация отжига (SimulatedAnnealing). Он получает в конструкторе систему (System) и конфигурационный файл с настройками. Есть возможность как заставить работать алгоритм до конца (Start()), так и выполнять по одному шагу (Step()).

Главное окно MainWindow унаследовано от QMainWindow. В нем, помимо обработчиков событий, содержатся текущий проект (Project) и список расписаний, полученных на каждой из итераций (ScheduleContainer). Когда выполняется некоторая операция с расписаниями (шаг алгоритма, его отмена, переход на некоторый шаг), из класса Project извлекается информация о текущем расписании и его характеристиках (время, надежность и так далее), она передается в ScheduleContainer, там, возможно, меняется указатель на текущее расписании, после чего вызывается функция loadSchedule(), обновляющая главное окно в соответствии с текущим расписанием. Класс ScheduleVisualizer унаследован от QWidget, и в нем происходит отрисовка графа текущего расписания, после чего он помещается в прокручиваемую область в центре главного окна.

Также имеется ряд классов, реализующих различные диалоговые окна: PreferencesDialog – окно настроек программы, SettingsDialog – окно настроек алгоритма. Отдельно остановимся на SettingsDialog. Это окно не имеет фиксированного дизайна и создается автоматически по словарю параметров, извлекаемому из экземпляра класса алгоритма. Это сделано для

того, чтобы была потенциальная возможность создать несколько алгоритмов и выбирать нужный из графического интерфейса.

5.2.2 Графический пользовательский интерфейс

Графический пользовательский интерфейс, построенный на основе программной библиотеки, представляет собой следующий набор взаимосвязанных окон: основное рабочее окно ГПИ, окно для просмотра и редактирования исходных данных, окно для просмотра и редактирования построенных расписаний.

Основное окно ГПИ служит для выполнения всех действий, необходимых в процессе работы с программой: ввод исходных данных, запуск алгоритма, просмотр результатов работы алгоритма, экспорт результатов в различные форматы (внутренний бинарный формат, исходный код процедур обмена, XML для средств имитационного моделирования). Действия представлены пунктами в меню, кнопками на панели инструментов и кнопками в самом окне.

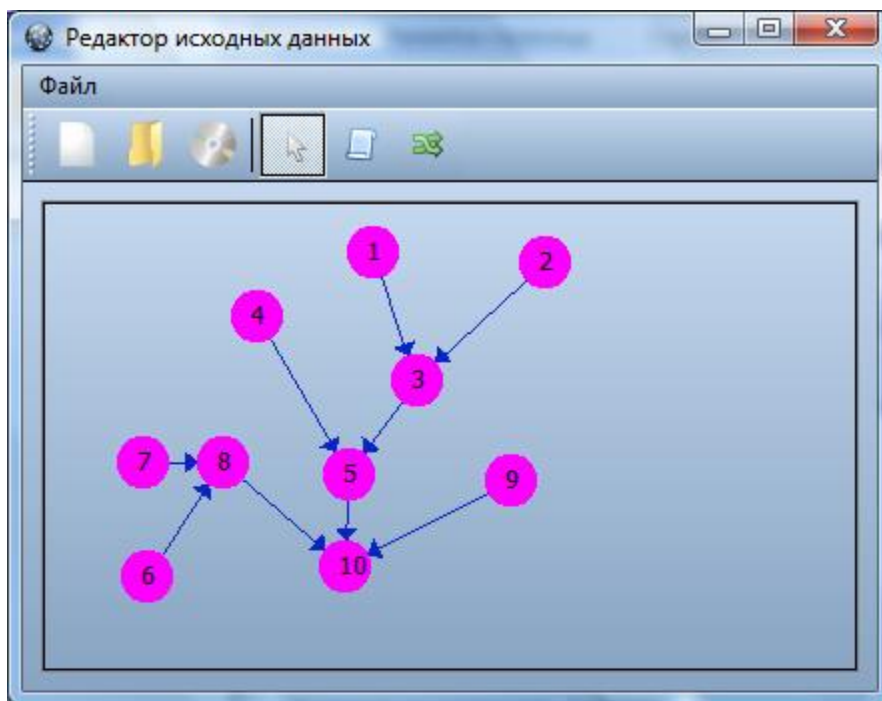


Рисунок 5.4. Окно редактора исходных данных.

Окно редактора исходных данных (рисунок 5.4) позволяет редактировать граф потока данных программы, для которой строится расписание. Исходные данные могут быть сохранены отдельно от проекта в виде XML-файла, который можно редактировать вручную.

В редакторе расписаний (рисунок 5.5) визуализируются расписания, найденные алгоритмом. До запуска алгоритма там доступно только начальное приближение, после появляется возможность просмотреть промежуточные результаты на каждой итерации алгоритма.

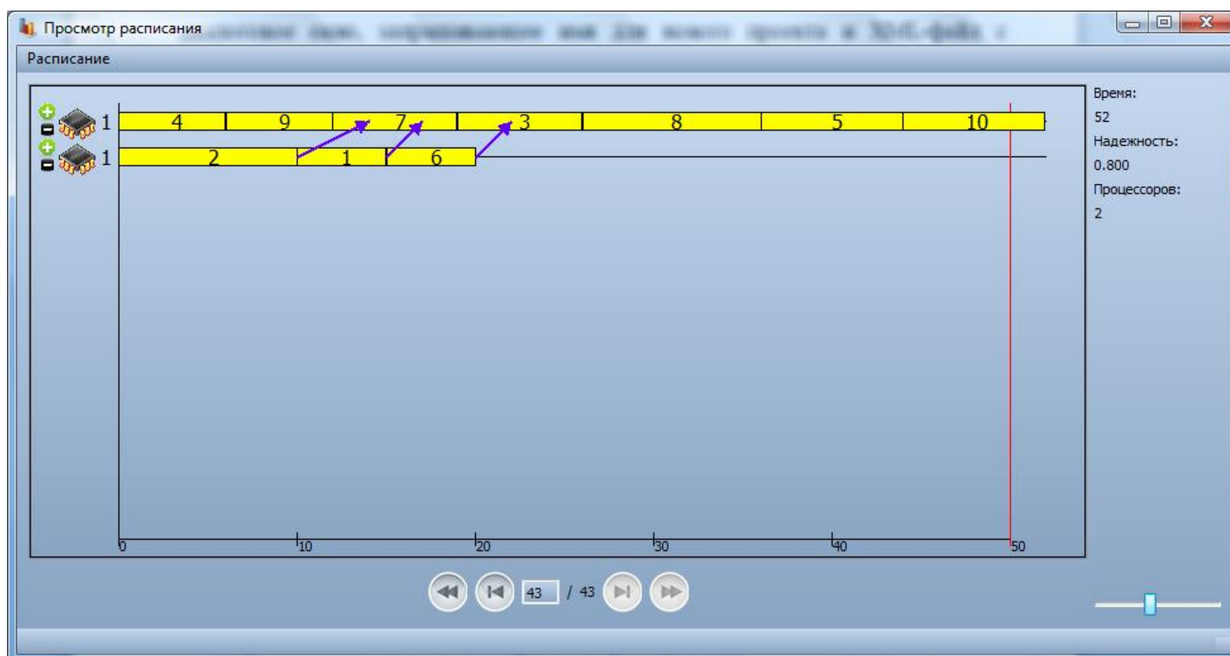


Рисунок 5.5. Окно редактора расписаний.

Ось времени изображена и размечена внизу основного поля; она направлена вправо, директивный срок обозначен красной линией. Горизонтальные линии соответствуют процессорам, прямоугольники – заданиям, стрелки – передачам данных. Координаты прямоугольников и стрелок точно соответствуют моментам времени, когда соответствующие задания и операции передачи данных происходили. Прямоугольники подписаны номерами заданий, версия с номером 1 не обозначается, остальные помечены v2, v3 и так далее.

Также в данном окне есть возможность ручного редактирования расписания. При применении операций вручную они добавляются в трассу так, как будто бы они были сделаны алгоритмом при автоматическом поиске, в частности, обновляется значение лучшего найденного расписания.

5.2.3 Подсистема для построения вычислительных систем для обработки данных от фазированных антенных решеток

Инструментальная система предоставляет пользователю возможность создавать шаблоны для генерации исходных данных для частных практических задач. В рамках инструментальной системы создан специализированный модуль для решения задачи построения вычислителей для обработки данных от фазированных антенных решеток [53][54]. Подробное описание задачи и параметры исходных данных приведены в приложении Г.

Обработка сигналов выполняется на многопроцессорной системе с одинаковыми по производительности и надежности процессорами. Заданы производительность и надежность процессоров, а также пропускная способность среды передачи данных между процессорами.

Данная задача сводится к описанной выше задаче построения системы с минимальным числом процессоров, необходимых для обработки данных в реальном масштабе времени и удовлетворения требований к надежности [108]. Вычислительная сложность заданий и объем передаваемых данных определяются исходя из параметров, приведенных в таблице Г.1 в приложении. Таким образом, все исходные данные для задачи, описанной в разделе 1, заданы.

На рисунке 5.6 показаны параметры исходных данных, которые может настраивать пользователь.

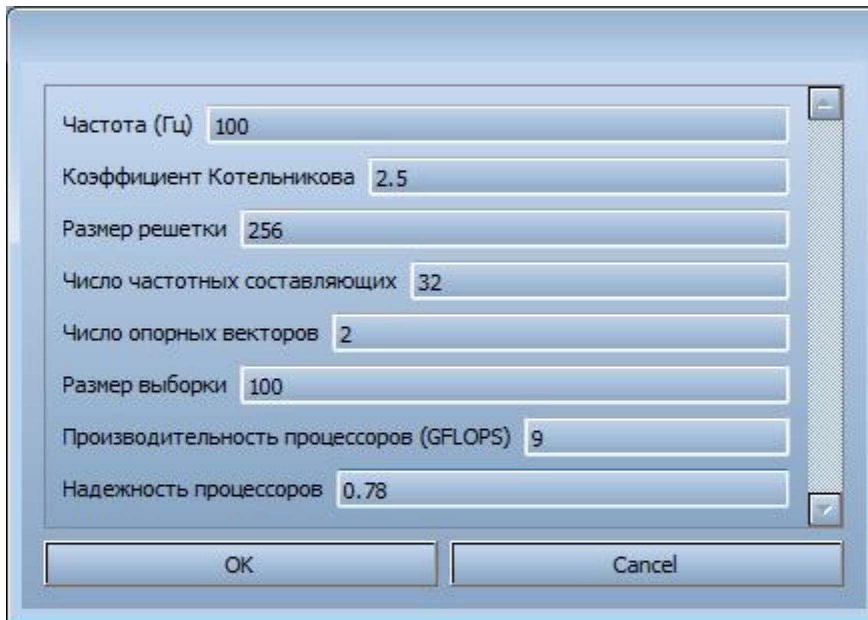


Рисунок 5.6. Параметры задачи планирования вычислений для антенных решеток.

На рисунке 5.7 приведен пример получаемого графа программы.

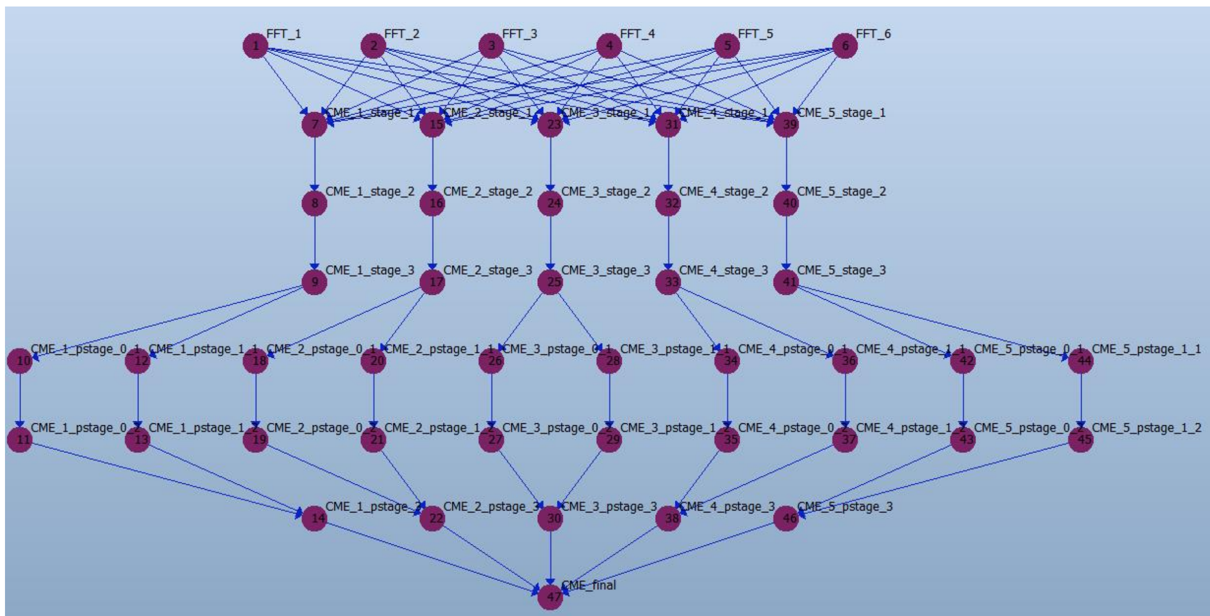


Рисунок 5.7. Граф программы для задачи планирования вычислений для антенных решеток.

В таблице 5.1 приведены результаты работы алгоритма (количество процессоров в построенной системе) для различных конфигураций пара-

метров. Рассматривалась задача радиолокации с различными размерами решетки, числом частотных составляющих и количеством опорных векторов. Надежность процессоров во всех случаях устанавливалась одинаковой.

Таблица 5.1. Количество процессоров в системе для различных конфигураций параметров: размер решетки (K), число опорных векторов (L), число частотных составляющих (M).

	$K = 16$	$K = 32$	$K = 64$	$K = 128$
$L = 8, M = 4$	2	4	7	13
$L = 8, M = 8$	2	4	7	14
$L = 8, M = 16$	3	4	8	14
$L = 8, M = 20$	3	5	8	14
$L = 16, M = 4$	3	5	8	14
$L = 16, M = 8$	4	5	9	15
$L = 16, M = 16$	4	6	9	16
$L = 16, M = 20$	5	6	10	16
$L = 32, M = 4$	5	7	10	17
$L = 32, M = 8$	6	8	11	18
$L = 32, M = 16$	8	9	12	19
$L = 32, M = 20$	9	10	13	20
$L = 64, M = 4$	10	11	14	21
$L = 64, M = 8$	12	13	17	23
$L = 64, M = 16$	14	16	19	25
$L = 64, M = 20$	16	18	21	27

5.3 Выводы

В данном разделе была описана инструментальная система для решения задачи структурного синтеза вычислительных систем с учетом ограничений на время выполнения программы и требований к надежности системы. Инструментальная система имеет графический пользовательский интерфейс, позволяет задавать исходные данные, запускать алгоритмы структурного синтеза, визуализировать результаты, генерировать по результатам исходный код для процедур обмена. Система поддерживает раз-

личные методы вычисления времени выполнения программы и надежности вычислительной системы, а также может быть настроена на решение частных задач структурного синтеза.

Заключение

Основные **результаты** диссертационной работы следующие.

- Предложена математическая постановка задачи структурного синтеза вычислительных систем реального времени с учетом ограничений реального времени и требований к надежности, которая позволяет использовать произвольные вычислимые функции для оценки времени выполнения расписания и надежности системы;
- Разработан алгоритм решения задачи на основе схемы имитации отжига. Теоретически обоснованы его корректность, доказана асимптотическая сходимость, и проведено исследование его точности и вычислительной сложности с использованием метода проверки статистических гипотез;
- Создана инструментальная система для решения задач структурного синтеза ВСРВ, позволяющая поддерживать процесс проектирования ВСРВ на различных стадиях.

Литература

1. Балашов В. В., Бахмуров А. Г., Волканов Д. Ю., Смелянский Р. Л., Чистолинов М. В., Ющенко Н. В. Стенд полунатурного моделирования для разработки встроенных вычислительных систем // Методы и средства обработки информации: Третья Всероссийская научная конференция. Труды конференции. – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова; МАКС Пресс. – 2009. – С.16-25.
2. Волканов Д. Ю., Зорин Д. А. Исследование применимости моделей оценки надежности для разработки программного обеспечения с открытым исходным кодом. // Программные системы и инструменты. Тематический сборник. – 2009. – № 10. – С. 125-134.
3. Гома Х. UML. Проектирование систем реального времени, распределённых и параллельных приложений //Изд.«ДМК», Москва. – 2002.
4. Зорин Д. А. Сравнение различных стратегий применения операций в алгоритме имитации отжига для задачи построения расписаний для многопроцессорных систем // "Параллельные вычисления и задачи управления" PACO'2012. Шестая международная конференция, Москва, 24-26 окт. 2012 г., Труды: в 3 т. М.: ИПУ РАН. – 2012. – Т. 1. – С. 278-291.
5. Зорин Д. А. Оценка сходимости алгоритма имитации отжига для задачи построения многопроцессорных расписаний // Вестник МГУ. Вычислительная математика и кибернетика. – 2014. – № 2. – С. 53–59.
6. Зорин Д. А., Костенко В. А. Алгоритм синтеза архитектуры вычислительной системы реального времени с учетом требований к надежности // Известия РАН. Теория и системы управления. – 2012. – № 2. – С. 76–83.

7. Ивченко Г. И., Медведев Ю. И. Математическая статистика. М.: Высшая школа, 1984. 248 р.
8. Костенко В. А., Романов В. Г., Смелянский Р. Л. Алгоритмы минимизации аппаратных ресурсов ВС // Искусственный интеллект (Донецк). – 2000. – № 2. – С. 383-388.
9. Костенко В. А. Алгоритмы построения расписаний для вычислительных систем реального времени, допускающие использование имитационных моделей // Программирование. – 2013. – №5 – С.53-71.
10. Кочетов Ю. А. Вероятностные методы локального поиска для задач дискретной оптимизации //Дискретная математика и ее приложения. Сборник лекций. – 2001. – №. 1. – С. 84-117.
11. Марков А. А., Распространение закона больших чисел на величины, зависящие друг от друга. – Известия физико-математического общества при Казанском университете. – 2-я серия. – Том 15. (1906) – С. 135–156.
12. Отчёт о научно-исследовательской работе «Создание прототипа интегрированной среды и методов комплексного анализа функционирования распределённых вычислительных систем реального времени (РВС РВ)» (Этап 4) // М.:, 2012. – Стр. 183.
13. Фуругян М. Г. Алгоритмы и модели вычислений. М.: Интернет-Университет Информационных Технологий, 2009.
14. Ширяев А. Н. Вероятность. М.: Наука, 1989. 574 р.
15. Aarts E., Korst J., Michiels W. Simulated annealing //Search methodologies. – Springer US, 2005. – P. 187-210.
16. Adelantado M., Busseno J.L., Rousselot J.Y., Siron P., Betoule M. HP-CERTI: towards a high performance, high availability open source RTI for composable simulations. // Fall simulation interoperability workshop, Orlando, USA – 2004.

17. Antonenko V.A. et al. DYANA: an integrated development environment for simulation and verification of real-time avionics systems // European Conference for Aeronautics and Space Sciences (EUCASS). Munich, Germany: Technische Universität München. – 2013.
18. Arkhangel'skii A. V., Pontryagin L. S. General Topology I. Basic concepts and constructions. Dimension theory, Enc //Math. Sci. – Vol. 17.
19. Avizienis A. The methodology of n-version programming //Software fault tolerance. – 1995. – Vol. 3. – P. 23-46.
20. Avižienis A., Laprie J. C., Randell B. Dependability and its threats: a taxonomy //Building the Information Society. – Springer US, 2004. – P. 91-120.
21. Banks D. C. et al. Fibre channel switching system and method : пат. 6160813 США. – 2000.
22. Bechta Dugan J., Lyu M. R. System reliability analysis of an N-version programming application //Reliability, IEEE Transactions on. – 1994. – Vol. 43. – №. 4. – P. 513-519.
23. Bicsi B. Network Design Basics for Cabling Professionals //City: McGraw-Hill Professional. – 2002.
24. Bray T. et al. Extensible markup language (XML) //World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210. – 1998.
25. Brigham E. O. The fast Fourier transform, 1974 //Englewood Cliffs, New Jersey, Prentices Hall.
26. Ceder A., Wilson N. H. M. Bus network design //Transportation Research Part B: Methodological. – 1986. – Vol. 20. – №. 4. – P. 331-344.
27. Chekuri C., Khanna S. A polynomial time approximation scheme for the multiple knapsack problem //SIAM Journal on Computing. – 2005. – Vol. 35. – №. 3. – P. 713-728.

28. Chen L., Avizienis A. N-version programming: A fault-tolerance approach to reliability of software operation //Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8). – 1978. – P. 3-9.
29. Chen Y. Solving nonlinear constrained optimization problems through constraint partitioning : дис. – University of Illinois, 2005.
30. Coello C. A. C. A comprehensive survey of evolutionary-based multiobjective optimization techniques //Knowledge and Information systems. – 1999. – Vol. 1. – №. 3. – P. 269-308.
31. Czarnowski I. et al. Scheduling multiprocessor tasks in presence of the correlated failures //Central European Journal of Operations Research. – 2003. – Vol. 11. – №. 2. – P. 163-182.
32. d'Ausbourg B. et al. Running real time distributed simulations under Linux and CERTI //Proceedings of the 2008 Summer Computer Simulation Conference. – Society for Modeling & Simulation International, 2008. – P. 43.
33. Davis R. I., Burns A. A survey of hard real-time scheduling for multiprocessor systems //ACM Computing Surveys (CSUR). – 2011. – Vol. 43. – №. 4. – P. 35.
34. Drozdowski M. Scheduling multiprocessor tasks–an overview //European Journal of Operational Research. – 1996. – Vol. 94. – №. 2. – P. 215-230.
35. Eckhardt D. E. et al. An experimental evaluation of software redundancy as a strategy for improving reliability //Software Engineering, IEEE Transactions on. – 1991. – Vol. 17. – №. 7. – P. 692-702.
36. Eckhardt D.E., Lee L. D. A theoretical basis for the analysis of multiversion software subject to coincident errors //Software Engineering, IEEE Transactions on. – 1985. – №. 12. – P. 1511-1517.
37. Fibre Channel Industry Association homepage [HTML] (http://fibrechannel.org/).

38. Fujimoto R. M. Parallel and distributed simulation systems. – New York : Wiley, 2000. – Vol. 300.
39. Goldberg D. E. et al. Genetic algorithms in search, optimization, and machine learning. – Reading Menlo Park : Addison-wesley, 1989. – Vol. 412.
40. Greenwood G. W., Gupta A. K., McSweeney K. Scheduling Tasks in Multiprocessor Systems Using Evolutionary Strategies //International Conference on Evolutionary Computation. – 1994. – P. 345-349.
41. Gu R. et al. Exploiting statically schedulable regions in dataflow programs //Journal of Signal Processing Systems. – 2011. – Vol. 63. – №. 1. – P. 129-142.
42. Guo B. et al. Hierarchical control and data flow graph modeling method in energy-aware hardware/software partitioning //Journal of Sichuan University: Engineering Science Edition. – 2011. – Vol. 43. – №. 4. – P. 83-88.
43. Henderson D., Jacobson S. H., Johnson A. W. The theory and practice of simulated annealing //Handbook of metaheuristics. – Springer US, 2003. – P. 287-319.
44. Holland J. H. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. – U Michigan Press, 1975.
45. Hou E. S. H., Ansari N., Ren H. A genetic algorithm for multiprocessor scheduling //Parallel and Distributed Systems, IEEE Transactions on. – 1994. – Vol. 5. – №. 2. – P. 113-120.
46. Hou E. S. H., Hong R., Ansari N. Efficient multiprocessor scheduling based on genetic algorithms //Proceedings of the 16th Annual Conference of IEEE Industrial Electronic Society (IECON'90). – 1990. – Vol. 2. – P. 1239-1243.
47. Jedrzejowicz P. et al. Evolution-based scheduling of fault-tolerant programs on multiple processors //Parallel and Distributed Processing. – Springer Berlin Heidelberg, 1999. – P. 210-219.

48. Jędrzejowicz P. et al. Population-based scheduling on multiple processors //Scheduling Multiple-version Programs on Multiple Processors Proceedings 4th Metaheuristics International Conference, MIC. – 2001. – P. 613-618.
49. Johnson Jr A. M., Malek M. Survey of software tools for evaluating reliability, availability, and serviceability //ACM Computing Surveys (CSUR). – 1988. – Vol. 20. – №. 4. – P. 227-269.
50. Kalashnikov A. V., Kostenko V. A. A parallel algorithm of simulated annealing for multiprocessor scheduling //Journal of Computer and Systems Sciences International. – 2008. – Vol. 47. – №. 3. – P. 455-463.
51. Kirkpatrick S., Jr. D. G., Vecchi M. P. Optimization by simulated annealing //science. – 1983. – Vol. 220. – №. 4598. – P. 671-680.
52. Knight J. C., Leveson N. G. An experimental evaluation of the assumption of independence in multiversion programming //Software Engineering, IEEE Transactions on. – 1986. – №. 1. – P. 96-109.
53. Kostenko V. A. Design of computer systems for digital signal processing based on the concept of “open” architecture // Automation and Remote Control. – 1994. – №. 12. – P. 151-162.
54. Kostenko V. A. Large-grain parallelism in signal processing problems //Programming and Computer Software. – 1997. – Vol. 23. – №. 2. – P. 109-115.
55. Kostenko V. A., Smelyanskii R. L., Trekin A. G. Synthesizing structures of real-time computer systems using genetic algorithms //Programming and Computer Software. – 2000. – Vol. 26. – №. 5. – P. 281-288.
56. Kuhn H. W. The Hungarian method for the assignment problem //Naval research logistics quarterly. ‐ 1955. ‐ Vol. 2. ‐ No. 1 ‐ 2. ‐ P. 83-97.
57. Laprie J. C. et al. Definition and analysis of hardware-and software-fault-tolerant architectures //Computer. – 1990. – Vol. 23. – №. 7. – P. 39-51.

58. Lawler E. L., Wood D. E. Branch-and-bound methods: A survey //Operations research. – 1966. – Vol. 14. – №. 4. – P. 699-719.
59. Levitin G. Computational intelligence in reliability engineering. – Springer-Verlag Berlin Heidelberg, 2007.
60. Littlewood B., Miller D. R. Conceptual modeling of coincident failures in multiversion software //Software Engineering, IEEE Transactions on. – 1989. – Vol. 15. – №. 12. – P. 1596-1614.
61. Locatelli M. Simulated annealing algorithms for continuous global optimization: convergence conditions //Journal of Optimization Theory and applications. – 2000. – Vol. 104. – №. 1. – P. 121-133.
62. Lundy M., Mees A. Convergence of an annealing algorithm //Mathematical programming. – 1986. – Vol. 34. – №. 1. – P. 111-124.
63. Mailloux R. J. Phased array antenna handbook. – Boston : Artech House, 2005. – P. 92-106.
64. Martello S., Toth P. Knapsack problems. – New York : Wiley, 1990.
65. Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules // IEEE, 2010 c. 26.
66. Monzingo R. A., Miller T. W. Introduction to adaptive arrays. – SciTech Publishing, 1980.
67. Moore M. An accurate and efficient parallel genetic algorithm to schedule tasks on a cluster //Parallel and Distributed Processing Symposium, International. – IEEE Computer Society, 2003. – P. 145a-145a.
68. Moore M. An accurate parallel genetic algorithm to schedule tasks on a cluster //Parallel Computing. – 2004. – Vol. 30. – №. 5. – P. 567-583.
69. Munkres J. Algorithms for the assignment and transportation problems //Journal of the Society for Industrial & Applied Mathematics. – 1957. – Vol. 5. – №. 1. – P. 32-38.
70. Nelson V. P. Fault-tolerant computing: Fundamental concepts //Computer. – 1990. – Vol. 23. – №. 7. – P. 19-25.

71. Nicola V. F., Goyal A. Modeling of correlated failures and community error recovery in multiversion software //Software Engineering, IEEE Transactions on. – 1990. – Vol. 16. – №. 3. – P. 350-359.
72. Nikravan M., Kashani M. H. A genetic algorithm for process scheduling in distributed operating systems considering load balancing //Proceedings 21st European Conference on Modelling and Simulation Ivan Zelinka, Zuzana Oplatkova, Alessandra Orsoni, ECMS. – 2007.
73. Nummelin E. General irreducible Markov chains and non-negative operators. – Cambridge University Press, 2004. – Vol. 83.
74. Oliviero A., Woodward B. Cabling: the complete guide to copper and fiber-optic networking. – John Wiley & Sons, 2009.
75. OMG Unified Modeling Language Specification [PDF] (http://www.omg.org/spec/UML/1.4/PDF)
76. Orsila H. et al. Optimal subset mapping and convergence evaluation of mapping algorithms for distributing task graphs on multiprocessor SoC //System-on-Chip, 2007 International Symposium on. – IEEE, 2007. – P. 1-6.
77. Orsila H., Salminen E., Hämäläinen T. D. Best practices for simulated annealing in multiprocessor task distribution problems //Simulated Annealing. – 2008. – P. 321-342.
78. Ottenstein K. J., Ottenstein L. M. The program dependence graph in a software development environment //ACM Sigplan Notices. – ACM, 1984. – Vol. 19. – №. 5. – P. 177-184.
79. Popov P. et al. Estimating bounds on the reliability of diverse systems //Software Engineering, IEEE Transactions on. – 2003. – Vol. 29. – №. 4. – P. 345-359.
80. Python v2.7 documentation [HTML] (https://docs.python.org/2.7/)
81. Python v3.1 documentation [HTML] (http://docs.python.org/py3k/).

82. Qin X., Jiang H. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters //Journal of Parallel and Distributed Computing. – 2005. – Vol. 65. – №. 8. – P. 885-900.
83. Qin X., Jiang H., Swanson D. R. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems //Parallel Processing, 2002. Proceedings. International Conference on. – IEEE, 2002. – P. 360-368.
84. Qt4 documentation [HTML] (http://qt-project.org/doc/)
85. Rumbaugh J., Jacobson I., Booch G. Unified Modeling Language Reference Manual, The. – Pearson Higher Education, 2004.
86. Spence L. E., Insel A. J., Friedberg S. H. Elementary linear algebra. – Pearson/Prentice Hall, 2008.
87. Srikanth G. U. et al. A survey on real time task scheduling //European Journal of Scientific Research. – 2012. – Vol. 69. – №. 1. – P. 33-41.
88. Sriram S., Bhattacharyya S. S. Embedded multiprocessors: Scheduling and synchronization. – CRC press, 2012.
89. Stankovic J. A. Real-time Computing. // Byte Magazine – 1992. – v. 17, N 8. – p. 155-160.
90. State Chart XML (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft 26 April 2011 [HTML] (http://w3.org/TR/scxml/)
91. Strang G. Introduction to linear algebra. – SIAM, 2003.
92. Szu H., Hartley R. Fast simulated annealing //Physics letters A. – 1987. – Vol. 122. – №. 3. – P. 157-162.
93. Treaster M. A survey of fault-tolerance and fault-recovery techniques in parallel systems //arXiv preprint cs/0501002. – 2005.
94. Trouvé A. Rough large deviation estimates for the optimal convergence speed exponent of generalized simulated annealing algorithms //Annales

de l'IHP Probabilités et statistiques. – Elsevier, 1996. – Vol. 32. – №. 3. – P. 299-348.

95. Van Loan C. Computational frameworks for the fast Fourier transform. – Siam, 1992. – Vol. 10.
96. Vanneschi M. Parallel paradigms for scientific computing //Reaction and Molecular Dynamics. – Springer Berlin Heidelberg, 2000. – P. 168-181.
97. Villalobos-Arias M., Coello C. A. C., Hernández-Lerma O. Asymptotic convergence of a simulated annealing algorithm for multiobjective optimization problems //Mathematical Methods of Operations Research. – 2006. – Vol. 64. – №. 2. – P. 353-362.
98. Wah B. W., Chen Y., Wang T. Simulated annealing with asymptotic convergence for nonlinear constrained optimization //Journal of Global Optimization. – 2007. – Vol. 39. – №. 1. – P. 1-37.
99. Wasserman F. Neurocomputer Techniques: Theory and Practice [Russian translation] //Mir, Moscow. – 1992. – 240 p.
100. Wattanapongsakorn N., Levitan S. P. Reliability optimization models for embedded systems with multiple applications //Reliability, IEEE Transactions on. – 2004. – Vol. 53. – №. 3. – P. 406-416.
101. Widrow B., Stearns S. D. Adaptive signal processing //Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p. – 1985. – Vol. 1.
102. Wild T. et al. Mapping and scheduling for architecture exploration of networking SoCs //VLSI Design, 2003. Proceedings. 16th International Conference on. – IEEE, 2003. – P. 376-381.
103. Wilhelm R. et al. The worst-case execution-time problem–overview of methods and survey of tools //ACM Transactions on Embedded Computing Systems (TECS). – 2008. – Vol. 7. – №. 3. – P. 36.
104. Wood A. Software reliability growth models //Tandem Computers, Technical report. – 1996. – Vol. 96.

105. Yao X. Simulated annealing with extended neighbourhood //International Journal of Computer Mathematics. – 1991. – Vol. 40. – №. 3-4. – P. 169-189.
106. Zorin D. A., Kostenko V. A. Co-design of real-time embedded systems under reliability constraints //Programmable Devices and Embedded Systems. – 2012. – Vol. 11. – №. 1. – P. 424-428.
107. Zorin D. A. Convergence and Accuracy Measurement of Scheduling on Multiprocessors with Simulated Annealing // VII Moscow International Conference on Operations Research (ORM2013): Proceedings, Vol. 1. Moscow: MAKS Press. – 2013. – P. 94-97.
108. Zorin D. A. Scheduling Signal Processing Tasks for Antenna Arrays with Simulated Annealing // Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). Kazan, Russia: Kazan National Research Technical University, – 2013. – P. 122-127.
109. Zorin D. A., Kostenko V. A. Job Shop Scheduling and Co-design of Real-Time Systems with Simulated Annealing // Proceedings of the 3rd International Conference on Operations Research and Enterprise Systems. Angers, France: ESEO. – 2014. – P. 17-26.

Приложение А. Способы оценки надежности

В рассматриваемой в данной работе задаче требуется построить систему, устойчивую как к отказам аппаратуры (процессоров), так и к некорректной работе программ. Поэтому необходимо иметь модель, позволяющую рассчитать надежность аппаратных и программных компонентов. В этой разделе будут описаны различные способы моделирования различных систем для расчета надежности.

А.1 Надежность процессоров. Резервирование

Резервирование процессоров – простой и эффективный способ повышения надежности аппаратуры. Одна и та же программа выполняется на n устройствах. Такая система выдерживает $n - 1$ отказов аппаратуры, так как достаточно лишь одного доработавшего до конца процессора [70][93].

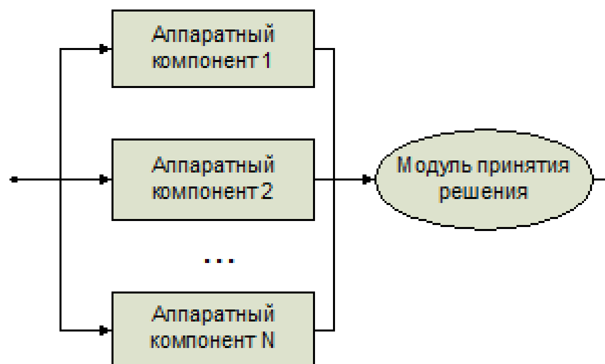


Рисунок А.1. Схема резервирования.

Эта модель несколько упрощенная, так как в ней неявно предполагается, что в случае отказа аппаратуры это событие можно определить. Это верно в случае, если процессор полностью перестает функционировать, например, до него не доходят никакие сигналы. В реальности могут встречаться более сложные отказы, когда процессор продолжает работу, но вычисления на нем ошибочны (например, магнитное воздействие как-то по-

вредило микросхему). Отказы такого вида в дальнейшем не рассматриваются.

Различают «горячее» и «холодное» резервирование. При горячем резервировании процессоры на самом деле работают параллельно, при холодном – реально работает только один процессор, а в случае его отказа подключается второй. Холодное резервирование сложнее в реализации, так как на резервных процессорах, даже если основной процессор не отказывает, необходимо отмечать контрольные точки и получать данные с основного, чтобы в случае отказа можно было начинать работу не с самого начала. В дальнейшем предполагается, что все процессоры находятся в горячем резерве.

Для оценки надежности аппаратуры обычно используют экспоненциальное распределение [59]. Пусть случайная величина T обозначает момент времени, когда процессор отказывает. Обозначим через $p(t)$ функцию надежности, то есть вероятность того, что в течение времени работы t отказа не будет: $p(t) = p(T > t)$.

При этом: $p(0) = 1$, то есть процессор изначально работоспособен и $p(t) \xrightarrow[t\to\infty]{} 0$ монотонно, то есть любая аппаратура когда-нибудь отказывает.

Тогда $q(t) = 1 - p(t)$ есть функция распределения величины T. При этих условиях распределение T абсолютно непрерывно, следовательно, существует плотность $f(t) = \frac{dq}{dt} = -\frac{dp}{dt}$. По определению плотности $q(t) = \int_0^t f(x)\,dx$.

Интенсивностью отказов $\lambda(t)$ называют условную плотность вероятности возникновения отказа компонента системы, определяемую для рассматриваемой наработки при условии, что до этой наработки отказ не возник. Вероятность отказа системы в достаточно малый промежуток времени $(t, t + dt)$ при условии, что до момента t отказа не было, есть с одной стороны $f(t)dt$ с другой, по определению интенсивности, $p(t) \cdot \lambda(t)\,dt$. Ре-

шая получающееся дифференциальное уравнение, находим $p(t) = e^{-\int_0^t \lambda(x)dx}$.

Если интенсивность является константой, получаем формулу плотности экспоненциального распределения: $f(t) = \lambda e^{-\lambda t}$. Используемая модель удовлетворяет требованиям однородности, монотонности и отсутствия последействия, поэтому применение данного распределения является оправданным.

Общее время работы точно не известно, однако приблизительно его всегда можно оценить сверху, поэтому чаще всего с использованием такой оценки получают константную оценку для надежности аппаратуры.

Отказы различных процессоров в дальнейшем считаются независимыми, поэтому приходим к формуле надежности системы из n процессоров, приведенной в разделе «Постановка задачи»: $R_H = \prod_{m_i \in M}(1 - R(m_i))$.

А.2 Надежность программ. N-версионное программирование

Принцип работы NVP

В отличие от анализа надежности аппаратуры, при рассмотрении надежности программных компонентов требуется более сложная классификация ошибочных состояний. При моделировании надежности оборудования предполагается, что отказ происходит из-за факторов, на которые не влияет работа системы, в то же время для программы в спецификации указано, какие данные она принимает на вход, поэтому отказ происходит вследствие неверной обработки *некоторых* входных данных. По умолчанию предполагается, что программа, сданная в эксплуатацию, на каких-то данных должна работать верно.

Отказы программы являются следствием *неисправностей* в ее реализации. Источником неисправностей может быть неверная или отсутствую-

щая реализация какой-то из операций. При выполнении программы с неисправностью может произойти *ошибка*, то есть программа окажется в неверном состоянии. Ошибка может потенциально привести к *отказу*, то есть прерыванию программы до завершения или выдаче неверного результата [2].

Приведем пример. Пусть программа принимает на вход целое число x и возвращает $x/10$, если x больше 20 и $x * 2$, если x меньше 20. Предположим, имеется следующая неверная реализация:

```
int f(int x)
{
   if (x > 20)
      return x/0;
   else
      return x+2;
}
```

В обеих ветках условного оператора имеются очевидные неисправности. Если x больше 20, то в программе произойдет отказ и она завершится аварийно. Если x меньше 20, то произойдет ошибка, когда возвращаемое значений будет $x + 2$ вместо $x * 2$, однако если x равно 2, то это не приведет к отказу, так как результат окажется верным.

Идея N-версионного программирования заключается в том, чтобы создать несколько версий одной и той же программы в расчете на то, что если в одной из версий будет неисправность, остальные версии проработают правильно и в итоге удастся избежать отказа [28]. Число версий всегда нечетно (обычно 3 либо 5), и результаты подвергаются простому сравнению, итоговым результатом объявляется тот, который выдали больше половины версий. Таким образом, при отказе не более чем $(N + 1)/2$ версий отказа не происходит.

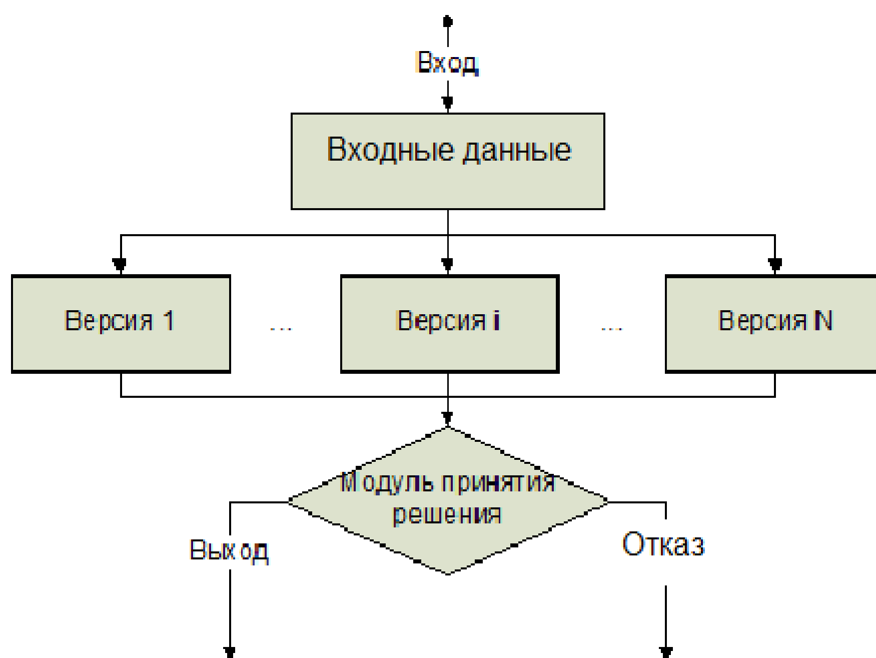


Рисунок А.2. Схема работы NVP.

Разные версии обычно разрабатываются разными группами программистов для того, чтобы по возможности неисправности были различны в разных версиях. Предполагается, что если неисправности различны, то версии отказывают на разных входных данных.

Статистическая оценка надежности системы с NVP

Приведем формулу [36] для надежности системы с NVP в общем случае. Отказ конкретной версии на конкретных входных данных можно считать детерминированным, значит, отказ произвольной версии на заданных входных данных есть случайная величина с тривиальным распределением, принимающим значение, равное доле отказавших на заданном входе версий. Обозначим через $\theta(x)$ вероятность отказа произвольно выбранной версии при входе x. Фактически, это доля имеющихся версий, отказывающих на входе x. Тогда, если вход является случайной величиной X, то $\theta(X)$ – также случайная величина, и вероятность отказа больше половины из N версий можно задать формулой:

$$P_N = \int_0^1 \sum_{l=(N+1)/2}^{N} \theta(x)^l \cdot \left(1-\theta(x)\right)^{N-l} dQ(x)$$

Вероятность отказа l версий из N имеет биномиальное распределение. Интеграл берется по всем входам x и считается по мере, заданной распределением случайной величины X, обозначающей вероятность получить на вход те или иные данные.

Практическая применимость данной формулы весьма затруднена. Распределение величины X в точности не известно, кроме того, множество ее значений может быть бесконечно, так что нельзя будет оценить $\theta(X)$ напрямую путем запуска каждой версии на каждом входе. Тем не менее, распределение X можно оценить с помощью сбора статистики о работе системы, и если протестировать все версии на некотором подмножестве X, можно получить точечную статистическую оценку для вероятности отказа системы и указать ее дисперсию [35].

Пусть тестируются n версий, и количество различных входных данных равно k. Обозначим через $u_y(x)$ индикатор события «y из n версий отказали». Тогда эмпирическая оценка вероятности отказа версий: $g(y) = \frac{1}{k}\sum_{i=1}^{n} u_y(x_i)$.

Пусть n версий выбираются случайно из N возможных. Обозначим через $\Psi = \{J\}$ множество всех сочетаний из N по n. $u_J(x_i, l)$ – индикатор события «l версий из сочетания J отказали». В этих обозначениях оценка для вероятности отказа более половины версий будет иметь следующий вид:

$$\tilde{P} = \frac{1}{k}\binom{N}{n}^{-1} \sum_{J\in\Psi} \sum_{i=1}^{k} \sum_{l=\frac{n+1}{2}}^{n} u_J(x_i, l)$$

Сумма по всем сочетаниям может быть записана как

$$\sum_{J\in\Psi} u_J(x_i, l) = \sum_{y=0}^{N} \binom{y}{l}\binom{N-y}{n-l} u_y(x_i)$$

В терминах функции $g(y)$ это можно записать как

$$\tilde{P} = \sum_{y=0}^{N} \binom{N}{n}^{-1} \sum_{l=\frac{n+1}{2}}^{n} \binom{y}{l} \binom{N-y}{n-l} g(y)$$

Поставляя $n = 1$ в эту формулу, можно получить оценку вероятности отдельной версии.

Дисперсия полученной оценки имеет следующий вид:

$$D\tilde{P} = \frac{1}{k}\left(\sum_{y=0}^{N} a_{ny}^{2} \Phi(y) - \left(\sum_{y=0}^{N} a_{ny} \Phi(y)\right)^{2}\right), a_{ny} = \binom{N}{n}^{-1} \sum_{l=\frac{n+1}{2}}^{n} \binom{y}{l} \binom{N-y}{n-l}$$

$\Phi(y)$ обозначает вероятность отказа y версий, оценкой для этой величины является введенная ранее функция $g(y)$.

Корреляция между ошибками в разных версиях. Модели Экхарда-Ли и Литтлвуда-Миллера

Теоретические исследования и эксперименты показывают, что даже при независимой разработке версий ошибки не всегда являются независимыми [19]. Исследования показали, что в программе есть более простые и более сложные для реализации части, и в более сложных обнаруживается больше ошибок. В этой связи предлагается отказаться от ранее использовавшейся модели $\theta(x)$ как случайной величины, принимающей одно значение, равное вероятности отказа на заданном входе. При таком определении ошибки, очевидно будут стохастически независимыми, поэтому для учета корреляции между ошибками $\theta(x)$ преобразуется в случайную величину с распределением Бернулли.

В работе [36] была подтверждена закономерность результатов экспериментов, показывавших наличие корреляции между ошибками в разных версиях. В основу легло следующее утверждение: несмотря на то, что команды разработчиков разные, они решают одну и ту же программистскую

задачу. В каждой программе есть более сложные и более простые для реализации части, при этом относительная сложность неформализуема, однако для разных людей сложными будут одни и те же части. Соответственно, вероятность найти ошибку в «сложной» части будет выше, а для двух случайных версий вероятность найти в них общую ошибку повышается, что позволяет говорить о средней по всем версиям вероятности общей ошибки.

Позже, в исследовании [60] была построена так называемая модель Литтлвуда-Миллера, в которой был предложен способ превратить проблему различной сложности отдельных частей программы в преимущество при разработке. Предлагается специально вести руководство разработкой разных версий так, чтобы в них особое внимание уделялось разным частям, за счет манипулирования сроками разработки и объемами тестирования. Тогда, если «сложный» момент для одной команды будет «простым» для другой, корреляция между ошибками снижается.

Модели Экхарда-Ли и Литтлвуда-Миллера являются скорее концептуальными, нежели практическими, так как оценить функцию «сложности» части в зависимости от подаваемых на вход данных не представляется возможным. Кроме того, у всех теоретических моделей есть еще один серьезный недостаток: поскольку все статистические оценки строятся на основе теории вероятностей, они дают представление о надежности в среднем. Это значит, что если бы можно было разработать бесконечное число версий, и выбирать из них всевозможные конечные подмножества, то полученные результаты в пределе совпадали бы с оценками. В реальности же число разрабатываемых версий сильно ограничено, и их поведение может сильно отличаться от теоретического, так как оценки обладают определенной дисперсией.

Классификация типов ошибок. Явные формулы для надежности

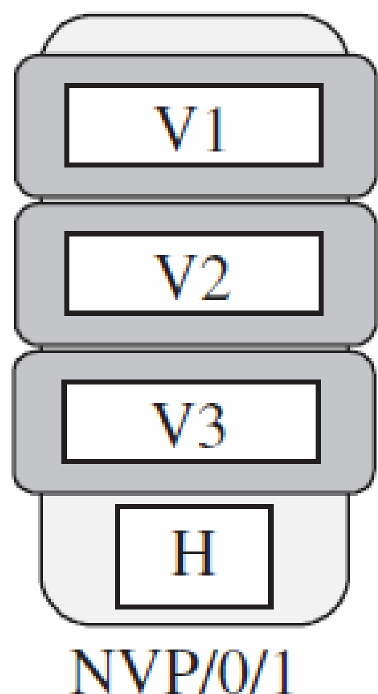


Рисунок А.3. Схема механизма NVP.

В [57] были рассмотрены практические варианты реализации N-версионного программирования в возможном сочетании с резервированием аппаратных устройств. При анализе надежности все виды ошибок были классифицированы на несколько типов, так что общая надежность системы явно выражалась через вероятности ошибок разных типов.

В [100] приведена систематизированная схема оценки надежности разных конфигураций NVP, являющаяся продолжением [57]. В качестве примера приведем разбор для простого и вместе с тем часто встречающегося случая: три версии программы на одном аппаратном устройстве (рисунок А.3). Эту схему часто обозначают как NVP/0/1, указывая, что она выдерживает 0 аппаратных отказов и 1 программный.

Имеется один аппаратный компонент, на котором параллельно исполняются три разные версии программы. Модуль принятия решений выбирает тот результат, который выдается двумя версиями из трех. Таким образом, система отказывает в следующих случаях:

- Ошибка в устройстве принятия решений,
- Ошибка в спецификации (все три версии неправильные),
- Отказ аппаратуры (все три версии не сработают),
- Общая ошибка (две разные версии содержат одну и ту же ошибку, система работает неверно),
- Отказ двух версий из трех.

Считается, что ошибки перечисленных выше типов не зависят друг от друга. В этом случае общую надежность можно рассчитать по формуле полной вероятности, если известны вероятности всех указанных выше типов ошибок.

В формуле используются следующие обозначения.

P_{v_i}– вероятность отказа -й версии программы.

$P_{rv_{ij}}$– вероятность отказа версий программы i и j из-за одинаковой ошибки.

P_d – вероятность отказа механизма принятия решений.

P_{all} – вероятность ошибки в спецификации программы.

P_{h_i} – вероятность отказа -й версии аппаратного обеспечения.

$Q_n = 1 - P_n$, каким бы ни был индекс n.

Общая надежность тогда имеет следующий вид:

$$
\begin{aligned}
P = P_{rv_{12}} + Q_{rv_{12}}P_{rv_{13}} + Q_{rv_{12}}Q_{rv_{13}}P_{rv_{23}} + Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}P_d \\
+ Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}Q_dP_{all} + Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}Q_dQ_{all}P_h \\
+ Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}Q_dQ_{all}Q_hP_{v_1}P_{v_2} \\
+ Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}Q_dQ_{all}Q_hQ_{v_1}P_{v_2}P_{v_3} \\
+ Q_{rv_{12}}Q_{rv_{13}}Q_{rv_{23}}Q_dQ_{all}Q_hQ_{v_2}P_{v_1}P_{v_3}
\end{aligned}
$$

Оценка самих вероятностей при данном подходе является нетривиальной задачей. В работе [22] описан реалистичный способ получения оценок

для всех вероятностей исходя из числа ошибок, обнаруженных при тестировании. Масштабное тестирование не всегда возможно, особенно в том случае, если тестирование идет до полного окончания разработки. Для таких случаев существуют методы оценки надежности, использующие данные о тестировании за некоторый период времени [2],[104].

Использование бета-распределения для моделирования надежности

В работе [71] было предложено для оценки вероятности отказа случайно выбранной версии на произвольных входных данных использовать модель, основанную на бета-распределении. Возможность применять такую модель вытекает из эксперимента, описанного в [52]. Приведем описание этого подхода по [31].

Возвращаясь к обозначению $\theta(X)$, введенному выше, напомним, что это вероятность отказа на случайных входных данных X, и это случайная величина с некоторым распределением. Описываемый в этой главе подход основан на предположении, что $\theta(x) \sim B(\alpha, \beta)$, где B – бета-распределение, задающееся плотностью следующего вида: $B(\alpha, \beta) \sim \frac{x^{\alpha-1} \cdot (1-x)^{\beta-1}}{B(\alpha,\beta)}$

На рисунке А.4 приведен график плотности бета-распределения при различных параметрах. Видно, что случайная величина с таким распределением принимает с ненулевыми вероятностями только значения из отрезка [0, 1], поэтому она может обозначать вероятность отказа.

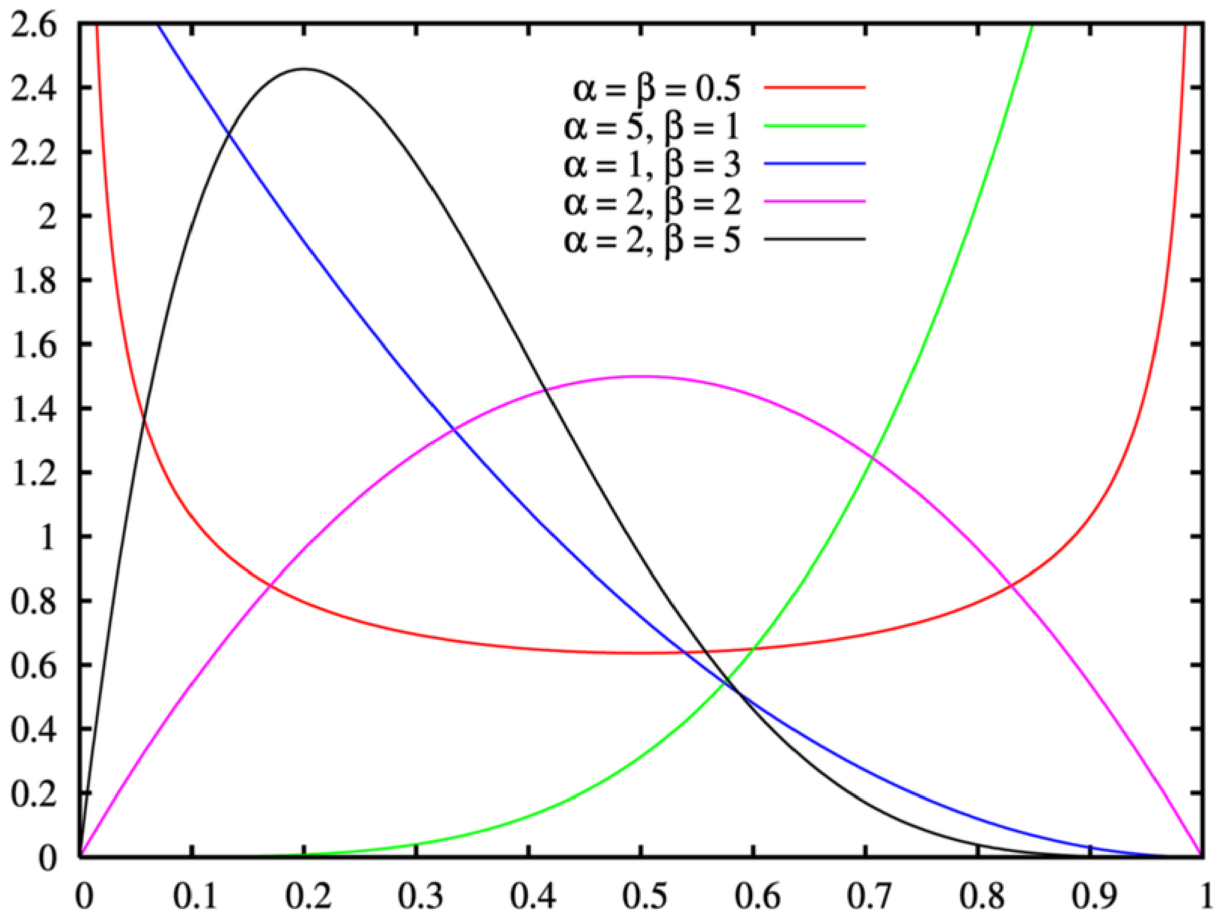


Рисунок А.4. Бета-биномиальное распределение.

Для бета-распределения производится репараметризация по следующей формуле: $\alpha = \rho/\theta, \beta = (1-\rho)/\theta$.

Содержательно ρ обозначает среднюю надежность версии, а θ – уровень корреляции между разными версиями. В частности, если $\theta \to 0$, получается, что версии стохастически независимы, а если $\theta \to \infty$, версии полностью зависимы, то есть если отказывает одна, то отказывают и все остальные.

Если использовать вышеописанную модель, то случайная величина, обозначающая число отказавших версий k, если всего версий N, имеет бета-биномиальное распределение с параметрами α, β, N:

$$\xi \sim P(\xi = k) = \binom{n}{k} \frac{B(k+\alpha, n-k+\beta)}{B(\alpha,\beta)}, k \in (0, \dots, N)$$

Содержательно бета-биномиальное распределение можно описать следующей урновой моделью. Пусть в урне α белых шаров и β черных. В каждом эксперименте вытаскивают шар, после чего возвращают его в урну и добавляют еще один того же цвета. Если проведено N экспериментов, то число вытащенных в них белых шаров будет иметь бета-биномиальное распределение. В случае с надежностью это означает, что отказ одной из версий повышает вероятность отказа других.

Оценка надежности на подмножествах входных данных. Модель Попова-Стриджини

В [79] описан подход к оценке вероятности отказа различных версий, основанный на разбиении множества входных данных на домены. За счет этого появляется возможность получить верхнюю и нижнюю оценки для вероятности совместного отказа.

Задача рассматривается для случая двух версий (вероятности совместного отказа трех и более версий не рассматриваются, но предложенные идеи можно распространить и на этот случай). Аналогично введенным ранее обозначениям, будем называть $\theta_A(X)$ индикатор отказа версии A на входных данных X. Очевидно, вероятность отказа версии есть не что иное, как:

$$P(A \text{ отказывает на } X) = P(A) = E\big(\theta(X)\big) = \sum_{x \in D} P(x) \cdot \theta(x)$$

Однако отказы версий A и B на случайных входных данных X не являются независимыми случайными величинами, следовательно, вероятность совместного отказа в общем случае задается формулой:

$$P(A, B \text{ отказывают на } X) = P(A, B) = P(A) \cdot P(B) + cov(\theta_A, \theta_B)$$

Ковариация задается формулой:

$$cov(\theta_A, \theta_B) = \sum_{x \in D} \big(\theta_A(x) - P(A)\big)\big(\theta_B(x) - P(B)\big) \cdot P(x)$$

Рассчитать ковариацию напрямую не представляется возможным, поэтому происходит следующее преобразование. Все пространство возможных входных данных разбивается на домены – непересекающиеся подмножества:

$$D = S_1 \cup S_2 \cup \ldots \cup S_n, \forall i, j \in [1, n]: S_i \cap S_j = \emptyset$$

На каждом домене вероятность совместного отказа рассчитывается по формуле, аналогичной общей:

$$P(A, B|S_i) = P(A|S_i) \cdot P(B|S_i) + cov_i(\theta_A, \theta_B)$$

Вероятность совместного отказа получается из вероятностей отказов на доменах по формуле полной вероятности:

$$P(A, B) = \sum_{i=1}^{n} P(A, B|S_i) \cdot P(S_i)$$

$$= \sum_{i=1}^{n} P(A|S_i) \cdot P(B|S_i) \cdot P(S_i)$$

$$+ \sum_{i=1}^{n} cov_i(\theta_A, \theta_B) \cdot P(S_i) = P_{subind}(A, B) + E(cov_i(\theta_A, \theta_B)$$

Первое слагаемое есть вероятность совместного отказа, если на каждом домене версии отказывают независимо. Преобразуем эту величину.

Пусть S_A – вероятность отказа версии А на случайно выбранном домене. По определению, $S_A = P(A|S_i) \cdot P(S_i)$ и $E(S_A) = P(A)$. Тогда

$$P_{subind}(A, B) = P(A) \cdot P(B) + cov(S_A, S_B) + E(cov_i(\theta_A, \theta_B))$$

Второе слагаемое в этом выражении обозначает уровень зависимости ошибок на разных доменах. Если эта величина меньше или равна нулю, значит ошибки независимы, однако более реалистичным будет случай, когда эта величина положительна и определенная корреляция между доменами имеет место.

Преимущество данного разложения в том, что величину $cov(S_A, S_B)$ *можно вычислить* в исходных предположениях, что заданы $P(A|S_i)$ и

$P(S_i)$. Таким образом, можно рассчитать нижнюю оценку вероятности отказа обеих версий:

$$P(A,B) \geq P(A) \cdot P(B) + \sum_{S_i} \big(P(A|S_i) - P(A)\big) \cdot \big(P(B|S_i) - P(B)\big) \cdot P(S_i)$$

Верхней оценкой для $P(A,B)$ может служить следующая величина:

$$P_{upper}(A,B) = \sum_{S_i} \min\big(P(A|S_i), P(B|S_i)\big) \cdot P(S_i)$$

Приложение Б. Модели среды передачи данных

Б.1 Вычисление времени для системы без конфликтов на портах

Пусть процессоры в системе подключены к такой среде передачи данных, которая обеспечивает возможность передачи данных между любыми двумя процессорами в любой момент [23][74]. Тогда время работы программы $t(S)$ может быть вычислено следующим образом:

1. Все задания на процессорах упорядочены по определению расписания. Для каждого процессора будем вычислять время выполнения заданий на нем. Изначально время на каждом из процессоров $\forall m \in M: t(S|m) = 0$. Изначально все задания на всех процессорах не просмотрены, то есть множество непросмотренных заданий $SN = S$. Задания, время выполнения которых уже учтено, будем считать просмотренными, и у каждого будет указано время $t(s)$, когда оно завершается.
2. Находим процессор m_i, первое непросмотренное задание s_{cur} на котором таково, что в соответствующую ему вершину графа входят дуги только из вершин, соответствующих просмотренным заданиям, то есть $Dep(s_{cur}) \subseteq S \backslash SN$. Такое задание всегда найдется, если расписание удовлетворяет свойству ацикличности. Если все задания уже просмотрены ($SN = \emptyset$), то переходим к пункту 5.
3. Пусть s_{pre} – задание, выполняющееся на m_i перед s_{cur}. Текущее время $t(S/m_i)$ вычисляется как сумма времени выполнения s_{cur} и времени поступления самых последних данных от других заданий: $\mathrm{H}(s_{cur}, m_i) + \max(t(s_{pre}), \max_{s_0 \in Dep(s_{cur})}(t(s_0) + F(s_0, s_{cur}))) + t_{wait}$.
4. Отмечаем s_{cur} как просмотренную и переходим к пункту 2.

5. Программа работает до тех пор, пока работает хотя бы один процессор: $t(S) = \max_{m_0 \in M} t(S|m_0)$.

Б.2 Вычисление времени для системы с общей шиной или коммутатором

Система с общей шиной [26] позволяет осуществлять одновременно только одну передачу данных между любыми двумя процессорами. Система с коммутатором Fibre channel [21][37] допускает произвольное количество одновременных передач данных, но каждый процессор в любой момент может *принимать* только одну передачу данных. Для этих двух систем время работы программы $t(S)$ может быть вычислено по следующей схеме.

Пусть в расписании используются M процессоров, соответственно, задания на процессорах можно обозначить как $s_{1i}, s_{2i}, \dots, s_{Mi}$. Время выполнения каждого задания $H(v)$ константно. Первоначально время $t = 0$, задания $s_{11}, s_{21}, \dots, s_{M1}$ помечаются как выполняющиеся, остальные – как ожидающие. С каждым выполняющимся заданием и передачей данных ассоциируется счетчик времени выполнения.

1. Продвинуть все счетчики времени выполняющихся передач данных. Если для некоторой передачи e значение счетчика совпало с $F(e)$, эта передача помечается как выполненная.
2. Продвинуть все счетчики времени выполняющихся заданий. Если для некоторого задания s_i значение счетчика совпало с $H(v_i)$, это задание помечается как выполненное. Если используется коммутатор, все дуги, исходящие из s_i, и входящие в задания, расположенные на процессорах, к которым в текущий момент нет передач данных, помечаются как выполняющиеся. Остальные дуги помечаются как ожидающие. Если используется шина, то только одна дуга помечается как выполняющаяся, остальные становятся ожидающими.

3. Если это возможно (все необходимые передачи данных завершились на шаге 1), ожидающие задания помечаются как выполняющиеся.
4. Если это возможно, одна (если используется шина) или несколько (если используется коммутатор) передач данных помечаются как выполняющиеся.
5. Счетчик времени t увеличивается на 1.
6. Если все задания помечены как выполненные, то значение t есть значение времени выполнения расписания. Иначе повторяются все шаги, начиная с 1.

Приложение В. Вычисление времени выполнения с помощью имитационного моделирования

Зачастую в ходе работы синтеза архитектур и построения расписаний вычислить время выполнения расписания аналитически чрезвычайно сложно. Это может быть даже принципиально невозможно, если программа задана исполнимым кодом, а характеристики производительности системы можно оценить только, запустив ее. В таких случаях используются имитационные модели или эмуляторы [1][38]. Таким образом, возникает задача интеграции синтеза архитектур и построения расписаний со средой имитационного моделирования. Так как большинство алгоритмов синтеза архитектур и построения расписаний требуют многократных запусков имитационной модели, то взаимодействие средств планирования со средой моделирования должно быть полностью автоматизированным.

Для таких ситуаций предлагается ввести единый формат представления расписаний и разработать отдельное программное средство, строящее модель системы по представленному в этом формате расписанию. После этого время работы программы может быть определено путем запуска построенной модели в среде CERTI [16][32].

Общая схема интеграции средства синтеза архитектур и построения расписаний и среды выполнения моделей представлена на рисунке В.1 [12][17].

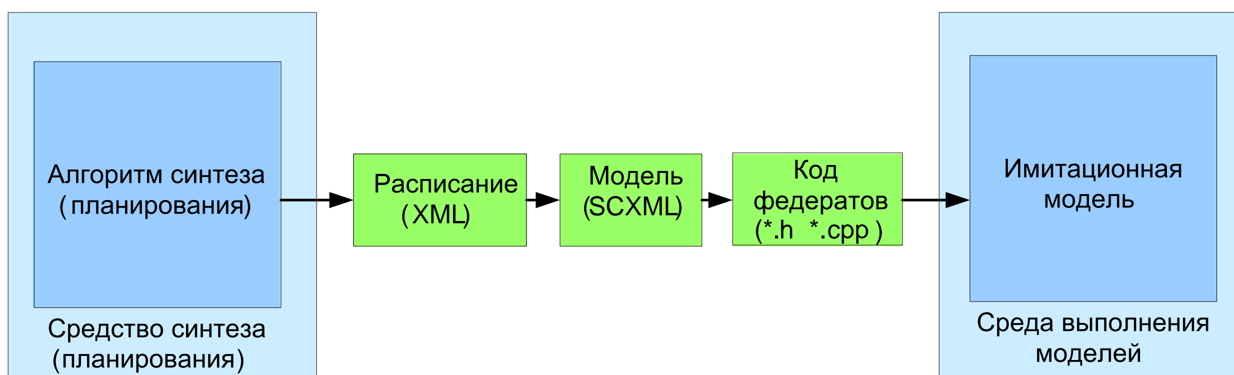


Рисунок В.1. Схема интеграции средств синтеза и имитационного моделирования.

Имитационная модель непосредственно возвращает временную диаграмму выполнения расписания в средство синтеза архитектуры.

В качестве формата представления расписаний используется формат, основанный на XML [24]. При описании расписания используются следующие теги:

<system> – корневой элемент в описании системы. Имеет атрибут rt, принимающий значение 1, если рассматриваемая система является системой жесткого реального времени, и 0 – иначе. Содержит внутри себя теги <processor>.

<processor> – описание процессора. Имеет атрибут id – уникальное имя процессора. Содержит внутри себя теги <task>.

<task> – описание задания. Имеет атрибуты num – порядковый номер задания в расписании; id – уникальное имя задания; time – время выполнения задания на процессоре, к которому привязано данное задание; dirtime – директивный срок выполнения задания; datavol – объем выходных данных. Внутри тега <task> могут содержаться теги <prev> и <next>.

<prev> – задание (атрибут id – имя), от которого текущее задание непосредственно зависит по данным.

<next> – задание (атрибут id – имя), которое зависит по данным от текущего задания.

В инструментальной системе, описанной в разделе 5, предусмотрена возможность экспорта расписания в файл такого формата.

Ниже приведен пример файла, описывающего расписание:

```
<system rt="0">
      <processor id="Processor_1">
            <task id="task_1" num="1" time="5" dirtime="15" datavol="1" >
                  <next id="task_4"></next>
            </task>
            <task id="task_4" num="2" time="10" dirtime="50" datavol="2">
                  <prev id="task_1"></prev>
                  <prev id="task_2"></prev>
                  <prev id="task_3"></prev>
            </task>
      </processor>
      <processor id="Processor_2">
            <task id="task_2" num="1" time="6" dirtime="25" datavol="3">
                  <next id="task_4"></next>
            </task>
            <task id="task_3" num="2" time="8" dirtime="35" datavol="4">
                  <next id="task_4"></next>
            </task>
      </processor>
</system>
```

По описанному в формате XML расписанию строится модель поведения системы, представляющая собой диаграмму состояний в формате SCXML [90]. Далее по ней будет сгенерирован код исполняемой имитационной модели, удовлетворяющей стандарту HLA [65], на языке C++. Данная модель описывает выполнение расписания задач на процессорах с учётом задержек на выполнение задач и обмен данными.

Опишем общую логику построения модели в виде диаграммы состояний.

Всей вычислительной системе соответствует параллельное состояние *system*, включающее в себя составные состояния, соответствующие процессорам.

Каждое состояние, соответствующее процессору, представляет собой последовательный автомат, моделирующий выполнение заданий, привязанных к заданному процессору. -ому заданию соответствует последовательность состояний, начинающаяся состоянием *task_<i>_entry* и завершающаяся состоянием *task_<i>_exit*. $\forall i \in [1, n-1]$существует переход из состояния *task_<i>_exit* в состояние *task_<i+1>_entry*. *task_1_entry* – начальное состояние автомата.

Рассмотрим последовательность состояний, соответствующих i-ому заданию (префикс *task_<i>_* будет опущен).

entry – начальное состояние.

Переход в **waiting.**

waiting – ожидание входных данных.

Переход в **working**:

- условие: для всех j *task_j_ready* == true, j – номера всех заданий других процессоров, от которых необходимо получать данные;
- действия: *current_time* = max{ *task_j_task_i_sending_end* } (значение счетчика времени становится равным времени получения последней порции входных данных).

working – выполнение задания.

Переход в **time_exceeded:**

- условие: (*current_time + time > dir_time)* (превышение директивного срока);
- действия: *task_i_ time_exceeded = true*.

Переход в **sending:**

- условие: (*current_time + time <= dir_time)*;

- действия: *task_i_time_exceeded=current_time+time>dir_time; current_time=time+current_time; task_i_task_j_sending_ready=false;* (*task_j* – непосредственно зависящие от *task_i* задания на других процессорах).

time_exceeded – нарушен директивный срок для системы жесткого реального времени.

Конечное состояние, переходов нет.

sending – состояние перед передачей данных.

Переход в **task_i_task_j_sending:**

- условие: *task_i_task_j_sending_ready*==false (*task_j* – непосредственно зависящие от *task_i* задания на других процессорах).

Переход в **end**:

- условие: для всех i *task_i_task_j_sending_ready*==true;
- действия: *task_i_ready*=true.

task_i_task_j_sending – передача данных от i-ой задачи к j-ой (передачи данных между заданиями одного процессора не моделируются).

Переход в **sending:**

- действие: task*_i_task_j_sending_ready* = true; *current_time = data_vol + current_time;* (время передачи пропорционально объему данных).

end – завершение работы задания.

Переход в **time_exceeded:**

- условие: (*current_time > dir_time)* (превышение директивного срока).

Переход в **exit :**

- условие: (*current_time <= dir_time*).

exit – выход.

Переход в **task_<i+1>_entry**, либо переходов нет.

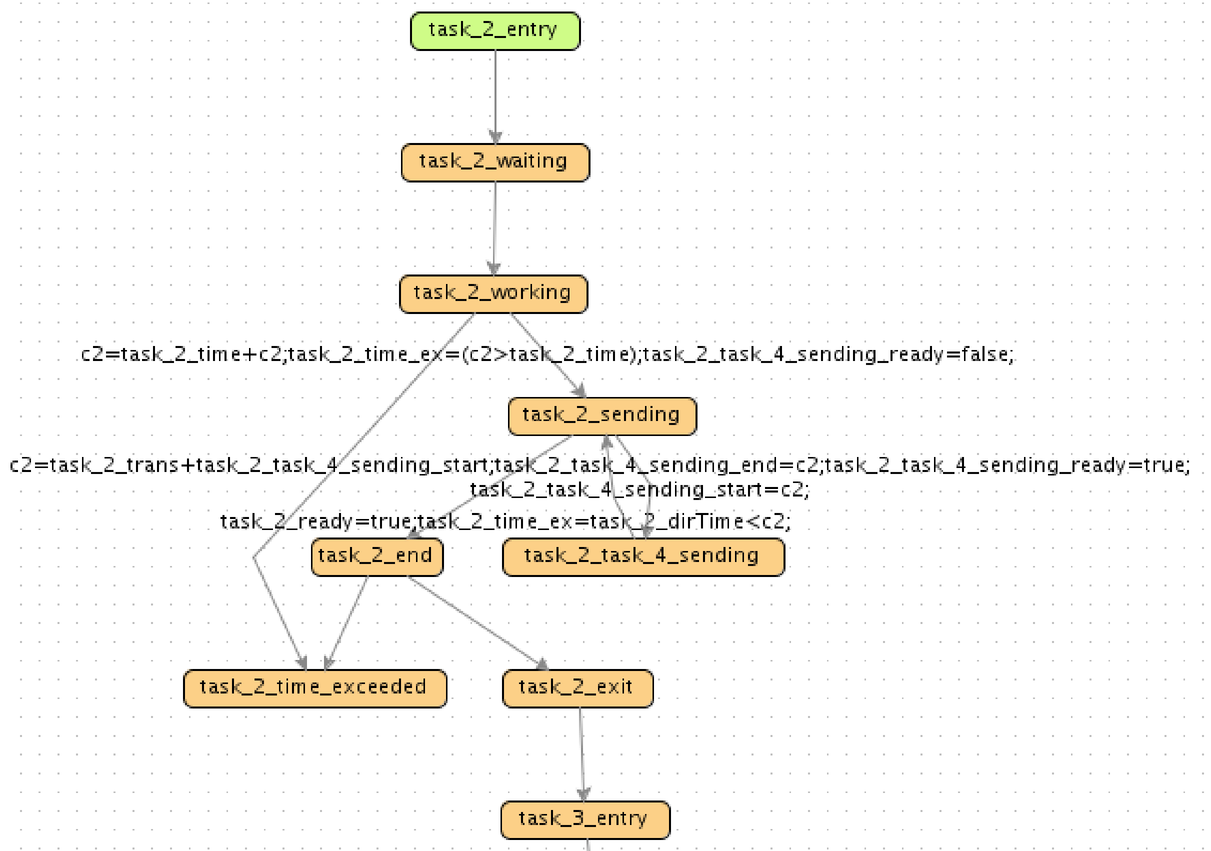


Рисунок В.2. Диаграмма состояний, описывающая процесс обработки задания.

Взаимодействие между автоматами, соответствующими процессорам, осуществляется с помощью глобальных переменных. В терминах среды моделирования CERTI при изменении значения глобальной переменой изменивший ее федерат должен выполнить вызов RTI send interaction с параметром-значением переменной, а все федераты, использующие эту переменную, – receive interaction. В UML-модели такое взаимодействие отображается как переход между состояниями параллельных регионов, соответствующих процессорам. Поле «event» такого перехода содержит имя

глобальной переменной. При генерации кода федерата переходы такого вида рассматриваются как указания на то, что при нахождении автомата, задающего логику работы процессора в некотором состоянии, необходимо выполнить вызов RTI send (receive) interaction.

На рисунке В.2 приведена вышеописанная диаграмма состояний.

Приложение Г. Описание задачи обработки данных от фазированных антенных решеток

В гидроакустике и радиолокации широко используются фазированные антенные решетки (АР) [63]. В зависимости от области применения решетки (бортовая, стационарная, гидроакустическая, радиолокационная и т.д.) требования к производительности и надежности вычислительной системы, осуществляющей обработку данных, могут существенно отличаться [66]. Например, в зависимости от размера решетки, требуемая производительность вычислительной системы может отличаться до 6-7 порядков [96].

Традиционные методы обработки сигналов основаны на быстрых преобразованиях Фурье (БПФ) [25][95]. Однако их потенциальная разрешительная способность ограничена размерами антенной решетки. Альтернативные методы, основанные на автоматической фильтрации шумов и на разложении взаимно-спектральной матрицы (ВСМ) [101] дают достаточно точные результаты даже на небольших антенных решетках, но вычислительная сложность этих методов гораздо выше.

Полагаем сигналы, принимаемые АР из K элементов в полосе частот $(-B, B)$. Пусть сигналы берутся с частотой $A \cdot B = 1/\tau$ (по теореме Котельникова а ≥ 2.5). Если Δf требуемая спектральная разрешающая способность, то число анализируемых элементов разрешения по частоте $L = B/\Delta f$. Входные вектора $r^T(t_i) = (r_1, r_2, \dots, r_k)$ поступают с периодом τ, далее на интервале времени $T = \text{а} \cdot L \cdot \tau$ выполняются K преобразований Фурье (ПФ) на $a \cdot L$ точек и выбирается частотное окно из L составляющих. Для методов сверхразрешения и адаптивных методов, которые работают с оценкой взаимно-спектральной матрицы (ВСМ), для каждого элемента разрешения по частоте проводится её вычисление $\hat{\Gamma}(f_i)_{K \times K} = \frac{1}{n}\sum_{j=1}^{n} X_j(f_j) X_j^{*T}(f_i), i = 1, \dots, L$, где $X_j(f_i)$ – дискретное ПФ

входного сигнала, вычисляемое для последовательных промежутков времени τ. Количество выборок n должно позволять оценивать статистики второго порядка анализируемых сигналов. На следующих этапах осуществляется идентичная обработка матриц $\hat{\Gamma}(f)$ для каждой частотной составляющей независимо.

В методах сверхразрешения (МСР) разрешающая способность не ограничивается отношением сигнал/шум, как это имеет место в адаптивных системах, а асимптотически возрастает с увеличением времени наблюдения до бесконечности. В этих методах выделяется подпространство сигналов E_3 и ортогональное ему подпространство шума $E_3^{\perp}$. Для чего вычисляются собственные значения вектора ВСМ: $\hat{\Gamma}(f_i) = V(f_i) \cdot \Lambda(f_i), i = 1, \dots, L$, где $V = [V_1, V_2, \dots, V_K]_{K \times K}$ – матрица собственных векторов, $\Lambda = diag[\lambda]$ – диагональная матрица собственных значений. Старшие M собственных векторов образуют подпространство сигналов $(V_i, i = 1, \dots, M)$ и соответствуют собственным значениям $\lambda_1 = \lambda_{si} + \delta$, где M – число источников сигналов, δ– интенсивность шума [86][91]. Младшие $K - M$ собственных векторов $(V_i, i = M + 1, \dots, K)$ ортогональны всем предыдущим векторам и всем векторам источников сигналов и соответствуют собственным значениям $\lambda_{M+1} \approx \lambda_{M+2} \approx \dots \approx \lambda_K \approx \delta$.

Далее после выделения E_3 и $E_3^{\perp}$ определяют координаты источников сигналов $F(f_i, \theta) = \left(\sum_{i=N+1}^{K} |V^{*T}(f_i) \cdot U(f_i, \theta)|^2\right)^{-1}, i = 1, \dots, L$, тогда $\lim_{\theta \to \theta_j} F(f_i, \theta) = \infty, i = 1, \dots, L, k = 1, \dots, N, N \geq M$, где θ – переменная, связанная с пространственными угловыми координатами, $U(f_i, \theta)$ – опорные вектора направления. Зная угловые координаты источников сигналов, легко можно сформировать вектора источников и определить спектральные плотности.

Адаптивные методы (АМ), основанные на непосредственном обращении матриц, имеют наибольшую скорость сходимости. Они до некоторого

критического уровня нечувствительны к плохой обусловленности ВСМ. Уровень нечувствительности может быть увеличен наращиванием разрядности вычислительных средств, осуществляющих обращение ВСМ. В этих методах после получения оценки ВСМ вычисляется обратная ей матрица $\Gamma^{-1}(f_i)$, затем производится оценка пространственно-частотного спектра поля, формирование весовых коэффициентов для диаграммообразования. Например, используя метод Клейптона (линейное ограничение) можно получить

$$W(f_i, \theta_j) = \frac{\Gamma^{-1}(f_i) \cdot U(f_i, \theta_j)}{U^{*T}(f_i, \theta_j) \cdot \Gamma^{-1}(f_i) \cdot U(f_i, \theta_j)}, i = 1, \dots, L, j = 1, \dots, N, N \geq M;$$

$$S(f_i, \theta_j) = \frac{1}{U^{*T}(f_i, \theta_j) \cdot \Gamma^{-1}(f_i) \cdot U(f_i, \theta_j)}, i = 1, \dots, L, j = 1, \dots, N, N \geq M,$$

где $U(f_i, \theta_j)$ – вектор управления, необходимый для фокусирования АР в требуемом направлении θ_j. Для получения значений W и S могут использоваться также алгоритмы с квадратичным ограничением и ограничением 3/2, но основная часть вычислительной нагрузки независимо от модификации метода связана с нахождением квадратичных и билинейных форм. Они могут параллельно вычисляться не только для каждой частоты, но и для каждого вектора.

В таблице Г.1 приведены основные этапы [53][54] методов нахождения координат источников сигнала, их вычислительная сложность, объёмы входных и выходных данных, т.е. в этой таблице приведено описание графа потока данных алгоритма нахождения координат источников сигнала (рисунок Г.1 [53]). При этом пятый этап вычисления для методов сверхразрешения и адаптивных методов имеет различную вычислительную сложность. Объёмы входных и выходных методов указаны для каждой входящей и исходящей дуги в словах (слово 4 байта).

Таблица Г.1. Основные этапы методов нахождения координат источников сигнала.

Этап	Операция	Сложность операции	Вх. данные	Вых. данные	Дир. срок	Кол-во операций в этапе
1	Нормализация	$O(aL)$	$-$	aL	$T = \mathrm{a} \cdot L \cdot \tau$	K
2	БПФ	$O(aL \cdot \log_2 aL)$	aL	1	$T = \mathrm{a} \cdot L \cdot \tau$	K
3	Внешнее произведение комплексных векторов	$O(K^2)$	1	K^2	$T = \mathrm{a} \cdot L \cdot \tau$	L
4	Нахождение с.з. и с.в., обращение матрицы	$O(K^3)$	K^2	K^2	$T \cdot n$	L
5 (МСР)	Вычисление F	$O(K)$	K^2	K	$T \cdot n$	L
5 (АМ)	Вычисление W, S	$O(K^2)$			$T \cdot n$	L
6	Скалярное произведение	$O(K)$	K	1	$T \cdot n$	$L \cdot M$
7	Сравнение	$O(K)$	1	K	$T \cdot n$	L

8	Сравнение векторов	$O(L \cdot K)$	K	$-$	$T \cdot n$	1

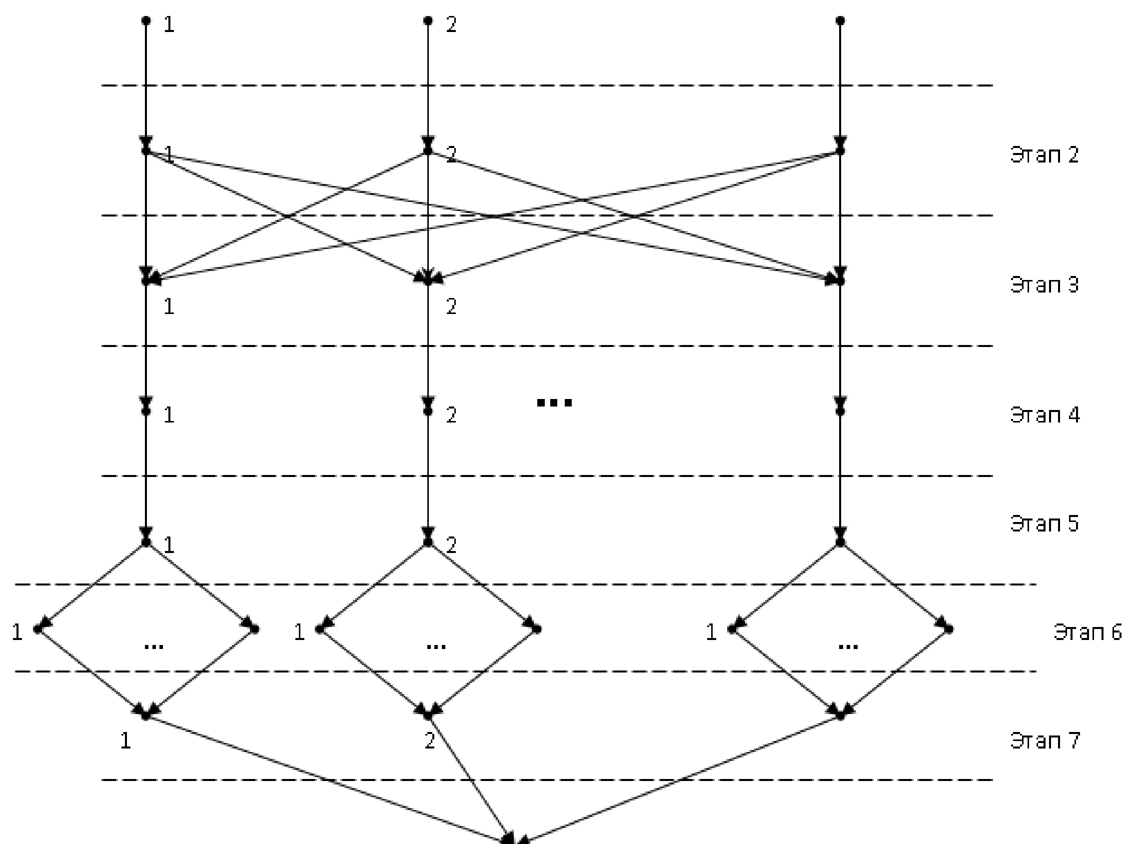


Рисунок Г.1. Граф работы алгоритма нахождения координат источников сигнала.