С.В. Назаров

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Монография

RU SCIence RU-SCIENCE.COM
Москва
2022

УДК 004.4
ББК 32+32.973
Н19

Рецензент:
Ю.Б. Зубарев, член-корреспондент РАН, д-р техн. наук, проф.

Назаров, Станислав Викторович.
Н19 Программное обеспечение систем реального времени : монография / С.В. Назаров. — Москва : РУСАЙНС, 2022. — 212 с.

ISBN 978-5-4365-9770-6

Компьютерные системы реального времени являются важнейшей составной частью многих объектов контроля и управления – от бортовых вычислительных систем космических аппаратов и ракет до ситуационных центров различного назначения. В промышленности они широко используются в управлении технологическими процессами, в робототехнике, здравоохранении, в высокоточных отраслях, таких как нефтегазовая промышленность, связь и энергетика, поскольку эти организации используют данные реального времени для постоянного совершенствования безопасности, эффективности и надежности.

В монографии рассмотрены вопросы разработки систем реального времени и приведены примеры решения ряда задач, связанных с коррекцией и оптимизацией архитектуры программного обеспечения таких систем

Ключевые слова: *программное обеспечение; системы реального времени; архитектура; коррекция; оптимизация.*

УДК 004.4
ББК 32+32.973

ISBN 978-5-4365-9770-6

Предисловие

Существует достаточно широкий класс объектов, в которых применяются аппаратно-программные комплексы, реагирующие в предсказуемые времена на непредсказуемый поток внешних событий в окружающей среде. Такие комплексы являются системами реального времени (СРВ или RTS – riel time system), которые способны обеспечить требуемый уровень сервиса в заданный промежуток времени. Реальным временем характеризуют такую реакцию объекта на входные сигналы или данные, при которой он успевает достаточно быстро выработать выходные сигналы (данные). Реальное время относится к системе или режиму работы, в котором вычисления проводятся в течение времени, определяемого внешним процессом, с целью управления или мониторинга внешним процессом по результатам этих вычислений. Таким образом, СРВ-система, в которой процесс обработки данных (сигналов) происходит настолько быстро, что его промежуточные результаты могут быть использованы в управлении этим процессом.

Условия работы системы в реальном времени выполняются в том случае, если ее быстродействие адекватно скорости протекания физических процессов на объектах контроля и управления, которые непосредственно связанны с функциями объекта. Из приведенных определений следует, что СРВ призваны решать задачи, в которых важны не только правильность решения, но и сроки, в которые эти решения принимаются. В зарубежной литературе срок, в пределах которого должно быть принято решение называется deadline (критический срок обслуживания).

Типичные времена реакции на внешние события в СРВ составляют от микросекунд до нескольких часов. Так, например, при решении задач математического моделирования – это несколько микросекунд, в радиолокации – несколько миллисекунд, в складском учете – несколько секунд, в системах резервирования различного назначения и торговых операциях – десятки секунд и минуты, в управление производством – несколько минут. В настоящее время определились предметные области с наиболее интенсивным использованием СРВ. Прежде всего следует отметить военную и космическую области с наиболее перспективными научными и техническими решениями по следующим направлениям:

- бортовое и встраиваемое оборудование;
- системы измерения и управления;
- радары;
- цифровые видеосистемы;

• симуляторы (тренажеры);

• ракеты различного назначения;

• системы определения положения и привязки к местности.

Следующей областью интенсивного использования СРВ является промышленность, здесь прежде всего отметим автоматизированные системы управления производством (АСУП) и автоматические системы управления технологическими процессами (АСУТП). К этой же области следует отнести следующие ветви промышленности:

• авиастроение – бортовые системы управления (авиадвигателем и другими подсистемами, в том числе и управление автопилотом самолета);

• автомобилестроение – системы управления мотором, системы антиблокировки колес, автоматическое сцепление, симуляторы и др.;

• энергетика – сбор информации, управление системами электроснабжения, данными и оборудованием;

• телекоммуникации – коммуникационное оборудование, сетевые коммутаторы, телефонные станции и др.;

• банковское оборудование: (банкоматы – ОСРВ QNX);

• товары широкого потребления – мобильные телефоны; цифровое телевидение; компьютерное и офисное оборудование (факсы – ОСРВ VxWorks), (CDROM – ОСРВ VRTX32).

Это далеко неполный перечень. Используются СРВ и в других областях народного хозяйства, например, в медицине для управления сложным медицинским оборудованием (томографы, сканеры и т.п.), для ведения электронных медицинских карточек, в системах наблюдения за больным в палате интенсивной терапии и много другого.

Во всех случаях использования СРВ к ним предъявляется следующие требования:

• гарантированное время выполнения задач, т.е. при выполнении любого набора задач, все задачи останутся внутри своих временных границ. Это значит, что система должна быть предсказуемой (predictable).

• поддержка параллельного выполнения нескольких задач;

• максимальное время отклика на событие, а не среднее, не должно превышать заданного значения;

• возможность безотказной работы в течение длительного периода времени.

В зависимости от временных требований, предъявляемых к СРВ, они могут быть разделены на МСРВ – системы мягкого реального времени (soft real time system, SRTS) и системы ЖСРВ – жесткого реального

времени – (hard real time system, HRTS). Предъявляемые временные характеристики обработки событий должны удовлетворяться обязательно и очень строго. Не допускается никаких задержек реакции системы, так как может произойти катастрофа в случае задержки реакции или результаты могут оказаться бесполезны в случае опоздания, т.е. стоимость опоздания может оказаться бесконечно велика. Примерами систем ЖСРВ могут служить бортовые системы управления, системы аварийной защиты, регистраторы аварийных событий системы безопасности, контроля и и управления, системы оцифровки звука/изображения.

В системах мягкого реального времени отступление от заданных временных параметров не приводит к нарушению работы системы. Задержка реакции не критична, хотя и может привести к увеличению стоимости результатов и снижению производительности системы в целом. В таких системах за опоздание результатов приходится платить – приемлемо снижение производительности системы, вызванное запаздыванием реакций. Примерами МСРВ могут быть интерактивные системы, для которых время реакции на действия пользователя должно быть, если не нормированным, то хотя бы предсказуемым и стабильным. Например, это работа с пакетами в сети, автомат розничной торговли, обработка данных с метеостанций.

Отличие ЖСРВ от МСРВ заключается в том, что ЖСРВ – никогда не опоздает с реакцией на событие, а МСРВ – не должна опаздывать с реакцией на событие. Вообще СРВ в большинстве случаев решают комбинацию задач жесткого и мягкого реального времени, а также задач, не имеющих критического срока обслуживания. Задача может переходить из статуса мягкого реального времени при пропуске некоторого срока обслуживания в статус задачи жесткого реального времени назначением критического срока обслуживания.

Существует несколько определений понятия реального времени, часто противоречащих друг другу, что не позволяет, к сожалению, принять единую терминологию. Близким к каноническому можно назвать следующее определение: "Система реального времени – это такая система, корректность работы которой зависит не только от выполнения неких заданий, но и от времени их выполнения. Если временные параметры задания нарушены - оно считается невыполненным". Дополнение к этому определению: "Следовательно, сама система должна иметь гарантированные временные параметры, т.е. поведение системы должно быть предсказуемым. Это позволяет минимизировать количество невыполненных (вследствие нарушения временных параметров) заданий".

Хорошим примером системы реального времени является робот, который берет деталь, движущуюся по конвейеру. Если он опоздает, то пропустит один цикл работы конвейера, а попытка взять деталь слишком рано может заблокировать движение других деталей. Другой пример - самолет, летящий на автопилоте. Специальные датчики определяют положение самолета в трехмерном пространстве. Только постоянное и своевременное получение этих данных бортовым компьютером гарантирует безопасность полета.

Актуальность тематики параллельных вычислений была осознана достаточно давно при решении сложных научно-технических задач, как в связи с низкой надежностью и производительностью компьютеров, так и в связи с появлением многопроцессорных систем и многоядерных процессоров. Технология обеспечения надежности и высокой производительности на основе параллельных вычислений естественным образом стала преобладающей в бортовых вычислительных системах (БВС). В настоящее время такие системы находят широкое применение в авиационной и космической технике, а также в наземных и водных подвижных объектах.

В данной монографии представлена совокупность математических моделей, формулировок задач и подходов к их решению, позволяющих построить расписание параллельного вычислительного процесса для реализации информационно-связанных задач на многопроцессорных бортовых вычислительных системах. Даны модели наборов решаемых задач в форме нагруженного графа и в ярусно-параллельной форме, решение задач о назначениях решаемых задач на процессоры и алгоритм составления расписания параллельного вычислительного процесса.

В монографии рассмотрены некоторые вопросы организация вычислительного процесса при выполнении информационно-связанных задач в СРВ. Организация вычислительного процесса должна учитывать особенности архитектуры компьютера и его операционной системы. Решение представляется последовательностью связанных этапов. Для решения используются методы сетевого планирования и управления, теория параллельных вычислительных процессов и расписаний.

В первой главе рассмотрены теоретические вопросы разработки методов и моделей коррекции архитектуры программных систем реального времени (СРВ). Дается общее понятие архитектуры программной системы и рассматриваются особенности программах систем реального времен. Приводится классификация таких систем, рассматриваются типовые архитектуры СВР, обсуждаются их достоинства и недостатки, а

также сферы использования в тех или иных системах. Рассмотрены модели формального представления архитектур программных систем. Подробно рассмотрены вопросы расслоения программной системы и алгоритмы выделения слоев программы.

Вторая глава монографии непосредственно посвящена методам коррекции архитектуры программной системы и архитектурным преобразованиям программной системы. Разработаны методы коррекции архитектуры программных систем и приведены примеры использования этих методов для изменения архитектуры программной системы по определенному требованию (например, в связи с архитектурой аппаратуры бортовой системы), а также оптимизации архитектуры для повышения производительности системы.

Следующие три главы книги рассматривают различные задачи, связанные с решением вопросов организации вычислительного процесса бортовых вычислительных систем жесткого реального времени. В третьей главе дается постановка задачи организации вычислительного процесса бортовой системы реального времени. Отмечаются особенности современных бортовых систем.

Технология обеспечения надежности и высокой производительности на основе параллельных вычислений естественным образом стала преобладающей в бортовых вычислительных системах (БВС). В настоящее время такие системы находят широкое применение в авиационной и космической технике, а также в наземных и водных подвижных объектах. Эффективность выполнения поставленных задач, безопасность, эксплуатационная пригодность и ряд других важных качеств подвижных объектов в значительной мере определяются способностью бортовой вычислительной системы выполнять свои функции.

Актуальность тематики параллельных вычислений не вызывает сомнений при решении сложных научно-технических задач, как в связи с низкой надежностью и производительностью компьютеров, так и в связи с появлением многопроцессорных систем и многоядерных процессоров. В четвертой главе монографии рассмотрены проблемные вопросы оптимизации параллельных вычислительных процессов бортовых систем жесткого реального времени. Для разработки постановок задач организации параллельных вычислительных процессов и их решения использованы методы сетевого планирования и управления в совокупности с математическим аппаратом ярусно-параллельных графов и методами математического программирования.

Известно, что современные облачные технологии обеспечивают высокий уровень безопасности и конфиденциальности хранения данных, высокую степень надежности и неограниченные вычислительные ресурсы. "Облако" позволяет сэкономить на покупке лицензионных продуктов, экономично и удобно в использовании. В результате применения облачных технологий уменьшаются расходы на приобретение дорогостоящих мощных компьютеров, серверов, сокращаются затраты на оплату труда ИТ-специалистов, обслуживающих локальный дата-центр. Естественно, возникает вопрос реализации некоторых классов СРВ на основе облачных ресурсов. В пятой главе книги рассматривается возможность создания системы реального времени на основе web-серверов, вычислительная мощность которых формируется за счёт арендованной инфраструктуры IaaS (Infrastructure-as-a-Service – инфраструктура как сервис).

В шестой главе монографии рассматриваются вопросы организации вычислительного процесса в системах мягкого реального времени. С появлением, суперкомпьютеров стало реальностью решение многих научных проблем в области математики, физики, механики, астрофизики, биоинформатики, науки о Земле и мировом океане, обучение искусственного интеллекта, распознавание изображений и многое другое. Решение этих задач не требует режима жесткого реального времени, но весьма желательно время решения ограничить. В противном случае решение теряет актуальность. Вспомним прогноз погоды или распознавание личности пассажира, проходящего через контрольно-пропускной пункт метро или сотрудника предприятия, пришедшего на свою смену.

Основной материал шестой главы посвящен вопросам оптимизация вычислительного процесса при выполнении информационно-связанных задач в суперкомпьютерах. Даже при высокой производительности суперкомпьютеров, предназначенные для них задачи, могут занимать значительное время и выполняться, как правило, неоднократно. Программы, реализующие эти задачи, имеют сложную многомодульную структуру с информационными и управляющими связями между ними. Поэтому они реализуются в пакетном режиме и часто в ночное время. В пакеты могут входить различные задачи, требующие высокой вычислительной мощности и большой основной и внешней памяти.

Желающих использовать суперкомпьютеры всегда много. В связи с этим даже в рамках мощнейших суперкомпьютеров требуется оптимизация вычислительного процесса с целью рационального расходования вычислительных ресурсов при одновременном требовании выполнения

определенного набора задач (пакета) в заданные временные ограничения. Организация вычислительного процесса суперкомпьютеров должна учитывать особенности их архитектуры. На это обращено внимание в шестой главе при рассмотрении задачи планирования вычислительного процесса в суперкомпьютере IBM.

В седьмой главе книги рассматривается задача распределения вычислительных ресурсов в системах мягкого реального времени применительно к широкому классу информационных систем управления предприятиями. Информационная система (ИС) предприятия по своей сути является автоматизированной системой управления (АСУ). Ее задача обеспечивать на основе использования экономико-математических методов, технических средств и организационных комплексов рациональное управление предприятием или технологическим процессом. Такие системы широко используются в хозяйственной практике научно-исследовательских, конструкторских и производственных предприятий, выполняющих разработку, производство и сбыт какой-либо продукции.

В заключительной восьмой главе монографии рассматриваются вопросы рефакторинга программных систем СРВ, получившие в настоящее время большую актуальность. Известно, что чаще приходится читать и модифицировать код, а не писать новый. В основе поддержки читаемости и модифицируемости кода лежит рефакторинг – как в частном случае для конкретной программы, так и для программного обеспечения в целом. С рефакторингом связан известный риск. Он требует внести изменения в работающий код, что может привести к появлению трудно находимых ошибок в программе. Неправильно осуществляя рефакторинг, можно потерять дни и даже недели. Существует несколько методов рефакторинга. Понимание техники таких методов рефакторинга важно для организованного осуществления рефакторинга. С помощью методов рефакторинга можно поэтапно модифицировать код, внося каждый раз небольшие изменения, благодаря чему снижается риск, связанный с развитием проекта.

Автор

Содержание

Глава 1. МЕТОДЫ И МОДЕЛИ КОРРЕКЦИИ АРХИТЕКТУРЫ ПРОГРАММНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

1.1. Понятие архитектуры ПС

Известно, что с появлением вычислительного машиностроения термин архитектура первоначально применялся только к аппаратному обеспечению (архитектура ЭВМ, вычислительных комплексов, сетей и др.). С развитием информационных технологий применение термина “архитектура” существенно расширилось. Сейчас широко используются понятия: архитектура программного обеспечения, архитектура операционной системы, архитектура системы сетевых протоколов, архитектура данных и др. В связи с ростом сложности компьютерных систем и необходимости рассмотрения систем на различном уровне их детализации появились такие понятия, как архитектура процессора, архитектура набора команд и даже более детальные уровни.

Начало архитектуре программного обеспечения как концепции было положено в научно-исследовательской работе Э. Дейкстры в 1968 году и Д. Парнаса в 1970-х. Это были первые работы, посвященные архитектуре программных систем. В этих работах была подчёркнута критическая важность разработки правильной структуры программной системы (термин архитектура еще не использовался). На сегодняшний день до сих пор нет единого мнения в отношении определения термина "архитектура программного обеспечения". Многие специалисты в настоящий момент считают эту дисциплину совокупностью положений без четких правил о "правильном" пути создания системы, а само проектирование архитектуры смесью науки и искусства. Популярность изучения этой области возросла с начала 1990-х годов вместе с научно-исследовательской работой по исследованию архитектурных стилей (шаблонов), языков описания и документирования архитектуры, и формальных методов проектирования. Результатами подобных исследований являются популярные монографии, например, книга Л. Басса и др. “Архитектура программного обеспечения на практике” [1].

Первым стандартом программной архитектуры явился стандарт IEEE 1471: ANSI / IEEE 1471 – 2000: Рекомендации по описанию преимущественно программных систем. Он был принят в 2007 году, под

названием ISO / IEC 42010:2007 [2]. Как ни странно, несмотря на многолетнее развитие соответствующей дисциплины, до сих пор нельзя дать краткий и ясный ответ на вопрос: что такое программная архитектура? Общепринятых определений не существует. На сайте Института программной инженерии (Software Engineering Institute, SEI) [3], посвященном практическим аспектам архитектуры, приводится несколько определений. И хотя отсутствие согласия относительно такого определения не стало существенным препятствием к развитию самой дисциплины, его достижение оказалось весьма трудной задачей.

При проектировании любая программная система (ПС) может рассматриваться с разных точек зрения: поведенческой (динамической), структурной (статической), логической (удовлетворение функциональным требованиям), физической (распределенность), реализационной (как детали архитектуры представляются в коде) и т.п. Эти различные архитектурные представления или виды (view) в совокупности дают полную информацию об архитектуре ПС. При этом каждое отдельное архитектурное представление может быть определено, как частный аспект программной архитектуры, представляющий специфические свойства программной системы, наиболее интересные для конкретного заинтересованного лица.

Классическим источником комплекса архитектурных точек зрения и представлений, построенных в системе координат "вопрос-уровень детализации", является модель Дж. Захмана [4]. Каждое архитектурное представление является результатом ответа на вопрос: как? что? где? и т.п. в контексте необходимого уровня абстракции. Например, физическая модель данных (Physical Data Model) является ответом на вопрос "что?" в контексте технологической модели, а логическая модель данных, отвечая на тот же вопрос, находится на один уровень абстракции выше – в контексте системной или логической модели.

Таким образом, можно говорить о том, что каждое архитектурное представление ПС A_i описывает некоторую сторону частного решения по архитектуре ПС. В совокупности можно считать, что полное решение по определению архитектуры AS ПС представляется объединением вида

$$AS = \cup A_i,\ i = 1,2,\dots,n,$$

где n – число различных представлений архитектуры ПС.

Стандарт IEEE 1471 определяет представление архитектуры как согласованный набор документов, описывающий архитектуру с точки зрения определенной группы заинтересованных лиц с помощью набора

моделей. Стандарт рекомендует для каждого представления фиксировать отраженные в нем взгляды и интересы, роли лиц, которые заинтересованы в таком взгляде на систему, причины, обусловливающие необходимость такого рассмотрения системы, несоответствия между элементами одного представления или между различными представлениями, а также различную служебную информацию об источниках информации, датах создания документов и пр.

Заметим, что каждое архитектурное представление ПС A_i может иметь множество возможных и допустимых вариантов, и задача архитектора заключается в нахождении такого варианта V_i архитектурного представления A_i , которое является оптимальным по некоторому частному показателю при условии непротиворечивости решений V_j по другим архитектурным представлениям $A_j,\ j \neq i,\ i,j = 1,2,\dots,n.$ Это условие можно записать следующим соотношением

$$V_i \cap V_j = \emptyset, j \neq i, \qquad i,j = 1,2,\dots,n.$$

Далее в статье рассматриваются архитектурные представления бортовых ПС и вопросы архитектурного синтеза таких систем.

1.2. Особенности систем реального времени

Специализированные системы реального времени как бортовые вычислительные системы имеют разнообразное применение. В данной монографии рассматриваются системы, которые используются для управления и иных целей (стабилизации, защиты, безопасности и др.) в подвижных объектах космического, воздушного и наземного характера, например, летательных аппаратах (дронах, ракетах боевого применения различных классов, самолетах и др.), транспортных средствах, автомобилях и т. п. Как правило, наборы программ, реализуемых в таких объектах современными бортовыми вычислительными системами (БВС), можно представить совокупностью информационно-связанных подпрограмм (заданным набором задач – ЗНЗ). Это позволяет представлять решаемые задачи в виде нагруженного графа и, соответственно, использовать для их анализа методы теории графов и сетевого планирования и управления. Обычно ЗНЗ обладают достаточным внутренним параллелизмом, который может быть удобно использован при реализации задач многопроцессорными (многоядерными) бортовыми системами.

Это особенно важно, поскольку БВС, как правило, учитывая специфику управляемых объектов, работают в режиме реального времени с ограничениями на время выполнения ЗНЗ. Использование многопроцессорных (многоядерных) часто характеризуется тем, что архитектура

аппаратных средств многопроцессорной системы зачастую фиксируется на этапе проектирования и остается неизменной на время выполнения программ. Это означает, что разработчик должен выбирать соответствующую систему для каждого предполагаемого приложения. С этой целью разработчик должен разбить приложение на части, рассмотреть возможности параллелизма при выполнении приложения и выбрать соответствующую систему для своего приложения. Ограничением такого подхода является недостаток распространенных существующих многопроцессорных систем, заключающийся в отсутствии адаптивности на стадии разработки и во время выполнения программы [5].

Программируемая разработчиком логическая матрица предлагает более гибкое решение, потому что с ее помощью аппаратные средства можно переконфигурировать с новыми функциями и использовать повторно с различными приложениями. Некоторые поставщики программируемых пользователем логических матриц (компания Xilinx) предлагают специальную функцию, называемую динамическим и частичным переконфигурированием [6]. Это означает, что часть аппаратных средств системы может быть изменена во время выполнения программы, а оставшаяся часть остается действующей и неизменной. Программируемая логика от Xilinx позволяют применять интеллектуальные решения в широчайшем спектре отраслей промышленной, научной и потребительской деятельности человека. Это программируемые логические интегральные схемы ПЛИС (FPGA), 3D микросхемы и системы на SoC (System-on-a-Chip), которые устанавливают стандарты низкой стоимости, высокой производительности и пониженного энергопотребления. Архитектура Xilinx UltraScale™ обеспечивает беспрецедентный уровень интеграции и возможностей, обеспечивая при этом производительность на системном уровне ASIC-класса для самых требовательных приложений, требующих высочайшей пропускной способности, быстродействия памяти, массовых потоков данных, DSP и производительности обработки пакетов [7].

Рассмотрим модели архитектуры бортовых программных систем, учитывая указанные выше особенности задач (внутренний параллелизм, временные ограничения), решаемых в бортовых системах, и возможности бортовых вычислительных средств (жесткие структуры, переконфигурация, адаптация и др.).

1.3. Типовые архитектуры программных систем

В 1999 году состоялась первая конференция по программной архитектуре, были созданы рабочая группа IFIP Working Group 2.10 и институт Worldwide Institute of Software Architects. В надежде повысить практическую ценность описания архитектуры группа Open Group на базе XML создала язык Architecture Description Markup Language, который обеспечил совместное использование архитектурных моделей. Объединенные усилия сообществ сторонников многократного использования кода привели к появлению специальных продуктов и методов, привлекших внимание крупных компаний. Так, например, появились и сформировались методы SAAM (Software Architecture Analysis Method), BAPO (Business Architecture Process Organization) и ATAM (Architecture Tradeoff Analysis Method). К имевшемуся общему стандарту архитектуры RM-ODP (Reference Model of Open Distributed Processing) добавился IEEE 1471 [8].

В крупных компаниях появились собственные главные архитекторы. Стало наблюдаться некоторое превосходство программного архитектора над разработчиком. Архитектор решает разные вопросы, но часто считает архитектурой и то, что ей не является. Появились выразительные языки описания архитектуры ADL, однако на практике используются лишь немногие из них. Исключением стали Koala [9] и UML [10, 11], если воспринимать их как язык описания архитектуры. Для ряда предметных областей появились готовые архитектурные решения в виде платформ, например, J2EE, .Net, Symbian/Series 60 и Websphere. Стандарты обмена данными на уровне приложений, например, XML и SOAP, оказали на них значительное влияние. Языки сценариев изменили привычные способы конструирования систем. Архитекторы стали строить системы на основе своих представлений о возможностях этих платформ. Удалось придумать надежные подходы, чтобы устранить недостатки проектирования без архитектуры. Ниже перечислены некоторые из самых известных архитектур программных систем [12]:

– многослойная архитектура (Layered Architecture),
– многоуровневая архитектура (Tiered Architecture),
– сервис-ориентированная архитектура (Service-oriented Architecture, SOA),
– микросервисная архитектура (Microservice Architecture).

Кратко рассмотрим каждую из них и отметим их положительные стороны и недостатки.

Многослойная архитектура (обычно трехслойная)

В данном случае подход работает по принципу разделения ответственностей. Программное обеспечение разделено на слои, лежащие друг на друге, и каждый слой выполняет определенные функции.

Слой представления (Presentation Layer) реализует пользовательский интерфейс и отвечает за обеспечение хорошего пользовательского опыта.

Слой бизнес-логики (Business Logic Layer) содержит бизнес-логику приложения. Он отделяет UI/UX от вычислений, связанных с бизнесом. Это позволяет легко изменять логику в зависимости от постоянно меняющихся бизнес-требований, никак не влияя на другие слои.

Слой передачи данных (Data Link Layer) отвечает за взаимодействие с постоянными хранилищами, такими как базы данных, и прочую обработку информации, которая не связана с бизнесом.

Данные и элементы управления проходят через каждый слой и передаются от одного к другому. Эта система повышает в некоторой степени стабильность ПО, поскольку предлагает абстракцию, благодаря разделению функций между уровнями. Кроме того, к преимуществам этой архитектуры относятся более простая реализация по сравнению с другими подходами. Изолирование защищает одни слои от изменений других. При этом повышается управляемость программного обеспечения за счет слабой связанности.

Есть и недостатки. Архитектура не предлагает большой масштабируемости. Система, созданная с таким подходом, будет иметь монолитную структуру, усложняющую внесение модификаций. Данные должны проходить по каждому слою, даже если нет необходимости передавать их с определенных слоев, что может снизить быстродействие.

Многоуровневая архитектура

Этот архитектурный подход разделяет комплекс ПО на уровни по принципу взаимодействия “клиент-сервер”. Архитектура может иметь один, два и больше уровней, разделяющих ответственности между поставщиком данных и потребителем. Такой подход использует шаблон Request Response (запрос-ответ) для связи между уровнями. В отличие от многослойной архитектуры, этот подход предлагает масштабируемость, которая может быть как горизонтальной (масштабирование сети с помощью высокопроизводительных узлов), так и вертикальной (масштабирование каждого узла путем повышения его производительности).

Одноуровневая система – единая система работает как на стороне сервера, так и клиента. Обеспечивает простоту развертывания и отличную скорость связи, а также устраняет необходимость межсистемного взаимодействия. Такая система подходит только для небольших однопользовательских приложений.

Двухуровневая система состоит из двух физических машин в качестве сервера и клиента. Она обеспечивает изоляцию операций управления данными, обработки данных и операций представления. Клиент содержит слои презентации, бизнес-логики и передачи данных, сервер включает хранилища и базы данных.

Трехуровневая и n-уровневая системы обладают высокой масштабируемостью как по горизонтали, так и по вертикали. Как правило, это сложные дорогие системы, обеспечивающие высокую производительность. Этот подход хорошо сочетается с современной сервис-ориентированной архитектурой. Его реализация может оказаться дорогостоящей с точки зрения времени и ресурсов и рекомендуется для сложных ПО, требующих высокой производительности и масштабируемости.

Сервис-ориентированная архитектура (SOA)

Эта архитектурная модель состоит из компонентов и приложений, которые связываются друг с другом с помощью четко определенных сервисов. Она состоит из 5 элементов: собственно сервисы; сервисная шина; сервисный репозиторий; безопасность SOA; управление SOA.

Клиент отправляет запрос с использованием стандартного протокола и формата данных по сети. Этот запрос обрабатывается сервисной шиной, которая считается основой сервис-ориентированной архитектуры и отвечает за оркестровку и маршрутизацию. С помощью сервисного репозитория запрос направляется в специальный сервис, который может взаимодействовать с другими сервисами и базами данных, чтобы составить полезную нагрузку (данные) ответа.

Микросервисная архитектура

При таком подходе приложение разрабатывается как набор небольших сервисов, каждый из которых работает в собственном процессе и связывается с легковесными механизмами, обычно API для HTTP-ресурса. Эти сервисы основываются на бизнес-возможностях и могут развертываться независимо друг от друга с помощью полностью автоматизированного механизма. Централизованное управление между сервисами минимально. Они могут быть написаны на разных языках, использовать разные технологии хранения данных.

Архитектура работает по принципу компонентизации сервисов. Она разделяет программное обеспечение на различные изолированные компоненты (сервисы), каждый из которых несет единую ответственность. Изменения в одной сервисе не должны затрагивать другие. Рассмотрим, какая из этих архитектур является наиболее подходящей для БПС указанных выше объектов.

1.4. Модели формального представления архитектуры БПС

Архитектура – это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы [13]. Архитектура программной системы определяет ее структуру, точнее – несколько структур, каждая из которых включает в себя элементы и взаимосвязи между ними. Элементы могут быть вычислительными объектами, связанными потоком управления или бизнес-объектами, связанными семантическими ограничениями.

В целом процесс проектирования архитектуры состоит из систематической декомпозиции элементов верхнего уровня на совокупности более мелких элементов. Наилучшим математическим аппаратом для формального представления архитектуры, исходя из этого определения следует считать теорию графов, поскольку ничего лучшего для наглядного представления множества элементов и связей между ними, пожалуй, нет. К этому надо добавить развитый математический аппарат теории графов, позволяющий исследовать, оценивать и модифицировать различные модели архитектуры программных систем, представленные графами.

Отметим положительные моменты многослойной архитектуры БПС. Прежде всего, заметим, что эта архитектура не подразумевает какое-то обязательное количество уровней, их может быть три, четыре, пять и больше. Это хорошо соответствует процессу декомпозиции элементов вышележащего уровня на совокупности нижележащих элементов программы. При реализации элементов каждого слоя может быть занято переменное количество процессорных единиц (что очень важно). Переход от реализации некоторого слоя программных элементов к следующему слою может потребовать увеличения или уменьшения числа процессорных элементов. В этом плане конфигурируемая матрица процессорных элементов наиболее эффективна. Завершение некоторого

программного элемента текущего слоя позволяет использовать освободившийся процессорный элемент для выполнения программного элемента следующего слоя.

Кроме того, к положительным сторонам многослойной архитектуры можно отнести:

более простую реализацию по сравнению с другими архитектурами;

более высокий уровень абстракции благодаря разделению ответственностей между слоями;

изолирование слоев обеспечивает защиту одного слоя от изменений, вносимых в другие слои;

в целом повышается управляемость программного обеспечения за счет слабой связанности.

Концепция уровней абстракции была предложена Э. Дейкстрой в1968 г. [14]. Согласно с ней программа разбивается на иерархические упорядоченные части $L(1), L(2), \dots, L(n)$, называемые слоями, удовлетворяющими определенным проектировочным критериям. Каждый слой является группой тесно связанных модулей. Суть идеи – минимизировать сложность системы за счет такого определения слоев, которое обеспечит высокую степень независимости слоев друг от друга. Это достигается благодаря тому, что свойства определённых объектов такой системы (ресурсы, представление данных и т.п.) упрятываются внутрь каждого слоя, что позволяет рассматривать каждый слой как абстракцию этих объектов.

На рис. 1.1 – 1.3 показаны две возможные структуры таких слоёв. На рис. 1.1 показан такой подход, когда задача рассматривается как создание машины пользователя, начиная с самого низшего уровня аппаратуры (или, возможно, операционной системы). Последовательность слоев, называемых абстрактными машинами, определяется так, что каждая следующая машина строится на основе предыдущих, расширяя их возможности. Каждый уровень может ссылаться только на один, отличный от него самого уровень, а именно тот, который ему предшествует. В структуре, изображенной на рис. 1.2, слои не являются полными абстракциями более низших слоев, каждый из них может ссылаться на любой предшествующий уровень.

Если в варианте по рис. 1.1 каждый слой имеет доступ к элементам только одного слоя, то разработчик должен меть в виду только предыдущий слой. Хотя с точки зрения проектирования этот вариант кажется привлекательным, он может оказаться неэффективным. Например, если некоторое средство, представляемое слоем $L(2)$, потребуется в слое $L(i)$, то каждый из слоев $L(3), L(4), \dots, L(i-1)$ должен обеспечить это средство. Это значит, что запрос данного средства слоем $L(i)$ должен "просачиваться" вниз через слой $L(i-1)$, пока не достигнет слоя $L(2)$, который способен выполнить запрос. Эти трудности, связанные с проблемой эффективности, могут склонить к принятию структуры по рис. 1.2, в которой каждый слой $L(i)$, где $2 < i < n$, может непосредственно обращаться к слою $L(2)$.

Модель многослойной архитектуры произвольной структуры удобно описывать множеством булевых переменных

$$X = \{x_{ij} \in \{0,1\} | \, i = n, n-1, \dots, 2; j = n-1, n-2, \dots, 1; i > j\}, \qquad (1)$$

где $x_{ij} = 1$, если существует связь между слоямиii j, и $x_{ij} = 0$, если такой связи нет. Так как между смежными слоями всегда имеется связь, то

$$(\forall i | i = j+1)(x_{ij} = 1), i = n, n-1, \dots, 2. \qquad (2)$$

Если в многослойной структуре программы, представленной выражением (1), принимают единичные значения только переменные, описываемые условием (2), то эта программа имеет структуру, соответствующую варианту по рис.1. Если справедливо условие

$$(\forall i | i = n, n-1, \dots, 2)(\forall j | (i-j \geq 1))(x_{ij} = 1),$$

то эта программа имеет структуру, соответствующую варианту по рис. 1.2.

Если справедливо условие

$$(\exists i | (i = n, n-1, \dots, 2)) \& (\exists j | (i-j \geq 2))(x_{ij} = 1),$$

то программа имеет структуру по рис.3, соответствующую промежуточному варианту между вариантами структур, представленными на рис. 1.1 и рис. 1.2.

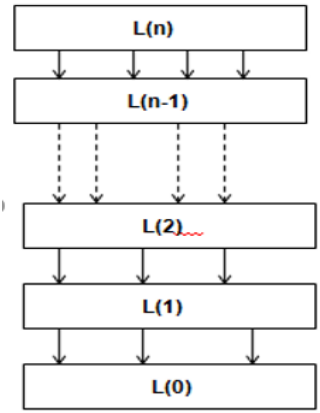


Рис. 1.1. Вариант классической

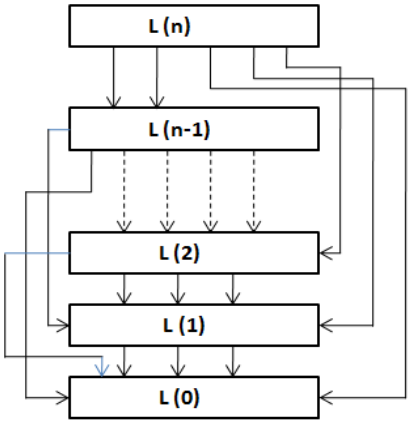


Рис. 1.2. Вариант структуры с полными связями

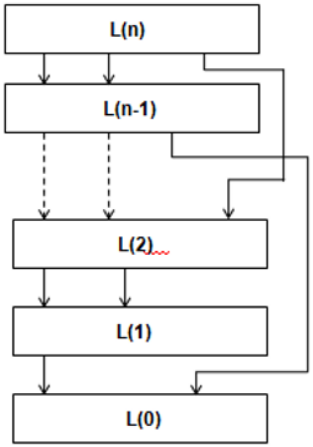


Рис. 1.3. Вариант структуры с произвольными связями

Многоуровневая архитектура предполагает разделение ПС на уровни по принципу взаимодействия “клиент-сервер”. Архитектура может иметь один, два и больше уровней, разделяющих ответственности между компьютером-поставщиком данных и компьютером-потребителем. В отличие от многослойной архитектуры, многоуровневая архитектура предлагает масштабируемость, которая может быть как горизонтальной (масштабирование сети с помощью высокопроизводительных узлов), так и вертикальной (масштабирование каждого узла путем повышения его производительности). Реализация многоуровневой системы, как правило, обходится дороже, но обеспечивает высокую производительность. Этот подход можно сочетать с современной сервис-ориентированной архитектурой, чтобы создавать сложнейшие модели. Поскольку реализация может оказаться дорогостоящей с точки зрения времени и ресурсов, рекомендуется использовать его для сложных ПО, требующих производительности и масштабируемости. Таким образом, наиболее целесообразной архитектурой бортовой программной системы в рамках данной статьи будем считать многослойную архитектуру.

1.5. Принципы расслоения программной системы

Изучение структуры программы, ее проектирование (или рефакторинг) связано с разбиением программы на слои в соответствии с некоторыми принципами расслоения. Наиболее распространенная идея расслоения программной системы по принципу равной доступности модулей слоев была разработана Рабочей группой по методологии программирования Международной федерации по обработке информации (в группу входили Н. Вирт, У. Дал, Э. Дейкстра и др.). Изложение этой идеи было дано в начале 80-х годов прошлого столетия в монографии проф. Варшавского университета В. Турского [15], однако не потеряло актуальности в настоящее время. В соответствии с этой идеей рассматриваются программные системы одного определенного языкового уровня. При этом исключаются вопросы, относящиеся к другим языковым уровням, например, интерпретация элементарных операторов в терминах более примитивных составляющих.

Операторы данного языкового уровня (например, машинные команды) рассматриваются как модули базового (нулевого) слоя, составляющие слой $L(0)$. Это определяется тем, что все элементарные операторы повсеместно (из любого слоя) доступны. Возможные ограничения на использование этих операторов касаются входящих в них данных, т.е.

фактических параметров активации модулей. Заметим, что здесь не учитываются привилегированные операторы машинного языка, которые не могут использоваться в прикладных программных системах.

Модули, построенные из модулей нулевого слоя, могут рассматриваться как модули первого слоя $L(1)$. При этом не требуется, чтобы все модули первого слоя были равнодоступны: например, в программах, написанных на языке, допускающем блочную структуру, некоторые модули первого слоя могут быть локализованы в каком-либо блоке и, следовательно, доступны только внутри этого блока и его подблоков. С другой стороны, при конструировании модулей высших слоев $L(2), \dots, L(n)$в некоторых языках можно использовать модули разных слоев и даже того же самого слоя, что и конструируемый модуль (рекурсия, сопрограммы). Это приводит к отказу от определения слоя как совокупности модулей одного и того же уровня (под уровнем понимается номер слоя). Под уровнем модуля должно пониматься нечто большее, чем наивысший уровень составляющих его модулей. Поэтому уточним понятие равной доступности.

Определение (семантическая спецификация) модуля, отличного от базового, зависит от спецификации других модулей. Будем предполагать, что в каком бы месте программы ни определялся модуль, спецификации всех требуемых модулей в том месте, где дается определение, будут синтаксически доступны (иначе в программе имеется ошибка). Будем писать $n \succ m$, если спецификация модуля m зависит от спецификации модуля n. Предположим теперь, что в данной программе выделены слои $L(0)$, $L(1)$, …, $L(n)$. Реально можно распознать только слой базовых модулей $L(0)$. Процесс, описываемый ниже, введет более высокие уровни и отношение порядка.

Рассмотрим множество модулей M, не входящих в выделенные слои и таких, что для каждого модуля $m \in M$ имеем:

1) если спецификация модуля m зависит от модуля ml, а модуль m1 принадлежит слою $L(i)$ то $0 \le i \le n$;

2) хотя бы один из модулей, используемых в спецификации m, принадлежит слою $L(n)$.

Любое множество M, удовлетворяющее этим двум требованиям, формирует страту над слоем $L(n)$. Максимальной стратой $\mathrm{S}\big(L(n)\big)$ над слоем $L(n)$ будем называть страту, содержащую все модули рассматриваемой программы, удовлетворяющие условиям 1) и 2) для страты.

Рассмотрим теперь множество всех модулей $K\big(L(n)\big)$ таких, что для каждого модуля $k \in K\big(L(n)\big)$ выполняются:

1) хотя бы один модуль, используемый в спецификации k, принадлежит слою $L(n)$;

2) все модули, используемые в спецификации k, не являющиеся членами

$L(0) \cup SHL(0) \cup \ldots \cup L(n-1)SH\big(L(n-1)\big) \cup L(n)$, должны быть членами $K\big(L(n)\big)$;

3) если $m \succ k$ и $m \in K\big(L(n)\big)$, то существует хотя бы один модуль $p \in K\big(L(n)\big)$, такой, что $p \succ m$.

Множество $SH\big(L(n)\big) = K\big(L(n)\big) \cup S\big(L(n)\big)$ называется покровом над слоем $L(n)$.

Таким образом, произвольная (или максимальна) страта над $L(n)$ является некоторым множеством (всех) модулей, специфицируемых с использованием слоев $L(0)$, $L(1)$, …, $L(n)$, которые по определению не могут быть реализованы при отсутствии доступа хотя бы к некоторым модулям слоя $L(n)$. На практике реализация модуля из страты над $L(n)$ потребует, вероятно, также доступа к модулям из других слоев $L(0)$, $L(1)$, …, $L(n-1)$. Взаимозависимые модули исключаются из страты, однако включаются в покровы как члены множеств $K\big(L(n)\big)$. Для некоторых языков программирования понятия покровов совпадают с понятием максимальной страты.

Дальнейший анализ расслоения программы обусловлен специфическим множеством синтаксических правил, налагающих ограничения на правила локализации. Предположим, что имеется общедоступный базовый слой модулей и покров над ними. Если все модули покрова равнодоступны во всех частях программы, исключая базовые модули и модули покрова, то весь покров можно рассматривать как первый слой модулей. Тем не менее, в большинстве случаев некоторые части покрова, т.е. некоторые модули, принадлежащие покрову, будут резервироваться для собственных нужд компонентов более высокого уровня. Следовательно, при анализе программы может возникнуть тенденция рассматривать в качестве первого слоя модулей только ту часть покрова, которая равнодоступна всем компонентам программы, не относящимся ни к базовому слою, ни к модулям самого покрова.

В хорошо структурированной программе спецификация программы как модуля будет семантически зависеть только от спецификаций нескольких модулей предыдущего слоя, и для связок будет использовать некоторые модули базового слоя. Присутствие базовых модулей

во всех покровах и слоях необходимо и связано с использованием в качестве базового слоя базовых операторов определенного языкового уровня.

Разбиение программы на слои показано на примере по рис. 1.4 (из монографии В. Турского), где отношение $n \succ m$ обозначается стрелкой, проведенной из n в m. Базовый слой обозначен горизонтальной линией в нижней части рисунка. Модули A, B, F, G и H принадлежат максимальной страте над базовым слоем; модули C, D, E составляет множество $K\big(L(0n)\big)$. Следовательно, все модули самого нижнего ряда принадлежат покрову над базовым слоем. Модули A, B, C, D, E и H равнодоступны "сверху" и, таким образом, принадлежат первому слою. Модули F и G используются только в модуле M, что дает основание рассматривать эту группу модулей как единый модуль M, а модули F и G как модуляризованные компоненты модуля M.

Модули I, J и L удовлетворяют условиям множества $K\big(L(1)\big)$ и, следовательно, принадлежат покрову над первым слоем. Поскольку они равнодоступны, они являются членами второго слоя.

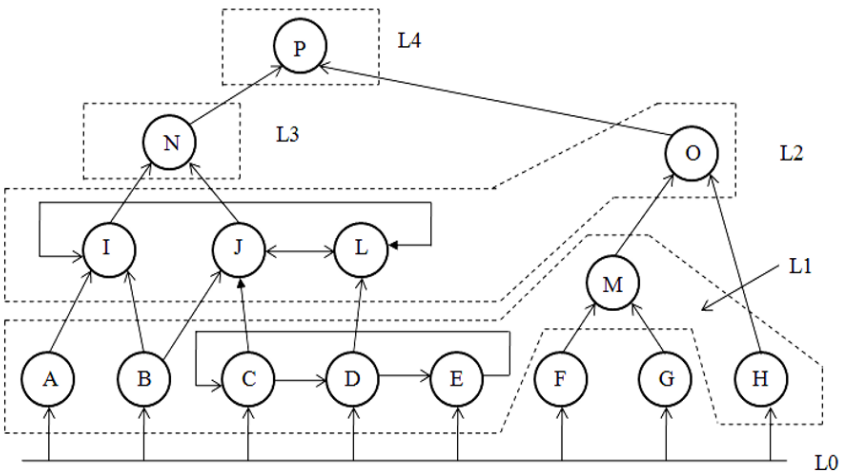


Рис. 1.4. Разбиение программы на слои

Куда следует отнести модуль M? Он зависит от базовых модулей и модулей F и G, которые являются членами страты над базовым слоем и не принадлежат никакому слою. Следовательно, модуль M принадлежит максимальной страте над нулевым слоем. Он также доступен, как и

модули A, B, C, D, E и H, поэтому модуль M является модулем первого слоя. Рассуждая таким же образом, считаем, что модули I, J, L и O относятся ко второму слою. Третий и четвертый слои содержат по одному модулю (N и P). Наконец, модуль P представляет программу в целом.

Таким образом, данный пример показывает, что чисто механические понятия, конструируемые большей частью методом “снизу-вверх”, можно использовать для модульного расслоения программы, отражающего взгляд “сверху-вниз” на структуру программы. Отвлекаясь от способа получения начальной архитектуры программной системы, рассмотрим вопросы коррекции и оптимизации архитектуры. Во второй части этой статьи предлагаются модели и методы совершенствования архитектуры программной системы и даются примеры их использования.

Глава 2. МЕТОДЫ КОРРЕКЦИИ АРХИТЕКТУРЫ ПРОГРАММНОЙ СИСТЕМЫ

2.1. Архитектурные преобразования программной системы

Отвлекаясь от метода получения начальной архитектуры программной системы, сосредоточимся на коррекции и оптимизации этой архитектуры. Вопросы оптимизации архитектуры остаются актуальными на всех этапах разработки программной системы от разработки технического задания до рефакторинга системы. На всех этих этанах процесс изменения внутренней структуры программы не должен затрагивать её внешнее поведение и имеет целью облегчить понимание её работы. В основе оптимизации лежит последовательность эквивалентных (то есть сохраняющих поведение) преобразований, сохраняющих функциональную семантику базового кода.

По ходу трансформаций часто встает задача выявления смысловой нагрузки модулей. Для решения подобных задач зачастую приходится исследовать реальный программный код, анализировать сигнатуры функций и комментарии, а при отсутствии последних и сам код функций. Задача специалиста, вовлеченного в процесс архитектурных преобразований, – по возможности минимизировать объем семантического анализа (например, путем удаления вспомогательных блоков) и сделать его последовательным и направленным.

Необходимость в архитектурных преобразованиях системы может быть связана со следующими причинами:

Развитие программы по причине изменений, обусловленных текущей необходимостью. Часто изменения вносят программисты, которые не до конца понимают архитектуру ПС в целом, и постепенно код становится менее структурированным, а разбираться в нем все труднее. Архитектурные преобразования улучшает композицию ПС.

Повышение производительности ПС. Изменения, уточнения и усложнения программного модуля, несомненно, заставляет программу выполняться медленнее, хотя и при этом делает ее более понятной и податливой для настройки производительности.

Потребность в функциональных изменениях ПС. Внедрение новой функциональности не должно затронуть логику системы. Изменение существующей архитектуры может быть хорошим шагом на пути

внедрения новой функциональности, облегчающим дальнейшую эволюцию системы.

Смена аппаратной платформы ПС (например, переход на программируемые логические матрицы). Смена платформы ПС должна минимально затрагивать существующий код. Желательно ограничиться изменениями только в узкой платформенно-зависимой прослойке системы. Выделение такой прослойки всегда сопряжено с необходимостью изменения архитектуры.

Возможность организации распараллеленного (многопоточного) вычислительного процесса при наличии в вычислителе нескольких ядер.

Обновление технологии разработки программного продукта, связанное, например, с переходом на более совершенную технологию программирования.

Прежде чем говорить о коррекции архитектуры, следует задаться вопросом, как оценить качество структуры ПС? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева. Степень отличия реальной проектной структуры от дерева характеризуется невязкой структуры. Известно, что полный граф с n вершинами имеет количество ребер равное $e_c = n \times (n-1)/2$, а дерево с таким же количеством вершин – существенно меньшее $e_t = n - 1$.

Формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева. Для проектной структуры с n вершинами и e ребрами невязка определяется по выражению:

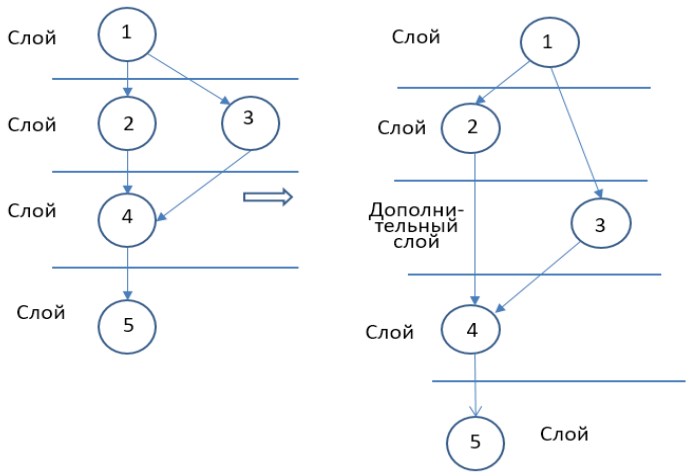
$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \\ = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}. \qquad (2.1)$$

Значение невязки лежит в диапазоне от 0 до 1. Если $Nev = 0$, то проектная структура является деревом, если Nev = 1, то проектная структура – полный граф. Ясно, что невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления. Использование таких характеристик для оценки качества программы показано в монографии [16].

Вообще заметим, что формализовать процесс коррекции архитектуры ПС или тем более построить алгоритм коррекции довольно затруднительно. Однако в ряде случаев, выделив отдельные фрагменты структуры ПС, можно их преобразовать, стремясь к получению наилучшей

структуры, например, к дереву. Чаще всего это удается сделать путем изменения числа слоев, перемещения модулей по слоям, объединения (поглощения) или разделения модулей. Некоторые примеры такой коррекции даны в статье [17]. Остановимся на основных приемах коррекции архитектуры, которые можно использовать для преобразования архитектуры ПС с целью наилучшего использования вычислительных возможностей бортовой вычислительной системы, обратив внимание на возможный эффект такого преобразования.

На рис. 5 показан вариант коррекции архитектуры ПС путем введения в систему дополнительного слоя. На рис 2.1 а) показана возможность параллельного выполнения модулей 2 и 3. Если в БВС имеется только один вычислитель, такая возможность не реализуется. В этом случае возможно только последовательное выполнение этих модулей. Коррекция архитектуры системы в этом случае – введение дополнительного слоя, как показано на рис. 2.1 б). Это, конечно приведет к увеличению времени выполнения ПС, но для однопроцессорной (одноядерной) БВС иного решения в данном случае нет.

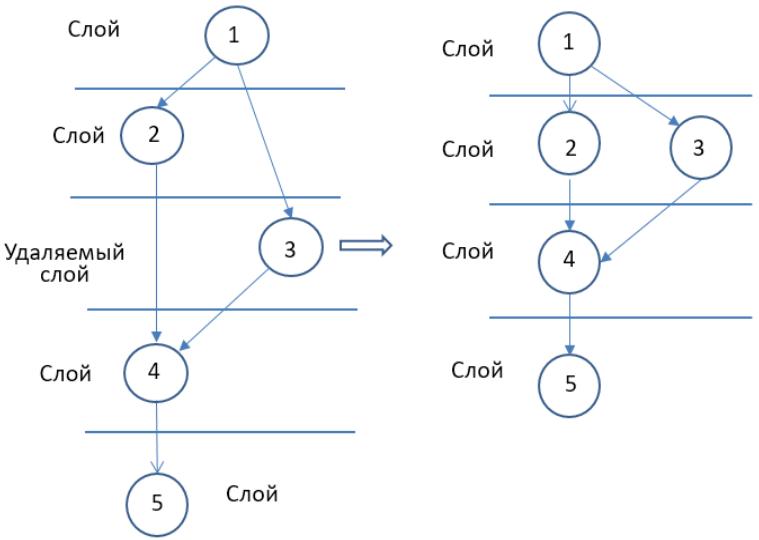


а) б)

Рис. 2.1. Коррекция перемещением за счёт введения дополнительного слоя

Вариант увеличения скорости вычислительного процесса показан на рис. 2.2. В данном случае коррекция идет по двум направлениям – уменьшается количество слоев ПС и увеличивается параллельность вычислительного процесса. Из рис. 2.2 а) ясно, что можно удалить слой, где размещается модуль 3, и передвинуть его в вышележащий слой. При

этом можно выполнять параллельно модули 2 и 3 (если в БВС есть свободный вычислитель), как это показано на рис. 2.2 б).



а) б)

Рис. 2.2. Коррекция перемещением счёт удаления слоя

На рис. 2.3 а) показана последовательная цепочка модулей 2 и 3, которые используются только модулем 1. Может быть это стало следствием желания распараллелить работу по программированию этих модулей. Возможный вариант улучшения структуры ПС путем объединения модулей 2 и 3 показан на рис. 2.3 б). Заметим, что если в этом случае модуль 1 обращается только к модулям 2-3 и 4, то сокращается число слоев ПС. Полезный эффект такого преобразования – сокращение числа операций обмена.

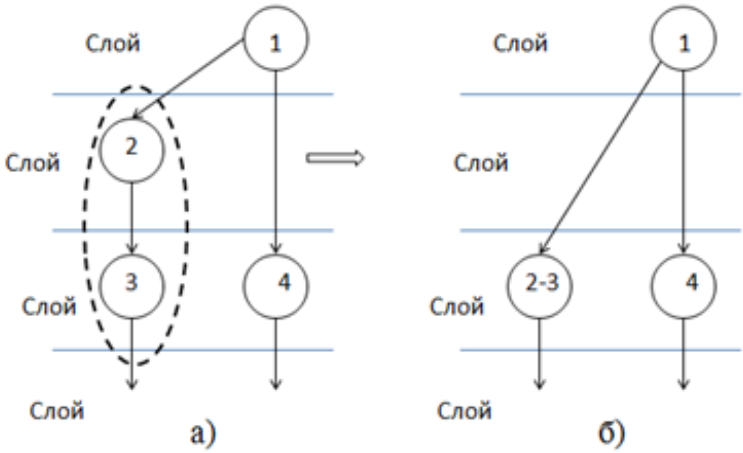


Рис. 2.3. Коррекция поглощением нижележащим слоем

На рис. 2.4 а) показан случай, когда результаты работы модулей 1 и 2 используются только модулем 3. Улучшение структуры ПС можно получить объединением модулей 1 и 2, как показано на рис. 2.4 б). Это может привести к сокращению числа вычислителей. Такая коррекция объединением модулей, принадлежащих одному слою, полезна в случае недостаточного количества вычислителей БВС.

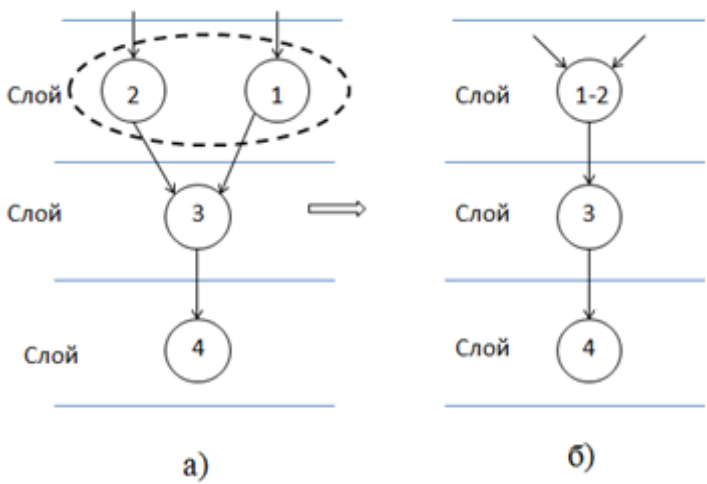


Рис. 2.4. Коррекция объединением (вариант 1)

Другой случай объединения модулей 2 и 3 приведен на рис. 2.5 а). Он возможен в том случае, если к модулям 2 и 3 обращается только модуль 1, а сами модули 2 и 3 обращаются только к модулю 4. В результате объединения модулей 2 и 3, как показано на рис. 2.5 б) упрощается структура ПС. Этот подход может быть использован, например, для сокращения числа вычислителей и сокращения операций обмена результатами текущих вычислений.

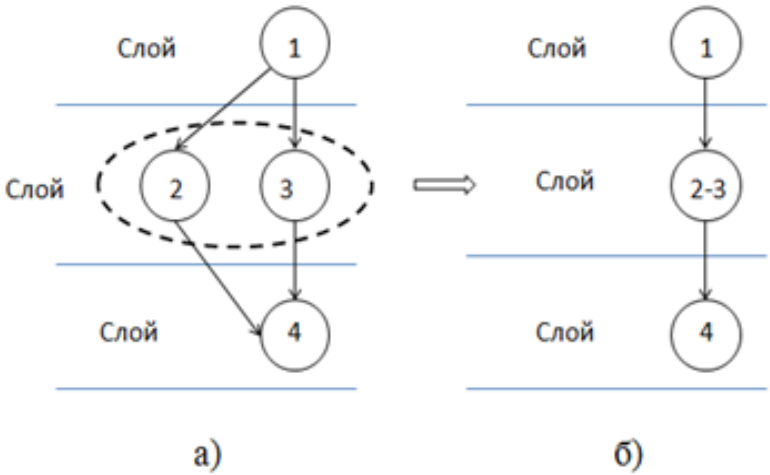


Рис. 2.5. Коррекция объединением (вариант 2)

При коррекции архитектуры ПС, кроме объединения модулей и перемещения по слоям, возможны ситуации разделения модулей, как показано на рис. 2.6. Это может быть полезно, например, для организации параллельного вычислительного процесса и снижения сложности отдельных модулей. В данном случае модуль 4 разделен функционально, поскольку результат его работы необходим различным модулям нижележащего слоя. Заметим, что такое разделение не всегда возможно. Например, если результат модуля 2 нужен для модуля 4-2, появится дополнительная межмодульная связь: модуль 2 – модуль 4-2. Также, если результат модуля 3 нужен для модуля 5, появится дополнительная связь: модуль 3 – модуль 4-1. Таким образом в этом случае вместо 6 связей будет 7, что ставит под сомнение такое функциональное разделение.

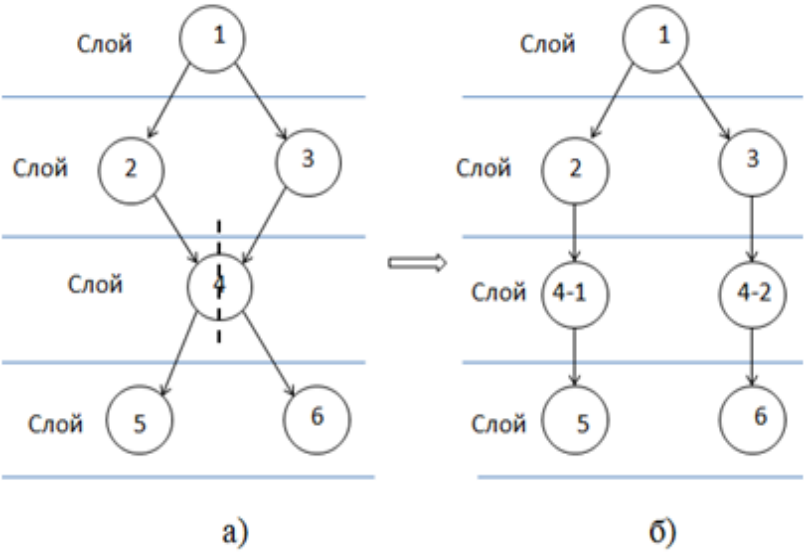
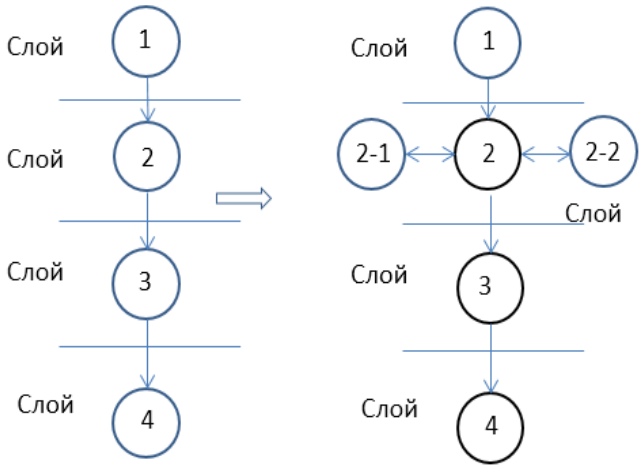


Рис. 2.6. Коррекция функциональным разделением

В других случаях разделение может понадобиться для увеличения скорости выполнения модуля. В этом случае необходимо переработать алгоритм модуля – найти возможность его распараллеливания для реализации на двух и более вычислителях (рис. 2.7). На рис. 2.7 а) показан модуль 2, время реализации которого существенно замедляет вычислительный процесс. На рис. 2.7 б) модуль перепрограммирован и содержит два параллельно выполняемых модуля 2-1 и 2-2, которые могут выполняться на отдельных вычислителях. Однако в каждом конкретном случае решение о той или иной коррекции структуры ПС должно приниматься после детального его анализа и оценки целесообразности.



а) б)

Рис. 2.7. Коррекция разделения распараллеливанием

Результат коррекции архитектуры в каждом из рассмотренных случаев должен быть спроецирован на реальный программный код системы. При проецировании удаления модулей из модели необходимо определить множество строк и файлов, которое соответствует удаленному блоку в программном коде. После этого необходимо удалить из программного проекта выявленные строки и файлы. При проецировании на код переноса модуля в модели переносятся соответствующие строки и файлы в исходном коде программной системы и т.д. Производимые таким образом трансформации можно рассматривать как архитектурно-управляемый рефакторинг программного кода.

2.2. Пример оптимизация архитектуры для повышения производительности ПС

В монографии [18] автор дал пример использования изложенных выше методов коррекции архитектуры ПС, имеющий целью повышения производительности БВС. Для простоты изложения структура многослойной ПС дана в обобщенном виде (рис. 2.8), каждый слой показан в виде одного модуля с возможностью организации связей с любым произвольным слоем системы. Такая модель позволяет рассмотреть любой вариант многослойной программной системы, лежащий в диапазоне структур, приведенных на рис. 1.1 – 1.3.

Учитывая, что число слоев в большинстве ПС, как правило, не превышает трех-пяти, рассмотрим пятислойную программу, по рис. 2.9. Пунктирными линиями показаны возможные дополнительные связи между слоями. Каждой линии поставлена в соответствие булева переменная, единичное значение которой означает наличие межслойной связи, нулевое - отсутствие такой связи.

Будем считать, что ПС прошла полный этап тестирования и в процессе отладки определены временные характеристики модулей и частоты обращения модулей любого слоя к модулям нижележащих слоев. Предположим, что передача (трансляция) запроса через слой i дополнительно загружает этот слой на некоторый промежуток времени t_i. Например, если $x_5 = 0$, (т.е. отсутствует связь между слоями 5 и 3), то модуль m4 дополнительно работает в течение промежутка времени t_5. Если x_5 = 1, то дополнительное время модулю m4 не потребуется. Однако в этом случае необходимо в программу добавить межмодульный интерфейс для взаимодействия модулей m5 и m3 , который несколько увеличит время работы модулей m5 и m3 (на величину t_5') и увеличит программу на некоторую величину e5. Аналогичные рассуждения справедливы и для других переменных, показанных на рис. 2.8.

Как правило, дополнительные связи между слоями программы сокращают время выполнения ее функций, но увеличивают размер программы. Необходимо также учесть тот факт, что дополнительно создаваемые связи между слоями могут работать с различной нагрузкой. Так, например, если создается связь, обозначаемая переменной x_5, то модуль m4 освобождается от трансляции только тех запросов, которые модуль m5 адресует модулю m3. Поэтому целесообразной каждой переменной x_i поставить в соответствие определенную интенсивность взаимодействия некоторой пары модулей λ_i. В этих условиях задача архитектурной оптимизации ПС сводится к определению такой структуры, которая обеспечивает наилучшую производительность программы при заданных ограничениях на размер дополнительных межмодульных интерфейсов.

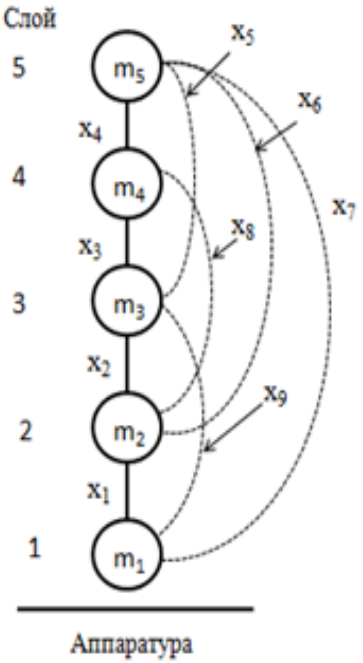


Рис. 2.8. Структура многослойной ПС в обобщенном виде

В нашем случае структура многослойной программной системы может быть представлена вектором $X = \{x_i | \, i = 5, 6, \ldots, 9\}$ (заметим, что всегда $x_1 = x_2 = x_3 = x_4 = 1$, так как эти переменные определяют связи между смежными слоями). Поэтому требуется найти такое значение X_{opt}, при котором обеспечивается максимальный выигрыш во времени работы БПС

$$MaxT = \sum_{i=5}^{i=9} \lambda_i \times (t_i - t_i') \times x_i \qquad (2.4)$$

при выполнении ограничения на допустимое увеличение программы $E_д$ за счет дополнительных межмодульных интерфейсов

$$\sum_{i=5}^{i=9} e_i \times x_i \; \leq E_д. \qquad (2.5)$$

Учитывая двоичный характер переменных, следует добавить ограничение

$$\forall\, i(x_i \in \{0,\ 1\}). \qquad (2.6)$$

Сформулированная задача (2.4) – (2.6) относится к классу задач о загрузке рюкзака. Малая размерность (в нашем примере) задачи позволяет ее легко решить полным перебором переменных, представляющих допустимые решения в условиях принятых ограничений. В реальных программных системах, содержащих по нескольку модулей в каждом слое, размерность задачи может существенно вырасти и потребуется применить боле сложные алгоритмы решения. Следует отметить, что в связи с неточностью получения исходных данных в рассматриваемой задаче, методы получения оптимального решения не имеют смысла. Поэтому можно ограничиться приближенными, быстро работающими алгоритмами. Результат решения должен быть спроецирован на реальный программный код системы.

Заметим, что кроме решения задачи, представляет интерес анализ возможных вариантов структур с целью получения ответа вида “что будет, если?”. Другими словами, хорошо было бы иметь модель, которая помогла бы детально исследовать возможные варианты архитектур ПС. Такую модель легко построить, используя электронные таблицы, например, Excel современной редакции.

2.3. Пример коррекции архитектуры ПС для заданного количества вычислителей

Представляет интерес использовать предложенные выше методы коррекции архитектуры ПС для оптимизации ее структуры в случае организации вычислительного процесса при заданном количестве вычислителей. В качестве ограничивающего фактора в этом случае выступает количество вычислительных элементов БВС, которое может быть использовано для выполнения ЗНЗ и возможности операционной системы БВС по адаптивному перераспределению вычислителей по модулям ПС. Для дальнейшего рассмотрения использования рассмотренных выше возможных архитектурных преобразований (коррекций) рассмотрим пример архитектуры программной системы (отвлекаясь от того, как она получена), приведенный на рис. 2.9. В этом примере и далее будем использовать модульное расслоение программы, отражающее взгляд “сверху-вниз” на структуру программы. При этом подразумевается наличие базового слоя модулей ниже слоя 0. Пусть требуется провести коррекцию исходной архитектуры для случая двухпроцессорной БВС.

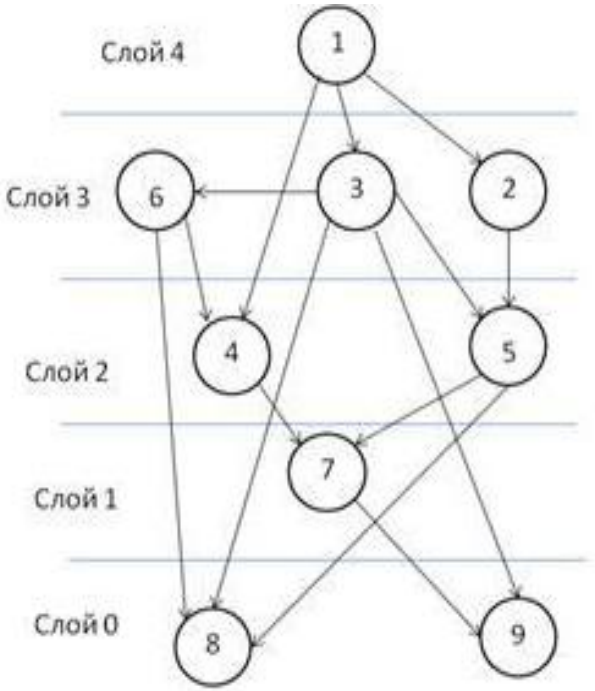


Рис. 2.9. Граф ПС

Анализируя предложенный граф архитектуры, следует отметить, что он не отвечает каноническим правилам многослойной структуры. В частности, модуль 6 не отвечает этим требованиям. Известно, что выделение слоев – хорошая основа для улучшения системы. Найти строгие слои в произвольной программной системе достаточно трудно, поскольку, как уже отмечалось, они могут содержать горизонтальные связи и сильносвязанные компоненты. Поэтому целесообразно расширить понятие слоя, позволив включать в произвольные слои сильносвязанные компоненты. Эти компоненты при таком подходе можно рассматривать, как атомарные модули. Заметим, что не всегда сильносвязанные компоненты на структурных диаграммах свидетельствуют о плохой архитектуре системы. Возможным дефектом архитектуры с поглощающими слоями может стать эффект "пропавшего слоя" – дефектная связь приводит к появлению модулей, которые по смыслу должны находиться на разных слоях.

Вернемся к структуре, приведенной на рис. 2.9. Оценим качество исходной архитектуры по характеристике, заданной формулой (2.1).

$$Nev = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)} = \frac{2 \times (14 - 9 + 1)}{(9-1) \times (9-2)} = 0{,}214.$$

Вторая важная характеристика структуры ПС – средняя ширина слоя. Это среднее количество модулей в одном слое программы (обозначим это значение через A). Средняя ширина слоя определяется по выражению

$$A = \sum_{i=1}^{k} w_i / k, \qquad (2.7)$$

где w_i– ширина слоя i (количество модулей в слое), k–число слоев. Прямоугольность графа ПС можно оценить средним отклонением ширины слоя от значения A. Обозначим это отклонение величиной D.

$$D = \sum_{i=1}^{k} |w_i - A| / k. \qquad (2.8)$$

Очевидно, что чем ближе значение Dк нулю, тем лучше соответствие архитектуры ПС бортовой вычислительной системе. Для исходной архитектуры ПС

$$A_{\text{и}} = \frac{(1 + 3 + 2 + 1 + 2)}{5} = 1{,}8.$$

$$D_{\text{и}} = \frac{|1 - 1{,}8| + |3 - 1{,}8| + |2 - 1{,}8| + |1 - 1{,}8| + |2 - 1{,}8|}{5} = 0{,}64.$$

Данные характеристики показывает приспособленность структуры ПС к реализации определенным количеством вычислителей (в нашем случае 2 вычислителя). Начинаем коррекцию исходной архитектуры. Ясно, что модуль 6 не может находиться с модулем 3 в одном слое. Так как модуль 3 для выполнения своих функций обращается к модулю 6, то последний должен быть перемещен в нижележащий слой. Возможный вариант новой структуры показан на рис. 2.10. Заметим, что в данном случае модуль 2 должен быть перемещен в нижележащий слой 3 или необходимо переместить модуль 5 в слой 3. Следует обратить внимание на увеличение количества слоев ПС после выполненной коррекции. Этот важный факт может привести к увеличению времени работы ПС. Отметим также, что после такой коррекции структуры не изменилась сложность ПС, определяемая по значению *Nev*, поскольку число вершин и ребер осталось прежним, т.е. $Nev_1 = Nev_{\text{и}}$.

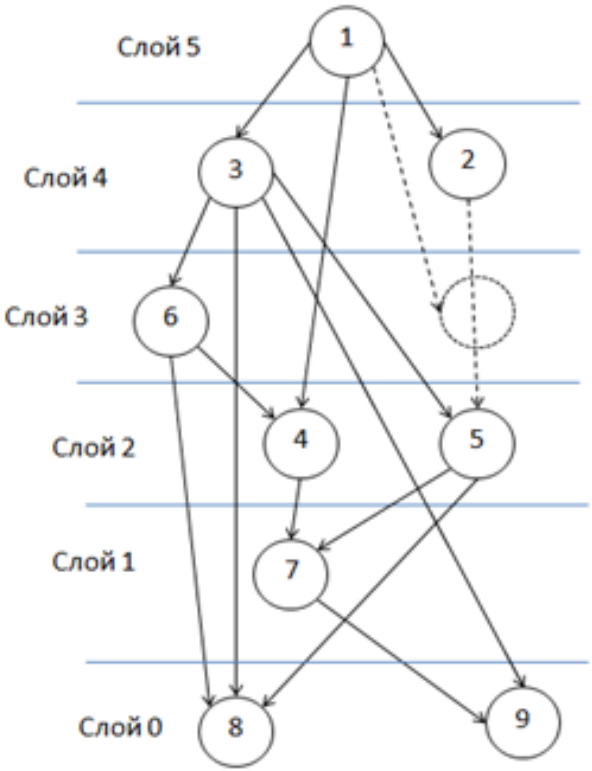


Рис. 2.10. 1-й вариант коррекции архитектуры ПС

Определим для первого варианта коррекции ПС значения показателей по выражениям (2.7) и (2.8):

$$A_1 = \frac{(1+2+1+2+1+2)}{6} = 1{,}5.$$

$$D_1 = \frac{|1-1{,}5| + |2-1{,}5| + |1-1{,}5| + |2-1{,}5| + |1-1{,}5| + |2-1{,}5|}{6} = 0{,}5.$$

Второй вариант коррекции исходной архитектуры ПС отличается от первого варианта тем, что перемещается модуль 5 на вышележащий слой. Легко убедиться, что в этом случае значения показателей Nev_2, A_2 и D_2 равны предыдущим значениям Nev_1, A_1 и D_1.

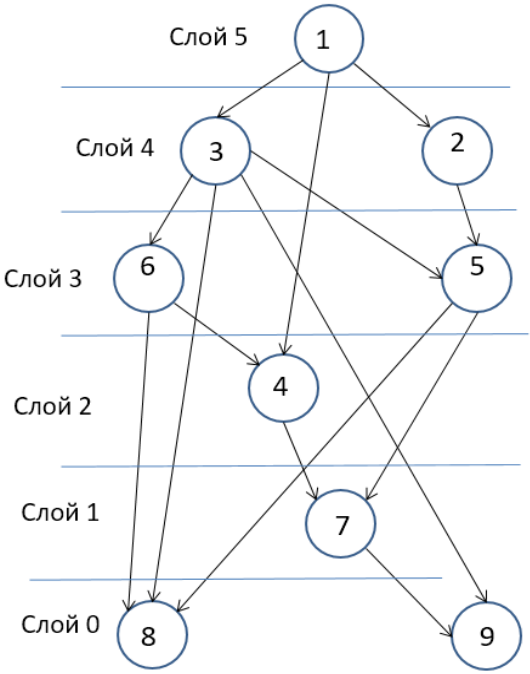


Рис. 2.11. 2-й вариант коррекции архитектуры ПС

Возможен третий вариант коррекции архитектуры, связанный с объединением модулей 3 и 6 (на рис. 2.12 это модуль 3-6). При этом меняется количество слоев ПС (их становится 5, что лучше), и изменяется число вершин и ребер (8 вершин и 12 ребер). Это несколько снижает сложность ПС по значению по значению невязки:

$$Nev_3 = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)} = \frac{2 \times (12 - 8 + 1)}{(8-1) \times (8-2)} = 0{,}24.$$

Определим также как изменились показатели прямоугольности графа ПС:

$$A_3 = \frac{(1 + 2 + 2 + 1 + 2)}{5} = 1{,}6.$$

$$D_3 = \frac{|1 - 1{,}6| + |2 - 1{,}6| + |2 - 1{,}6| + |1 - 1{,}6| + |2 - 1{,}6|}{5} = 0{,}48.$$

Однако объединенный модуль 3-6 возрастает по объему и усложняется его программирование. Дальнейшая коррекция ПС возможна перемещением модуля 8 на соседний вышележащий слой, этим улучшается прямоугольность графа структуры ПС. Если возможно разделение модуля 9 распараллеливанием его алгоритма, то получим окончательную (четвертую) структуру ПС, представленную на рис. 2.12. Определим основные характеристики этого варианта структуры ПС (9 вершин и 14 ребер):

$$Nev_4 = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)} = \frac{2 \times (14 - 9 + 1)}{(9-1) \times (9-2)} = 0{,}214.$$

$$A_4 = \frac{(1 + 2 + 2 + 2 + 2)}{5} = 1{,}8.$$

$$D_4 = \frac{|1 - 1{,}8| + |2 - 1{,}8| + |2 - 1{,}8| + |2 - 1{,}8| + |2 - 1{,}8|}{5} = 0{,}32.$$

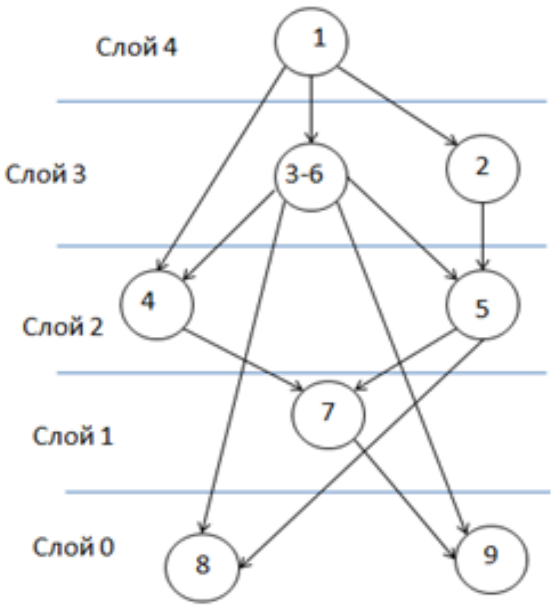


Рис. 2.12. 3-й вариант коррекции архитектуры ПС

Для удобства сравнения полученных структур ПС результаты вычисления их характеристик представлены в табл. 1.

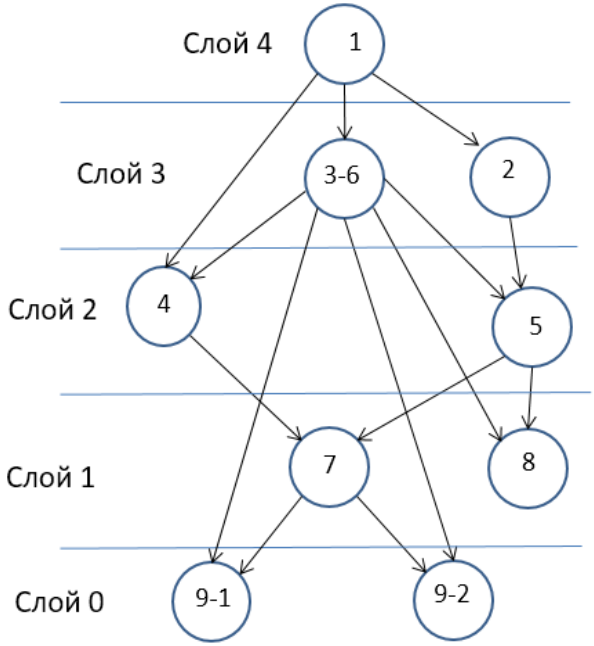


Рис. 2.13. 4-й вариант коррекции архитектуры ПС

Таблица 1

Характеристики вариантов структур

	Характеристики			
Струк-тура	Невязка Nev	Средняя ширина слоя A	Отклонение от средней ширины D	Число слоев
Исходная	0,214	1,8	0,64	5
Вариант 1	0,214	1,5	0,5	6
Вариант 2	0,214	1,5	0,5	6
Вариант 3	0,24	1,6	0,48	5
Вариант 4	0,214	1,8	0,32	5

Анализируя характеристики полученного семейства архитектур программной системы, можно сделать вывод о наилучшем приспособлении варианта 4 для реализации ПС на двухпроцессорном вычислителе. Однако вопрос о действительной оптимальности выбранной архитектуры остается на данном этапе открытым. Это связано с асинхронным выполнением модулей в ходе вычислительного процесса. Потому вполне возможны простои, и даже низкие загрузки вычислителей, что может поставить под сомнение целесообразность использования двухпроцессорной БВС. Принять окончательное решение можно только, построив диаграммы загрузки процессоров, используя временные характеристики модулей. Пример такого решения дан в статьях [19, 20].

Бытует высказывание: “архитектура – это то, за что увольняют системного архитектора и руководителя проекта”. Именно архитектор занимается проектированием архитектуры программной системы и разработкой архитектурного описания этой системы. Важнейшая обязанность архитектора заключается в разработке ключевых проектных решений относительно внутреннего устройства программной системы. Это начальный этап создания ПС и, как известно, ошибки, допущенные на этом этапе, исправляются многократно сложнее и дороже, чем позже они выявляются. В тоже время не существует хорошо разработанных формальных методов проектирования архитектуры программных систем. И если само программирование до сих пор во многом считается искусством, то в значительной большей мере это можно сказать относительно проектирования архитектуры программных систем.

Неоценима важность разработки архитектуры бортовой программной системы для класса летательных аппаратов, в которых ресурсы вычислительной системы существенно ограничены, особенно в части памяти. Иногда бывает так, что программная система разрабатывается без предварительного архитектурного описания. Но как бы она не разрабатывалась, в ней уже заложена определенная архитектура. Далее начинается понимание важности и необходимости иметь ее архитектурное описание и начинается процесс “раскопки архитектуры” – представление созданной архитектуры, после чего начинается ее коррекция (рефакторинг). По мнению автора, данная монография может оказать определенную помощь в процессах коррекции архитектуры создаваемой программной системы.

Литература к г. 1 и 2

1. БассЛ., КлементсП., КацманР. [BassL., ClementsP., KazmanR.] Архитектура программного обеспечения на практике. 2-е изд: пер. с англ. СПб.: Питер. 2006.

2. Systems and software engineering – Recommended practice for architectural description of software-intensive systems. [Электронныйресурс] URL: https://www.iso.org/standard/45991.html (дата обращения 10.08.2021).

3. Software Engineering Institute. [Электронныйресурс] URL: https://www.sei.cmu.edu/ (дата обращения 10.08.2021).

4. The Concise Definition of The Zachman Framework by: John A. Zachman. [Электронный ресурс] URL: https://www.zachman.com/about-the-zachman-framework (дата обращения 12.08.2021).

5. Назаров С.В., Барсуков А.Г. Оптимизация параллельных вычислений бортовых систем реального времени (часть 1) // ЭЛЕКТРОНИКА: Наука, Технология, Бизнес. 2020. № 10.С. 110 - 116.

6. Хитт Д. Стратегия Xilinx – быстрее двигаться к адаптируемому, интеллектуальному миру // ЭЛЕКТРОНИКА: Наука, Технология, Бизнес. 2018. № 4. С. 84–87.

7. XilinxUltrascale + Обзор. [Электронный ресурс]. URL:https://developer.ridge-run.com/wiki/index.php?title=Xilinx_Ultrascale%2B_Overview (дата обращения 12.08.2021).

8. The Reference Model of Open Distributed Processing (RM-ODP). [Электронный ресурс]. URL: http://www.rm-odp.net/(дата обращения 12.08.2021).

9. R. van Ommering et al., The Koala Component Model for Consumer Electronics. IEEE Trans. Computers. 2000. vol. 33. no. 3.

10. БучГ., РамбоД., ЯкобсонИ. [BoochG., RumbaughJ., Jacobson I.]Язык UML. Руководство пользователя. 2-е изд.:пер. с англ. Мухин Н. М.: ДМК Пресс. 2007.

11. Гурьянов В. И. Имитационное моделирование на UML SP: монография / В.И. Гурьянов. – Чебоксары: Филиал СПбГЭУ в г. Чебоксары. 2014. – 135 с.

12. Крачтен Ф., Оббинк Х., Стаффорд Д. Ретроспектива программных архитектур. Открытые системы. . СУБД. 2006. № 03. [Электронный ресурс]. URL:https://www.osp.ru/os/2006/03/1156577(дата обращения 12.08.2021).

13. IEEE 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems.[Электронный ресурс].

URL:https://standards.ieee.org/standard/1471-2000.html(дата обращения 12.08.2021).

14. Дейкстра Э. Дисциплина программирования. [Электронный ресурс]. URL:http://khpi-iip.mipk.kharkiv.edu/library/extent/dijkstra/pp/ewdx01.pdf(дата обращения 12.08.2021).

15. Турский В. Методология программирования: пер. с англ. – М.: Мир, 1981. – 264 с

16. Назаров С.В. Архитектура и проектирование программных систем. Монография. М.: Инфра-М. 2020.

17. Назаров С.В., Вилкова Н.Н. Структурный рефакторинг многослойных программных систем // Информационные технологии и вычислительные системы. 2014. № 4. – С. 13 – 23.

18. Назаров С.В. Эффективность и оптимизация компьютерных систем: монография /С.В. Назаров. – 2-е изд. – М.: РУСАЙНС, 2020.

19. Назаров С.В., Барсуков А.Г. Оптимизация параллельных вычислений бортовых систем реального времени (часть 2). ЭЛЕКТРОНИКА: Наука, Технология, Бизнес.2021. № 1. С. 130 – 136.

20. Назаров С.В. Оптимизация параллельных вычислений в системах реального времени и пакетной обработки//Norwegian Journal of the International Science. 2020. №39. P. 42 – 55.

Глава 3. ОПТИМИЗАЦИЯ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА БОРТОВЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

3.1. Особенности современных бортовых систем

Актуальность тематики параллельных вычислений была осознана достаточно давно при решении сложных научно-технических задач, как в связи с низкой надежностью и производительностью компьютеров, так и в связи с появлением многопроцессорных систем и многоядерных процессоров. Технология обеспечения надежности и высокой производительности на основе параллельных вычислений естественным образом стала преобладающей в бортовых вычислительных системах (БВС). В настоящее время такие системы находят широкое применение в авиационной и космической технике, а также в наземных и водных подвижных объектах. Эффективность выполнения поставленных задач, безопасность, эксплуатационная пригодность и ряд других важных качеств подвижных объектов в значительной мере определяются способностью бортовой вычислительной системы выполнять свои функции.

Развитие бортового оборудования характеризуется постоянным увеличением числа решаемых задач и повышением их сложности, расширением интеллектуальных и адаптивных возможностей. Это неизбежно приводит к усложнению БВС, ее операционной системы и специального программного обеспечения. На время решения большинства задач, возлагаемых на БВС, накладываются жесткие временные ограничения. Выполнение этих требований приводит к необходимости организации параллельных вычислительных процессов.

В данной главе представлена совокупность математических моделей, формулировок задач и подходов к их решению, позволяющих построить расписание параллельного вычислительного процесса для реализации информационно-связанных задач на многопроцессорных бортовых вычислительных системах. Даны модели наборов решаемых задач в форме нагруженного графа и в ярусно-параллельной форме, решение задач о назначениях решаемых задач на процессоры и алгоритм составления расписания параллельного вычислительного процесса.

Большая часть алгоритмов (программ), используемых в современных бортовых вычислительных системах (БВС), обладает достаточным

внутренним параллелизмом, который может быть реализован многопроцессорными (многоядерными) бортовыми системами. Сложность этого подхода связана с тем, что архитектура аппаратных средств многопроцессорной системы зачастую фиксируется на этапе проектирования и остается неизменной на время выполнения программ. Это означает, что разработчик должен выбирать соответствующую систему для предполагаемых приложений. С этой целью разработчик должен разбить приложение на части, рассмотреть возможности параллелизма при выполнении приложения и выбрать соответствующую систему для своего приложения. Ограничением такого подхода является недостаток существующих многопроцессорных систем, заключающийся в отсутствии адаптивности на стадии разработки и во время выполнения программы [1–7].

Программируемая разработчиком логическая матрица предлагает более гибкое решение, потому что с ее помощью аппаратные средства можно переконфигурировать с новыми функциями и использовать повторно с различными приложениями. Некоторые поставщики программируемых пользователем логических матриц (компания Xilinx) предлагают специальную функцию, называемую динамическим и частичным переконфигурированием [8]. Это означает, что часть аппаратных средств системы может быть изменена во время выполнения программы, а оставшаяся часть остается действующей и неизменной. В данной статье рассмотрим возможное решение в рамках первого подхода, позволяющее на основе анализа приложения определить целесообразную структуру многопроцессорной системы с выполнением требований на время выполнения приложений.

3.2. Постановка задачи

Пусть задана структура приложения, состоящего из некоторого множества информационно-связанных частей (задач) с известным (ожидаемым) временем выполнения каждой задачи. Предполагается, что это время определяется элементарным вычислителем (процессором или ядром) многопроцессорной бортовой вычислительной системы. Структуру подлежащего выполнению приложения удобно представить графом, например, как это показано на рис. 3.1, который будем далее использовать для иллюстрации решения поставленной задачи [9]:

$$G = \{< z_i , t_i > | i = 1, \dots , M \},$$

где z_i – номер задачи (вершины) в графе (первая цифра); t_i – время решения задачи (вторая цифра в вершине графа); M — количество задач в пакете G.

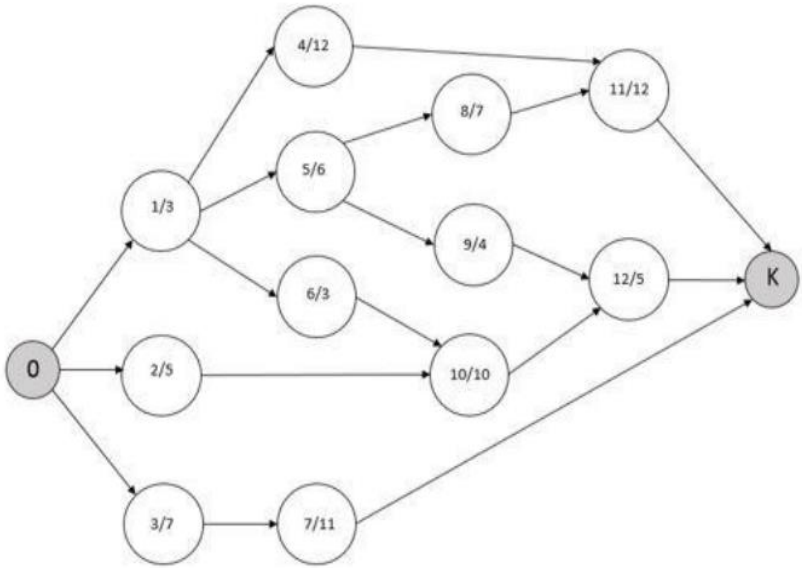


Рис. 3.1. Граф решаемых задач

В нашем примере вычисления начинаются в вершине О и завершаются в вершине *К* (здесь вершины *О* и *К* – фиктивные, используются как исходная и завершающая точки процесса выполнения задач). Дуги графа представляют собой передачу результатов вычислений от задачи z_i к задаче z_j Длительность передачи является весом дуги, обозначим ее соответственно t_{ij}. Поскольку предполагается, что решение набора задач производится на мультипроцессоре и оперативная память является общей для всех процессоров, будем считать, что $t_{ij} = 0$. Требуется организовать вычислительный процесс многопроцессорной (многоядерной) бортовой вычислительной системы таким образом, чтобы выполнить все задачи набора за минимально возможное в данной системе время, т.е. найти такое расписание выполнения задач по процессорам:

$$Sch\ eduleZ = \{z_i\ ,t_{н(i)}\ ,t_{з(i)}\ ,\ n_i\ \},$$

которое обеспечивает $min\ T_Z$ при заданном количестве вычислителей, где $t_{н(i)}$ – время начала выполнения задачи z_i , $t_{з(i)}$ – время завершения выполнения задачи z_i , n_i – номер процессора, выделенного для выполнения задачи z_i .

3.3. Решение задачи

3.3.1. Проверка возможности организации параллельного вычислительного процесса

Известно, что существующий в графе задач G критический путь определяет минимальный срок выполнения всех работ, т.е. в нашем случае минимальное время выполнения заданного набора задач. Для нахождения перечня вершин, образующих критический путь, и его длины можно использовать различные методы, в том числе метод линейного программирования. Потенциальную параллельность выполнения заданного набора задач можно определить, преобразовав граф задач в ярусно-параллельную форму.

Ярусно-параллельная форма графа (ЯПФ) – деление вершин ориентированного ациклического графа на перенумерованные подмножества V_j такие, что, если дуга идет от вершины $v_l \in V_j$ к вершине $v_m \in V_k$, то обязательно $j < k$. Каждое из множеств V_j называется ярусом ЯПФ, j – его номером, количество вершин $|V_j|$ в ярусе – его шириной. Количество ярусов в ЯПФ называется её высотой, а максимальная ширина её ярусов – шириной ЯПФ. Для ЯПФ графа алгоритма важным является тот факт, что операции, которым соответствуют вершины одного яруса, не зависят друг от друга (не находятся в отношении связи), и поэтому заведомо существует параллельная реализация алгоритма, в которой они могут быть выполнены параллельно на разных устройствах вычислительной системы.

Поэтому ЯПФ графа алгоритма может быть использована для подготовки такой параллельной реализации алгоритма. Минимальной высотой всех возможных ЯПФ графа является его критический путь. Построение ЯПФ с высотой, меньшей критического пути, невозможно. Если в составе яруса могут быть вершины, находящиеся в различных отношениях (например, параллельности или альтернативы, что типично для граф-схем параллельных алгоритмов), ярус называется сечением, а ЯПФ – множеством сечений. Наличие более одного отношения между вершинами сечения существенно усложняет большинство алгоритмов обработки.

Для получения ЯПФ предварительно необходимо построить матрицу смежности исходного графа. Матрица смежности – это квадратная матрица размерностью (M+1) × (M+1), (где M – число вершин графа), однозначно представляющая его структуру. Обозначим ее как $A = |a_{ij}|$, где каждый элемент матрицы определяется следующим образом: $a_{ij} = 1$, если есть дуга (i,j), $a_{ij} = 0$, если нет дуги (i,j). В нашем примере матрица смежности будет иметь следующий вид, приведенный на рис. 3.2. Алгоритм распределения модулей системы по уровням:

1. Находим в матрице нулевые строки. В нашем случае это только одна строка с номером 13.

2. Вершина с этим номером образует нулевой уровень ЯПФ.

3. Вычеркиваем столбцы с номерами найденных вершин. В нашем случае – столбец 13.

4. Находим в матрице нулевые строки (7, 11, 12). Это вершины 1-го уровня.

5. Вычеркиваем столбцы с номерами 7, 11, 12.

6. Находим в матрице нулевые строки (3, 4, 8, 9 и 10). Это вершины 2-го уровня.

7. Вычеркиваем столбцы с номерами найденных вершин.

8. Находим в матрице нулевые строки (2, 5, 6). Это вершины 3-го уровня.

9. Вычеркиваем столбцы с номерами 5, 6.

10. Вершина с номером 1 образует 4-й уровень.

номер вершины															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
номер вершины	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	3	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	5	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	6	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	8	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	9	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	11	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рис. 3.2. Матрица смежности

Полученная таким образом ЯПФ заданного набора задач представлена на рис. 3. Алгоритм определения критического в ЯПФ приведен в [10]. В нашем случае это довольно просто сделать перебором всех возможных путей. Такой путь длиной в 28 единиц процессорного времени образует последовательность вершин $W_{кр} = \{1, 5,8,11\}$. Из рисунка

следует, что высота ЯПФ равна четырем, минимальная ширина – $B_{мин}$ = 1, а максимальная – $B_{макс}$ = 5. Таким образом граф задач далеко не прямоугольный, что неудобно для организации параллельного вычислительного процесса. Визуально из рис. 3 понятно, что граф можно легко переформатировать, не меняя связей между вершинами, например, можно переместить вершину 3 на 4-й уровень. Что же касается вершины 2, то для ее перемещения на уровень 4, нужно проверить, не приведет ли этот шаг к задержке выполнения вершины 10. А если и задержит, то это не должно привести к увеличению длины критического пути.

Действительно, начало времени выполнения вершины 10 зависит от времени завершения вершин 2 и 6. Однако пока неизвестно, какая вычислительная производительность будет выделена на каждый ярус, ответить на этот вопрос невозможно. Поэтому окончательное расписание вычислительного процесса можно составить только после решения вопроса о количестве процессоров, выделенных для решения задач, представленных графом G.

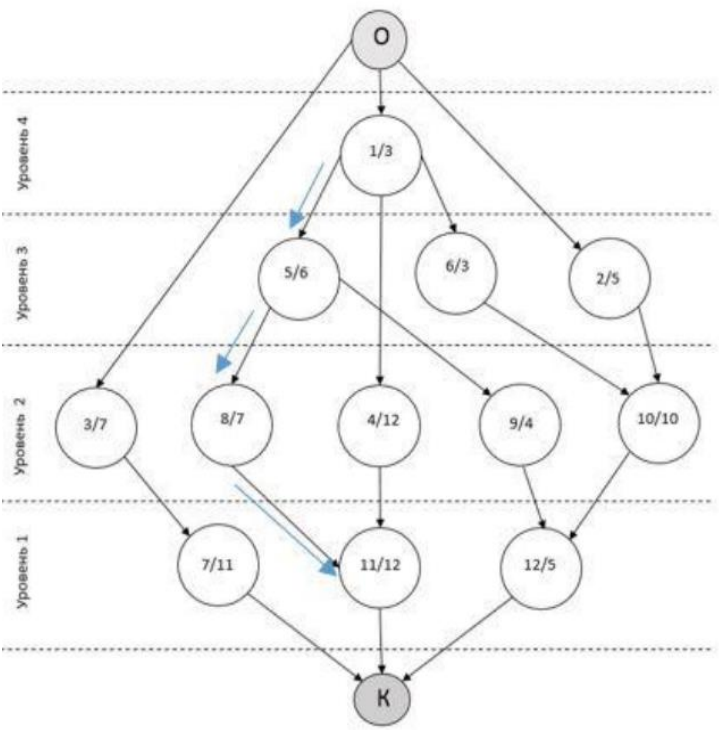


Рис. 3.3. Граф задач в ЯПФ

3.3.2. Выбор количества процессоров для параллельного вычислительного процесса

Далее будем считать, что можно использовать некоторое количество процессоров в заданном диапазоне. Предварительно целесообразно определить минимальную вычислительную мощность $T_{\text{мин}}$, которая позволить реализовать заданный набор задач. Ее легко найти, просуммировав времена реализации всего набора задач:

$$T_{\text{мин}} = \sum_{i=1}^{M-1} t_i.$$

В нашем примере Tмин= 85. При этом вычислительная нагрузка по ярусам неравномерна и составляет следующие значения:

$$T1 = 28,\ T2 = 40,\ T3 = 14,\ T4 = 3. \qquad (3.1)$$

Также неравномерна и ширина ярусов графа G:

$$B1 = 3,\ B2 = 5,\ B3 = 3,\ B4 = 1. \qquad (3.2)$$

Очевидной является необходимость перестройки ЯПФ графа решаемых задач. Однако, как следует из рис. 3.3, вершины 2 и 3 можно передвинуть в четвертый ярус, а вершину 4 – в третий ярус, получив тем самым более прямоугольную ЯПФ (рис. 3.4). После такой реорганизации графа вычислительная нагрузка по ярусам составит следующие значения:

$$T1 = 28,\ T2 = 21,\ T3 = 21,\ T4 = 15. \qquad (3.3)$$

Ширина ярусов становится одинаковой и равна трем. Для дальнейшего упрощения задачи можно поступить следующим образом. Выделим отдельный процессор на реализацию критического пути в графе. Его загрузка составит 28 единиц. Для выполнения оставшейся вычислительной работы, равной 57 единицам, потребуется как минимум 2 процессора. Однако при этом неизбежно превышение времени выполнения всего набора задач по сравнению с найденным временем критического пути. Кроме того, встает вопрос о том, какие из оставшихся задач (кроме входящих в критический путь) на какой процессор необходимо назначить. При этом желательно обеспечить равномерность загрузки процессоров.

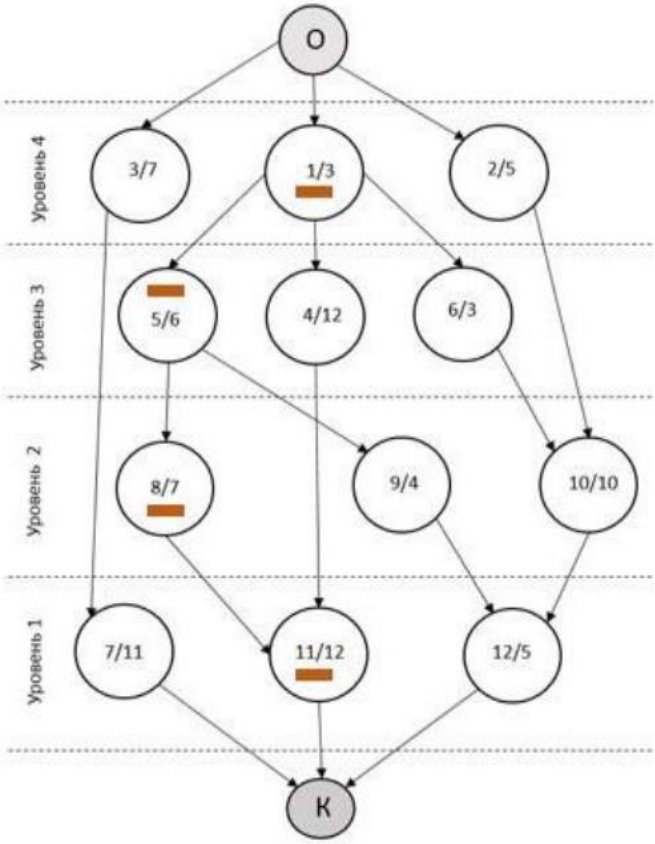


Рис. 3.4. Преобразованная ЯПФ

Рассмотрим возможную постановку задачи в данной ситуации. Размещение задач по процессорам удобно представить двудольным графом вида $G = (Z, P, E)$, где E – множество дуг, отображающих размещение множества задач. Требуется так распределить множество задач

$$Z' = \{z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10}, z_{11}, z_{12}, \}$$

по процессорам P (второй и третий), чтобы разница в загрузке процессоров была минимальной и все задачи Z были выполнены. Если ограничений на размещение задач по процессорам нет, то в таком графе каждая вершина $z_i \in Z'$ соединена дугами со всеми вершинами P (рис. 3.5).

Поставим в соответствие дугам E графа G множество переменных $X = \{x_{ij} | i \in I, \quad j = 2,3\}$. Каждая переменная x_{ij} образуется по правилу $x_{ij} = 1$, если будет принято решение выполнять задачу z_i на втором процессоре p_2 (первый процессор выделен на задачи критического пути). В противном случае $x_{ij} = 0$ и задача выполняется на третьем процессоре.

Применительно к графовой интерпретации задача сводится к отысканию частичного графа

$$G_o = (Z, P, E_o),$$

где $E_o \subseteq E$. При этом множество найденных дуг E_o определит оптимальное значение булевых переменных $X_o = \{x_{ij} | i \in I, \ j = 2,3\}$ в соответствии с целевой функцией

$$C = \sum_{i \in I} \sum_{j=2} t_{ij} \cdot x_{i2} - \sum_{i \in I} \sum_{j=3} t_{ij} \cdot x_{i3} \to min. \qquad (3.4)$$

Определим ограничения задачи. Первая группа ограничений связана с тем фактом, что каждая задача в результате решения должна быть назначена только на один вычислитель. Это условие записывается следующим соотношением:

$$\forall i \sum_{j=2}^{j=3} x_{ij} = 1. \qquad (3.5).$$

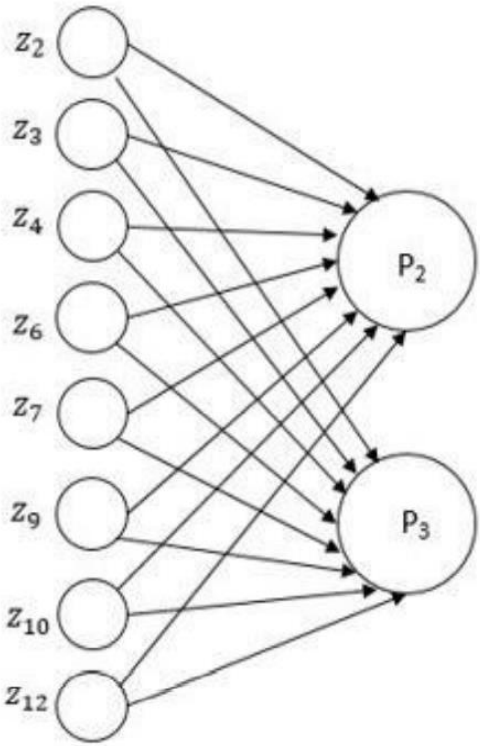


Рис. 3.5. Графовая интерпретация задачи о назначения

Вторая группа ограничений должна обеспечить равномерную нагрузку процессоров. Поскольку общая загрузка второго и третьего процессоров составляет 57 единиц, целесообразно задать следующие ограничения по загрузке каждого процессора:

$$\sum_{i \in I} t_{ij} \cdot x_{ij} \leq 28 \;\; j = 2{,}3. \qquad (3.6)$$

Поскольку все переменные задачи – двоичные, вторая группа ограничений имеет следующий вид:

$$\forall xij \in \{0{,}1\}. \qquad (3.7)$$

Решение легко получить в электронных таблицах (рис. 6). По полученному решению распределение задач по процессорам представлено на рис. 3.6.

t_1	значение	X_{22}	X_{23}	
t_2	5,00	1,00	0,00	1,00
t_3	7,00	1,00	0,00	1,00
t_4	12,00	1,00	0,00	1,00
t_6	3,00	0,00	1,00	1,00
t_7	11,00	0,00	1,00	1,00
t_9	4,00	1,00	0,00	1,00
t_{10}	10,00	0,00	1,00	1,00
t_{12}	5,00	0,00	1,00	1,00

Цел. Функция	-1,00		
Ограничения по загрузке процессора 2	28	28	
Ограничения по загрузке процессора 3	29	28	

Рис. 3.6. Решение задачи (4) – (7) о распределении задач по процессорам

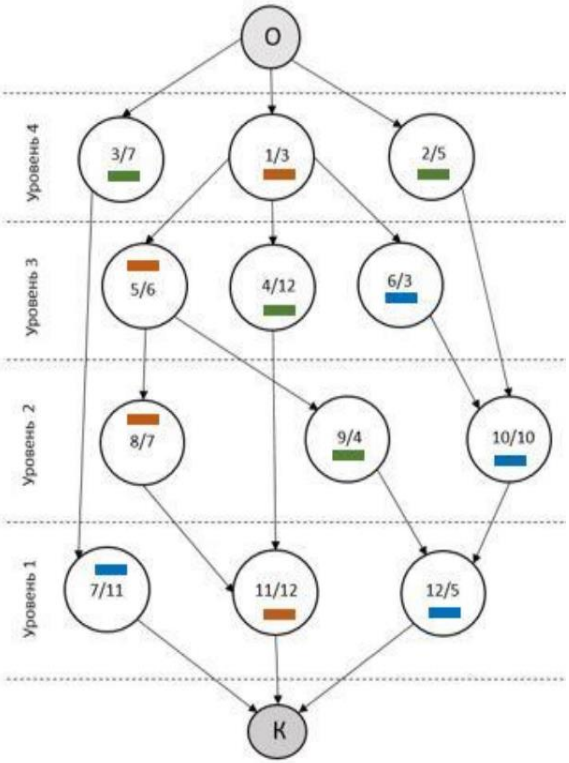


Рис. 3.7. Распределение задач по процессорам: –
1-й процессор, –2-й процессор, – 3-й процессор)

3.3.3. Разработка расписания

Согласно полученному решению, загрузка процессоров распределяется следующим образом: процессор 1 – 28; процессор 2 – 26; процессор 3 – 29. Однако анализируя граф, представленный на рис. 3.8, легко видеть, что на первом и четвёртом уровнях занято не три, а два процессора. Кроме того, такое распределение задач по процессорам не учитывает их информационной связности. Поэтому при разработке расписания необходимо учитывать следующие два обстоятельства:

1. Задержка в завершении задач, не лежащих на критическом пути, до определенного момента может не влиять на срок завершения всего набора задач. Такие задачи обладают резервом времени – таким промежутком времени, на который может быть отсрочено завершение задачи без нарушения сроков завершения критического пути.

2. При составлении расписания с целью сокращения возможных простоев можно переносить выполнение запланированной к выполнению задачи с одного процессора на другой, если при этом не меняется последовательность выполнения, обусловленная информационными связями задач. Такая возможность связана с тем, что процессоры имеют общую память. Резерв времени показывает, сколько имеется в запасе времени для выполнения данной задачи, на которое можно увеличить продолжительность данной задачи, не изменяя при этом продолжительности пути, которому принадлежит задача.

Резерв времени задачи z_i определяется как разность между наиболее поздним $t_i^{\text{п}}$ и ранним $t_i^{\text{р}}$ сроками выполнения задачи и временем выполнения самой задачи: $R_i = t_i^{\text{п}} - t_i^{\text{р}} - t_i$. Поздний срок $t_i^{\text{п}}$ – это такой срок завершения задачи, превышение которого вызовет задержку в реализации задач, лежащих на критическом пути. Определим резервы времени исходя из условия, что время завершения задачи не должно увеличивать величину критического пути. Для этого вначале определяем наиболее ранние сроки начала выполнения задач, рассматривая вычислительный процесс, начиная с вершины *O*:

Для задач уровня 4: $t_1^{\text{р}} = 0$; $t_2^{\text{р}} = 0$; $t_3^{\text{р}} = 0$.

Для задач уровня 3 (здесь надо учитывать наиболее раннее выполнение задач предшествующего уровня):

$t_4^{\text{р}} = 3$; $t_5^{\text{р}} = 3$; $t_6^{\text{р}} = 3$.

Для задач уровня 2:

$t_8^{\text{р}} = 9$; $t_9^{\text{р}} = 9$; $t_{10}^{\text{р}} = 6$.

Для задач уровня 1: $t_7^{\text{р}} = 7$; $t_{11}^{\text{р}} = 16$; $t_{12}^{\text{р}} = 16$.

После этого определяем наиболее поздние сроки завершения задач, начиная с вершины *K*:

Для задач уровня 1: $t_7^{\text{п}} = 28$; $t_{11}^{\text{п}} = 28$; $t_{12}^{\text{п}} = 28$.

Для задач уровня 2: $t_8^{\text{п}} = 16$; $t_9^{\text{п}} = 23$; $t_{10}^{\text{п}} = 23$.

Для задач уровня 3: $t_4^{\text{п}} = 16$; $t_5^{\text{п}} = 9$; $t_7^{\text{п}} = 13$.

Для задач уровня 4: $t_1^{\text{п}} = 3$; $t_2^{\text{п}} = 13$; $t_3^{\text{п}} = 17$.

Определяем резервы времени для выполняемых задач, которые могут быть использованы для снижения возможных простоев процессоров:

Для задач уровня 4:

R1 = 3 – 0 – 3 = 1;

R2 = 13 – 0 – 5 = 8;

R3 = 17 – 0 – 7 = 10.

Для задач уровня 3:
R4 = 16 – 3 – 12 = 1;
R5 = 9 – 3 – 6 = 0;
R6 = 13 – 3 – 3 = 7.
Для задач уровня 2:
R8 = 16 – 9 – 7 = 0;
R9 = 23 – 9 – 4 = 10;
R10 = 23 – 6 – 10 = 7.
Для задач уровня 1:
R7 = 28 – 7 – 11 = 10;
R11 = 28 – 16 – 12 = 0;
R12 = 28 – 16 – 5 = 7.

Построим диаграмму загрузки процессоров, исходя из поярусной последовательности выполнения задач. Очевидно, что задачи с нулевым значением резерва времени принадлежат критическому пути и должны выполняться без задержек. Следующими должны выполняться задачи с минимально возможными резервами времени и т.д. При этом следует учитывать наличие информационной связи между задачами. Построить оптимальный план распределения по процессорам с учетом этих условий достаточно сложно, поскольку данная задача относится к классу NP-полных. Поэтому целесообразно использовать эвристические алгоритмы, дающие достаточно хорошее решение.

Пример эвристического алгоритма:

Рассматриваем задачи высшего уровня N. Назначаем на процессоры задачи, готовые к выполнению.

Переходим к следующему уровню N = N−1 (уровень 3). Если N=0, конец.

На процессор с минимальным временем освобождения назначаем задачу яруса, имеющую минимальный резерв времени на выполнение.

Если не все задачи текущего уровня назначены на процессоры, переходим к п.3 алгоритма. Если все задачи текущего уровня назначены на процессоры, переходим к п. 2.

Построим расписание, используя этот алгоритм.

Рассматриваем задачи высшего уровня N=4. Назначаем на процессоры задачи, готовые к выполнению. В нашем случае это задачи z_1, z_2, z_3.

Переходим к следующему уровню N = N−1 (уровень 3).

На процессор с минимальным временем освобождения (в нашем случае процессор 1, красный цвет на диаграмме) назначаем задачу

яруса, имеющую минимальный резерв времени на выполнение. Это задача z_5.

Так как не все задачи уровня 3 назначены на процессоры выбираем задачу текущего уровня с минимальным резервом времени. Это задача z_4. Назначаем ее на процессор с минимальным временем освобождения (процессор 3).

Оставшуюся задачу уровня 3 z_6 назначаем на процессор 2. Все задачи текущего уровня назначены. Переходим к п. 2 алгоритма, N= 3 – 1. Переход на уровень 2.

На процессор 1 назначаем задачу z_8, имеющую нулевой резерв времени. На процессоры 2 и 3 назначаем задачи z_{10} и z_9. Все задачи уровня 2 назначены. Переходим к п. 2 алгоритма, N= 2 – 1. Переход на уровень 1.

На процессор 1 назначаем задачу z_{11}, имеющую нулевой резерв времени. На процессоры 2 и 3 назначаем оставшиеся задачи.

Все задачи уровня 1 назначены. Переходим к п. 2 алгоритма, N= 1 – 1 =0. Конец. Как видно из диаграммы (рис. 3.8), время реализации заданного пакета задач составляет 31 единицу времени при длине критического пути 28 единиц.

Рис. 3.8. Диаграмма загрузки процессоров для поярусного выполнения задач

Дальнейшее улучшение расписания можно получить, учитывая тот факт, что в полученном расписании загрузка процессоров неравномерная и, следовательно, имеется возможность улучшения плана за счет увеличения загрузки третьего процессора. Это можно получить, если вместо задач 9 и 12 на этом процессоре выполнять задачу 7, а задачи 9 и 12 выполнять на втором процессоре. Для проверки возможности такой замены нужно пересчитать времена наиболее позднего завершения этих задач, исходя из реально возможного значения критического пути, который в нашем примере равен 29. Такая проверка показывает, что перестановка возможна. Скорректированный план загрузки процессоров представлен на рис. 3.9. Заметим, что еще одну возможность оптимизации

плана дает допустимость прерывания задачи и продолжение ее решения на другом процессоре.

Рис. 3.9. Итоговая диаграмма загрузки процессоров

Приведенный пример оптимизации параллельного вычислительного процесса в бортовых вычислительных системах позволяет сделать следующие выводы. Для планирования параллельного вычислительного процесса в бортовых вычислительных системах возможно использование ярусно-параллельных форм представления наборов решаемых задач. Такое представление позволяет предварительно определить требуемое количество процессоров и минимально возможное время реализации заданного набора задач. Предварительное распределение решаемых задач по процессорам системы можно получить решением задачи о назначениях методом булевого линейного программирования. Определение наиболее ранних и наиболее поздних моментов начала и завершения задач в ряде случаев позволяет улучшить первоначальное распределение задач по процессорам. Улучшение полученного плана реализации задач и устранение обнаруженных простоев процессоров слабо поддается формализации и требует “ручной доработки”.

Литература к гл. 3

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ – Петербург, 2004. 608 с.

2. Боресков А. В. [и др.] Параллельные вычисления на GPU. Архитектура и программная модель CUDA. М.: Изд-во МГУ, 2012. 336 с.

3. Бурцев В.С. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ. М.: Изд-во “Нефть и газ”, 1997. 150 с.

4. Таненбаум Э. Распределенные системы. Принципы и парадигмы, Ван Сеен. М.СПб.: Питер, 2003. 877с.

5. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: Издательский дом «Вильямс», 2003. 512 с.

6. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Н.Новгород, 2001. 386 с.

7. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем, СПб.: Петербург, 2002. 360 с.

8. Гохрингер Д., Хюбнер М., Бекер Ю. Архитектура адаптивных многопроцессорных системна кристалле: новая степень свободы при проектировании систем и в поддержке при выполнении // Мир радиоэлектроники. М.: Техносфера, 2012. С. 146 – 173.

9. Карпов В.Е. Введение в распараллеливание алгоритмов и программ // Компьютерные исследования и моделирование. 2010. Т. 2 № 3. С. 231 272.

10. Назаров С.В. Операционные системы специализированных вычислительных комплексов: теория построения и системного проектирования. М.: Машиностроение, 1989. 400 с.

Глава 4. ОПТИМИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ БОРТОВЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

4.1. Особенности современных бортовых систем

Актуальность тематики параллельных вычислений осознана достаточно давно как в связи с низкой надежностью и производительностью компьютеров, так и в связи с появлением многопроцессорных систем и многоядерных процессоров. Последние годы активно используются программируемые логические матрицы и, видимо, недалеко то время, когда появятся программируемые матрицы процессорных элементов. Технология обеспечения надежности и высокой производительности на основе параллельных вычислений естественным образом стала преобладающей в бортовых вычислительных системах реального времени (БВС). Такие системы находят широкое применение в авиационной и космической технике, а также в наземных и водных подвижных объектах.

На время решения большинства задач, возлагаемых на БВС, накладываются жесткие временные ограничения. Часто встает вопрос максимально возможного сокращения времени выполнения этих задач. Реализация этих требований приводит к необходимости организации параллельных вычислительных процессов. В данной работе представлена совокупность математических моделей, формулировок задач и подходов к их решению, позволяющих построить параллельный вычислительный процесс реализации информационно-связанных задач в минимально возможное время в условиях заданных ограничений по вычислительной среде. Математической основой решения являются теория графов, сетевого планирования и управления и расписаний.

Как правило, наборы программ, реализуемых современными бортовыми вычислительными системами (БВС), можно представить совокупностью информационно-связанных подпрограмм (заданным набором задач – ЗНЗ). Это позволяет представлять решаемые задачи в виде нагруженного графа и, соответственно, использовать для их анализа методы теории графов и сетевого планирования и управления. ЗНЗ обладают достаточным внутренним параллелизмом, который может быть удобно использован при реализации задач многопроцессорными (многоядерными) бортовыми системами. Основной недостаток этого под-

хода связан с тем, что архитектура аппаратных средств многопроцессорной системы зачастую фиксируется на этапе проектирования и остается неизменной на время выполнения программ. Это означает, что разработчик должен выбирать соответствующую систему для предполагаемых приложений. С этой целью разработчик должен разбить приложение на части, рассмотреть возможности параллелизма при выполнении приложения и выбрать соответствующую систему для своего приложения. Ограничением такого подхода является недостаток распространенных существующих многопроцессорных систем, заключающийся в отсутствии адаптивности на стадии разработки и во время выполнения программы [1].

Программируемая разработчиком логическая матрица предлагает более гибкое решение, потому что с ее помощью аппаратные средства можно переконфигурировать с новыми функциями и использовать повторно с различными приложениями. Некоторые поставщики программируемых пользователем логических матриц (компания Xilinx) предлагают специальную функцию, называемую динамическим и частичным переконфигурированием [2]. Это означает, что часть аппаратных средств системы может быть изменена во время выполнения программы, а оставшаяся часть остается действующей и неизменной.

Программируемая логика от Xilinx позволяют применять интеллектуальные решения в широчайшем спектре отраслей промышленной, научной и потребительской деятельности человека. Это программируемые логические интегральные схемы ПЛИС (FPGA), 3D микросхемы и системы на Кристалле (SoCs), которые устанавливают стандарты низкой стоимости, высокой производительности и пониженного энергопотребления мощности. Архитектура Xilinx UltraScale™ обеспечивает беспрецедентный уровень интеграции и возможностей, обеспечивая при этом производительность на системном уровне ASIC-класса для самых требовательных приложений, требующих высочайшей пропускной способности, быстродействия памяти, массовых потоков данных, DSP и производительности обработки пакетов [3].

4.2. Постановка задачи

Пусть задана структура приложения, состоящего из некоторого множества информационно-связанных частей (задач) с известным (ожидаемым) временем выполнения каждой задачи заданного набора задач набора (ЗНЗ). Примем следующие ограничения и допущения по ЗНЗ:

1. Предполагается, что это время выполнения каждой задачи набора определяется элементарным вычислителем (процессором или ядром) многопроцессорной бортовой вычислительной системы (БВС), и это время (в условных временных единицах) установлено в процессе отладки.

2. Каждая задача ЗНЗ (или часть задач) обладает внутренним параллелизмом. Время выполнения задачи сокращается пропорционально числу выделенных задаче вычислителей.

3. Задано максимальное количество вычислителей, которое может быть выделено задаче. Оно определяется уровнем параллелизма задачи и изменяется от одного (параллелизма в программе нет) до трех (для случая максимально возможного параллелизма).

4. Определено максимальное количество вычислителей, которое может быть использовано для выполнения ЗНЗ.

Требуется определить с учетом заданных ограничений:

1. Минимально возможное время решения системой ЗНЗ – $T_{\text{ЗНЗ}}^{min}$.

2. Количество вычислителей V_{min}, необходимое для выполнения требования 1.

3. Количество вычислителей V_i, выделяемых каждой задаче ЗНЗ, необходимое для выполнения требования 1.

4. Количество вычислителей $V_{\text{тр}}$, необходимое для выполнения ЗНЗ с заданным требованием по времени реализации – $T_{\text{ЗНЗ}}^{\text{тр}}$.

5. Количество вычислителей V_i, выделяемых каждой задаче ЗНЗ, необходимое для выполнения требования 4.

В данной разделе рассмотрим возможное решение, позволяющее на основе анализа приложения определить целесообразную структуру многопроцессорной системы с выполнением требований на время выполнения приложений. Все далее принимаемые решения и разрабатываемые модели будем сразу иллюстрировать конкретными примерами.

Структуру подлежащего выполнению приложения удобно представить графом, например, как это показано на рис. 4.1, который будем далее использовать для иллюстрации решения поставленной задачи:

$G = \{< z_i, z_j >, t_{ij} \mid j > i,\ \ i = 0, 1, \dots, M-2,\ \ j = 1,2, \dots, M-1\}$, где z_i, z_j – номера событий (вершины) в графе, t_{ij} – время решения задачи (вес соответствующей дуги) при использовании одного вычислителя, M – количество задач в пакете G.

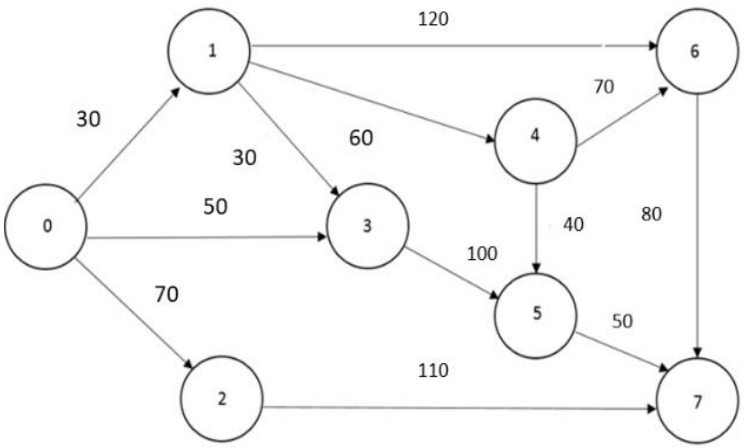


Рис. 4.1. Структура ЗНЗ

Для дальнейшей формализации задачи будем использовать понятия сетевого планирования и управления. В нашем случае работы представляются дугами графа $< z_i, z_j >$, или просто (i, j), причем для любой дуги $j > i$. Обмен информацией между задачами инициируется событиями, например, событие z_1 инициирует завершение работы $< z_0, z_1 >$, длительностью t_{01} ,возможность запуска вычислений $< z_1, z_4 >, < z_1, z_5 >$, и $< z_1, z_6 >$. Организация вычислительного процесса при выполнении данного приложения сводится к определению временных параметров сетевого графика и к его оптимизации по длительности выполнения всего комплекса задачи и затратам вычислительных ресурсов в соответствии с заданными требованиями.

4.3. Решение задачи

4.3.1. Определение величины критического пути и резервов времени по отдельным вычислительным работам при выделении для каждой работы одного вычислителя

Критический путь T_{kr}– это полный путь, определяющий длительность всего комплекса вычислительных работ и имеющий наибольшую продолжительность. Для решения поставленной задачи необходимо найти длину критического пути и возможные резервы времени при выполнении отдельных вычислительных работ. Определение длины критического пути можно проводить различными способами. Наиболее удобным способом расчёта сетевого графика при небольшом количестве работ является расчёт непосредственно на сети графика и заключается в нахождении критического пути и определении резервов времени для работ, которые не располагаются на этом пути.

При производстве расчетов сетевых моделей применяют следующие наименования его параметров [4, 5]:

Продолжительность работы (t_{ij}) (здесь i и j – номера соответственно начального и конечного событий).

Раннее начало работы $t_{ij}^{\text{р.н}}$– характеризуется выполнением всех предшествующих работ и определяется продолжительностью максимального пути от исходного события всей модели до начального события рассматриваемой работы.

Раннее окончание работы $t_{ij}^{\text{р.о}}$– определяется суммой раннего начала и продолжительности рассматриваемой работы.

Позднее начало работы $t_{ij}^{\text{п.н}}$– определяется разностью позднего окончания и продолжительности рассматриваемой работы.

Позднее окончание работы $t_{ij}^{\text{п.о}}$– определяется разностью продолжительности критического пути и максимальной продолжительности пути от завершающего события всей модели до конечного события рассматриваемой работы.

Общий резерв времени работы R_{ij} – характеризуется возможностью роста продолжительности работы без увеличения продолжительности критического пути и определяется как разность между поздним и ранним окончанием рассматриваемой работы. Общий резерв работы принадлежит не только первой работе, но и всем последующим работам данного пути. В случае использования на одной из работ общего резерва критический путь не изменит своей продолжительности, но все последующие работы окажутся критическими и лишатся резерва.

Частный резерв времени работы r_{ij} – характеризуется возможностью увеличения продолжительности работы без изменения раннего начала последующей работы и определяется разностью между ранним началом последующей работы и ранним окончанием рассматриваемой работы. Частный резерв имеет место, когда одним событием заканчивается не менее двух работ. Отличие частного резерва от общего заключается в том, что частный резерв может быть использован только на рассматриваемой или предшествующих работах и не может быть использован на последующих.

Полным резервом некоторого пути в сетевой модели R называют разность между продолжительностью критического пути модели и продолжительностью рассматриваемого пути. Результаты расчета сетевой модели выполнения ЗНЗ, представленного на рис. 4.1, приведены в табл. 4.1. Работы, не имеющие резерва времени и выделенные жирным шрифтом, образуют критический путь T_{kr} = 240.

Если, заданное значение времени $T_{\text{ЗНЗ}}^{\text{тр}} \geq T_{kr}$ и разработчика это устраивает, то можно считать, что задача в части директивного времени решения ЗНЗ выполнена и можно предварительно определить требуемое количество вычислителей V_{min}. Полное время занятости вычислителей можно найти, просуммировав времена выполнения отдельных задач:

$$T_{\text{знз}} = \sum_{i=0}^{i=6} \sum_{j=1}^{j=7} t_{ij} = \ 810. \qquad (4.1)$$

Таблица 4.1

Расчет параметров сетевого графика по рис. 4.1

A	B	C	D	E	F	G	H
Работа (I,J)	Продолжительность работы t_{ij}	Раннее начало работы $t_{ij}^{р.н}$	Раннее окончание работы $t_{ij}^{р.о}$	Позднее начало работы $t_{ij}^{п.н}$	Позднее окончание работы $t_{ij}^{п.о}$	Общий резерв времени работы R_{ij}	Частный резерв времени работы R_{ij}
(0,1)	**30,00**	**0,00**	**30,00**	**0,00**	**30,00**	**0,00**	**0,00**
(0,2)	70,00	0,00	70,00	60,00	130,00	60,00	0,00
(0,3)	50,00	0,00	50,00	40,00	90,00	40,00	0,00
(1,3)	30,00	30,00	60,00	60,00	90,00	30,00	0,00
(1,4)	**60,00**	**30,00**	**90,00**	**30,00**	**90,00**	**0,00**	**0,00**
(1,6)	120,00	30,00	150,00	40,00	160,00	10,00	0,00
(2,7)	110,00	70,00	180,00	130,00	240,00	60,00	0,00
(3,5)	100,00	60,00	160,00	90,00	190,00	30,00	0,00
(4,5)	40,00	90,00	130,00	150,00	190,00	60,00	0,00
(4,6)	**70,00**	**90,00**	**160,00**	**90,00**	**160,00**	**0,00**	**0,00**
(5,7)	50,00	160,00	210,00	190,00	240,00	30,00	0,00
(6,7)	**80,00**	**160,00**	**240,00**	**160,00**	**240,00**	**0,00**	**0,00**

Минимальное количество вычислителей в этом случае можно определить, разделив вычислительную нагрузку на длину критического пути, т.е.

$$V_{min} \geq T_з / T_{kr} = 4 \qquad (4.2)$$

Если выделить один вычислитель на выполнение задач, образующих критический путь, то оставшаяся вычислительная нагрузка потребует для своего выполнения еще не менее трех вычислителей. Знак больше или равно в соотношении (4.2) определяется тем фактом, что организация параллельного вычислительного процесса может потребовать дополнительного числа вычислителей.

4.3.2. Минимизация длины критического пути

Проведенные расчеты показывают, что при выделении для каждой задачи ЗНЗ одного вычислителя время решения заданного набора задач не может быть меньше T_{kr} , равного 240 условным временным единицам. Рассмотрим, какие имеются возможности для сокращения времени решения ЗНЗ, а в нашем случае – сокращения длины критического пути.

4.3.2.1. Повышение производительности вычислителей БВС

Наиболее простое решение – использование вычислителей более высокой производительности. Это – "лобовое" решение, которое, конечно, приведет к уменьшению времени критического пути. Однако оно, во-первых, не всегда возможно и, во-вторых, может привести к неполному использованию вычислительных возможностей бортовой вычислительной системы, и, наконец, удорожает всю систему.

Отметим еще одну особенность использования вычислителей БВС. Как все современные процессорные элементы, вычислитель БВС работает в режиме квантования процессорного времени, что позволяет организовать его мультипрограммную работу. Режим разделения времени позволяет рассматривать один физический вычислитель как совокупность виртуальных вычислителей (процессоров), которые могут выделяться отдельным задачам ЗНЗ. При этом суммарная производительность физического вычислителя, за исключением издержек мультипрограммирования (которыми далее пренебрегаем) равна сумме производительности его виртуальных вычислителей.

4.3.2.2. Использование имеющегося резерва времени вычислителей БВС

Имеющийся резерв времени по работам, не лежащим на критическом пути, свидетельствует о том, что имеется возможность изъятия ресурсов, выделенных на некоторые работы, с целью добавления ресурсов на работы, лежащие на критическом пути. Это приведет к уменьшению длины критического пути, т.е. к сокращению всего цикла выполняемых вычислений. В нашем примере суммарный резерв времени составляет значение

$$R = \sum_{i=0}^{M-2} \sum_{j=1}^{M-1} R_{ij} = 320.$$

Передача ресурса от какой-либо работы означает увеличение ее выполнения в зависимости от величины изъятого ресурса, добавление ресурса к другой работе, лежащей на критическом пути, означает уменьшение ее выполнения в соответствии с величиной добавленного ресурса. Например, изъятие половины резерва, т.е. 30 еди-

ниц ресурса от работы (0,2) означает ее удлинение до 100 единиц времени. Это означает, что для ее выполнения нужен будет виртуальный процессор со следующим значением доли физического процессора

$$V_{02} = \frac{t_{02}}{t_{02} + 0{,}5 * R_{02}} = \frac{70}{70 + 0{,}5 * 60} = 0{,}7.$$

Изъятый ресурс можно добавить с целью сокращения критического пути, например, к работе (6,7). При этом время выполнения этой работы уменьшится с 80 до 50 единиц времени, но для выполнения этой работы потребуется виртуальный процессор со следующим значением доли физического процессора

$$V_{67} = \frac{t_{67}}{t_{67} - 0{,}5 * R_{02}} = \frac{80}{80 - 0{,}5 * 60} = 1{,}6$$

Для решения задачи минимизации критического пути необходимо построить модель линейного программирования с целью определения значений T_{kr} для различных вариантов перераспределения ресурсов. Удобно это сделать в электронных таблицах, например, Excel. Воспользуемся для этого формализацией задачи поиска критического пути, предложенной в [6]. Исходные данные удобно представить в форме таблицы (табл. 4.2).

Таблица 4.2

Исходные данные для решения задачи поиска критического пути

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	Переменные	x_{01}	x_{02}	x_{03}	x_{13}	x_{14}	x_{16}	x_{27}	x_{35}	x_{45}	x_{46}	x_{57}	x_{67}	Ограничив. фукция	Ограничкние
	Значения переменных	1	0	0	0	1	0	0	0	0	1	0	1		
Номера узлов	0	1	1	1										1	1
	1	-1			1	1	1							0	0
	2		-1					1						0	0
	3			-1	-1				1					0	0
	4					-1				1	1			0	0
	5								-1	-1		1		0	0
	6						-1				-1		1	0	0
	t_{ij}	30	70	50	30	60	120	110	100	40	70	50	80		

Задача поиска критического пути на основе табл. 2 заключается в определении значения функции

$$T_{kr} = \sum_{i=0}^{i=6} \sum_{j=1}^{j=7} t_{ij} \cdot x_{ij} \rightarrow \max \qquad (4.3)$$

при следующих ограничениях:

$$x_{01} + 1 \cdot x_{02} + 1 \cdot x_{03} = 1; \qquad (4.4)$$

$$-1 \cdot x_{01} + 1 \cdot x_{13} + 1 \cdot x_{14} + 1 \cdot x_{16} = 0 \qquad (4.5)$$

$$-1 \cdot x_{02} + 1 \cdot x_{27} = 0; \qquad (4.6)$$

$$-1 \cdot x_{03} - 1 \cdot x_{13} + 1 \cdot x_{35} = 0; \qquad (4.7)$$

$$-1 \cdot x_{14} + 1 \cdot x_{45} + 1 \cdot x_{46} = 0; \qquad (4.8)$$

$$-1 \cdot x_{35} - 1 \cdot x_{45} + 1 \cdot x_{57} = 0; \qquad (4.9)$$

$$-1 \cdot x_{16} - 1 \cdot x_{46} + 1 \cdot x_{67} = 0; \qquad (4.10)$$

$$\forall x_{ij} \in \{0,1\} | \; i = 0,1, \dots 6; j = 1,2, \dots 7. \qquad (4.11)$$

Решение задачи (4.3) – (4.11) для исходного графа сетевой модели показано на рис. 4.2.

Определим теперь, как изменится величина критического пути, если 40 единиц резерва изъять у работы (4,5) и передать их для выполнения работе (6,7). При этом задача (4,5) будет выполняться 80 единиц времени, а задача (67) – 40 единиц времени. Решение задачи в этом случае показано на рис. 4.3 и дает значение критического пути, равное 220. Если рассчитанное значение критического пути окажется больше директивного, можно рассмотреть другие варианты использования резервов других работ, пока не будет полученное требуемое значение критического пути. Если этого не удается сделать за счет резервов в выполнении работ, необходимо увеличить количество ресурсов для их выполнения. В данном случае – перейти к более производительным вычислителям или переписать программы для возможности организации параллельного вычислительного процесса.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
		Переменные	x_{01}	x_{02}	x_{03}	x_{13}	x_{14}	x_{16}	x_{27}	x_{35}	x_{45}	x_{46}	x_{57}	x_{67}	Ограничив. функция	Ограничение
		Значения переменных	1	0	0	0	1	0	0	0	0	1	0	1		
	Номера узлов	0	1	1	1										1	1
		1	-1			1	1	1							0	0
		2		-1					1						0	0
		3			-1	-1				1					0	0
		4					-1				1	1			0	0
		5								-1	-1		1		0	0
		6						-1				-1		1	0	0
		t_{ij}	30	70	50	30	60	120	110	100	40	70	50	80		
		T_{kr}	240													

Результаты поиска решения

Решение найдено. Все ограничения и условия оптимальности выполнены.

Тип отчета: Результаты, Устойчивость, Пределы

Сохранить найденное решение

Восстановить исходные значения

OK | Отмена | Сохранить сценарий... | Справка

Рис. 4.2. Критический путь исходного графа сетевой модели

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
		Переменные	x_{01}	x_{02}	x_{03}	x_{13}	x_{14}	x_{16}	x_{27}	x_{35}	x_{45}	x_{46}	x_{57}	x_{67}	Ограничив. функция	Ограничение
		Значения переменных	1	0	0	0	1	0	0	0	1	0	1	0		
	Номера узлов	0	1	1	1										1	1
		1	-1			1	1	1							0	0
		2		-1					1						0	0
		3			-1	-1				1					0	0
		4					-1				1	1			0	0
		5								-1	-1		1		0	0
		6						-1				-1		1	0	0
		t_{ij}	30	70	50	30	60	120	110	100	80	70	50	40		
		T_{kr}	220													

Результаты поиска решения

Решение найдено. Все ограничения и условия оптимальности выполнены.

Тип отчета: Результаты, Устойчивость, Пределы

Сохранить найденное решение

Восстановить исходные значения

OK | Отмена | Сохранить сценарий... | Справка

Рис. 4.3. Вариант расчета критического пути сетевой модели выполнения графа задач при использовании резерва задачи (4,5)

4.3.2.3. Использование потенциальной параллельности задач ЗНЗ для минимизации критического пути

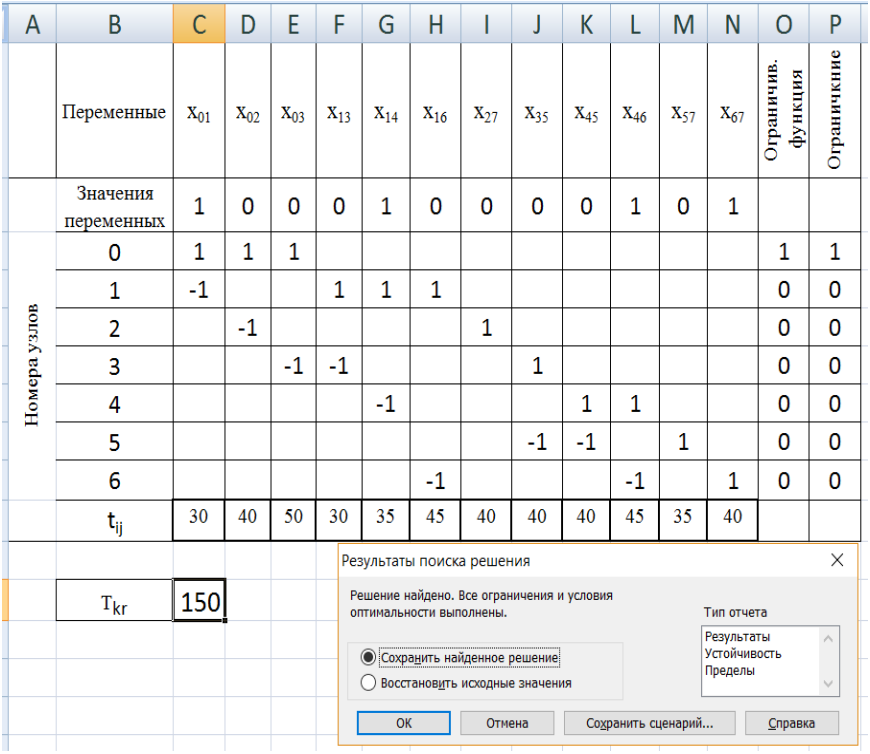
Как отмечено в постановке задачи данной статьи, каждая задача ЗНЗ обладает внутренним параллелизмом. Время выполнения задачи сокращается пропорционально числу выделенных задаче вычислителей. Пусть известен возможный уровень параллелизма по каждой задаче и, соответственно, время выполнения каждой задачи ЗНЗ при максимально возможном распараллеливании, табл. 4.3.

Таблица 4.3

Время выполнения задач ЗНЗ при максимальном распараллеливании

№ задачи	00,1	00,2	00,3	11,3	11,4	11,6
Время выполнения на одном вычислителе	330	770	550	330	660	1120
Возможный уровень параллелизма	1	11-2	1	1	11-2	11-3
Время выполнения при максимальном рапараллеливании	**330**	**440**	**550**	**330**	**335**	**445**
Время выполнения на одном вычислителе	1110	1100	870	770	550	440
Возможный уровень параллелизма	11-3	11-3	11-2	11-2	11-2	1
Время выполнения при максимальном рапараллеливании	**440**	**440**	**440**	**445**	**335**	**440**

Определим длину критического пути для случая достаточного количества вычислителей в БВС. В данном случае распараллеленное выполнение задач (0,2), (1,4), (1,6), (2,7), (3,5), (4,5), (4,6) и (5,7) потребует кроме основных 12 вычислителей (по одному на каждую задачу) дополнительно 11 вычислителей. Для учета этого факта достаточно в разработанной электронной таблице по рис. 4.3 изменить содержимое ячеек, содержащих значения t_{ij}. Решение в этом случае показано на рис. 4.4. Таким образом, T_{kr} = 150. Можно ли еще сократить длину критического пути? За счет распараллеливания – нет. Здесь все возможности использованы в полной мере. Осталась только одна задача критического пути (0,1), но ее распараллелить невозможно, разве что написать заново.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
		Переменные	x_{01}	x_{02}	x_{03}	x_{13}	x_{14}	x_{16}	x_{27}	x_{35}	x_{45}	x_{46}	x_{57}	x_{67}	Ограничив. функция	Ограничкние
		Значения переменных	1	0	0	0	1	0	0	0	0	1	0	1		
Номера узлов		0	1	1	1										1	1
		1	-1			1	1	1							0	0
		2		-1					1						0	0
		3			-1	-1				1					0	0
		4					-1				1	1			0	0
		5								-1	-1		1		0	0
		6						-1				-1		1	0	0
		t_{ij}	30	40	50	30	35	45	40	40	40	45	35	40		
		T_{kr}	150													

Рис. 4.4. Расчет критического пути выполнения графа задач при использовании максимального распараллеливания задач ЗНЗ

Обновленные данные по параметрам сетевого графика приведены в табл. 4.4. Критический путь длиной 150 условных единиц образует цепочка вершин 0-1-4-5-7. Суммарная трудоемкость критического пути определяется суммой времени выполнения задач (0,1), (1,4), (4,6), (6,7) и составляет 150 условных единиц. Имеющийся общий резерв, как видно из табл. 4, составляет 240 условных единиц. Этот факт свидетельствует о возможности уменьшения числа вычислителей, выделяемых на задачи некритического пути.

Таблица 4.4

Расчет параметров сетевого графика с параллельным выполнением ЗНЗ

Работа (I,J)	Продолжительность работы t_{ij}	Раннее начало работы $t_{ij}^{р.н}$	Раннее окончание работы $t_{ij}^{р.о}$	Позднее начало работы $t_{ij}^{п.н}$	Позднее окончание работы $t_{ij}^{п.о}$	Общий резерв времени работы R_{ij}	Частный резерв времени работы R_{ij}
(0,1)	**30,00**	**0,00**	**30,00**	**0,00**	**30,00**	**0,00**	**0,00**
(0,2)	40,00	0,00	40,00	70,00	110,00	70,00	0,00
(0,3)	50,00	0,00	50,00	25,00	75,00	25,00	0,00
(1,3)	30,00	30,00	60,00	45,00	75,00	15,00	0,00
(1,4)	**35,00**	**30,00**	**65,00**	**30,00**	**65,00**	**0,00**	**0,00**
(1,6)	45,00	30,00	75,00	65,00	110,00	35,00	0,00
(2,7)	40,00	40,00	80,00	110,00	150,00	70,00	0,00
(3,5)	40,00	60,00	100,00	75,00	115,00	15,00	0,00
(4,5)	40,00	65,00	105,00	75,00	115,00	10,00	0,00
(4,6)	**45,00**	**65,00**	**110,00**	**65,00**	**110,00**	**0,00**	**0,00**
(5,7)	35,00	115,00	115,00	115,00	115,00	0,00	0,00
(6,7)	**40,00**	**110,00**	**150,00**	**110,00**	**150,00**	**0,00**	**0,00**

4.3.3. Минимизация количества ресурсов выполнения пакета ЗНЗ без увеличения длины критического пути

Будем считать, определенный выше критический путь, равный 150 единицам времени, соответствует директивному времени решения заданного пакета задач. В этом случае становится актуальной задача минимизации ресурсов (в нашем случае количества процессоров), необходимых для реализации всех задач при условии непревышения директивного значения длины критического пути. Другими словами, длительности выполнения всех работ пакета нужно изменить так, чтобы длина любого пути в графе была в идеале равна или меньше длины критического пути. Практически все вычислительные работы будут увеличены с учетом возможных резервов времени для их выполнения.

Для формализации задачи введем дополнительные обозначения:

t_{ij}^{x}– время выполнения работы (i,j) после минимизации количества выделяемых ресурсов;

T_i^x – время свершения событий в графе работ (события отождествляются с вершинами графа);

V_{ij} – исходное число процессоров, которое позволяет выполнить все работу с учетом возможного параллельного выполнения задачи;

N_m –минимальное количество процессоров, которое будет получено в результате решения оптимизационной задачи (пока еще без учета возможности параллельной работы различных задач ЗНЗ).

d_{ij} – доля физического вычислителя соответствующего виртуального процессора, необходимая для выполнения работы (i,j) после минимизации количества выделяемых ресурсов,

$$d_{ij} = V_{ij} - V_{ij}\frac{\left(t_{ij}^x - t_{ij}\right)}{t_{ij}^x}.$$

В качестве исходного количества вычислителей можно принять сумму V_{min} (см. выражение (2)) с дополнительным числом вычислителей, которые надо добавить для выполнения параллельных ветвей задач ЗНЗ (см. табл.3). Таким образом, $N = 4+11=15$. Значение минимального количества процессоров определяется как сумма долей физических процессоров в виртуальных вычислителях, т.е.

$$N_{min} = \sum_{i=0}^{i=6}\sum_{j=1}^{j=7} d_{ij},$$

В качестве целевой функции задачи выбираем время занятости процессоров (полное машинное время) на решение пакета задач ЗНЗ. Его нужно минимизировать за счет использования имеющихся резервов времени на выполнение отдельных задач. Таким образом, необходимо найти такие значения множества переменных t_{ij}^x выполнения работ (i,j) ,которые минимизируют целевую функцию времени выполнения ЗНЗ

$$T_{\text{знз}} = \sum_{i=0}^{i=6}\sum_{j-1}^{j=7} t_{ij} - \sum_{i=0}^{i=6}\sum_{j=1}^{j=7}\left(t_{ij}^x - t_{ij}\right) \to min \qquad (4.12)$$

Первое слагаемое этой функции представляет собой полные затраты машинного времени на решение всего пакета задач. Второе экономию машинного времени при условии того, что можно увеличить время решения некоторых задач за счет имеющегося резерва

времени на их выполнение (здесь$(t_{ij}^{x} - t_{ij}) \geq 0$). Рассмотрим ограничения, которые должны учитываться при решении этой задачи.

Первый вид ограничений связан с принятым условием использования для решения задачи только имеющихся резервов для выполнения задач пакета. Отсюда следует система ограничений для продолжительности выполнения работ следующего вида:

$$t_{ij} + R_{ij} \geq t_{ij}^{x} \geq t_{ij}. \qquad (4.13)$$

Второй вид ограничений должен обеспечить такое изменение длительности выполнения всех работ пакета, чтобы длина любого пути в графе была в идеале равна длине критического пути. В рассматриваемом примере таких путей шесть (перечислим их последовательностями вершин графа): P1 : 0 – 2 – 7; P2 : 0 – 3 – 5 – 7; P3: 0 – 1 – 6 – 7; P4 : 0 – 1 – 3 – 5 – 7; P5 : 0 – 1 – 4 – 5 – 7; P6 : 0 – 1 – 4 – 6 – 7. Ограничения на длину этих путей имеют следующий вид:

$$L_i\,(P_i) \leq T_{kr} | 1 = 1,2,\ldots,M-1 \qquad (4.14)$$

Решение задачи (12) – (14) показано на рис. 4.5.

A	B	C	D	E	F	G	H	I	J
Работа (I,J)	Продолжительность работы t_{ij}	Общий резерв времени работы R_{ij}	$t_{ij} + R_{ij}$	t_{ij}^{x}	$t_{ij}^{x} - t_{ij}$	Исходное число вычислителей V_{ij}	Минимальное число вычислителей d_{ij}	Путь	Длина пути
(0,1)	**30,00**	**0,00**	**30,00**	30,00	**0,00**	**1,00**	1,00	0-2-7	150,00
(0,2)	40,00	70,00	110,00	75,00	35,00	2,00	1,07	0-3-5-7	150,00
(0,3)	50,00	25,00	75,00	75,00	25,00	1,00	0,67	0-1-6-7	150,00
(1,3)	30,00	15,00	45,00	45,00	15,00	1,00	0,67	0-1-3-5-7	150,00
(1,4)	**35,00**	**0,00**	**35,00**	35,00	**0,00**	**2,00**	2,00	0-1-4-5-7	150,00
(1,6)	45,00	35,00	80,00	80,00	35,00	3,00	1,69	0-1-4-6-7	150,00
(2,7)	40,00	70,00	110,00	75,00	35,00	3,00	1,60		
(3,5)	40,00	15,00	55,00	40,00	0,00	3,00	3,00		
(4,5)	40,00	10,00	50,00	50,00	10,00	2,00	1,60		
(4,6)	**45,00**	**10,00**	**55,00**	**45,00**	**0,00**	**2,00**	2,00		
(5,7)	35,00	0,00	35,00	35,00	0,00	2,00	2,00		
(6,7)	**40,00**	**0,00**	**40,00**	40,00	**0,00**	**1,00**	1,00		
$T_{знз}$	315,00								

Результаты поиска решения

Решение найдено. Все ограничения и условия оптимальности выполнены.

Тип отчета: Результаты, Устойчивость, Пределы

Сохранить найденное решение

Восстановить исходные значения

OK | Отмена | Сохранить сценарий... | Справка

Рис. 4.5. Решение задачи минимизации вычислительных ресурсов

Как видно из результата решения задачи минимизации использования ресурсов, число процессоров при назначении отдельного вычислителя на одну и только одну задачу для реализации ЗНЗ без увеличения длины критического составляет $N_m = 23$ физических процессоров (сумма по столбцу G по рис. 4.5). При этом физические вычислители могут простаивать значительное количество времени. Переход на виртуальные процессоры более низкой производительности, но работающие без простоев, позволяет сократить число вычислителей до $N_{min} = 18{,}29$ виртуальных процессоров (сумма по столбцу H на рис. 4.5). Однако структура пакета (см. рис. 4.1) свидетельствует о возможности параллельной и мультипрограммной работы этих процессоров при выполнении задач пакета ЗНЗ.

4.3.4. Возможности организации мультипроцессорного выполнения пакета задач, представленного сетевой моделью

Потенциальную параллельность выполнения заданного набора задач можно определить, преобразовав граф задач в ярусно-параллельную форму [7]. Ярусно-параллельная форма графа (ЯПФ) – деление вершин ориентированного ациклического графа на перенумерованные подмножества V_j такие, что, если дуга идет от вершины $v_l \in V_j$ к вершине $v_m \in V_k$,то обязательно $j < k$.

Каждое из множеств V_j называется ярусом ЯПФ, j – его номером, количество вершин $|V_j|$ в ярусе – его шириной. Количество ярусов в ЯПФ называется её высотой, а максимальная ширина её ярусов – шириной ЯПФ. Для ЯПФ графа важным является тот факт, что операции, которым соответствуют вершины одного яруса, не зависят друг от друга (не находятся в отношении связи), и поэтому заведомо существует параллельная реализация алгоритма, в которой они могут быть выполнены параллельно на разных устройствах вычислительной системы. Поэтому ЯПФ графа алгоритма может быть использована для подготовки такой параллельной реализации алгоритма.

Для получения ЯПФ пакета задач, представленного в форме сетевой модели, как показано на рис. 4.1, его предварительно необходимо преобразовать, заменив дуги графа (работы) вершинами. В пре-

образованном графе (рис. 4.6) номера вершин соответствуют работам, длительность которых пересчитана в соответствии с решением задачи (12 – 14) по рис. 4.5.

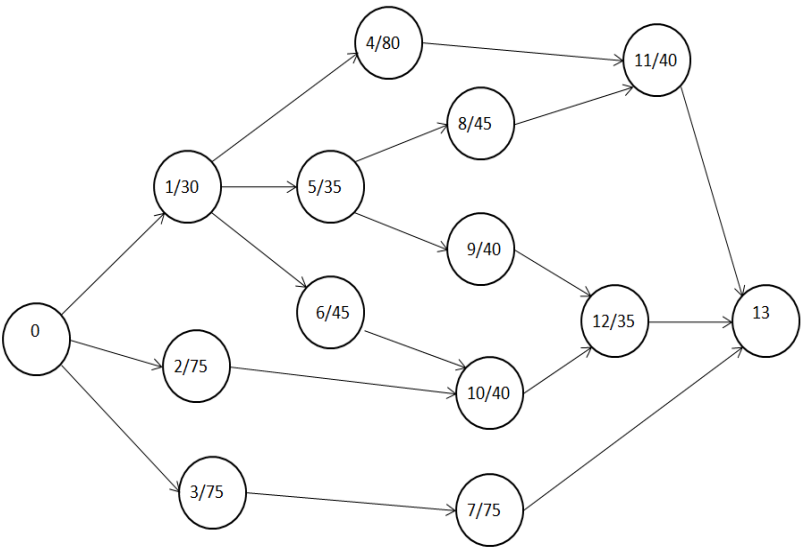


Рис. 4.6. Преобразованный граф пакета задач приложения

Получить ЯПФ можно, построив матрицу смежности графа. Матрица смежности – это квадратная матрица размерностью $(M+1) \times (M+1)$, где M – число вершин графа, однозначно представляющая его структуру. Обозначим ее как $A = |a_{ij}|$, где каждый элемент матрицы определяется следующим образом:aij =1,если есть дуга (i, j), $a_{ij} = 0$, если нет дуги (i, j). В нашем примере матрица смежности будет иметь следующий вид, приведенный на рис. 4.7.

Алгоритм распределения модулей системы по уровням:

1. Находим в матрице нулевые строки. В нашем случае это только одна строка с номером 13.
2. Вершина с этим номером образует нулевой (низший) уровень ЯПФ.
3. Вычеркиваем столбцы с номерами найденных вершин. В нашем случае – столбец 13.
4. Находим в матрице нулевые строки (7, 11, 12). Это вершины 1-го

уровня.
5. Вычеркиваем столбцы с номерами 7, 11, 12.
6. Находим в матрице нулевые строки (3, 4, 8, 9 и 10). Это вершины 2-го уровня.
7. Вычеркиваем столбцы с номерами найденных вершин.
8. Находим в матрице нулевые строки (2, 5, 6). Это вершины 3-го уровня.
9. Вычеркиваем столбцы с номерами 5, 6.
10. Вершина с номером 1 образует 4-й уровень.

ЯПФ графа задач пакета, поученная на основе матрицы смежности, представлена на рис. 8. В каждой вершине графа в виде дроби указан номер вершины (числитель) и время выполнения (знаменатель). Рядом с вершиной указано количество виртуальных процессоров, которые необходимы для выполнения задачи соответствующего номера.

	номер вершины														
номер вершины		0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	3	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	5	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	6	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	8	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	9	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	11	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рис. 4.7. Матрица смежности

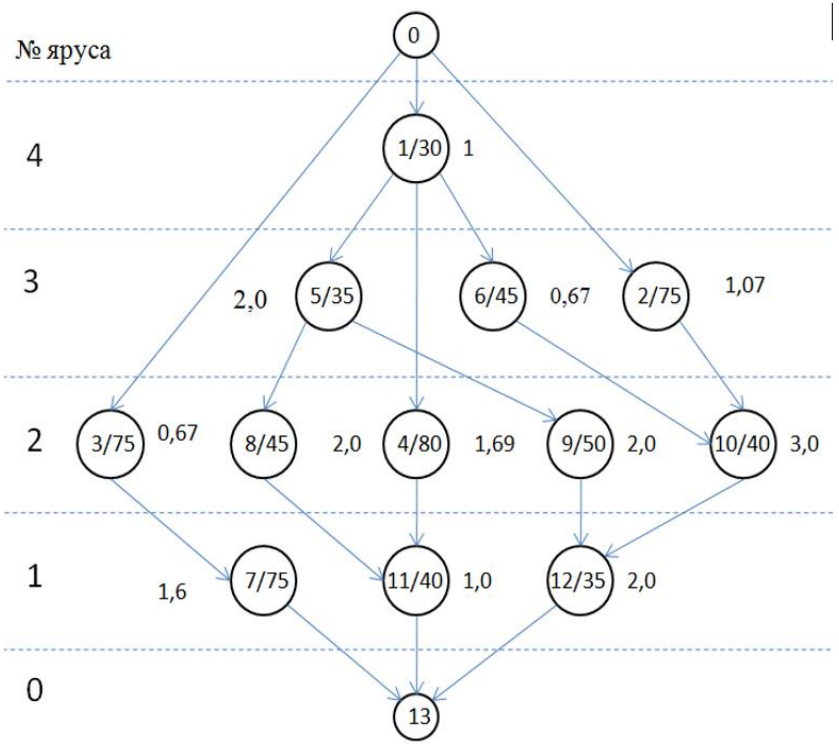


Рис. 4.8. ЯПФ графа пакета задач приложения

Построенный граф задач в ЯПФ далеко не прямоугольный, что неудобно для организации параллельного вычислительного процесса и определения минимально необходимого количества вычислителей. Визуально по рис. 8 понятно, что граф можно легко перестроить, не меняя связей между вершинами, например, можно переместить вершины 2 и 3 на 4-й уровень, а вершину 4 – на 3-й. После таких преобразований граф примет вид, показанный на рис. 4.9.

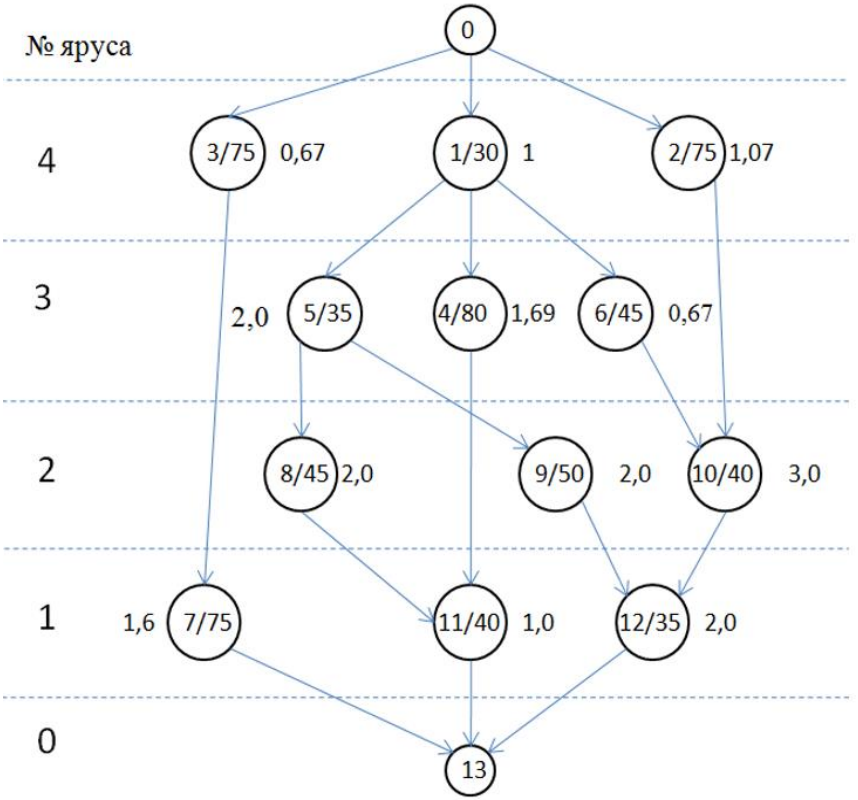


Рис. 4.9. Преобразованный ЯПФ пакета ЗНЗ

По ЯПФ (рис. 4.9) легко построить план реализации вычислительного процесса во времени. На рис. 4.10 сверху показана шкала времени (равная длине критического пути), под которой обозначены временные промежутки реализации задач пакета. В нижней части диаграммы приведены сведения о необходимом количестве виртуальных вычислителей. Красными стрелками показаны передачи данных между задачами ЗНЗ. В нашем примере минимальное количество виртуальных вычислителей $V_{min} = 9{,}6$. В этом случае можно получить минимально возможное время решения системой ЗНЗ со значением $T_{ЗНЗ}^{min} = 150$ условных единиц. Для этого результата БВС должна содержать $N_m = 10$ физических процессоров.

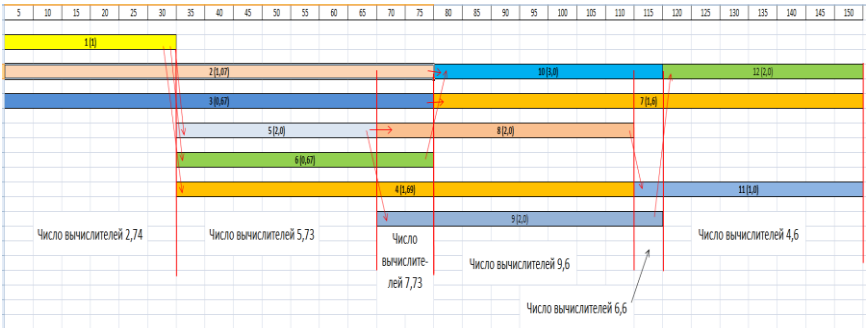


Рис. 4.10. Диаграмма выполнения вычислительных работ пакета ЗНЗ

4.3.5. Минимизация загрузки БВС

Как показывает диаграмма по рис. 4.10, требуемое количество вычислителей для реализации ЗНЗ с требуемым минимальным значением $T_{\text{знз}} = 150$ условных единиц процессорного времени равно10. При этом полная величина затрат процессорного времени БВС составит 1500 условных единиц. Реальные затраты (без учета простоя процессоров) легко определить по рис. 4.10. Здесь можно выделить 4 этапа выполнения задач пакета ЗНЗ. Каждый этап характеризуется неизменным количеством вычислителей и продолжительностью. Так на первом этапе выполнения ЗНЗ занято 2,74 вычислителя, а продолжительность этапа – 30 условных единиц. Таким образом, затраты этого этапа составляют 30*2,74=82,0 единиц процессорного времени. Аналогично вычисляются затраты процессорного времени на последующих этапах (табл. 4.5).

Таблица 4.5

Затраты процессорного времени на выполнении ЗНЗ

№ этапа	1	2	3	4	5	6	Суммарное время
Продолжительность этапа	30	35	10	35	5	35	
Количество вычислителей	2,74	5,73	7,73	9,6	6,6	4,6	
Затраты процессорного времени	82,2	200,55	77,3	336	33	161	890,05

Полагая, что ПЛИС вычислителей обладает свойством реконфигурации, имеет смысл заранее запрограммировать переключения освободившихся от выполнения текущей задачи процессоров на последующие задачи согласно диаграмме выполнения работ ЗНЗ. Другими словами, надо составить расписание загрузки и переключения процессоров. Постановка и формализация задачи построения такого расписания достаточно сложна и могла бы быть темой отдельной статьи. Однако в нашем небольшом примере можно легко увидеть возможности уменьшения простоя вычислителей. После завершения задачи 1 (желтый цвет на диаграмме) освободившийся вычислитель можно назначить на выполнение задачи 5, затем 9 и 12. Таким образом, этот вычислитель будет работать без простоя.

Приведенные в этой главе математические модели и пример оптимизации параллельного вычислительного процесса в бортовых вычислительных системах позволяет сделать следующие выводы:

1. Эффективное планирование параллельного вычислительного процесса в бортовых вычислительных системах можно обеспечить, используя методы сетевого планирования и управления в совокупности с математическим аппаратом ярусно-параллельных графов и математического программирования.

2. Реализация в программируемых логических интегральных схемах ПЛИС процессорных элементов с разделяемой производительностью позволяет удобно использовать аппарат виртуальных процессоров для решения задачи распараллеливания вычислительного процесса и минимизации физических ресурсов для выполнения ЗНЗ с заданными требованиями по времени выполнения.

3. Возможности программируемой реконфигурации разрабатываемых в настоящее время ПЛИС процессорных элементов с разделяемой производительностью целесообразно использовать для построения эффективных расписаний загрузки процессоров БВС.

Литература к гл. 4

1. Гохрингер Д., Хюбнер М., Бекер Ю. Архитектура адаптивных многопроцессорных систем на кристалле: новая степень свободы при проектировании систем и в поддержке при выполнении // Мир радиоэлектроники. М.: Техносфера, 2012. С. 146 – 173.

2. Хитт Д. Стратегия Xilinx – быстрее д вигаться к адаптируемому, интеллектуальному миру // ЭЛЕКТРОНИКА. Наука. Технология. Бизнес. 2018. № 4. С. 84 – 87.

3. Xilinx Ultrascale + Обзор. [Электронныйресурс]. URL:https://developer.ridgerun.com/wiki/index.php?title=Xilinx_-Ultrascale%2B_Overview

4. Кофман, А.В. Сетевые методы планирования: Применение системы ПЕРТ и ее разновидностей при управлении производ. и науч.-исслед. проектами : Пер. с фр. М. Прогресс, 1968. 181 с.

5. Сетевой график. [Электронныйресурс]. URL: http://www.stroitelstvonew.ru/1/sete-voy_grafik.shtml.

6. Вагнер Г. Основы исследования операций. Том 1. Пер. с англ. Изд. «МИР». – М.: 1972.– 336 с.

7. Назаров С.В. Операционные системы специализированных вычислительных комплексов: теория построения и системного проектирования. – М.: Машиностроение,1989. – 400 с.

8. Назаров С.В., Барсуков А.Г. Оптимизация параллельных вычислений бортовых систем реального времени (часть 1) // ЭЛЕКТРОНИКА: Наука, Технология, Бизнес. 2020. № 10.С. 110 - 116.

Глава 5. Управление предоставлением облачных вычислительных ресурсов реального времени

5.1. Использование облачных ресурсов

Рынок облачных технологий уверенно растет, отмечают эксперты, ежедневно появляются новые пользователи, особенно среди предприятий малого и среднего бизнеса. Ведущие коммерческие компании и государственные службы приходят к пониманию удобства применения облачных вычислений. Современные высокотехнологичные вычислительные мощности позволяют эффективно хранить, анализировать и обрабатывать данные. Ведущие ИТ-компании и инженеры давно обсуждают плюсы и минусы использования облачных технологий. Эксперты прогнозируют проблемы и возможные потери конфиденциальных данных в "облаках" из-за широкого спроса и притока пользователей.

К основным достоинствам облачных технологий относятся высокий уровень безопасности и конфиденциальности хранения данных, степень надежности и неограниченные вычислительные ресурсы. "Облако" удобно в использовании, поскольку требуется лишь установка простых приложений на любые пользовательские терминалы, и позволяет сэкономить на покупке лицензионных продуктов, ресурсов и программ. В результате применения облачных технологий уменьшаются расходы на приобретение дорогостоящих мощных компьютеров, серверов, сокращаются затраты на оплату труда ИТ-специалистов, обслуживающих локальный дата-центр.

Рассмотрим создание системы реального времени (СРВ), сочетающей в себе web-серверы, вычислительная мощность которых формируется на основе арендованной инфраструктуры IaaS (Infrastructure-as-a-Service – инфраструктура как сервис). Арендатор создает собственный парк виртуальных серверов (виртуальный дата-центр) и разворачивает коммерческую систему для обслуживания пользователей, получая прибыль. В предложенной модели с точки зрения теории стратегических игр два партнера: СРВ (виртуальный дата-центр – арендуемая часть IaaS) и природа (сеть клиентов), как принято в теории игр именовать непредсказуемого партнера. Модель и система мониторинга СРВ позволяют управлять предоставлением облачной вычислительной мощности и сбалансировать расходы на аренду и прибыль, получаемую в результате обслуживания пользователей.

Остановимся подробнее на одном из вопросов организации функционирования широкого класса компьютерных систем, используемых в электронном бизнесе. Имеются в виду электронная коммерция, онлайновый маркетинг, оформление заявок, осуществление платежей, информационная поддержка доставки товаров, электронный документооборот, информационно-справочные системы, финансовые системы, системы взаимодействия с клиентами и т.п. [1–7]. Подобные системы используют интернет-технологии для передачи данных и предоставления web-сервисов на основе специализированных web-сайтов. Последние представляют собой системы реального времени на основе web-серверов.

Современные облачные технологии позволяют получать требуемые для этого ресурсы в форме инфраструктуры IaaS, на базе которой предоставляются услуги аренды вычислительных ресурсов и систем хранения, таких как виртуальные серверы с заданной вычислительной мощностью и каналы связи нужной пропускной способности для доступа к хранилищам данных и внешним ресурсам. При этом клиент может использовать любые операционные системы и приложения [8–11].

5.2. «Облачная» система реального времени

Рассмотрим вопросы организации работы систем реального времени и требования к вычислительным ресурсам (количеству процессоров), обеспечивающим работу систем реального времени. Как правило, СРВ предназначены для своевременного и предсказуемого реагирования на запросы, поступающие в систему. Для таких систем характерен определенный набор запросов некоторого типа $Z = \{z_i, i = 1, 2, \dots, M\}$, для каждого из них в системе предусмотрена заранее разработанная программа P_i, хранящаяся в памяти системы. Основное требование к СРВ заключается в своевременности обработки запросов. Реакция на запрос z_i должна уложиться в заранее заданный интервал времени R_i.

Все системы реального времени принято подразделять на жесткие системы реального времени, в которых недопустимо превышение заданного значения R_i для реализации запроса z_i, и мягкие (гибкие системы реального времени). В последних допускается "опоздание" при обработке запроса z_i, но повышается "стоимость" опоздания, которую обозначим C_i.

Внешние запросы (события), на которые СРВ должна реагировать, можно разделить на периодические (возникающие через регулярные интервалы времени) и непериодические (непредсказуемые). Если в систему поступает M потоков периодических запросов и запрос z_i по-

ступает с периодом T_i, а на его обработку затрачивается t_i времени системы, то все потоки могут быть своевременно обработаны в однопроцессорной системе только при выполнении условия:

$$\sum_{i=1}^{M} \frac{t_i}{T_i} \leq 1. \qquad (5.1)$$

Системы реального времени, удовлетворяющие условию (5.1), считают поддающимися планированию или планируемыми. Классический пример статического алгоритма планирования реального времени для прерываемых периодических процессов – алгоритм DMS (Date Monotonic Scheduling) [12]. Основанный на статических приоритетах алгоритм подходит для планирования независимых периодических процессов с заданным директивным временем выполнения. Как было показано Лю (Liu) и Лейлэнд (Layland) в [13], использование статических приоритетов целесообразно только при не слишком высокой загруженности центрального процессора. Алгоритм DMS гарантированно работает в любой системе периодических процессов при условии:

$$U = \sum_{i=1}^{M} \frac{t_i}{T_i} \leq M\left(2^{\frac{1}{M}} - 1\right). \qquad (5.2)$$

Например, U ≤ 0,8284 для двух процессов, когда количество процессов M стремится к бесконечности, это выражение имеет вид:

$$lim_{M\to\infty} M\left(\sqrt[M]{2} - 1\right) = ln2 \approx 0{,}69 \ldots \qquad (5.3)$$

Гиперболическая граница (5.3) – более жесткое условие планирования, чем то, которое было представлено Лю и Лейлэнд, то есть имеем:

$$\prod_{i=1}^{M} (U_i + 1) \leq 2,$$

где U_i – использование ЦП для каждой задачи. Приблизительная оценка заключается в том, что DMS может удовлетворить все предельные сроки, если загрузка процессора составляет менее 69,32%. Другие 30,7% ЦП могут быть выделены для задач с меньшим приоритетом (не для реального времени). Известно, что случайно созданная система периодических задач будет соответствовать всем предельным срокам, когда ис-

пользование ЦП составляет 85% или меньше, однако это зависит от знания точной статистики задачи (периодов, крайних сроков), которая не может быть гарантирована для всех наборов задач.

Таким образом, алгоритм DMS надежен при относительно невысокой загрузке процессора. К тому же статическое планирование, используемое алгоритмом, не всегда возможно. Другой недостаток алгоритма DMS – неэффективность для планирования непериодических процессов и непостоянных временных интервалов использования центрального процессора. А именно эти условия характерны для систем мягкого реального времени.

Наиболее подходящий алгоритм планирования для таких систем – EDF (Earliest Deadline – First) – процесс с ближайшим сроком завершения первым. Это динамический алгоритм планирования, не требующий периодичности процессов и постоянства временных интервалов использования центрального процессора. Каждый раз при поступлении в систему процесс объявляет о своем присутствии и о сроке выполнения задания. Планировщик хранит список процессов, отсортированный по срокам выполнения. Алгоритм запускает процесс с ближайшим по времени сроком. В случае перехода нового процесса в состояние готовности система сравнивает его срок выполнения со сроком текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса. Алгоритм EDF работает с любым набором процессов, для которого возможно планирование. При этом коэффициент загрузки процессора может достигать 100%.

5.3. Постановка задачи использования облачных ресурсов

Алгоритм EDF можно считать достаточно неплохим для планирования работы систем мягкого реального времени. В целом с учетом непредвиденного характера нагрузки систем рассматриваемого класса следует ввести некоторую метрику (показатель) функционирования системы. Наиболее целесообразной метрикой считается "штраф" за опоздание в обслуживании запросов, поступающих в систему. Размер штрафа C_i, получаемого системой за опоздание с обработкой процесса z_i, можно сформировать следующим образом:

$C_i^- = K_1(D_i, t_i)$, если имеется дефицит производительности процессоров СРВ, в результате которого превышено время обработки запроса t_i по сравнению с заданным директивным значением D_i (в данном случае $t_i > D_i$);

$C_i^+ = K_2(D_i, t_i)$, если имеется избыточная производительность процессора СРВ, в результате чего запрос z_i обрабатывается быстрее директивного значения (в данном случае $t_i < D_i$);

$$C_i = 0, \text{ если } t_i = D_i.$$

Коэффициент K_1 можно трактовать как удельные финансовые издержки, связанные с потерями клиентов системой, которые отказались от ее услуг из-за неудовлетворительного обслуживания. Коэффициент K_2 можно трактовать как удельные финансовые издержки, обусловленные эксплуатацией СРВ избыточной производительности.

Будем считать, что СРВ на платформе IaaS строится как многопроцессорная (и возможно, многоядерная) система с возможностью динамического выделения некоторого числа процессоров (от 1 до n) для обслуживания входящих запросов. Выбор количества работающих в конкретный момент времени процессоров зависит от интенсивности поступления запросов в систему, которая в общем случае слабо предсказуемая либо непредсказуемая. В условиях полной неопределенности можно попробовать решить задачу на основе теории стратегических игр.

5.3.1. Решение задачи методами теории игр

С точки зрения теории стратегических игр в данной игре два партнера: СРВ (арендуемая часть IaaS) и природа (сеть клиентов), как принято в теории игр именовать полностью непредсказуемого партнера [14]. Стратегии СРВ обозначим $R_1, R_2, \ldots, R_n$, а стратегии природы – $P_1, P_2, \ldots, P_k$. Предположительно потребность в производительности в некоторые периоды функционирования (например, рабочий день, вечер, праздничные дни и т.п.) составляет $P_1, P_2, \ldots, P_k$, а СРВ может состоять из 1, 2, …, n процессоров, обеспечивающих соответственно значения реальной производительности $R_1, R_2, \ldots, R_n$.

Схематично матрицу выигрышей можно записать в следующем виде:

	P_1	P_2	…	P_k
R_1	C_{11}	C_{12}	…	C_{1k}
R_2	C_{21}	C_{22}	…	C_{2k}
…	…	…	…	…
R_n	C_{n1}	C_{n2}	…	C_{nk}

Здесь C_{ij} – штраф, получаемый системой при имеющейся производительности системы реального времени R_i и требуемой производительности P_j. Предположим, что с использованием тех или иных методов матрица выигрышей получена. В соответствии с теоремой стратегических игр для нашего случая, когда значения из множеств $P = \{P_1, P_2, \dots, P_k\}$ и $R = \{R_1, R_2, \dots, R_n\}$ могут принимать конечное число, оптимальное решение заключается в поиске смешанных стратегий.

Из теории стратегических игр следует, что при использовании смешанных стратегий есть, по крайней мере, одно оптимальное решение с ценой игры V, которое находится между верхним и нижним значениями [14–16]. Следует заметить, что всегда $V > 0$.

Допустим, оптимальная смешанная стратегия СРВ складывается из стратегий $R_1, R_2, \dots, R_n$ с вероятностями, равными $p_1, p_2, \dots, p_n$ $(p_1 + p_2 + \dots + p_n = 1)$, а оптимальная стратегия клиентской сети – из стратегий $P_1, P_2, \dots, P_k$, которые применяются с вероятностями, равными $q_1, q_2, \dots, q_n$ $(q_1 + q_2 + \dots + q_n = 1)$. Если СРВ применяет оптимальную стратегию, а клиентская сеть чистую стратегию P_j $(j = 1,2, \dots k)$, то средний штраф, получаемый системой, составит: $C_j = p_1 \cdot c_{1j} + p_2 \cdot c_{2j} + \dots + p_n \cdot c_{nj}$ $(j = 1,2, \dots, k)$.

Особенность оптимальной стратегии СРВ состоит в том, чтобы при произвольном поведении противника (клиентской сети) она обеспечивала штраф не больший, чем цена игр V. Отсюда имеем систему ограничений:

$$\left.\begin{array}{l} p_1 \cdot c_{11} + p_2 \cdot c_{21} + \dots + p_n \cdot c_{n1} \le V, \\ p_1 \cdot c_{12} + p_2 \cdot c_{22} + \dots + p_n \cdot c_{n2} \le V, \\ \dots \\ p_1 \cdot c_{1k} + p_2 \cdot c_{2k} + \dots + p_n \cdot c_{nk} \le V. \end{array}\right\} \qquad (5.4)$$

Систему (4.18) можно преобразовать, разделив по частям на V:

$$\left.\begin{array}{l} c_{11} \cdot x_1 + c_{21} \cdot x_2 + \dots + c_{n1} \cdot x_n \le 1, \\ c_{12} \cdot x_1 + c_{22} \cdot x_2 + \dots + c_{n2} \cdot x_n \le 1, \\ \quad \dots \\ c_{1k} \cdot x_1 + c_{2k} \cdot x_2 + \dots + c_{nk} \cdot x_n \le 1, \end{array}\right\} \qquad (5.5)$$

Здесь $x_1 = P_1/V$, $x_2 = P_2/V$ …, $x_n = P_n/V$. Из условия $p_1 + p_2 + \cdots + p_n = 1$ следует, что

$$x_2 + \cdots + x_n = 1/V. \quad (5.6)$$

Значения величин $p_1, p_2, \ldots, p_n$ должны быть подобраны таким образом, чтобы гарантированное значение штрафа СРВ было по возможности минимальным, то есть чтобы достигалось

$$V = \min \text{ или } 1/V = max.$$

Таким образом, задача сводится к нахождению таких значений $x_1, x_2, \ldots, x_n$, чтобы

$$x_1 + x_2 + \cdots + x_n = max. \quad (5.7)$$

Кроме того, должны выполняться дополнительные граничные условия, а именно $p_i \geq 0$ $(i = 1,2, \ldots, n)$, следовательно, имеем:

$$x_i = P_i/V \geq \text{для } i = 1,2, \ldots, n \quad (5.8)$$

Из этого следует, что нахождение оптимальной смешанной стратегии СРВ сводится к решению классической задачи линейного программирования с целевой функцией (5.7), ограничениями (5.5) и (5.8).

В результате решения этой задачи по определенным значениям $x_1 + x_2 + \cdots + x_n$ из уравнения (5.6) можно определить значение *V*, а затем из соотношений (5.8) значения $p_1, p_2, \ldots, p_n$, которые определяют оптимальную смешанную стратегию СРВ.

5.3.2. Пример решения задачи

Задана матрица игры (рис. 5.1), в которой стратегии СРВ (арендуемая часть IaaS) обозначены $R_1, R_2, \ldots, R_5$. Каждая стратегия предполагает включение в работу одного, двух, …, пяти процессоров. Возможные стратегии клиентов, обозначенные $P_1, P_2, \ldots, P_4$, предусматривают нужную производительность, обеспечиваемую загрузку 0,5; 1,25; 2,5 и 4,5 процессоров (здесь 0,5 следует понимать как загрузку одного процессора на 50%, соответственно это относится к другим данным). Пусть за дефицит производительности СРВ получает штраф, равный 5 условным единицам ($K_1 = 5$), если не достает производительности, обеспечиваемой одним процессором, то есть за избыточную производительность система получает штраф в 4 единицы за каждый лишний выделенный процессор ($K_2 = 4$). Например, для совокупности стратегий {R2, P3} клиенту необходимо 2,5 процессора, система предоставляет 2 процессора, имеет место нехватка производительности. Штраф составляет 5×(2,5 – 2) = 2,5 штрафной единицы.

МАТРИЦА ИГРЫ					
		Стратегии клиентов			P
		P 1	P 2	P3	P4
Стратегия СРВ		0,5	1,25	2,5	4,5
R1	1	2	1,25	7,5	17,5
R2	2	6	3	2,5	12,5
R3	3	10	7	2	7,5
R4	4	14	11	6	2,5
R5	5	18	15	10	2

Рис. 5.1. Матрица игры

Для совокупности стратегий {R4, P2} клиенту нужно 1,25 процессора, система предоставляет 4 процессора. В этом случае имеет место избыточная производительность, штраф за которую составляет 4×(4 – 1,25) = 11 штрафных единиц. Решение задачи показано на рис. 4.37.

Как видно из решения (см. рис. 5.2), цена игры в данном примере равна 7,5 штрафной единицы. Решение определяет использование стратегий R_2, R_3 и R_4 с вероятностями, соответственно равными 0,36451; 0,27097 и 0,36451. Это позволяет считать целесообразным выбор числа процессоров СРВ из соотношения

$$N = n(R_2) \cdot p_2 + n(R_3) \cdot p_3 + n(R_4) \cdot p_4 = \\ = 2 \cdot 0{,}36451 + 3 \cdot 0{,}27097 + 4 \cdot 0{,}36451 \approx 3.$$

За дефицит производительности	5				
За избыточную производительность	3				
	Переменные		Вероятности		Цена игры V
	x1	0	p1	0	7,5
	x2	0,048602	p2	0,36451	
	x3	0,036129	p3	0,27097	
	x4	0,048602	p4	0,36451	
	x5	0	p5	0	
Целевая функция		0,133333		1	

Ограничения		
1	<=	1
0,7	<=	1
0,394408	<=	1
1	<=	1

Рис. 5.2. Решение задачи

Использование представленной модели предполагает создание системы реального времени в форме совокупности web-серверов, вычислительная мощность которых формируется на основе арендованной инфраструктуры IaaS. Арендатор создает собственный парк виртуальных серверов (виртуальный дата-центр) и разворачивает коммерческую систему для обслуживания пользователей, получая определенную прибыль. Предложенная игровая модель позволяет сбалансировать расходы на аренду и прибыль, получаемую от обслуживания пользователей. Для решения этой задачи большое значение имеет установление значений коэффициентов K_1 и K_2. Наблюдение за работой системы позволяет решить задачу. Что касается значений множеств D_i, t_i, зависящих от задач пользователей, то встроенные средства мониторинга вычислительного процесса, имеющиеся в операционных системах, позволяют найти простое решение.

С использованием средств мониторинга и установленных значений коэффициентов K_1 и K_2 определяется текущее значение штрафа (цены игры) Sэ, характерного для эффективной работы арендуемой системы. В процессе функционирования в зависимости от сложившейся ситуации (интенсивности потока запросов) средства мониторинга СРВ получают текущее значение штрафа, которое может отличаться от Sэ. В этом случае необходимо уточнить множество возможных стратегий $P = \{P_1, P_2, \dots, P_k\}$ и $R = \{R_1, R_2, \dots, R_n\}$ и вновь решить задачу определения требуемой производительности СРВ. В идеале можно построить адаптивную СРВ на основе средств мониторинга и предложенной игровой задачи.

Литература к гл. 5

1. Облачные сервисы 2013. Cnews-аналитика. – . URL: http://www.cnews.ru/reviews/new/oblachnye_ servisy_2013/
2. Батаев А.В. Анализ использования облачных сервисов в банковском секторе // Молодой ученый. 2015. № 5. С. 234–240. URL: https://moluch.ru/archive/85/15818/
3. Батаев А.В. Перспективы внедрения облачных технологий в банковском секторе России // Научно-технические ведомости СПбГПУ. Экономические науки № 2 (192). 2014. С. 156–164.
4. Монахов Д.Н., Монахов Н.В., Прончев Г.Б., Кузьменков Д.А. Облачные технологии. – М.: Издательство МГУ им. М.В. Ломоносова, 2013.
5. Риз Дж. Облачные вычисления. – БХВ-Петербург, 2011. 288 с.

6. Гордюшин А.В., Лебедева С.В. Облачные технологии: технология создания «облака» // Вестник молодых ученых Санкт-Петербургского государственного университета технологии и дизайна. 2014. № 3. С. 53–57.

7. Романова И. Облачные технологии и их применение // Молодой ученый. 2016. № 17.1. С. 109–112. [Электронный ресурс]. – Режим доступа: https://moluch.ru/archive/121/33593/

8. Макаров Д.В., Романчук В.А. Облачные SaaS, IaaS, PaaS системы для искусственного интеллекта // Современная техника и технологии. 2015. № 5 [Электронный ресурс]. – Режим доступа:: http://technology.snauka.ru/2015/05/6731

9. Круликовский А.П., Тупота Е.С. Инструментарий для управления “облачными” технологиями // В сб.: Актуальные проблемы и перспективы развития экономики Труды Юбилейной XV международной научно-практической конференции. Крымский федеральный университет им. В. И. Вернадского. 2016. С. 218–219.

10. Кондратьев А.А., Тищенко И.П., Фраленко В.П. Разработка распределенной системы защиты облачных вычислений // Программные системы: теория и приложения. 2011. № 4(8). С. 61–70.

11. Гатиятуллин Т.Р., Сухова А.Р. Проблемы безопасности в облачных технологиях // Проблемы развития современной науки // Сб. ст. Международной научно-практической конференции / Отв. ред. Сукиасян А.А.. 2015. С. 44–46.

12. Enrico Bini, Giorgio C. Buttazzo and Giuseppe M. Buttazzo. Rate Monotonic Analysis: the Hyperbolic Bound // IEEE Transactions on Computers. 2003 52: 933–942.

13. Liu C.L., Layland J.W. Scheduling Algorithms foR Multiprogramming in a Hard Real-Time Environment. J. ACM, 1973-20, pp. 46–61.

14. Оскар Ланге. Оптимальные решения. – М.: Прогресс, 1967. 286 с.

15. Романьков В.А. Введение в Теорию Игр: уч. пособ. / Романьков В.А. – М.: РГГУ, 2014. 699 с.

16. Захаров А.В. Теория игр в общественных науках: учебник для вузов / А.В. Захаров; Нац. исслед. ун-т "Высшая школа экономики". – М.: Изд. дом Высшей школы экономики, 2015. - 304 с.

Глава 6. ПРОГРАММНЫЕ СИСТЕМЫ МЯГКОГО РЕАЛЬНОГО ВРЕМЕНИ

6.1. Области применения высокопроизводительных систем мягкого реального времени

Во всем мире непрерывно возрастает интерес к высокопроизводительным вычислениям. Он обусловлен как потребностями различных отраслей науки, образования, медицины, инженерной и другой практической деятельности человечества, и наконец, просто жизни человека. С появлением, суперкомпьютеров стало реальностью решение многих научных проблем в области математики, физики, механики, астрофизики, биоинформатики, науки о Земле и мировом океане, обучение искусственного интеллекта, распознавание изображений и многое другое. Наконец, суперкомпьютер – сам продукт развития информатики, вычислительной техники и программирования, позволяет решать задачи своего совершенствования – создание перспективных архитектур квантовых и биокомпьютеров, технологий, методов и языков программирования.

Данный раздел посвящен вопросам оптимизация вычислительного процесса при выполнении информационно-связанных задач в суперкомпьютерах. Даже при высокой производительности суперкомпьютеров, предназначенные для них задачи, могут занимать значительное время и выполняться, как правило, неоднократно. Программы, реализующие эти задачи, имеют сложную многомодульную структуру с информационными и управляющими связями между ними. Поэтому они реализуются в пакетном режиме и часто в ночное время. В пакеты могут входить различные задачи, требующие высокой вычислительной мощности и большой основной и внешней памяти.

Желающих использовать суперкомпьютеры всегда много. В связи с этим даже в рамках мощнейших суперкомпьютеров требуется оптимизация вычислительного процесса с целью рационального расходования вычислительных ресурсов при одновременном требовании выполнения определенного набора задач (пакета) в заданные временные ограничения (не в жесткие), т.е. во многих случаях суперкомпьютеры работают в режиме мягкого реального времени. Организация вычислительного процесса суперкомпьютеров должна учитывать особенности их архитектуры. В этом разделе дается обзор возможных архитектур, но предлагается решение оптимизации вычислительного процесса для серверов и мэйнфреймов IBM на процессорах Power9 и z14 применительно к операционным системам, которые используются на этих компьютерах.

Решение предлагается последовательностью связанных задач. В начале представляется структура приложения и дается постановка задачи. Решение начинается с определения величины критического пути и резервов времени по отдельным вычислительным работам. Далее рассматриваются возможности минимизация длины критического пути. После чего ставится и решается задача минимизация количества ресурсов выполнения приложения без увеличения длины критического пути. В заключение предлагается временная диаграмма вычислительного процесса с фиксацией итого результата занятости вычислительных ресурсов.

Как правило, на суперкомпьютерах решаются задачи научного характера из самых различных областей науки, техники и бизнеса, сложнейшие математические и физические задачи (например, квантовой теории гравитации, размерности пространства-времени, поиска простых чисел в шифровании, обработки данных результатов сложных экспериментов, например, на большом адронном коллайдере (эксперимент ATLAS [1, 2]), задачи предсказания погоды, климата и глобальных изменений в атмосфере, задачи генетики человека, моделирования мирового океана и многие другие. Каждая задача уникальна, специфична и требует концептуального построения модели задачи с характеристиками адекватными оригиналу, учета и выявления условий, при которых эта задача решается, и определения входных параметров. Далее формально построенная модель преобразуется в математическую модель, в которой установленные параметры и характеристики и их взаимосвязи описываются походящим математическим аппаратом с определенной точностью, допущениями и ограничениями.

Выбирается метод решения (иногда разрабатывается новый), и на его основе алгоритм решения. При обосновании метода решения рассматриваются вопросы влияния различных факторов и условий на конечный результат, в том числе на точность вычислений, время решения задачи на компьютере, требуемый объем памяти и др. Алгоритм, как правило, состоит из некоторой совокупности относительно самостоятельных блоков и определенными возможностями последовательности их выполнения. Построить сразу параллельный алгоритм решения задачи непросто, необходимо знание не только теории алгоритмов и программирования, но и нужно быть специалистом в исследуемой проблеме, что не всегда совмещается. Поэтому, если это возможно, следует остановиться на известном последовательном алгоритме, а далее в последовательной программе рассматривать ациклические и циклические участки, применяя для их распараллеливания различные приемы.

Современные технологии и стандарты, например, OpenMP (Open Multi-Processing) – открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB) [3].

Интерфейс OpenMP был определен как стандарт для программирования в модели компьютера с общей памятью. OpenMP реализует параллельные вычисления на основе многопоточности, в которой основной поток создает набор подчиненных потоков и задача распределяется между ними. Разработчик просто добавляет в текст последовательной программы OpenMP-директивы. Программа разбивается на последовательные и параллельные области. Все последовательные области выполняет главный поток, порождаемый при запуске программы, а при входе в параллельную область главный поток порождает дополнительные потоки. Предполагается, что OpenMP-программа без какой-либо модификации должна работать как на многопроцессорных системах, так и на однопроцессорных.

В системах с распределенной памятью на каждом вычислительном узле функционирует собственные копии операционной системы, под управлением которых выполняются независимые программы. Это могут быть как независимые программы, так и параллельные ветви одной программы. В этом случае механизмом взаимодействия между ними является передача сообщений. В 1994 г. был принят стандарт механизма передачи сообщений MPI (Message Passing Interface) [4]. MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Основная цель стандарта – это обеспечение полной независимости приложений, написанных с использованием MPI, от архитектуры многопроцессорной системы. Альтернативный подход предоставляет парадигма параллельной обработки данных, которая реализована в языке высокого уровня HPF. От программиста требуется только задать распределение данных по процессорам, а компилятор автоматически генерирует вызовы функций синхронизации и передачи сообщений. Для распараллеливания циклов используются либо специальные конструкции языка, либо директивы

компилятору, задаваемые в виде псевдокомментариев. Язык HPF реализует идею инкрементального распараллеливания и модель общей памяти на системах с распределенной памятью. Эти два обстоятельства и определяют простоту программирования и соответственно привлекательность этой технологии. Одна и та же программа, без какой-либо модификации, должна эффективно работать как на однопроцессорных системах, так и на многопроцессорных вычислительных системах.

Как правило, задачи, решаемые на суперкомпьютерах даже при их высокой производительности, могут занимать значительное время и выполняться неоднократно. Сами программы, реализующие эти задачи, имеют достаточно сложную структуру из отдельных модулей с информационными и управляющими связями между ними. Поэтому они реализуются в пакетном режиме и часто в ночное время. В пакет могут входить различные задачи, требующие высокой вычислительной мощности и основной и внешней памяти. В связи с этим даже в рамках мощнейших суперкомпьютеров требуется оптимизация вычислительного процесса с целью рационального расходования вычислительных ресурсов при одновременном требовании выполнения определенного набора задач (пакета) в заданные временные ограничения [5 – 7].

Рассмотрим теперь архитектуры суперкомпьютеров, которые могут использоваться для реализации подобных сложных задач, требующих очень высокой производительности.

6.2. Вычислительные средства решения сложных задач

Предоставить ресурсы для решения задач, рассмотренных выше, можно только на основе суперкомпьютеров или сверхмощных серверах. Производительность таких компьютеров измеряется в сотнях и тысячах терафлопов (Тфлопс, FLOPS – Floating point Operations per Second). Первенство в Top500 на 21.06.2017 принадлежит двум китайским компьютерам – Sunway TaihuLight (93 Пфлопс; удерживает первую позицию с июня 2016 г.) и Tianhe-2 (33,8 Пфлопс; с июня 2013 г. по ноябрь 2015 г. был мощнейшей машиной мира) [8]. В России суперкомпьютеры сосредоточены в научных, научно-производственных и учебных заведениях. В рейтинг самых мощных суперкомпьютеров мира Top500 на 2018 год вошли лишь три российские машины против пяти в прошлогоднем ноябрьском рейтинге. Компьютер Lomonosov-2 из МГУ занимает в рейтинге 63-е место (производительность 2,1 Пфлопс), Lomonosov – 227-е место (0,9 Пфлопс), а Polytechnic RSC Tornado из Санкт-Петербургского политехнического университета – 412-е место [9]. Суперкомпьютер

МГУ «Ломоносов», в МГУ проводят обработку сейсмических данных, в результате которой научные группы университета выделили ранее неизвестные месторождения природных ресурсов на Сахалине и в Казахстане.

В МСЦ РАН установлен суперкомпьютер производительностью около 124 Тфлопс. Мощности суперкомпьютера МСЦ РАН на безвозмездной основе предоставляются различным академическим организациям, число пользователей их системы превышает 1000 человек. Основные направления исследований, для которых использовалась вычислительная система МСЦ РАН, велись в области математики, механики, физики, информатики и вычислительной техники, астрономии, химии, науки о Земле, биологии, биофизики и информатики, затрагивались все приоритетные направления модернизации России.

Специалисты ИВМ РАН и Института океанологии им. П.П. Ширшова разработали и запустили на суперкомпьютере МСЦ РАН математическую модель динамики океана, которую применили для исследования внутригодовой изменчивости циркуляции вод и уровня Каспийского моря. С применением модели стало возможным доказать существование подповерхностных струйных течений вдоль восточного берега Среднего Каспия и правильно интерпретировать данные наблюдений. Сейчас перед специалистами стоит задача создать модель Мирового океана с пространственным разрешением, лучшим, чем было использовано в модели Каспийского моря.

Два суперкомпьютера имеется в Южно-уральском государственном университете (ЮУрГУ) [10]. Пиковая производительность этих суперкомпьютеров составляет 117,6 и 12,3 Тфлопс. Распределение задач по приоритетным направлениям науки на их суперкомпьютерных ресурсах выглядит следующим образом: 52,2% задач приходится на информационные технологии, 33,7% – на энергоэффективность и энергосбережение, 9,4% – на космические технологии, 3,5% – на медицинские технологии и 1,2% – на ядерные технологии. Если же брать распределение задач по отраслям, то естественно-научные задачи составляют 65% от общего потока, инженерные – 33%, социально-экономические – 2%.

Согласно данным 2018 года редакции списка Топ-50 самых мощных компьютеров СНГ, представленной НИВЦ МГУ имени М.В.Ломоносова и МСЦ РАН в рамках Международной научной конференции «Параллельные вычислительные технологии (ПаВТ) 2018», наметился заметный рост производительности входящих в список систем. Суммарная производительность систем из списка Топ-50 на тесте Linpack за

полгода выросла с 8,7 ПФЛОПС (квадриллионов (1015) операций с плавающей точкой в секунду) до 10,7 Пфлопс. Суммарная пиковая производительность систем списка при этом достигла 17,4 ПфлопсС, хотя еще полгода назад в предыдущей редакции списка она составляла только 13,4 Пфлопс [11].

В 2020 году лидером списка Топ-50 в России является суперкомпьютер «Скиф МГУ». В списке самых мощных компьютеров в странах СНГ Top50 остается суперкомпьютер «Lomonosov-2» производства компании «Т-Платформы» в МГУ им. М.В.Ломоносова. После обновления его производительность на тесте Linpack возросла с 2,1 Пфлопс до 2,48 ПФЛОПС, пиковая производительность возросла до 4,95 Пфлопс [11]. По-прежнему работает суперкомпьютер производства «Т-Платформы» и CRAY в Главном вычислительном центре Федеральной службы по гидрометеорологии и мониторингу окружающей среды. Его производительность на тесте Linpack составляет 1,29 Пфлопс. Поднялся в списке Топ-50 суперкомпьютер в НИЦ «Курчатовский Институт», созданный в результате объединения двух ранее установленных в институте систем. Его производительность на тесте Linpack составила 755,53 Тфлопс [12].

Сравнение наиболее мощных суперкомпьютеров России и СНГ с мировыми лидерами в последнее время не в пользу отечественных систем. Так, например, для того, чтобы попасть в Топ-500 наиболее мощных суперкомпьютеров планеты, система должна иметь пиковую мощность не ниже 700 Тфлопс. Судя по последней редакции российского рейтинга Топ-50, таких систем в России всего пять, хотя полгода назад было всего три.

6.3. Архитектуры суперкомпьютеров

На сегодняшний день суперкомпьютеры являются уникальными системами, создаваемыми "традиционными лидерами" компьютерного рынка, такими как IBM, Hewlett–Packard, NEC и другими, которые приобрели множество компаний, вместе с их опытом и технологиями. Компания Cray Inc. по-прежнему занимает достойное место в ряду производителей суперкомпьютерной техники [13].

Существует две основные архитектуры современных суперкомпьютеров – системы с общей памятью и кластеры. Каждый подход не исключает другого и имеет свои достоинства и недостатки. Достоинство систем с общей памятью – универсальность модели параллельного ПО, не требующей какого-либо дополнительного кода, или значительных изменений кода. Плюс кластерных систем – отказоустойчивость и лучшая масштабируемость. Системы с общей памятью SMP (общая память для

всех процессоров) и NUMA (у каждого процессора, кроме доступа к общей памяти, есть локальная память) плохо масштабируются при росте числа процессоров. Кластеры масштабируются плохо из-за возрастающей сложности сети при добавлении узлов, но это происходит, когда число процессоров измеряется сотнями и даже тысячами [14 – 16].

С давних времен сложилось так что Intel продвигает SMP системы на рынке, а AMD NUMA. В случае Intel-а связь между процессорами осуществляется на основе QPI (QuickPath Interconnect), соответственно для AMD это HyperTransport. Конец 1980-х и начало 1990-х годов охарактеризовались сменой магистрального направления развития суперкомпьютеров от векторно-конвейерной обработки данных к большому и сверхбольшому числу параллельно соединенных скалярных процессоров. Массивно-параллельные системы стали объединять в себе сотни и даже тысячи отдельных процессорных элементов, причем ими могли служить не только специально разработанные, но и общеизвестные и доступные в свободной продаже процессоры. Большинство массивно-параллельных компьютеров создавалось на основе мощных процессоров с архитектурой RISC (Reduced Instruction Set Computer), наподобие Power PC или PA-RISC [17].

Использование серийных микропроцессоров позволило не только гибко менять мощность установки в зависимости от потребностей и возможностей, но и значительно удешевить производство. Примерами суперкомпьютеров этого класса могут служить Intel Paragon, IBM SP, Cray T3D/T3E и ряд других. В ноябре 2002 года фирма Cray Inc. анонсировала решение Cray X1 с производительностью 52,4 Тфлопс. В это же время был опубликован список Top 500 [15], в который входили вычислительные системы, официально показавшие максимальную производительность. Его возглавила "Компьютерная модель Земли" (Earth Simulator) с результатом 35,86 Тфлопс (5120 процессоров), созданная одноименным японским центром и NEC [13].

В настоящее время развивается технология построения больших и суперкомпьютеров на базе кластерных решений. По мнению многих специалистов, на смену отдельным, независимым суперкомпьютерам должны прийти группы высокопроизводительных серверов, объединяемых в кластер. Удобство построения кластерных ВС заключается в том, что можно гибко регулировать необходимую производительность системы, подключая к кластеру с помощью специальных аппаратных и программных интерфейсов обычные серийные серверы до тех пор, пока не будет получен суперкомпьютер требуемой мощности. Кластеризация

позволяет манипулировать группой серверов как одной системой, упрощая управление и повышая надежность. Со списком ведущих производителей серверов и кластерных систем можно познакомиться на сайте [16].

Одной из ведущих компаний мира, имеющей, пожалуй, самую мощную в отрасли исследовательскую команду, способную решать многогранные внедренческие задачи практически любого уровня сложности, на протяжении более столетней деятельности является IBM. Вполне очевидно, что реальной физической базой кластерных систем высокой готовности от IBM может выступать практически любой сервер компании. Это простые и недорогие стандартные машины из серии eServer xSeries на базе 32-разрядной архитектуры x86 (в том числе с использованием 64-разрядных расширений EM64T и AMD64) и 64-разрядной архитектуры Itanium, производительные серверы eServer iSeries и eServer pSeries на базе архитектуры POWER5 и мощные мэйнфреймы eServer zSeries [17, 18].

Разработчики IBM выпустили RISC-процессор Power9, претендующий на роль ведущего обработчика больших данных [19]. Следует заметить, что SPARC64 и архитектура Power – единственные сегодня действующие представители RISC-архитектуры высокой производительности.

Power9 производят по 14-нанометровой технологии Fin-FET (транзисторы с трехмерными затворами), а процессор содержит 8 млрд транзисторов на площади 695 кв. мм [20]), что чуть больше, чем в Power8. Структура конвейеров разделяется на фронтальный компонент (внешний, до начала выполнения команд отвечающий, например, за диспетчирование) и компонент блоков выполнения (EU). Усовершенствованы были оба компонента. Процессорное ядро Power9 типа SMT4 (аппаратная поддержка четырех программных нитей, Simultaneous MultiThreading 4) основывается на двух SS и содержит еще и традиционные блоки. Но Power9 может базироваться и на ядрах других типов – SMT8, которые вдвое больше по размерам, чем SMT4, и включают четыре 128-разрядных SS плюс те же увеличенного размера традиционные блоки. Общее число процессорных ядер SMT4 в микросхеме Power9 равно 24, а в варианте с SMT8 – 12. SMT-ядра поддерживают внеочередное выполнение команд; SMT4 обеспечивает возможность завершения до 128 команд на каждом такте, а SMT8 – до 256 команд.

17 августа 2020 года IBM представила процессор IBM POWER10. Он выполнен с использованием 7-нм техпроцесса и производственных мощностей полупроводниковой фабрики Samsung. Производитель заяв-

ляет, что новинка имеет трехкратное преимущество в энергоэффективности по сравнению с POWER9. Процессор IBM POWER10 поддерживает полное шифрование содержимого оперативной памяти и имеет специальные аппаратные блоки для ускорения криптографических процедур. IBM уточнила, что у POWER10 зафиксирован четырехкратный прирост по скорости шифрования AES на ядро по сравнению с POWER9. Также в POWER10 поддерживается защита и изоляция контейнеров на аппаратном уровне.

17 июля 2017 года IBM анонсировала z14 – это микропроцессор , созданный для мэйнфреймов IBM Z , IBM заявила, что это самый быстрый в мире микропроцессор с тактовой частотой 5,2 ГГц, [2] с повышением производительности на 10% на ядро и на 30% для всего чипа по сравнению с его предшественником z13 . Десяти ядерный процессор может обрабатывать 40 потоков. Микропроцессор z14 поддерживает повышенную эффективность виртуализации ядер Linux и более высокую пропускную способность встроенных процессоров z-Integrated Information Processor (zIIP). Последний представляет собой специализированный процессор, работающий асинхронно с основным процессором мейнфрейма, который предназначен для повышения эффективности использования вычислительных ресурсов.

Поскольку кластерная архитектура суперкомпьютеров в настоящее время является преобладающей, а основными ее компонентами (вычислительными узлами), судя по публикациям, во многих случаях являются серверы и мэйнфреймы IBM на процессорах Power9 и z14, вопросы оптимизация вычислительного процесса при выполнении информационно-связанных задач в суперкомпьютерах будем рассматривать применительно к операционным системам, которые используются на этих компьютерах.

6.4. Операционные системы суперкомпьютеров

В качестве базовых ОС в каждой серверной серии IBM используются как свои системы, так и ОС сторонних производителей -- z/OS и варианты Linux для zSeries; i5/OS, Linux и AIX 5L для iSeries; AIX 5L и Linux для pSeries; Microsoft Windows Server, Novell Netware и Linux для xSeries, оптимизированные для построения кластеров высокой готовности. Необходимо особо отметить, что IBM как многопрофильная технологическая компания, выпускающая все без исключения виды аппаратных ресурсов, применяемых для построения кластерных систем высокой готовности, ориентируется на собственные изделия, лишь изредка предлагая список совместимого оборудования от сторонних поставщи-

ков. Система кластеризации Parallel Sysplex для z/OS позволяет объединять в рамках одного решения до 32 узлов, обеспечивая при этом разделение ресурсов и параллельное исполнение задач. Используемые в ней технические решения гарантируют практически линейный рост производительности при добавлении в кластер новых серверов.

Кроме того, ее расширение Geographically Dispersed Parallel Sysplex (GDPS) [21] с помощью технологии HyperSwap позволяет строить территориально разнесенные, катастрофоустойчивые системы. В качестве промежуточного кластеризующего ПО на системах iSeries предлагается использовать продукты, поставляемые технологическими партнерами IBM. Основное рекомендуемое решение - MIMIX High Availability Cluster от фирмы Lakeview Technology. Модульность и широкие возможности мониторинга и автоматизации процессов репликации данных и системных объектов, переключения нагрузки между производственной и резервными системами позволяют создавать решения, оптимальные с точки зрения требований бизнес-пользователей.

Система кластеризации High Availability Cluster Multi-Processing версии 5.2, используемая на серверах pSeries с установленной AIX 5L, позволяет организовать на базе ПО Oracle Application Server 10g как системы с разделением ресурсов (для чего используется технология Oracle Real Application Cluster), так и системы активного дублирования без разделения ресурсов – Cold Failover Cluster, использующие один из узлов в режиме горячего резерва на случай сбоя в основном узле.

В проектах создания кластерных систем высокой готовности на базе серверов стандартной архитектуры из семейства xSeries IBM полагается на бизнес-партнеров и их программные разработки - в системе Windows предлагается использовать сервисы Microsoft Cluster Services, для Netware применяются Novell Cluster Services, а для Linux используется пакет LifeKeeper компании SteelEye и собственный продукт IBM – Tivoli System Automation, работающий в среде разных версий Linux не только на xSeries, но и на всех аппаратных платформах IBM. Этот продукт позволяет объединять механизмы высокой доступности, уже имеющиеся в программных комплексах более высокого уровня, а также организовывать кластерные системы активного дублирования для тех продуктов, в которых такой поддержки нет.

Современные технологии виртуализации, используемые в серверах и мэйнфреймах IBM, начинают свое развитие от разработки в IBM гипервизора, который был использован для работы виртуальных машин в операционной системе VM/370. В более ранних системах IBM для реализации мультипрограммирования разработчики использовали технологию

разделов. В дальнейшем эта технология была использована в виртуальных средах (виртуальных серверах) на базе гипервизора. Корпорация IBM существенно расширила технологию разделов, разработав так называемые динамические LPAR (DLPAR), позволяющие перемещать ресурсы между разделами без перезагрузки системы или самих разделов.

Физический сервер может содержать несколько изолированных друг от друга виртуальных серверов, работающих одновременно, и разделяющих ресурсы. Каждый логический (виртуальный сервер) выполняется в логическом разделе LPAR (Logical Partition Access Resources), в который устанавливается гостевая операционная система. Максимальное количество LPAR на сервере зависит от количества процессоров и памяти: на один процессор физического сервера можно создать максимум 10 LPAR, выделив на каждый LPAR минимальную долю в 0,1 часть ресурсов процессора и минимум по 256 Мб оперативной памяти. Теоретическое ограничение тоже есть: например, для Power 795 модели можно создать максимум 254 LPAR. Таким образом можно оптимизировать нагрузку на серверы, поделив процессорный пул между виртуальными серверами, например, какому-то LPAR дать 0,5 процессора, а более нагруженному – 1,2.

Существует большое количество операционных систем совместимых с LPARs, включая z/OS, z/VM, z/VSE, z/TPF, AIX, Linux и i/OS. На универсальных компьютерах типа IBM, LPAR управляет PR/CM. Универсальные компьютеры типа IBM работают исключительно в режиме LPAR, даже когда только один раздел на машине. Всеми LPAR управляет гипервизор. Для управления LPAR применяются следующие гипервизоры: для zSeries - PR/SM (Processor Resource/System Manager); для IBM Power Systems - Power Hypervisor (PowerVM). Для взаимодействия между LPAR внутри одного физического сервера используется виртуальная сеть – HiperSocket. PR/SM (Processor Resource/System Manager) представляет собой гипервизор 1-го типа (монитор виртуальных машин), который позволяет использовать несколько логических разделов для разделения физических ресурсов, таких как процессоры, каналы ввода-вывода и прямых запоминающих устройств доступа (DASD). Работает на основе компонента CP VM/XA и непосредственно на уровне машины, и выделяет системные ресурсы через LPAR'ы, чтобы совместно использовать физические ресурсы. Это стандартная функция на Системе z IBM, Системе p и i машин [27].

6.5. Постановка, формализация и решение задачи

6.5.1. Представление структуры приложения и постановка задачи

Выполнение приложения производится на суперкомпьютере IBM типа Power или z. Известна структура приложения, состоящего из некоторого множества информационно-связанных частей (задач) с известным (ожидаемым) временем выполнения каждой задачи. Будем считать, что каждая задача выполняется отельным логическим (виртуальным сервером) в логическом разделе LPAR, в который устанавливается гостевая операционная система. Предполагается, что это время определяется элементарным вычислителем (процессором) многопроцессорного компьютера. Взаимодействие LPAR внутри физического сервера обеспечивается общей оперативной памятью или использованием виртуальной сети – HiperSocket [23]. Структуру подлежащего выполнению приложения удобно представить взвешенным ориентированным графом, как это показано на рис. 1, который будем далее использовать для иллюстрации решения поставленной задачи:

$$G = \{< s_i, s_j >, t_{ij} \big| j > i, \;\; i = 0, 1, \dots, M-2, \;\; j = 1,2, \dots, M-1\},$$

где s_i, s_j – номера событий (вершины) в графе, t_{ij} – время решения задачи (вес соответствующей дуги), M – количество задач в пакете G. Считаем, что временные характеристики приложения известны по результатам его разработки и определены из условия выполнения каждой задачи приложения на одном процессоре (в одном LPAR) суперкомпьютера, выбранного для реализации этого приложения.

Для дальнейшей формализации задачи будем использовать понятия сетевого планирования и управления. В нашем случае работы представляются дугами графа $< s_i, s_j >$, или просто (i, j), причем для любой дуги $j > i$. Обмен информацией между задачами инициируется событиями, например, событие s_1 инициирует завершение работы $< s_0, s_1 >$, длительностью t_{01}, возможность запуска вычислений $< s_1, s_4 >, < s_1, s_6 >$, и $< s_1, s_{31} >$. Организация вычислительного процесса при выполнении данного приложения сводится к определению временных параметров сетевого графика и к его оптимизации по длительности выполнения всего комплекса задач и затратам вычислительных ресурсов в соответствии с заданными требованиями. Следует иметь ввиду, что, как правило, на суперкомпьютере выполняется в пакетном режиме несколько различных приложений, и вычислительные ресурсы для реализации каждого из них ограничены.

6.5.2. Решение задачи

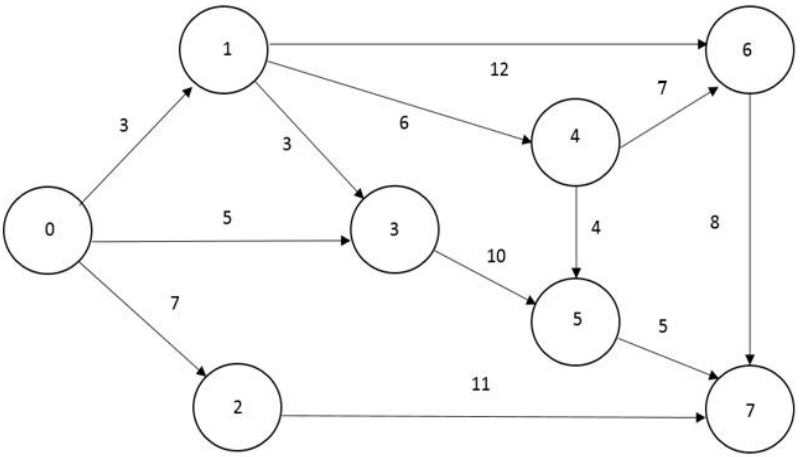


Рис. 6.1. Структура приложения

Для дальнейшей формализации задачи будем использовать понятия сетевого планирования и управления. В нашем случае работы представляются дугами графа $< s_i, s_j >$, или просто (i, j), причем для любой дуги $j > i$. Обмен информацией между задачами инициируется событиями, например, событие s_1 инициирует завершение работы $< s_0, s_1 >$, длительностью t_{01}, возможность запуска вычислений $< s_1, s_4 >, < s_1, s_6 >$, и $< s_1, s_{31} >$. Организация вычислительного процесса при выполнении данного приложения сводится к определению временных параметров сетевого графика и к его оптимизации по длительности выполнения всего комплекса задач и затратам вычислительных ресурсов в соответствии с заданными требованиями. Следует иметь ввиду, что, как правило, на суперкомпьютере выполняется в пакетном режиме несколько различных приложений, и вычислительные ресурсы для реализации каждого из них ограничены.

6.5.2.1. Определение величины критического пути и резервов времени по отдельным вычислительным работам

Критический путь T_{kr} – это полный путь, определяющий длительность всего комплекса вычислительных работ и имеющий наибольшую продолжительность. Для решения поставленной задачи необходимо найти длину критического пути и возможные резервы времени при

выполнении отдельных вычислительных работ. Определение длины критического пути можно проводить различными способами. Наиболее удобным способом расчёта сетевого графика при небольшом количестве работ является расчёт непосредственно на сети графика и заключается в нахождении критического пути и определении резервов времени для работ, которые не располагаются на этом пути.

При производстве расчетов сетевых моделей применяют следующие наименования его параметров [24, 25]:

Продолжительность работы (t_{ij}) (здесь i и j – номера соответственно начального и конечного событий.

Раннее начало работы $t_{ij}^{\text{р.н}}$– характеризуется выполнением всех предшествующих работ и определяется продолжительностью максимального пути от исходного события всей модели до начального события рассматриваемой работы.

Раннее окончание работы $t_{ij}^{\text{р.о}}$ – определяется суммой раннего начала и продолжительности рассматриваемой работы.

Позднее начало работы $t_{ij}^{\text{п.н}}$ – определяется разностью позднего окончания и продолжительности рассматриваемой работы.

Позднее окончание работы $t_{ij}^{\text{п.о}}$ – определяется разностью продолжительности критического пути и максимальной продолжительности пути от завершающего события всей модели до конечного события рассматриваемой работы.

Общий резерв времени работы R_{ij} – характеризуется возможностью роста продолжительности работы без увеличения продолжительности критического пути и определяется как разность между поздним и ранним окончанием рассматриваемой работы. Общий резерв работы принадлежит не только первой работе, но и всем последующим работам данного пути. В случае использования на одной из работ общего резерва критический путь не изменит своей продолжительности, но все последующие работы окажутся критическими и лишатся резерва.

Частный резерв времени работы r_{ij} – характеризуется возможностью увеличения продолжительности работы без изменения раннего начала последующей работы и определяется разностью между ранним началом последующей работы и ранним окончанием рассматриваемой работы. Частный резерв имеет место, когда одним событием заканчивается не менее двух работ. Отличие частного резерва от общего заключается в том, что частный резерв может быть использован только на рассматриваемой или предшествующих работах и не может быть использован на последующих.

Полным резервом некоторого пути в сетевой модели R называют разность между продолжительностью критического пути модели и продолжительностью рассматриваемого пути.

Результаты расчета сетевой модели выполнения пакета задач, представленного на рис. 1, приведены в табл.1. Работы, не имеющие резерва времени и выделенные жирным шрифтом, образуют критический путь T_{kr} = 24.

Таблица 1

Таблица расчетов параметров сетевого графика

Работа (ij)	Продолжительность работы t_{ij}	Ранее начало работы $t_{ij}^{р.н}$	Ранее окончание работы $t_{ij}^{р.о}$	Позднее начало работы $t_{ij}^{п.н}$	Позднее окончание работы $t_{ij}^{п.о}$	Общий резерв работы R_{ij}	Частный резерв работы r_{ij}
(01)	3,00	0,00	3,00	0,00	3,00	**0,00**	0,00
(02)	7,00	0,00	7,00	6,00	13,00	6,00	0,00
(03)	5,00	0,00	5,00	4,00	9,00	4,00	0,00
(13)	3,00	3,00	6,00	6,00	9,00	3,00	0,00
(14)	6,00	3,00	9,00	3,00	9,00	**0,00**	0,00
(16)	12,00	3,00	15,00	4,00	16,00	1,00	0,00
(27)	11,00	7,00	18,00	13,00	24,00	6,00	0,00
(35)	10,00	6,00	16,00	9,00	19,00	3,00	3,00
(45)	4,00	9,00	13,00	15,00	19,00	6,00	0,00
(46)	7,00	9,00	16,00	9,00	16,00	**0,00**	0,00
(57)	5,00	16,00	21,00	19,00	24,00	3,00	3,00
(67)	8,00	16,00	24,00	16,00	24,00	**0,00**	0,00

6.5.2.2. Минимизация длины критического пути

Имеющийся резерв времени по работам, не лежащим на критическом пути, свидетельствует о том, что имеется возможность изъятия ресурсов, выделенных на некоторые работы, с целью добавления ресурсов на работы, лежащие на критическом пути. Это приведет к уменьшению длины критического пути, т.е. к сокращению всего цикла выполняемых вычислений. В нашем примере суммарный резерв времени составляет значение

$$R = \sum_{i=0}^{M-2} \sum_{j=1}^{M-1} R_{ij} = 32.$$

Передача ресурса от какой-либо работы означает увеличение ее выполнения на величину изъятого ресурса, добавление ресурса к другой работе, лежащей на критическом пути, означает уменьшение ее выполнения на величину добавленного ресурса. Например, изъятие половины резерва, т.е. трех единиц ресурса от работы (02) означает ее удлинение до 10 единиц времени. Это означает, что для ее выполнения нужен будет виртуальный процессор со следующим значением доли физического процессора

$$V_{02} = \frac{t_{02}}{t_{02} + 0{,}5 * R_{02}} = \frac{7}{7 + 0{,}5 * 6} = 0{,}7.$$

Изъятый ресурс можно добавить с целью сокращения критического пути, например, к работе (67). При этом время выполнения этой работы уменьшится с 8 до 5 единиц времени, но для выполнения этой работы потребуется виртуальный процессор со следующим значением доли физического процессора

$$V_{67} = \frac{t_{67}}{t_{67} - 0{,}5 * R_{02}} = \frac{8}{8 - 0{,}5 * 6} = 1{,}6.$$

Для решения задачи минимизации критического пути необходимо построить модель линейного программирования с целью определения значений T_{kr} для различных вариантов перераспределения ресурсов. Удобно это сделать в электронных таблицах, например, Excel. Воспользуемся для этого формализацией задачи поиска критического пути, предложенной в [26]. Исходные данные удобно представить в форме таблицы (табл. 2).

Таблица 2. Исходные данные для решения задачи поиска критического пути

	Переменные	x_{01}	x_{02}	x_{03}	x_{13}	x_{14}	x_{16}	x_{27}	x_{35}	x_{45}	x_{46}	x_{57}	x_{67}	Огранич. функция	Ограниче-ние
	Значения переменных	1	1	1	1	1	1	1	1	1	1	1	1		
Номера узлов	0	1	1	1										3	1
	1	-1			1	1	1							2	0
	2		-1					1						0	0
	3			-1	-1				1					-1	0
	4					-1				1	1			1	0
	5								-1	-1		1		-1	0
	6						-1				-1		1	-1	0
	t_{ij}	3	7	5	3	6	12	11	10	4	7	5	8		

Задача поиска критического пути на основе табл. 2 заключается в определении значения функции

$$T_{kr} = \sum_{i=0}^{i=6} \sum_{j=1}^{j=7} t_{ij} \cdot x_{ij} \rightarrow max \quad (6.1)$$

при следующих ограничениях:

$$1 \cdot x_{01} + 1 \cdot x_{02} + 1 \cdot x_{03} = 1; \quad (6.2)$$

$$-1 \cdot x_{01} + 1 \cdot x_{13} + 1 \cdot x_{14} + 1 \cdot x_{16} = 0 \quad (6.3)$$

$$-1 \cdot x_{02} + 1 \cdot x_{27} = 0; \quad (6.4)$$

$$-1 \cdot x_{03} - 1 \cdot x_{13} + 1 \cdot x_{35} = 0; \quad (6.5)$$

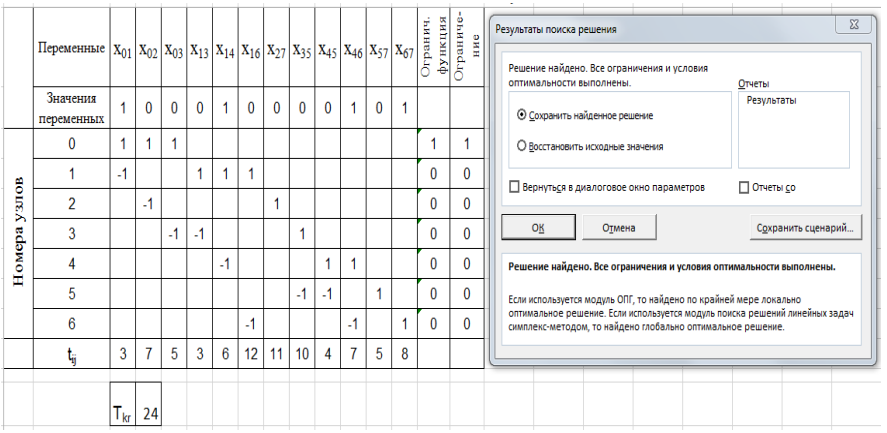
$$-1 \cdot x_{14} + 1 \cdot x_{45} + 1 \cdot x_{46} = 0; \quad (6.6)$$

$$-1 \cdot x_{35} - 1 \cdot x_{45} + 1 \cdot x_{57} = 0; \quad (6.7)$$

$$-1 \cdot x_{16} - 1 \cdot x_{46} + 1 \cdot x_{67} = 0; \quad (6.8)$$

$$\forall x_{ij} \in \{0,1\} | \; i = 0,1, \dots 6; j = 1,2, \dots 7. \quad (6.9)$$

Решение задачи (6.) – (6.9) для исходного графа сетевой модели показано на рис. 6.2.



	Переменные	X_{01}	X_{02}	X_{03}	X_{13}	X_{14}	X_{16}	X_{27}	X_{35}	X_{45}	X_{46}	X_{57}	X_{67}	Огранич. функция	Ограниче-ние
	Значения переменных	1	0	0	0	1	0	0	0	0	1	0	1		
Номера узлов	0	1	1	1										1	1
	1	-1			1	1	1							0	0
	2		-1					1						0	0
	3			-1	-1				1					0	0
	4					-1				1	1			0	0
	5								-1	-1		1		0	0
	6						-1				-1		1	0	0
	t_{ij}	3	7	5	3	6	12	11	10	4	7	5	8		

T_{kr}	24

Результаты поиска решения

Решение найдено. Все ограничения и условия оптимальности выполнены.

Отчеты

Результаты

Сохранить найденное решение

Восстановить исходные значения

Вернуться в диалоговое окно параметров

Отчеты со

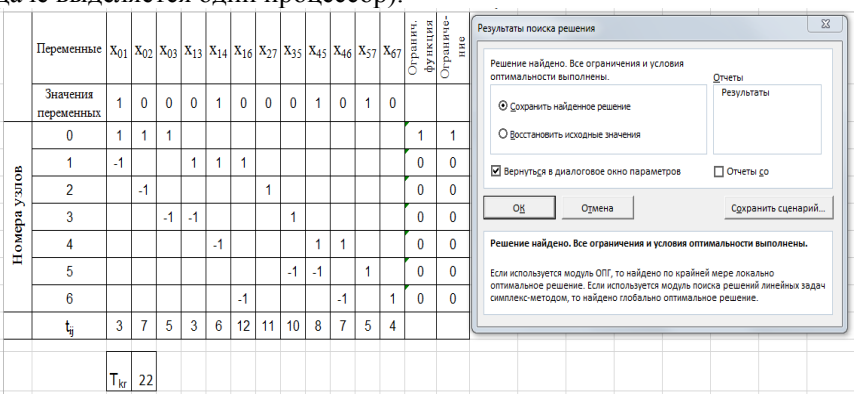
ОК | Отмена | Сохранить сценарий...

Решение найдено. Все ограничения и условия оптимальности выполнены.

Если используется модуль ОПГ, то найдено по крайней мере локально оптимальное решение. Если используется модуль поиска решений линейных задач симплекс-методом, то найдено глобально оптимальное решение.

Рис. 6.2. Критический путь исходного графа сетевой модели

Определим теперь, как изменится величина критического пути, если 4 единицы резерва изъять у работы (45) и передать их для выполнения работе (67). При этом задача (45) будет выполняться 8 единиц времени, а задача (67) – 4 единицы времени. Решение задачи в этом случае показано на рис. 6.3 и дает значение критического пути, равное 22. Если рассчитанное значение критического пути окажется больше директивного, можно рассмотреть другие варианты использования резервов других работ, пока не будет полученное требуемое значение критического пути. Если этого не удается сделать за счет резервов в выполнении работ, необходимо увеличить количество ресурсов для их выполнения (в рассматриваемой задаче мы исходили из предположения, что каждой задаче выделяется один процессор).



	Переменные	X_{01}	X_{02}	X_{03}	X_{13}	X_{14}	X_{16}	X_{27}	X_{35}	X_{45}	X_{46}	X_{57}	X_{67}	Огранич. функция	Ограниче-ние
	Значения переменных	1	0	0	0	1	0	0	0	1	0	1	0		
Номера узлов	0	1	1	1										1	1
	1	-1			1	1	1							0	0
	2		-1					1						0	0
	3			-1	-1				1					0	0
	4					-1				1	1			0	0
	5								-1	-1		1		0	0
	6						-1				-1		1	0	0
	t_{ij}	3	7	5	3	6	12	11	10	8	7	5	4		

T_{kr}	22

Результаты поиска решения

Решение найдено. Все ограничения и условия оптимальности выполнены.

Отчеты

Результаты

Сохранить найденное решение

Восстановить исходные значения

Вернуться в диалоговое окно параметров

Отчеты со

ОК | Отмена | Сохранить сценарий...

Решение найдено. Все ограничения и условия оптимальности выполнены.

Если используется модуль ОПГ, то найдено по крайней мере локально оптимальное решение. Если используется модуль поиска решений линейных задач симплекс-методом, то найдено глобально оптимальное решение.

Рис. 6.3. Вариант расчета критического пути сетевой модели выполнения графа задач при использовании резерва задачи (45)

6.5.2.3. Минимизация количества ресурсов выполнения пакета без увеличения длины критического пути

Будем считать, определенный выше критический путь, равный 24 единицам времени, соответствует директивному времени решения заданного пакета задач. В этом случае становится актуальной задача минимизации ресурсов (в нашем случае количества процессоров), необходимых для реализации всех задач при условии непревышения длины критического пути. Другими словами, длительности выполнения всех работ пакета нужно изменить так, чтобы длина любого пути в графе была в идеале равна длине критического пути. Практически все вычислительные работы будут увеличены с учетом возможны резервов времени для их выполнения.

Для формализации задачи введем дополнительные обозначения:

t_{ij}^{x} – время выполнения работы (ij) после минимизации количества выделяемых ресурсов;

T_i^x – время свершения событий в графе работ (события отождествляются с вершинами графа);

N – исходное число процессоров, которое позволяет выполнить все работы по принципу: один процессор на одну работу;

N_m – минимальное количество процессоров, которое будет получено в результате решения оптимизационной задачи (пока еще без учета возможности их параллельной работы). Значение минимального количества процессоров определяется как сумма долей физических процессоров в виртуальных единичных процессорах логических разделах LPAR, т.е.

$$N_m = \sum_{i=0}^{i=6} \sum_{j=1}^{j=7} d_{ij},$$

где d_{ij} – доля физического процессора соответствующего виртуального процессора LPAR, необходимая для выполнения работы (ij) после минимизации количества выделяемых ресурсов,

$$d_{ij} = 1 - \frac{\left(t_{ij}^{x} - t_{ij}\right)}{t_{ij}^{x}}.$$

В качестве целевой функции задачи выбираем время занятости процессоров (полное машинное время) на решение пакета задач – его нужно минимизировать за счет использования имеющихся резервов времени на выполнение отдельных задач. Таким образом, целевая функция имеет следующий вид:

$$\sum_{i=0}^{i=6}\sum_{j=1}^{j=7} t_{ij} - \sum_{i=0}^{i=6}\sum_{j=1}^{j=7}(t_{ij}^{x} - t_{ij}) \rightarrow min \qquad (6.10)$$

Первое слагаемое этой функции представляет собой полные затраты машинного времени на решение всего пакета задач. Второе – экономию машинного времени при условии того, что можно увеличить время решения некоторых задач за счет имеющегося резерва времени на их выполнение (здесь $(t_{ij}^{x} - t_{ij}) \geq 0$). Рассмотрим ограничения, которые должны учитываться при решении этой задачи.

Первый вид ограничений связан с принятым условием использования для решения задачи только имеющихся резервов для выполнения задач пакета. Отсюда следует система ограничений для продолжительности выполнения работ следующего вида:

$$t_{ij} + R_{ij} \geq t_{ij}^{x} \geq t_{ij}. \qquad (6.11)$$

Второй вид ограничений должен обеспечить такое изменение длительности выполнения всех работ пакета, чтобы длина любого пути в графе была в идеале равна длине критического В рассматриваемом примере таких путей шесть (перечислим их последовательностями вершин графа): P_1 : 0 – 2 – 7; P_2 : 0 – 3 – 5 – 7; P_3 : 0 – 1 – 6 – 7; P_4 : 0 – 1 – 3 – 5 – 7; P_5 : 0 – 1 – 4 – 5 – 7; P_6 : 0 – 1 – 4 – 6 – 7. Ограничения на длину этих путей имеют следующий вид:

$$L_i\,(P_i) \leq T_{kr} | 1 = 1,2, \dots, M-1 \qquad (6.12)$$

Решение задачи (6.10) – (6.13) показано на рис. 6.4 и 6.5.

Операция	Продолжительность работы t_{ij}	R_{ij}	$t_{ij} + R_{ij}$	t^x_{ij}	$t^x_{ij} - t_{ij}$	Исходное число процессоров N	Минимальное число процессоров Nm	Путь	Сумма работ по пути
(01)	3	0	3	3	0	1	1,00	0-2-7	24
(02)	7	6	13	13	6	1	0,54	0-3-5-7	24
(03)	5	4	9	9	4	1	0,56	0-1-6-7	24
(13)	3	3	6	6	3	1	0,50	0-1-3-5-7	24
(14)	6	0	6	6	0	1	1,00	0-1-4-5-7	24
(16)	12	1	13	13	1	1	0,92	0-1-4-6-7	24
(27)	11	6	17	11	0	1	1,00		
(35)	10	3	13	10	0	1	1,00		
(45)	4	6	10	10	6	1	0,40		
(46)	7	0	7	7	0	1	1,00		
(57)	5	3	8	5	0	1	1,00		
(67)	8	0	8	8	0	1	1,00		
	81			101	20	12	9,92		
	Fцел	61							

Рис. 6.4. Представление задачи (6.10) – (6.12) в электронных таблицах

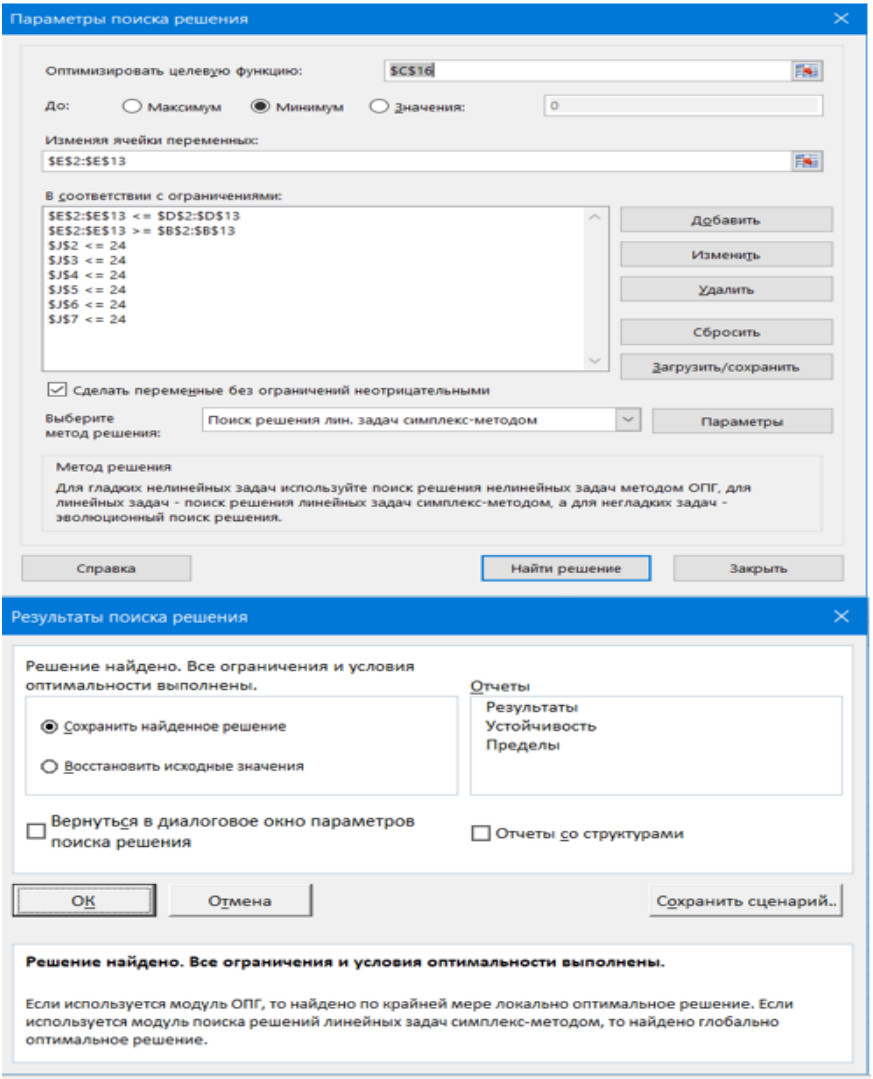


Рис. 6.5. Решение задачи (6.10) – (6.12) в электронных таблицах

Как видно из результата решения задачи минимизации использования ресурсов, число процессоров для реализации пакета задач без увеличения длины критического пути можно сократить с 12 до 10. Однако структура пакета свидетельствует о возможной параллельности работы этих процессоров при выполнении задач пакета.

6.5.3. Возможности организации мультипроцессорного выполнения пакета задач, представленного сетевой моделью

Потенциальную параллельность выполнения заданного набора задач можно определить, преобразовав граф задач в ярусно-параллельную форму [27]. Ярусно-параллельная форма графа (ЯПФ) – деление вершин ориентированного ациклического графа на перенумерованные подмножества V_j такие, что, если дуга идет от вершины $v_l \in V_j$ к вершине $v_m \in V_k$,то обязательно $j < k$.

Каждое из множеств V_j называется ярусом ЯПФ, j – его номером, количество вершин $|V_j|$ в ярусе – его шириной. Количество ярусов в ЯПФ называется её высотой, а максимальная ширина её ярусов – шириной ЯПФ. Для ЯПФ графа алгоритма важным является тот факт, что операции, которым соответствуют вершины одного яруса, не зависят друг от друга (не находятся в отношении связи), и поэтому заведомо существует параллельная реализация алгоритма, в которой они могут быть выполнены параллельно на разных устройствах вычислительной системы. Поэтому ЯПФ графа алгоритма может быть использована для подготовки такой параллельной реализации алгоритма.

Для получения ЯПФ пакета задач, представленного в форме сетевой модели, как показано на рис. 1, его предварительно необходимо преобразовать, заменив дуги графа (работы) вершинами. В преобразованном графе (рис. 6.6) номера вершин соответствуют работам, длительность которых пересчитана в соответствии с решением задачи (6.10 – 6.12) по рис. 6.6.

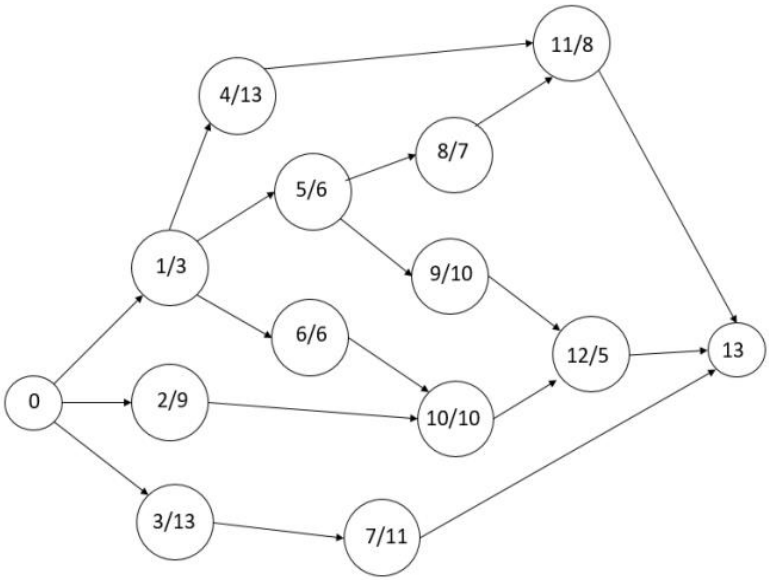


Рис. 6.6. Преобразованный граф пакета задач приложения

Получить ЯПФ можно, построив матрицу смежности графа. Матрица смежности – это квадратная матрица размерностью (M+1) × (M+1), (где M – число вершин графа), однозначно представляющая его структуру. Обозначим ее как $A = |a_{ij}|$, где каждый элемент матрицы определяется следующим образом: aij =1, если есть дуга (i, j), $a_{ij} = 0$, если нет дуги (i, j). В нашем примере матрица смежности будет иметь следующий вид, приведенный на рис. 6.7.

Алгоритм распределения модулей системы по уровням:

1. Находим в матрице нулевые строки. В нашем случае это только одна строка с номером 13.
2. Вершина с этим номером образует нулевой (низший) уровень ЯПФ.
3. Вычеркиваем столбцы с номерами найденных вершин. В нашем случае – столбец 13.
4. Находим в матрице нулевые строки (7, 11, 12). Это вершины 1-го уровня.
5. Вычеркиваем столбцы с номерами 7, 11, 12.
6. Находим в матрице нулевые строки (3, 4, 8, 9 и 10). Это вершины 2-го уровня.
7. Вычеркиваем столбцы с номерами найденных вершин.
8. Находим в матрице нулевые строки (2, 5, 6). Это вершины 3-го уровня.
9. Вычеркиваем столбцы с номерами 5, 6.
10. Вершина с номером 1 образует 4-й уровень.

ЯПФ графа задач пакета, поученная на основе матрицы смежности представлена на рис. 8.

	номер вершины														
номер вершины		0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	3	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	5	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	6	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	8	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	9	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	11	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рис. 6.7. Матрица смежности

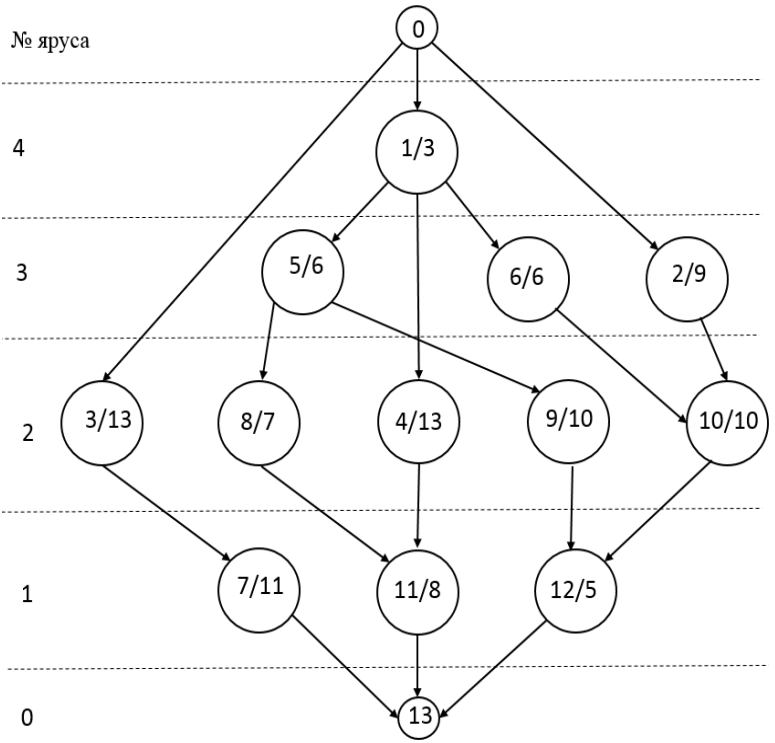


Рис. 6.8. ЯПФ графа пакета задач приложения

Построенный граф задач в ЯПФ далеко не прямоугольный, что неудобно для организации параллельного вычислительного процесса. Визуально из рис. 8 понятно, что граф можно легко перестроить, не меняя связей между вершинами, например, можно переместить вершины 2 и 3 на 4-й уровень, а вершину 4 – на 3-й. После таких преобразований граф примет вид, показанный на рис. 6.9. Здесь у каждой вершины курсивом дано число долей физического процессора, которое необходимо разделу LPAR выполнения задачи в соответствии с решением, минимизирующим количество ресурса процессоров для выполнения всего пакета задач (рис. 6.4).

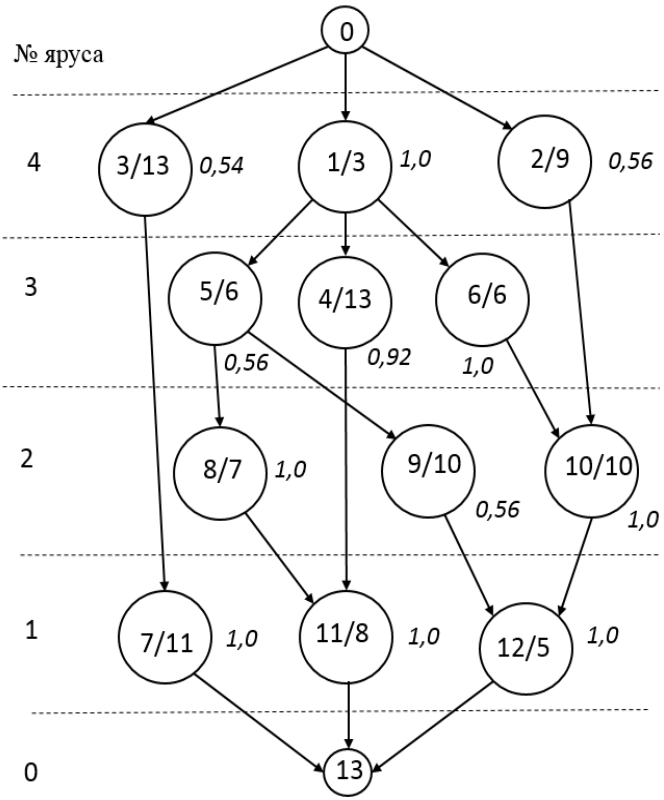


Рис. 6.9. ЯПФ пакета задач приложения

6.5.4. Построение плана вычислительного процесса

По ЯПФ (рис. 6.9) легко построить план реализации вычислительного процесса во времени. На рис. 6.10 сверху показана шкала времени (равная длине критического пути), под которой обозначены временные промежутки реализации задач пакета. В нижней части диаграммы приведены сведения о необходимом количестве логических разделов LPAR и количестве долей физических процессорных модулей (здесь необходимо округление с учетом дискретности выделения 0,1), образующих виртуальные процессоры разделов. Вертикальными стрелками показаны передачи данных между разделами.

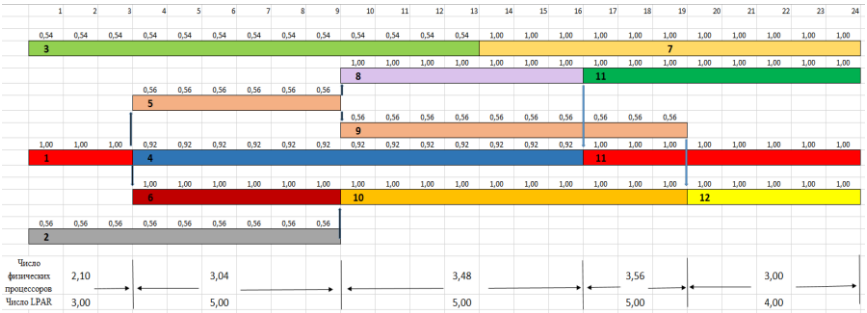


Рис. 6.10. Диаграмма выполнения вычислительных работ пакета

6.6. Обсуждение результатов решения задачи

Рассмотрим на нашем примере результаты использования предложенной последовательности задач оптимизация вычислительного процесса при выполнении информационно-связанных задач в суперкомпьютерах типа IBM типа Power или z. Цель разработанной методики – обеспечить решение заданного пакета задач с ограничением на время выполнения пакета при минимизации занятых ресурсов. Применительно к компьютерам указано типа и операционным системам, используемым на этих компьютерах, итогом решения задачи является определение числа логических разделов LPAR (по сути – виртуальных машин) и выделение каждому разделу виртуального процессора необходимой доли физического процессора, которую можно назначать с дискретностью 0,1.

Первым шагом является определение длины критического пути, которое задает минимально возможное время реализации задач пакета и возможных резервов времени в выполнении отдельных задач пакета. Исходными данными для решения на этом этапе являются результаты выполнения каждой задачи (после ее отладки) на одном процессоре компьютера (см. п. 6.2.1). Если длина критического пути превышает желаемое время реализации пакета, задачам, лежащим на критическом пути, выделяется большая вычислительная мощность за счет задач, имеющих резерв времени выполнения (см. п. 6.2.2). В нашем примере принято решение оставить первоначально определенный критический путь, равный 24 единицам процессорного времени. Следующий шаг – минимизация количества ресурсов выполнения пакета без увеличения длины критического пути (см. п. 6.2.3). В результате его выполнения учтены резервы времени, имеющиеся у работ, не лежащих на критическом пути, и суммарное количество процессоров, необходимое для выполнения

всех работ в расчете на их раздельное выполнение в однопрограммном режиме сокращено с 12 до 9,92 (см. рис. 6.4). Наконец последний шаг – учет возможности организации мультипроцессорного выполнения пакета задач, представленного сетевой моделью (см. п. 3.2.4). Исходный граф пакета задач представляется в ЯПФ и определяется динамика изменения количества LPAR (3 – 5 – 5 – 5 – 4) и числа долей физических процессоров (2,1 – 3,1 – 3,5 – 3,6 – 3,0) при реализации пакта задач в мультипроцессорном режиме и временная диаграмма выполнения вычислительного процесса (см. рис. 6.10).

Перспективной архитектурой суперкомпьютеров для решения многих сложных задач, требующих больших вычислительных мощностей, следует считать серверные компьютеры (мэйнфреймы) и кластерные системы на базе процессоров высокой производительности типа Power9 и z14, поскольку они обеспечивают гибкое создание и управление динамическими разделами (виртуальными машинами) при широкой номенклатуре гипервизоров и гостевых операционных систем. Наилучшим применением мета-компьютеров является решение таких задач, в которых вычислительные узлы практически не взаимодействуют друг с другом и основную часть работы производят в автономном режиме.

Актуальной задачей функционирования мэйнфреймов и кластерных систем является оптимизация вычислительного процесса при выполнении информационно-связанных задач в суперкомпьютерах. Это позволяет обеспечить одновременное решение сложных пакетов информационно-связанных задач для большого количества пользователей подобных систем.

Для решения пакетов информационно-связанных задач на мэйнфреймах и кластерах их необходимо предварительно подготовить таким образом, чтобы обеспечить такой вычислительный процесс, при котором каждый пакет выполняется с требуемыми временными ограничениями при минимально необходимых вычислительных ресурсах. Такую подготовку задач можно провести, используя методы сетевого планирования и управления и методы организации мультипроцессорного планирования и управления.

Литература к гл. 6

1. Большой_адронный_коллайдер. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/
2. Глянцев А. Большой адронный коллайдер достиг рекордной светимости. [Электронный ресурс]. URL: http://www.vesti.ru/doc.html?id=2954624

3. Спецификация API OpenMP для параллельного программирования. [Электронный ресурс]. URL: https://www.openmp.org/
4. Функции передачи сообщений. [Электронный ресурс]. URL: http://www.opennet.ru/docs/RUS/linux_parallel/node142.html.
5. Polychronopoulos C.D. and Kuck D.J. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers IEEE TRANSACTIONS ON COMPUTERS, VOL. C-36, NO 12, pp. 1425–1439 (Dec 1987).
6. Smallen, S., Crine, W., Frey, J., Berman, F., Wolski, R., Su M.H., Kesselman, C., Young, S., Ellisman, M.: Combining workstations and supercomputers to support grid applications: the parallel tomography experience. In: Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556). pp. 241–252 (2000).
7. Scogland, T.R.W., Feng, W.C.: Runtime adaptation for autonomic heterogeneous computing. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 562–565 (May 2014).
8. Среди 500 самых мощных суперкомпьютеров мира осталось лишь три российских
[Электронный ресурс]. URL: https://www.vedomosti.ru/technology/articles/2017/06/21/695300-superkompyuterov-rossiiskih
9. Россия вошла в мировой топ-500 суперкомпьютеров лишь с 3 моделями. [Электронный ресурс]. URL: http://www.yugopolis.ru/news/rossiya-voshla-v-mirovoj-top-500-superkomp-yuterov-lish-s-3-modelyami-107614
10. Какие задачи в России решаются на суперкомпьютерах. [Электронный ресурс]. URL: http://fea.ru/news/3635.
11. Российские суперкомпьютеры рванули вверх.http://www.cnews.ru/news/top/2018-04-03_top50_rossijskie_superkompyutery_rvanuli_vverh
12. Климентов А.А., Машинистов Р.Ю., Новиков А.М., Пойда А.А, Рябинкин Е.А., Тертычный И.С. Интеграция суперкомпьютера НИЦ «Курчатовский институт» с центром Грид первого уровня // Суперкомпьютерные дни в России 2015. с.700 – 705.
13. Суперкомпьютеры и кластеры. [Электронный ресурс]. URL: https://poznayka.org.
14. Архитектура современных компьютеров. [Электронный ресурс]. URL: http://openmp.ru
15. McLay R., Schulz K.W., Barth W.L., Minyard T.: Best practices for the deployment and management of production HPC clusters. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 1–11 (Nov 2011).

16. Ведущие российские производители высокопроизводительных компьютеров. [Электронный ресурс]. URL:https://parallel.ru/computers/vendors.html
17. Левин В.К. Тенденции развития суперкомпьютеров. [Электронный ресурс]. URL: https://cyberleninka.ru/article/n/tendentsii-razvitiya-superkompyuterov
18. Кластеры и массивно-параллельные системы различных производителей. Примеры кластерных решений IBM. [Электронный ресурс]. URL: https://revolution.allbest.ru/programming/00715899_1.html
19. Кузьминский М. Пополнение в семействе Power // Открытые системы. СУБД. – 2014. – №7. – С. 16 –18.
20. POWER9 - Микроархитектура – IBM. [Электронный ресурс]. URL: https://en.wikichip.org/wiki/ibm/microarchitectures/power9
21. Varrette S., Bouvry P., Cartiaux H., Georgatos F.: Management of an academic HPC cluster: The UL experience. In: 2014 International Conference on High Performance Computing Simulation (HPCS). pp. 959–967 (July 2014).
22. Geographically dispersed parallel sysplex support for peer-to-peer VTS. [Электронный ресурс]. URL: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.idao300/o3035.htm
23. Руководство по внедрению IBM HiperSockets. [Электронный ресурс]. URL: http://www.redbooks.ibm.com/abstracts/sg246816.html
24. Сетевой график. [Электронный ресурс]. URL: http://www.stroitelstvo-new.ru/1/setevoy_grafik.shtml
25. Таха Х.А. Введение в исследование операций, 7-е издание: Пер. с англ. – М.: Издательский дом “Вильямс”, 2005. – 912 с.
26. Вагнер Г. Основы исследования операций. Том 1. Пер. с англ. Изд. «МИР». – М.: 1972., 336 с.
27. Карпов В. Е. Введение в распараллеливание алгоритмов и программ. Компьютерные исследования и моделирование 2010. Т. 2 № 3 С. 231–272.

Глава 7. РАСПРЕДЕЛЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ РЕСУРСОВ В СИСТЕМАХ МЯГКОГО РЕАЛЬНОГО ВРЕМЕНИ

7.1. Информационные системы предприятий

В современных условиях конкуренции эффективность предприятия во многом определяется организацией управления на основе информационно-коммуникационных технологий, реализуемых на автоматизированных системах планирования и управления ресурсами, запасами и кадрами предприятия. Информационная система предприятия (ИС), обеспечивающая получение своевременной и достоверной информации, содержит ряд подсистем, необходимых для принятия управленческих решений. По своей сути такая ИС является системой мягкого реального времени.

Создание ИС требует высокой квалификации участвующих в ней специалистов. Однако часто создание таких систем выполняется на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования системы. Одной из важнейших задач создания и сопровождения ИС является распределение вычислительной мощности ИС между ее подсистемами. В данной главе предлагается формализованный подход решения задачи распределения вычислительных ресурсов ИС между ее подсистемами, основанный на теории стратегических игр. Приводится пример распределения ресурсов ИС между подсистемами системы, даются предложения по совершенствованию системы распределения ресурсов.

Функционирование любого предприятия невозможно без технологий информационной поддержки производственных, финансовых, маркетинговых и других процессов предприятия. Эффективная информационная система значительно упрощает процесс управления, поскольку позволяет оперативно собрать, отсортировать, обработать необходимую информацию и помочь принять правильное решение управленцам (менеджерам) предприятия различного уровня. Цель ИС не просто увеличение эффективности обработки данных и помощь управленцу, а создание и поддержание высокоэффективного производства.

Информационная система управления предприятием должна автоматизировать все или, по крайней мере, большинство из видов деятельности предприятия [1–4]. Понятно, что автоматизация должна быть выполнена не ради автоматизации как таковой и удобства работы сотрудников, а должна обеспечить реальный эффект в результатах финансово-хозяйственной деятельности предприятия с учетом затрать на создание и эксплуатацию ИС.

В идеале автоматизированная информационная система предприятия представляется автоматизированной системой управления (АСУ). В классическом определении АСУ (как это определял еще академик Глушков В. М.) – это «совокупность экономико-математических методов, технических средств и организационных комплексов, обеспечивающих рациональное управление предприятием или технологическим процессом» [5].

В хозяйственной практике научно-исследовательских, конструкторских и производственных предприятий, выполняющих разработку, производство и сбыт какой-либо продукции, можно выделить основные типовые виды деятельности: производственная, маркетинговая, финансовая, кадровая.

Производственная деятельность связана с непосредственным выпуском продукции, управлением запасами и направлена на создание и внедрение в производство технических разработок и научно-технических новшеств.

Маркетинговая деятельность включает анализ рынка производителей и потребителей выпускаемой продукции, анализ продаж; организацию рекламной кампании по продвижению продукции и организацию материально-технического снабжения.

Финансовая деятельность связана с организацией контроля и анализа финансовых ресурсов фирмы на основе бухгалтерской, статистической, оперативной информации.

Кадровая деятельность направлена на подбор и расстановку необходимых фирме специалистов, а также ведение служебной документации по различным аспектам.

Перечисленные направления деятельности предприятия позволяют определить типовой набор информационных подсистем:

- производственная подсистема;
- подсистема маркетинга;
- финансовая и учётная подсистема;
- подсистема кадров.

В зависимости от масштаба предприятия каждая подсистема может иметь расширенный набор функций и представляться отдельной системой. Возможны и другие типы систем, выполняющие вспомогательные функции в зависимости от специфики деятельности фирмы. Кроме того, обязательно присутствует система администрирования ИС предприятия, задачей которой является рациональная организация вычислительного процесса в интересах качественного выполнения задач всех подсистем ИС.

Не углубляясь в дальнейшее рассмотрение особенностей построения современных ИС предприятия, в интересах рассмотрения и решения задачи данной главы будем далее считать, что рассматриваемая ИС состоит из перечисленных выше четырех подсистем и подсистемы администрирования. Внедрение ИС управления предприятием обычно производится силами специально сформированной рабочей группы или группы внедрения. На нее ложится значительная часть работы по внедрению системы и дальнейшему её сопровождению. Такой подход объясняется, во-первых, тем, что предприятие заинтересовано в том, чтобы у него были специалисты, которые могут оперативно решать большинство рабочих вопросов при настройке и эксплуатации системы, а во-вторых, обучение своих сотрудников и их использование всегда существенно дешевле аутсорсинга.

Особенно важным является выбор руководителя такой группы и администратора системы. Руководитель, помимо знаний компьютерных технологий, должен обладать глубокими знаниями в области ведения бизнеса и управления. Специалистов рабочей группы необходимо назначать с учётом знания современных компьютерных технологий и желания осваивать их в дальнейшем. Очень ответственно следует подходить к выбору и назначению администратора системы, так как ему будет доступна практически вся корпоративная информация.

Одной из важнейших задач, возлагаемых на группу внедрения ИС предприятия и администратора системы, является распределение имеющихся в системе вычислительных, информационных и программных ресурсов между подсистемами ИС. В данной главе рассмотрен возможный подход к решению задачи распределения вычислительных ресурсов между подсистемами ИС.

7.2. Постановка задачи распределения вычислительных ресурсов ИС

Вычислительные системы (или просто – вычислители) ИС предприятия (например, клиент-серверной архитектуры) представляют собой некоторое множество физических (аппаратных, программных и файловых) ресурсов. Обозначим их следующим образом:

$$F_{\text{ис}} = \{F_{\text{в}}, F_{\text{оп}}, F_{\text{взу}}, F_{\text{по}}, F_{\text{ф}}\}, \quad (7.1)$$

где $F_{\text{в}}$ – физические вычислительные ресурсы (процессоры), $F_{\text{оп}}$ – физические ресурсы оперативной памяти, $F_{\text{взу}}$ – физические ресурсы внешних запоминающих устройств, $F_{\text{по}}$ – физические ресурсы общего и специального программного обеспечения, $F_{\text{ф}}$ – физические ресурсы файловой системы.

Встает вопрос правильного (эффективного) распределения физических ресурсов системы между подсистемами ИС. Наиболее подходящим способом осуществления такого распределения следует считать механизм виртуализации [6-8]. На основе этого механизма, по сути, строится отображение физических ресурсов системы в некоторое множество виртуальных машин вида:

$$F_{\text{ис}} \rightarrow V_{\text{ис}}, \quad (7.2)$$

где $V_{\text{ис}} = \{V_1, V_2, \dots, V_n\}$ – множество виртуальных машин с некоторой выделенной долей физических ресурсов.

Такой подход позволяет разделить физические ресурсы между подсистемами ИС. В нашем дальнейшем примере определено четыре подсистемы (производственная, маркетинга, финансово-учетная и кадровая) и, кроме того, подсистема администрирования. Таким образом, следует создать пять виртуальных машин и распределить физические ресурсы системы между этими виртуальными машинами.

Техническое решение задачи выделения физических ресурсов созданной виртуальной машине решается администратором системы с помощью специальных средств администрирования, которые широко известны и имеются в арсенале администратора. Однако нетривиальным является вопрос определения доли физического ресурса, которую следует выделить каждой виртуальной машине.

Цель решения задачи заключается в определении доли вычислителя системы, которую требуется выделить каждой виртуальной машине. Для удобства дальнейшего изложения решения этой задачи примем производительность вычислителя системы (например, сервера) за единицу, тогда справедливо соотношение

$$\sum_{i=1}^{i=5} C_i \ \leq 1 \qquad (7.3)$$

где C_i – доля производительности вычислителя, выделенная виртуальной машине V_i и $0 \leq C_i \leq 1$.

Заметим, интенсивность работы подсистем ИС, как и поступление запросов пользователей этих подсистем, в общем случае слабо предсказуема. Обозначим множество параметров запросов подсистем ИС, поступающих на сервер в виде

$$P_i = \{p_j, j = 1,2, \dots, N_i\}, \qquad (7.4)$$

где p_j – значение некоторого параметра, а N_i – количество различных параметров. Такими параметрами могут быть, например, интенсивности запросов пользователей подсистемы на выполнение некоторой функциональной задачи, поиск, запись или редактирование необходимого документа и др.

Таким образом, постановка задачи может быть сформулирована следующим образом:

требуется найти такое отображение физических ресурсов системы в множество виртуальных машин, при котором обеспечивается полное выполнение функций подсистем с заданными значениями параметров рабочей нагрузки в условиях суммарной производительности виртуальной системы по ограничению, накладываемому физическими вычислительными ресурсами. С учетом введенных обозначений кратко постановка задачи представляется в виде:

найти такое отображение

$$F_{\text{ис}} \rightarrow \{V_1, V_2, \dots, V_n\}, \qquad (7.5)$$

которое при следующих ограничениях

$$P_i = \{p_j, j = 1,2, \dots, N_i\}, \ \ i = 1,2, \dots, n, \qquad (7.6)$$

$$0 \leq C_i \leq 1, i = ,2, \dots, n, \qquad (7.7)$$

$$\sum_{i=1}^{i=5} C_i \ \leq 1 \qquad (7.8)$$

обеспечивает полное выполнение запросов всех подсистем ИС.

В этих условиях полной неопределенности можно попробовать решить задачу на основе теории стратегических игр.

7.3. Решение задачи

В соответствии с теорией стратегических игр в данной игре два партнера: администратор системы и природа (сеть клиентов подсистем ИС), как принято в теории игр именовать полностью непредсказуемого партнера [9,10]. Под администратором системы будем понимать одно лицо, что касается природы, то – это клиенты ИС в целом. Однако потребности в вычислительных ресурсах каждой подсистемы могут иметь конфликт с такими же потребностями других подсистем, да и стратегии разных подсистем могут существенно отличаться. Поэтому целесообразна декомпозиция игровой задачи на несколько задач (игр) по подсистемам ИС. В каждой отдельной задаче стратегии администратора системы обозначим $R_1, R_2, \dots, R_k$, а стратегии природы (клиентов подсистемы ИС) $S_1, S_2, \dots, S_l$.

В этом случае в каждой частной задаче матрицу выигрышей можно представить в следующем виде

	S_1	S_2	….	S_j	….	S_l
R_1	d_{11}	d_{12}	….	d_{1j}	….	d_{1l}
R_2	d_{21}	d_{22}	….	d_{2j}	….	d_{2l}
…	…	…	….	…	….	…
R_i	d_{i1}	d_{i2}	….	d_{ij}	….	d_{il}
…	…	…	….	…	….	
R_k	d_{k1}	d_{k2}	….	d_{kj}	….	d_{kl}

Здесь d_{ij} – штраф, получаемый системой при имеющейся доли виртуальной машины R_i и требуемой доле S_j. Предположим, что с использованием тех или иных методов (средства операционной системы, программные анализаторы и т.п.) матрица получена. В соответствии с теоремой стратегических игр для нашего случая, когда значения из множеств $R = \{R_1, R_2, \dots, R_k\}$ и $S = \{S_1, S_2, \dots, S_l\}$ могут принимать конечное число, оптимальное решение заключается в поиске смешанных стратегий.

Из теории стратегических игр следует, что при использовании смешанных стратегий есть, по крайней мере, одно оптимальное решение с ценой игры V, которое находится между верхним и нижним значением игры [9,10]. Следует заметить, что всегда $V > 0$.

Допустим, что оптимальная стратегия администратора системы складывается из стратегий $R_1, R_2, \dots, R_k$ с вероятностями, равными $p_1, p_2, \dots, p_k$ $(p_1 + p_1 + \cdots + p_k = 1)$, а оптимальная стратегия клиентов подсистем ИС – из стратегий $S_1, S_2, \dots, S_l$, которые применяется с вероятностями, равными $q_1, q_2, \dots, q_l$ $(q_1 + q_2 + \cdots + q_l = 1)$. Если администратор применяет оптимальную стратегию, а клиенты подсистемы чистую стратегию $S_j (j = 1,2, \dots, l)$, то средний штраф, получаемый системой, составит

$$D_j = p_1 \cdot d_{1j} + p_2 \cdot d_{2j} + \cdots + p_k \cdot d_{kj} (j = 1,2, \dots, l).$$

Особенность оптимальной стратегии администратора состоит в том, чтобы при произвольном поведении противника (клиентов подсистемы) она обеспечивала штраф не больший, чем цена игры V. Отсюда имеем систему ограничений:

$$\left.\begin{array}{l} p_1 \cdot d_{11} + p_2 \cdot d_{21} + \dots + p_k \cdot d_{k1} \le V, \\ p_1 \cdot d_{12} + p_2 \cdot d_{22} + \dots + p_k \cdot d_{k2} \le V, \\ \dots. \\ p_1 \cdot d_{1k} + p_2 \cdot d_{2k} + \dots + p_k \cdot d_{kl} \le V. \end{array}\right\} \qquad (7.9)$$

Для удобства дальнейшего решения преобразуем систему ограничений (7.9), разделив по частям на V.

$$\left.\begin{array}{l} d_{11} \cdot x_1 + d_{21} \cdot x_2 + \dots + d_{k1} \cdot x_k \le 1, \\ d_{12} \cdot x_1 + d_{22} \cdot x_2 + \dots + d_{k2} \cdot x_k \le 1, \\ \dots. \\ d_{1k} \cdot x_1 + d_{2k} \cdot x_2 + \dots + d_{kl} \cdot x_k \le 1. \end{array}\right\} \qquad (7.10)$$

Здесь $x_1 = p_1 / V$, $x_2 = p_2 / V$, …,$x_k = p_k / V$.

Из условия $p_1 + p_2 + \cdots p_k = 1$следует, что

$$x_1 + x_2 + \cdots x_k = 1/V. \qquad (7.11)$$

Значения величин $p_1, p_2, \dots p_k$ должны быть такими, чтобы гарантированное значение штрафа системы было минимальным, т.е. чтобы достигалось

$$V = min \text{ или } \frac{1}{V} = max. \qquad (7.12)$$

Таким образом, решение задачи заключается в определении таких значений множества переменных $x_1, x_2, \dots x_k$, которое обеспечивает максимальное значение их суммы, т.е.

$$x_1 + x_2 + \cdots x_k = max. \qquad (7.13)$$

Кроме того, должны выполняться дополнительные граничные условия $p_i \geq 0$ $(i = 1,2, \dots, n)$, следовательно, имеем

$$x_i = p_i / V \geq 0 \ \ (i = 1,2, \dots, n) \qquad (7.14)$$

В итоге можно резюмировать, что нахождение оптимальной смешанной стратегии сводится к решению классической задачи линейного программирования с целевой функцией (7.13) и ограничениями (7.10) и (7.14). В результате решения этой задачи по определенным значениям $x_1, x_2, \dots x_k$ из уравнения (7.11) можно определить значение V, а затем из соотношения (7.14) значения $p_1, p_2, \dots p_k$, которые определяют оптимальную стратегию администратора.

7.4. Пример

Поскольку в постановке задачи отмечено, что каждая подсистема ИС может рассматриваться как независимая подсистема, которая выполняется в отдельной виртуальной машине, задачей администратора является определение оптимальной стратегии на выделение ресурсов для каждой подсистемы. В качестве примера определения такой стратегии рассмотрим производственную подсистему с ее параметрами запросов $P_{\text{пр}} = \{p_j, j = 1,2, \dots, N_{\text{пр}}\}$, которые должны выполняться сервером вычислительной системы предприятия. Построим матрицу игры [11]. Обозначим стратегии администратора системы $R_1, R_2, \dots, R_5$. Администратор (исходя из опыта) предполагает, что для производственной подсистемы достаточно выделение производительности виртуальной машины от 0,05 до 0,13 полной вычислительной мощности сервера (см. табл. 1).

Стратегии подсистемы, обозначенные как $S_1, S_2, \dots, S_5$, предусматривают требуемую производительность, обеспечивающую работу производственной подсистемы в следующих режимах:

Расчет и планирование загрузки технологического и другого оборудования. Выполнение этих задач требует производительности $S_1 =$ 0,08 полной вычислительной мощности сервера.

Контроль за ходом производства и координация работы подразделений предприятия. Эти задачи требует производительности $S_2 = 0{,}03$ полной вычислительной мощности сервера.

Ежедневный оперативный учет хода производства. Требует производительности $S_3 = 0{,}06$ полной вычислительной мощности сервера.

Оформление технической и другой документации. Эти работы связаны с затратами производительности в S_4= 0,045 вычислительной мощности сервера.

Планирование закупки материалов для производства продукции. Выполнение соответствующих задач требует $S_5 = 0{,}025$ вычислительной мощности сервера.

Пусть за дефицит производительности администратор получает штраф 5 условных единиц ($K_д = 5$, величина штрафа не имеет принципиального значения при решении задачи), а за избыточную производительность, выделенную подсистеме – штраф $K_и = 4$. Например, для совокупности стратегий {R3, S2} подсистеме необходимо 0,03 производительности полной виртуальной машины, а администратором выделено 0,1 виртуальной машины. В этом случае имеет место избыточная производительность, штраф за которую составляет 4(|0,1-0,03|) = 0,28 штрафных единиц. Для совокупности стратегий {R4, S5} подсистеме необходимо 0,025 производительности полной виртуальной машины, а администратором выделено 0,1 виртуальной машины. В этом варианте дефицит производительности и штраф составит 4(|0,1-0,025|) =0,3 штрафных единицы.

Таблица 1

Матрица игры

		Стратегии подсистемы				
		S_1	S_2	S_3	S_4	S_5
Стратегия администратора		0,08	0,03	0,06	0,045	0,025
R_1	0,05	5(0,08 - 0,05) = 0,15	4(0,05 -0,03) = 0,1	5(0,06-0,05) = 0,05	4(0,05-0,045) = 0,02	4(0,05-0,025) = 0,1
R_2	0,07	5(|0,07-0,08|) =0,05	4(0,07-0,03) =0,16	4(0,07-0,06) =0,04	4(0,07-0,045) =0,1	4(0,07-0,025) =0,18
R_3	0,09	4(0,09-0,08) =0,04	4(0,09-0,03) =0,24	4(0,09-0,06) =0,12	4(0,09-0,045) =0,18	4(0,09-0,025) =0,354
R_4	0,11	4(0,11-0,08) =0,12	4(0,11-0,03) =0,32	4(0,11-0,06) =0,2	4(0,11-0,045) =0,26	4(0,11-0,025) =0,354
R_5	0,13	4(0,13-0,08) =0,2	4(0,13-0,03) =0,4	4(0,13-0,06) =0,28	4(0,13-0,045) =0,34	4(0,13-0,025) =0,42

Решение задачи для составленной матрицы игры представлено в табл.2. Цена игры в данном случае равна 8,18 штрафной единицы. Решение определяет использование стратегий R1 и R2 с вероятностями соответственно равными 0,72 и 0,28. Это позволяет считать целесообразным выбор производительности виртуальной машины для реализации производственной подсистемы из соотношения

$$C_{\text{пр}} = R_1 \cdot p_1 + R_2 \cdot p_2 = 0{,}05 \cdot 0{,}72 + 0{,}07 \cdot 0{,}28 = 0{,}06.$$

Таблица 2

Решение задачи

Решение задачи

Стратегии подсистемы			S1	S2	S3	S4	S5	Переменные		Вероятности		Ограничения		
			0,08	0,03	0,06	0,045	0,03	X		P				
Стратегии администратора	R1	0,05	0,15	0,1	0,05	0,02	0,1	x_1	5,91	p_1	0,72	1	1	1
	R2	0,07	0,05	0,16	0,04	0,1	0,18	x_2	2,27	p_2	0,28	2	0,95	1
	R3	0,09	0,04	0,24	0,12	0,18	0,35	x_3	0	p_3	0	3	0,39	1
	R4	0,11	0,12	0,32	0,2	0,26	0,35	x_4	0	p_4	0	4	0,35	1
	R5	0,13	0,2	0,4	0,28	0,34	0,42	x_5	0	p_5	0	5	1	1
						Целевая функция			8,18		1			
						Цена игры V			0,12					

Аналогичным образом определяется необходимая производительность виртуальной машины для других подсистем ИС, после чего проверяется выполнение выражения (7.8). Если оно выполняется, то распределение реальных ресурсов вычислительной системы считается законченным. В противном случае (ресурсов недостаточно) можно увеличить штраф за предоставление избыточных ресурсов для подсистем, не критичных для выполнения задач, требующих режима реального времени. Существенное нарушение выражении (8) может свидетельствовать о необходимости увеличения физических вычислительных ресурсов ИС предприятия.

Предложенный подход и метод (методический аппарат) распределения вычислительных ресурсов в информационных системах предприятия может быть эффективно использован на начальном этапе работы

ИС предприятия. По мере вхождения ИС в режим нормальной эксплуатации может возникнуть необходимость перераспределения ресурсов ИС между подсистемами. Учет текущей загрузки вычислительных ресурсов подсистемами ИС может проводиться средствами операционной системы. Однако перераспределение ресурсов системы между подсистемами ИС потребует выполнения нетривиальных действий со стороны системного администратора и не всегда приведет к требуемому результату.

Реальные возможности решения задачи адаптивного распределения вычислительных ресурсов ИС предоставляют системы искусственного интеллекта с задачей поддержки принятия решений (ППР). По мнению многих специалистов, теория игр хорошо подходит для систем ППР, созданных для принятия стратегических решений [12]. По тематике данной задачи возможно два варианта использования обучаемых нейронных сетей. Первый вариант – построение нейронной сети для решения задач, рассматриваемых в теории игр, в том числе игровых задач, где один из партнеров – природа. Второй вариант связан с созданием обучаемой нейронной сети, включаемой в состав операционной системы компьютера для распознавания ситуации с текущей загрузкой системы, что позволит решить задачу адаптивного распределения вычислительных ресурсов системы.

Литература к гл. 7

1. Шакурин В. Подходы и процедуры выбора Корпоративных Информационных Систем. [Электронный ресурс]. URL:https://vc.ru/u/313313-victor-shakhurin.

2. Капулин Д. Информационная структура предприятия. М.: Сибирский федеральный университет, 2014. 270 с.

3. Современные информационные технологии в бизнесе.[Электронный ресурс]. URL:https://www.kom-dir.ru/article/2291-informatsionnye-tehnologii-biznesa

4. Юрченко Т. В. Информационные системы в экономике и управлении. [Электронный ресурс]. URL:https://bibl.nngasu.ru/electronicresources/uch-metod/automatics/850497.pdf

5. Глушков В.М. Введение в АСУ. – Киев: Техника, 1972. – 312 с.

6. Анализ современных технологий виртуализации. [Электронный ресурс]. URL:https://habr.com/ru/company/southbridge/blog/212985/

7. Виртуализация: новый подход к построению IT-инфраструктуры [Электронный ресурс]. URL:https://www.ixbt.com/cm/virtualization.shtml

8. Платформы виртуализации. Обзор. Hyper-V, KVM, vSphere и XenServer. [Электронный ресурс]. URL:https://itglobal.com/ru-kz/company/blog/about-virtual-solutions/

9. Ланге О. Оптимальные решения. М. Прогресс, 1967. - 285 с.

10. Варфоломеев В.И., С. Н. Воробьев С.Н. Принятие управленческих решений – М. : Кудиц-Образ, 2001. – 287 с.

11. Назаров С.В. Барсуков А.Г. Управление предоставлением облачных вычислительных ресурсов в виртуальных дата-центрах. Электроника: наука, технология, бизнес. 2018. № 4. С 108 – 112.

12. Интеллектуальные системы поддержки принятия решений. Краткий обзор [Электронный ресурс]. URL:https://habr.com/ru/company ods/blog/359188/

Глава 8. РЕФАКТОРИНГ ПРОГРАММНЫХ СИСТЕМ

8.1. Что такое рефакторинг

Начиная с последнего десятилетия прошлого века, получили широкое распространение эволюционные методологии разработки программного обеспечения, часто называемые адаптивными, такие как экстремальное программирование (Extreme Programming - XP), метод Scrum, унифицированный процесс компании Rational (Rational Unified Process - RUP), адаптивный унифицированный процесс (Agile Unified Process - AUP) и др. [1]. По своему характеру эти методологии является и итеративными, и инкрементными, а адаптивный подход является эволюционным и вместе с тем характеризуется высокой степенью взаимодействия участников разработки проектов.

В организациях, применяющих подобные технологии, внедряются такие адаптивные методики, как рефакторинг, программирование в паре, разработка на основе тестирования (Test-Driven Development - TDD) и адаптивное проектирование на основе модели (Agile Model Driven Development – AMDD). Концепция рефакторинга (refactoring) возникла в кругах, связанных со Smalltalk, но вскоре нашла себе дорогу и в лагеря приверженцев других языков программирования [2]. Несколько позже Мартин Фаулер в своей фундаментальной монографии, выдержавшей в нашей стране несколько изданий [3], дал определение рефакторинга как небольшого изменения в исходном коде, которое способствует улучшению проекта кода без изменения его семантики.

Иными словами, рефакторинг – это улучшение качества сделанной программистом работы без нарушения или добавления чего-либо. В своей книге М. Фаулер говорит также о том, что если возможно подвергнуть рефакторингу прикладной исходный код, то, видимо, есть возможность подвергнуть рефакторингу схему базы данных. Однако он считает, что рефакторинг баз данных – достаточно сложная и отдельная проблема и исключил эту тематику из своей книги. Однако это замечание не осталось незамеченным, и несколько позже появилась еще одна фундаментальная книга, посвященная именно рефакторингу баз данных [1].

Эволюция сложных программных систем требует от разработчика повышенного внимания к выбору архитектуры [4]. Практически всегда во время разработки, появляются новые требования со стороны заказчика, и приходиться пересматривать первоначальную архитектуру, в

том числе и структуру базы данных. Возможна и другая ситуация. Часто в фазе поддержки и эволюции приложения появляется желание переделать всё "с нуля".

Что же такое рефакторинг? Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. При проведении рефакторинга разработчик улучшает дизайн кода уже после того, как он написан [5, 6].

Что дает рефакторинг? Вот некоторые преимущества:

1. Рефакторинг улучшает композицию программной системы (ПС). По мере развития программы часто приходится вносить изменения, которые обусловлены текущей необходимостью. Изменения вносят и программисты, которые не до конца понимают архитектуру ПС в целом. Поэтому постепенно код становится менее структурированным и разбираться в нем все труднее.

2. Рефакторинг облегчает понимание ПС. Проведение рефакторинга обычно приводит к более глубокому пониманию того, как работает программа. Такое понимание существенно ускоряет процесс программирования.

3. Рефакторинг помогает найти ошибки. Поскольку рефакторинг увеличивает понимание кода, то он и позволяет быстрее находить ошибки.

4. Рефакторинг позволяет быстрее писать программы. Все вышеперечисленные пункты сводятся к тому, что рефакторинг позволяет быстрее разрабатывать код.

Рефакторинг является одной из основных практик экстремального программирования (XP) и позволяет создавать хорошую архитектуру программы, но несколько непривычным путем. В традиционном подходе архитектура программы создается еще до того, как написана первая строчка кода. Архитекторы на основе требований к ПС создают все то, что необходимо кодировщикам для написания кода. Конечно, обычно архитекторы – опытные разработчики, тем не менее очень сложно сразу создать хорошую архитектуру.

Если полагаться на рефакторинг, то вначале можно сделать лишь предварительный набросок будущей системы, а детали выяснять в процессе разработки. Чем больше разработчики знают о системе, тем больше возникает мыслей по ее улучшению. Мелкие и средние изменения в коде сами по себе не оказывают огромного влияния на систему, но суммарный эффект часто превосходит все ожидания. Спустя несколько

итераций архитектура системы становится стабильной и красивой. Когда программист чувствует целостность и внутреннюю красоту системы, он испытывает удовлетворение от работы.

Может показаться, что такой подход ненадежен. Действительно, чтобы смело полагаться на рефакторинг, необходимо наличие следующих условий:

1. Наличие автоматических тестов. Если они отсутствуют, то сложно контролировать изменения. Предположим, программист изменил название метода. Теперь ему нужно узнать, где он вызывается. При наличии автоматических тестов это делается элементарно. Если же изменения менее очевидные, то ценность автоматических тестов многократно возрастает.

2. Система контроля исходного кода. Управление исходным кодом является одной из главных практик разработки ПС. Если имеется работающая версия системы, то можно без опасений вносить изменения, потому что всегда есть возможность вернуться к стабильной версии. Контроль кода дает свободу.

3. Итерационная разработка. Рефакторинг можно проводить и при обычном водопадном процессе, но его ценность в таком случае уменьшается. Итерации позволяют лучше организовать процесс рефакторинга и сделать его последовательным.

4. Регулярность осуществления рефакторинга. Только методичное применение может принести ощутимую пользу. Эпизодический рефакторинг может быть даже опасным в случае отсутствия контроля исходного кода.

Есть одна опасность, которая подстерегает тех, кто начинает использовать рефакторинг. Надо четко разделять процесс разработки нового кода от процесса изменения уже существующего. Например, разработчик нашел в программе метод, который можно изменить (улучшить). Он начинает его изменять, не доводит изменения до конца, и вдруг вспоминает, что надо добавить в этот метод немного новой функциональности. Программист начинаете ее добавлять, увлекается и забывает завершить рефакторинг. В результате придется тратить лишнее время на то, чтобы все заработало, как надо. Поэтому, когда выполняется рефакторинг, надо обязательно доводить его до конца или вернуться к начальному состоянию и только после этого начинать писать новый код.

Проблема выбора при добавлении в систему новой возможности возникает при разработке достаточно сложной ПС практически всегда. Самым сложным при выборе того или иного решения является доведение разработчиком своего выбора непосредственному руководителю,

чтобы он смог принять взвешенное решение. С точки зрения многих руководителей «взвешивание» заканчивается сразу же после того, как он услышит предполагаемые сроки реализации. Не удивительно, что программисты и архитекторы, с таким подходом не согласны. С подобной ситуацией сталкивались известные специалисты, которые придумали типовые «паттерны», описывающие такую ситуацию. Одним из таких паттернов, является метафора технического долга, впервые описанная В. Каннингемом более двадцать лет назад [7].

Под техническим долгом понимается осознанное компромиссное решение, когда заказчик и ключевые разработчики четко понимают все преимущества от быстрого, пусть и не идеального технического решения, за которое придется расплатиться позднее. И хотя с точки зрения многих разработчиков ситуация, когда плохое решение может быть хорошим, может показаться нереальной, на самом деле, это вполне возможно. Если краткосрочное решение позволит компании получить видимые преимущества, выпустить продукт раньше конкурентов, удовлетворить ключевого заказчика или получить какие-то преимущества перед конкурентами, тогда такое решение совершенно оправданно.

Можно провести параллель между техническим и финансовым долгом. Финансовый долг означает, что вы получаете дополнительные средства сейчас, однако вам придется выплачивать фиксированную процентную ставку, и в конце срока вернуть весь долг кредитору. Аналогичная ситуация происходит и в случае принятия неоптимального технического решения.

Как признается в ряде публикаций [7 - 9], именно такой подход к построению приложений является оптимальным с архитектурной точки зрения. Если принимая архитектурное решение, разработчик не чувствует компромисса между краткосрочными и долгосрочными выгодами, то в любом случае не стоит считать его окончательным. Вполне возможно, что неправильное решение принято неосознанно, из-за непонимания предметной области, или из-за будущего изменения требований заказчиком.

Значительно проще снизить стоимость будущих изменений путем инкапсуляции важных решений в наименьшее число модулей, чем стараться продумать архитектуру до мельчайших подробностей, в надежде учесть все возможные и невозможные сценарии. Для определенного круга задач идеальная продуманная архитектура возможна, а иногда просто необходима, но в большинстве случаев – это не так; рано или поздно придет момент, когда видение системы разработчиком или

заказчиком изменится, что потребует внесения в нее существенных изменений.

Однако даже при разумном накоплении технического долга существует вероятность того, что команда (или заказчики) пойдут по старому принципу – работает – не трогай, и никогда не вернутся к этому решению снова, чтобы расплатиться за свой технический долг. Кроме того, существует множество случаев, когда технический долг накапливается постепенно и неосознанно, и если разработчики не будут об этом задумываться, то придут к ситуации, когда стоимость добавления новой возможности становится очень дорогой. В этом случае команда усиленно работает, исполнители устают, злятся друг на друга, не получив при этом никаких преимуществ перед конкурентами. Результатом является "грязный код" – второй источник технического долга [7].

Технический долг в классическом понимании является преднамеренным и в основном касается стратегических решений, поэтому и ответственность за него лежит на заказчиках и архитекторах, но все, что связано с грязным кодом, касается по большей части простых разработчиков [10]. Во время кодирования, как и во время принятия любых других решений, разработчик должен рассматривать краткосрочные и долгосрочные выгоды. Обычно в процессе развития приложения накапливается как груз крупных стратегических ошибок или непониманий требований, так и масса мелких тактических ошибок, типа длинных функций с неочевидными обязанностями и сложными побочными эффектами; расплывчатых классов, с нечеткими границами и обязанностями; отсутствие идиом именования и документации и т.п.

И если команда не отдает свои долги путем переосмысливания существующих решений и их последующего рефакторинга, если она не старается держать качество кода на высоком уровне, то рано или поздно это повысит стоимость развития и сопровождения кода настолько, что все существующие средства будут уходить на оплату процентов по техническому долгу.

Существует несколько способов выплаты технического долга. Наиболее простым из них является рефакторинг системы или некоторых ее частей или полное переписывание некоторых частей системы. Однако, часто бывает вместо прагматичного подхода к улучшению некоторых частей системы для получения разумной выгоды в будущем, программист начинает переписывать все подряд: что нужно, что не обязательно и что вообще переписывать не стоит. Это приводит к еще одной метафоре, которая описывает подобное неустранимое желание к переписыванию старого кода – к синдрому рефакторинга [8].

Первый симптом такого синдрома – неуважительное отношение к чужому коду. Заметим, что, во-первых, это не красит разработчика, а во-вторых, в большинстве случаев не уменьшает технический долг, который должен уменьшаться благодаря этому рефакторингу. Система в целом не становится более сопровождаемой и понятной, просто она теперь ближе и понятнее одному конкретному человеку. Конечно, на некоторое время сопровождаемость системы улучшается, но все выгоды закончится, когда за этот кусок кода берется другой разработчик.

Стремление к идеальному коду (симптом 2) является наиболее благим намерением при изменении существующего кода. Слепое следование непродуманным стандартам кодирования не многим лучше, чем отсутствие стандартов кодирования вовсе. Красота кода – не самоцель; код может быть красиво оформленным, все функции могут быть небольшого размера с достаточным количеством комментариев и даже проверены тестами. Но это не значит, что этот код будет справляться со своей основной задачей, поскольку никто не удосужился узнать эту задачу у пользователя.

С течением времени, с высоты своего собственного профессионального опыта, благодаря более четкому пониманию требований пользователя и, пониманию своей собственной системы, даже свой код начинает выглядеть ужасным (симптом 3). Поэтому улучшение своего собственного кода – процесс настолько же полезный, как и улучшение кода чужого. Но бывают случаи, когда даже свой код подвергается значительным переделкам не раз за год, но при этом в нем толком ничего и не меняется.

Это нормально, если человек это делает для своего собственного проекта, когда при этом нарабатываются новые решения уже хорошо известных задач. Но это не совсем разумно для коммерческого продукта. Получается, что код переписывается просто потому, что программист узнал о новой возможности в языке программирования или о новой библиотеке, которая значительно лучше решает эту же задачу. Проходит время, появляются новые возможности, процесс опять повторяется с начала, но при этом никакие долги не выплачиваются и ничего ценного в систему не добавляется.

Практически в любом деле прагматизм и отсутствие крайностей является лучшим выбором, и рефакторинг существующего кода – не исключение. Не стоит забывать о законе Парето – принципе 80/20. В соответствии с ним двадцати процентов усилий на улучшение кода и поддержание его в адекватном состоянии достаточно, чтобы улучшить его на 80%. А если это не так, то инвестировать дополнительные средства

на покрытие столь большого технического долга просто нет смысла, и стоит начать все с чистого листа.

И тут начинает проявляться эффект второй системы [9]. Когда технический долг команды начинает превышать все мыслимые и немыслимые границы, у команды разработчиков появляется как минимум два способа его погашения: провести рефакторинг системы таким образом, чтобы стоимость будущих изменений была не столь высокой или оставить текущую версию системы в покое и переписать все заново. В первом случае легко столкнуться с синдромом рефакторинга, когда изменения делаются не с расчетом уменьшения стоимости будущих изменений, а вносятся просто ради изменений.

Во втором случае может возникнуть «эффект второй системы», когда развивается система с чужим кодом. И хотя в классическом понимании «эффект второй системы» немного отличается от патологической нелюбви к чужому коду и постоянному его переписыванию, оба эти случая имеют и что-то общее, так что имеет смысл оба эти симптома рассмотреть совместно.

Поскольку многие разработчики и архитекторы вкладывают свою душу при создании программных систем, совсем не удивительно, что, приступая к новой версии своего детища, они стараются исправить все ошибки и недочеты, и пытаются создать идеальную систему. Кроме того, окрыленные успехом первой системы, разработчики начинают думать, что у них достаточно знаний и опыта в данной предметной области, чтобы на основе существующих частных знаний делать общие выводы. Это зачастую приводит к преждевременному обобщению решения, что очень часто считается злом не меньшим, чем преждевременная оптимизация.

Другой проблемой при создании второй системы является неправильная переоценка ценностей, когда доводятся до совершенства морально устаревшие функции системы, а принципиально новые подходы и решения не развиваются. Это приводит к идеальной реализации никому не нужных функций и вряд ли полезно для успеха системы. Все эти проблемы описаны еще Ф. Бруксом в его «Мифическом человеко-месяце», где помимо описания этой проблемы даются и рекомендации по ее решению. И хотя она вышла задолго до замечательной книги Э. Ханта и Д. Томаса «Программист-прагматик» [11], Брукс дает тот же самый совет, что и постоянно звучит в книге прагматиков: не нужно крайностей, и больше прагматизма и самодисциплины.

Этот прагматизм проявится в разумном балансе причин изменения программного кода. Код лучше подвергать рефакторингу сразу, как

только выявлена необходимость. Не стоит забывать про баланс между желаниями команды разработчиков и требованиями от заказчика по срокам. Баланс заключается в том, чтобы разумно совмещать введение новых функций и рефакторинга. Во многих случаях это экономит время. Разработчику следует всегда отдавать себе отчет – тратить ли время на рефакторинг или же заняться новым функционалом. Заказчики легче расстаются с деньгами, чем с благодарностью за хорошо сделанную работу, тем более за качественный код, в котором они ничего не понимают.

У. Апдайк, написавший главу в книге М. Фаулера [3], задается вопросом: почему разработчики не хотят применять рефакторинг к своим программам? Он считает, что если проект начинается с нуля и понятна задача, для решения которой предназначена система, и тот, кто финансирует проект, готов поддерживать его, пока разработчик не будете удовлетворен результатами, то разработчику крупно повезло. Такой сценарий идеален для применения объектно-ориентированной технологии, но большинство может о нем только мечтать.

Значительно чаще предлагается расширить возможности уже имеющегося программного обеспечения. Если у разработчика далеко не полное понимание того, что он делает, или он поставлен в жесткие временные рамки, что можно предпринять? Можно переписать программу заново. Применить свой опыт проектировщика, исправить все прежние грехи, работать творчески и с удовольствием. Но кто оплатит расходы? И можно ли быть уверенным, что новая система сможет делать все то, что делала старая? Можно скопировать и модифицировать части существующей системы, чтобы расширить ее возможности. Это может показаться оправданным, и даже будет рассматриваться как демонстрация повторного использования кода. Не надо даже разбираться в том, что будет повторно использоваться.

Однако с течением времени происходит размножение ошибок, программы разбухают, их архитектура разрушается, и нарастают дополнительные издержки на модификацию. Рефакторинг лежит посередине между этими двумя крайностями. Он дает возможность изменить существующее программное обеспечение, сделав понимание конструкции более явным, развить строение и извлечь повторно используемые компоненты, сделать яснее архитектуру программы, а также создать условия, облегчающие ввод дополнительных функций. Рефакторинг может способствовать извлечению выгоды из сделанных ранее инвестиций, уменьшить дублирование и рационализировать программу.

Допустим, что разработчика привлекают эти преимущества. Он согласен с Ф. Бруксом в том, что модернизация представляет собой “внутренне присущую сложность” разработки программного обеспечения. Он согласен с тем, что теоретически рефакторинг дает указанные преимущества. Почему же он все-таки не применяет рефакторинг к своим программам?

Возможны четыре причины:

1. Разработчик может не понимать, как выполнять рефакторинг.

2. Если выгоды ощутимы лишь в далекой перспективе, зачем тратить силы сейчас? В долгосрочной перспективе, когда придет время пожинать плоды, разработчик может оказаться вне проекта.

3. Рефакторинг кода является непроизводительной деятельностью, а разработчику платят за новые функции.

4. Рефакторинг может нарушить работу имеющейся программы.

Все это законные причины для беспокойства, отмечает У. Апдайк. Ему не раз доводилось выслушивать их от персонала коммуникационных и высокотехнологических компаний. Одни из них относятся к технологиям, а другие – к администрированию. Все они должны быть рассмотрены, прежде чем разработчики взвесят возможность рефакторинга своего программного обеспечения. Разберемся с каждой из них по очереди.

Как и когда применять рефакторинг? Как можно научиться рефакторингу? Каковы инструменты и технологии? В каких сочетаниях они могут принести какую-нибудь пользу? Когда следует их применять?

Ответы на эти вопросы, по мнению У. Апдайка, даны в книге М. Фаулера [3]. В этой книге описано несколько десятков различных рефакторингов, которыми автор пользуется в своей работе. Приведены примеры применения рефакторингов для внесения в программы важных изменений. С этим можно согласиться, тем не менее, в Интернете можно найти и отрицательные мнения об этой книге [12]. Однако, следует заметить, что негативное мнение связано с ожиданием рекомендаций архитектурного рефакторинга ПС, в то время как М. Фаулер в своей монографии говорит в основном о рефакторинге программного кода, причем чаще на уровне внутри класса и значительно реже на уровне классов. Рефакторингу архитектуры объектно-ориентированных ПС уделено внимания значительно меньше.

В работе [13] в довольно концентрированной форме даны ответы на основные вопросы, связанные с проведением рефакторинга ПС, а также ряд рекомендаций, как проводить рефакторинг. Обсуждаются

следующие вопросы: цель рефакторинга, простые и сложные методы рефакторинга, когда начинать рефакторинг? Как начать рефакторинг? Как не поломать рабочий код? Когда рефакторинг не нужен? И в заключение дается каталог методов рефакторинга и примеры рефакторинга.

Перейдем к рассмотрению этих вопросов с позиций автора этой работы.

Цель рефакторинга. Определите задачи, которые перед вами стоят, в большинстве случаев это упрощение ввода новых функций. Но, есть еще и другие цели. Профессиональные программисты, знакомые с паттернами проектирования [13], стараются привести структуру кода в порядок в соответствии с подходящими паттернами для улучшения поддержки, расширяемости и повторного использования кода.

Имеется также возможность использовать автоматизированные средства поиска кода подлежащего рефакторингу, это могут быть функции с большим количеством аргументов или слишком длинные функции. С помощью автоматических средств можно также выявлять структурное сходство различных методов. Такие функции являются кандидатами на проведение рефакторинга.

Простые и сложные методы рефакторинга. Простые методы рефакторинга – это базовые, фундаментальные методы, применяемые в рефакторинге. Например, это: выделение метода, перемещение поля класса, выделение класса, переименование метода, класса или поля и т.д. Простые методы рефакторинга – это фундамент, на основе которого строится любой вид рефакторинга. Нельзя начать рефакторинг, не зная фундаментальные методы. Сложные методы рефакторинга – это по сути комбинация простых методов, но решающих одну базовую задачу. Примером сложного метода рефакторинга может быть разделение наследования, который включает в себя: выделение класса, перемещение метода, подъем метода и подъем поля.

Когда начинать рефакторинг? Рефакторинг обычно применяется при добавлении новой функции. Причина, по которой стоит проводить рефакторинг, добавляя новую функцию, это возможно несовместимый или неясный дизайн кода для добавления новых функций. Рефакторинг следует применять, если требуется исправить ошибку. При исправлении ошибок польза рефакторинга в том, что код становиться более понятным. Если получается сообщение об ошибке, то это признак необходимости рефакторинга, потому что код не был достаточно ясным, и разработчик не смог увидеть ошибку. Целесообразно применять рефакторинг при разборе кода. Разбор кода – это хорошая практика, когда в команде разработчики проверяют код друг друга.

Как начать рефакторинг? Нужно четко убедиться, над какой частью кода будет производиться рефакторинг. Нужно точно знать, какую часть кода могут затронуть изменения. Определить методы, с помощью которых будет производиться рефакторинг. Выписать эти методы и для каждого составить последовательность шагов, которые будет необходимо выполнить. Проделать эту работу один раз для каждого метода, чтоб быть уверенным, что ничего не пропущено. Большинство методов рефакторинга уже имеют порядок действий и описание применения, ознакомьтесь с этими методами и применяйте их.

Как не поломать рабочий код? Постройте набор тестов для перерабатываемой части кода. Тесты важны, так как даже при последовательном проведении рефакторинга можно допустить ошибки, тесты помогут исключить возможные ошибки. Проводя рефакторинг, можно полагаться на тесты. Старайтесь покрыть тестами достаточную часть кода для работы функции, тесты должны быть как чистые, проверяющие вычисления правильных данных, так и грязные, посылающие в тест недопустимые, не ожидаемые данные.

Когда рефакторинг не нужен? Иногда рефакторинг не нужен. Например, когда надо переписать программу с нуля. Иногда имеющийся код настолько запутан, что подвергнуть его рефакторингу, конечно, можно, но проще начать все с самого начала. Явный признак необходимости переписать код – это его неработоспособность. Это обнаруживается при его тестировании, когда ошибок так много, что сделать код устойчивым не удается. Другой случай, когда следует воздержаться от рефакторинга, это близость даты завершения проекта. Однако приближение срока окончания работ – единственный случай, когда можно отложить рефакторинг, ссылаясь на недостаток времени.

Каталог методов рефакторинга и примеры рефакторинга. Большой каталог методов рефакторинга можно найти на refactoring.com вместо примеров кода, тут применяются примеры на языке UML. Также, хороший каталог методов рефакторинга можно найти на wikipeadia, каждый метод снабжен примером на C#. Применение методов рефакторинга требует хорошего знания ООП, умения писать тесты, быть терпеливым, делая рефакторинг небольшими шагами, проверять каждый шаг и постоянно учиться.

Автор статьи [13] дает несколько советов для проведения рефакторинга:

1. Всегда выполняйте рефакторинг короткими шагами с перерывом на перекомпиляцию и запуском тестов. Чем меньше ваши шаги, тем

лучше вы локализуете потенциальные ошибки и тем быстрее вы их устраните. То, что ошибки будут, можно не сомневаться.

2. Проводите рефакторинг снизу-вверх, особенно, если у вас длинная и запутанная цепочка наследования классов. Старайтесь всегда сначала производить изменения в потомственных классах, прежде чем приступать к базовым классам.

3. Вы должны знать основные методы рефакторинга, для этого советую вам купить книгу Мартина Фаулера – "Рефакторинг".

4. У вас всегда должна быть цель, с которой вы производите рефакторинг, не делайте его там, где он не нужен или только, если это не первоочередная задача. Чем лучше вы понимаете, зачем вы это делаете, тем качественнее будет результат.

5. Не увлекайтесь рефакторингом (рефакторинг – не панацея!) [14], рефакторинг не добавляет функционала в программу, поэтому и каких-либо видимых результатов вы также не получите. Делайте перерывы для того, чтоб сделать, что то, что можно увидеть и оценить. Тогда ваш руководитель будет вами доволен.

6. Любой вид рефакторинга можно сделать за 5, 20 минут, максимум за один час. Но в основном рефакторинг является комплексной задачей, которая может выполняться в течение недель или месяцев над действующим проектом. Идея в том, что двигаться нужно постепенно и небольшими шагами, и возможно уделяя этому не больше одного часа в день. Это хороший метод рефакторинга, потому как не занимает много времени и убедить начальство в необходимости будет намного проще.

7. В долгосрочной перспективе у вас будет красивый и легко сопровождаемый код. Код, который написан для людей, а не для машин. Код, которым вы можете гордиться и показывать в пример. Код, который работает, так как вы этого хотите, и все это за невысокую цену рефакторинга.

8.2. Рефакторинг, проектирование и производительность программ

В своей монографии [3] М. Фаулер отмечает, что рефакторинг играет особую роль в качестве дополнения к проектированию. Большинство начинающих программистов (в том числе и автор этой монографии в свое время) пишут программу и доводят ее до конца. Со временем становится ясно, что если заранее подумать об архитектуре программы, то можно избежать последующей дорогостоящей переработки. Поэтому

программисты привыкают к этому стилю предварительного проектирования. Многие считают, что проектирование важнее всего, а программирование представляет собой механический процесс.

Но программа весьма отличается от физического устройства. Это более гибкий объект. Программа значительно более податлива и целиком связана с обдумыванием. Существует утверждение, что рефакторинг может быть альтернативой предварительному проектированию. В крайнем сценарии проектирование вообще отсутствует. Первое решение, пришедшее в голову, воплощается в коде, доводится до рабочего состояния, а потом обретает требуемую форму с помощью рефакторинга. Такой подход фактически может действовать. Действительно есть программисты, которые так работают и получают в итоге систему с очень хорошей архитектурой.

Тех, кто поддерживает экстремальное программирование, часто изображают пропагандистами такого подхода. Подход, ограничивающийся только рефакторингом, применим, но не является самым эффективным. Даже «экстремальные» программисты сначала разрабатывают некую архитектуру будущей системы. Они пробуют разные идеи с помощью CRC-карт или чего-либо подобного [14], пока не получат внушающего доверия первоначального решения. Только после этого приступают к кодированию, а затем к рефакторингу. Смысл в том, что при использовании рефакторинга изменяется роль предварительного проектирования. Если не рассчитывать на рефакторинг, то ощущается необходимость как можно лучше провести предварительное проектирование.

Возникает чувство, что любые изменения проекта в будущем, если они потребуются, окажутся слишком дорогостоящими. Поэтому в предварительное проектирование вкладывается больше времени и усилий – во избежание таких изменений впоследствии. С применением рефакторинга акценты смещаются. Предварительное проектирование сохраняется, но теперь оно не имеет целью найти единственно правильное решение. Все, что от него требуется, – это найти приемлемое решение. По мере реализации решения, с углублением понимания задачи становится ясно, что наилучшее решение отличается от того, которое было принято первоначально. Но в этом нет ничего страшного, если в процессе участвует рефакторинг, потому что модификация не обходится слишком дорого.

Важным следствием такого смещения акцентов является большее стремление к простоте проекта. До введения рефакторинга в свою работу, как отмечает автор монографии [3], он всегда искал гибкие реше-

ния. Для каждого технического требования он рассматривал возможности его изменения в течение срока жизни системы. Поскольку изменения в проекте были дорогостоящими, он старался создать проект, способный выдержать изменения, которые можно было предвидеть. Недостаток гибких решений в том, что за гибкость приходится платить. Гибкие решения сложнее обычных. Создаваемые по ним программы в целом труднее сопровождать, хотя их легче перерабатывать в том направлении, которое предполагалось изначально.

И даже простые решения не избавляют от необходимости разбираться, как модифицировать проект. Для одной-двух функций это сделать не очень трудно, но изменения происходят по всей системе. Если предусматривать гибкость во всех этих местах, то вся система становится значительно сложнее и дороже в сопровождении. Весьма разочаровывает, конечно, то, что вся эта гибкость и не нужна. Потребуется лишь какая-то часть ее, но невозможно заранее сказать какая. Чтобы достичь гибкости, приходится вводить ее гораздо больше, чем требуется в действительности.

Рефакторинг предоставляет другой подход к рискам модификации. Возможные изменения все равно надо пытаться предвидеть, как и рассматривать гибкие решения. Но вместо реализации этих гибких решений следует задаться вопросом: «Насколько сложно будет с помощью рефакторинга преобразовать обычное решение в гибкое?» Если, как чаще всего случается, ответ будет «весьма несложно», то надо просто реализовать обычное решение. Рефакторинг позволяет создавать более простые проекты, не жертвуя гибкостью, благодаря чему процесс проектирования становится более легким и менее напряженным.

С рефакторингом обычно связан вопрос о его влиянии на производительность программы. С целью облегчения понимания работы программы часто осуществляется модификация, приводящая к замедлению выполнения программы. Это важный момент. Программное обеспечение часто отвергалось как слишком медленное, а более быстрые машины устанавливают свои правила игры. Рефакторинг, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более податливой для настройки производительности.

Секрет создания быстрых программ, если только они не предназначены для работы в жестком режиме реального времени, состоит в том, чтобы сначала написать программу, которую можно настраивать, а затем настроить ее так, чтобы достичь приемлемой скорости. М. Фаулер отмечает три подхода к написанию быстрых программ. Наиболее трудный из них связан с ресурсами времени и часто применяется в системах,

работающих в режиме реального времени. В этой ситуации при декомпозиции проекта каждому компоненту выделяется бюджет ресурсов (по времени и памяти).

Компонент не должен выйти за рамки своего бюджета, хотя разрешен механизм обмена временными ресурсами. Такой механизм жестко сосредоточен на соблюдении времени выполнения. Это важно в таких системах, как, например, кардиостимуляторы, в которых данные, полученные с опозданием, всегда ошибочны. Данная технология избыточна в системах другого типа, например, в корпоративных информационных системах.

Второй подход предполагает постоянное внимание. В этом случае каждый программист в любой момент времени делает все от него зависящее, чтобы поддерживать высокую производительность программы. Это распространенный и интуитивно привлекательный подход, однако он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет. Это распространенный и интуитивно привлекательный подход, однако, однако он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет работу с программой. Это замедляет создание программы. На это можно было бы пойти, если бы в результате получалось более быстрое программное обеспечение, но обычно этого не происходит.

С производительностью связано то интересное обстоятельство, что при анализе большинства программ обнаруживается, что большая часть времени расходуется небольшой частью кода (известное правило 80/20 – на 20% кода затрачивается 80% времени работы программы). Если в равной мере оптимизировать весь код, то окажется, что 90% оптимизации произведено впустую, потому что оптимизировался код, который выполняется не слишком часто. Время, ушедшее на ускорение программы, и время, потерянное из-за ее непонятности – все это израсходовано напрасно [15].

Третий подход к повышению производительности программы основан как раз на этой статистике. Он предполагает создание программы с достаточным разложением ее на компоненты без оглядки на достигаемую производительность вплоть до этапа оптимизации производительности, который обычно наступает на довольно поздней стадии разработки и на котором осуществляется особая процедура настройки программы. Начинается все с запуска программы под профайлером, контролирующим программу и сообщающим, где расходуются время и память.

Благодаря этому можно обнаружить тот небольшой участок программы, в котором находятся узкие места производительности. На этих

узких местах сосредоточиваются усилия, и осуществляется та же самая оптимизация, которая была бы применена при подходе с постоянным вниманием. Но благодаря тому, что внимание сосредоточено на выявленных узких местах, удается достичь больших результатов при значительно меньших затратах труда. Процесс поиска и ликвидации узких мест продолжается до достижения производительности, которая удовлетворяет пользователей.

Хорошее разделение программы на компоненты способствует оптимизации такого рода в двух отношениях. Во-первых, благодаря ему появляется время, которое можно потратить на оптимизацию. Имея хорошо структурированный код, можно быстрее добавлять новые функции и выиграть время для того, чтобы заняться производительностью. (Профилирование гарантирует, что это время не будет потрачено зря.) Во-вторых, хорошо структурированная программа обеспечивает более высокое разрешение для анализа производительности. Профайлер указывает на более мелкие фрагменты кода, которые легче настроить.

Благодаря большей понятности кода легче осуществить выбор возможных вариантов и разобраться в том, какого рода настройка может оказаться действенной. Таким образом, можно сделать вывод, что рефакторинг позволяет писать программы быстрее. На некоторое время он делает программы более медленными, но облегчает настройку программ на этапе оптимизации. В конечном счете, достигается большой выигрыш. он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет работу с программой. Это замедляет создание программы. На это можно было бы пойти, если бы в результате получалось более быстрое программное обеспечение, но обычно этого не происходит.

С производительностью связано то интересное обстоятельство, что при анализе большинства программ обнаруживается, что большая часть времени расходуется небольшой частью кода (известное правило 80/20 – на 20% кода затрачивается 80% времени работы программы). Если в равной мере оптимизировать весь код, то окажется, что 90% оптимизации произведено впустую, потому что оптимизировался код, который выполняется не слишком часто. Время, ушедшее на ускорение программы, и время, потерянное из-за ее непонятности – все это израсходовано напрасно [15].

Третий подход к повышению производительности программы основан как раз на этой статистике. Он предполагает создание программы с достаточным разложением ее на компоненты без оглядки на достигае-

мую производительность вплоть до этапа оптимизации производительности, который обычно наступает на довольно поздней стадии разработки и на котором осуществляется особая процедура настройки программы. Начинается все с запуска программы под профайлером, контролирующим программу и сообщающим, где расходуются время и память.

Благодаря этому можно обнаружить тот небольшой участок программы, в котором находятся узкие места производительности. На этих узких местах сосредоточиваются усилия, и осуществляется та же самая оптимизация, которая была бы применена при подходе с постоянным вниманием. Но благодаря тому, что внимание сосредоточено на выявленных узких местах, удается достичь больших результатов при значительно меньших затратах труда. Процесс поиска и ликвидации узких мест продолжается до достижения производительности, которая удовлетворяет пользователей.

Хорошее разделение программы на компоненты способствует оптимизации такого рода в двух отношениях. Во-первых, благодаря ему появляется время, которое можно потратить на оптимизацию. Имея хорошо структурированный код, можно быстрее добавлять новые функции и выиграть время для того, чтобы заняться производительностью. (Профилирование гарантирует, что это время не будет потрачено зря.) Во-вторых, хорошо структурированная программа обеспечивает более высокое разрешение для анализа производительности. Профайлер указывает на более мелкие фрагменты кода, которые легче настроить.

Благодаря большей понятности кода легче осуществить выбор возможных вариантов и разобраться в том, какого рода настройка может оказаться действенной. Таким образом, можно сделать вывод, что рефакторинг позволяет писать программы быстрее. На некоторое время он делает программы более медленными, но облегчает настройку программ на этапе оптимизации. В конечном счете, достигается большой выигрыш.

8.3. Когда применять рефакторинг

Дублирование кода. Если в программе есть одинаковые кодовые структуры в нескольких местах, при их объединении программа только выиграет. Простейшая задача с дублированием кода возникает, когда одно и то же выражение присутствует в двух методах одного и того же класса. В этом случае надо лишь применить *Выделение метода* (Extract Method) и вызывать код созданного метода из обеих точек.

Другая задача с дублированием встречается, когда одно и то же выражение есть в двух подклассах, находящихся на одном уровне. Устранить такое дублирование можно с помощью выделения метода для

обоих классов с последующим *Подъемом поля* (Pull Up Field). Если код похож, но не совпадает полностью, нужно применить выделение метода для отделения совпадающих фрагментов от различающихся. После этого может оказаться возможным применить *Формирование шаблона метода* (Form Template Method).

Если оба метода делают одно и то же с помощью разных алгоритмов, нужно выбрать более четкий из этих алгоритмов и применить *Замещение алгоритма* (Substitute Algorithm). Если дублирующийся код находится в двух разных классах, можно применить *Выделение класса* (Extract Class) в одном классе, а затем использовать новый компонент в другом. Бывает, что в действительности метод должен принадлежать только одному из классов и вызываться из другого класса, либо метод должен принадлежать третьему классу, на который будут ссылаться оба первоначальных. Необходимо решить, где оправдано присутствие этого метода, и обеспечить, чтобы он находился там и нигде более.

Длинный метод. Программы, использующие объекты, работают надежно, когда методы этих объектов короткие. Программистам, не имеющим опыта работы с объектами, часто кажется, что никаких вычислений не происходит, а программы состоят из нескончаемой цепочки делегирования действий. Однако, общаясь с такой программой на протяжении нескольких лет, становится ясным, какую ценность представляют собой маленькие методы. Уже на заре программирования стало ясно, что чем длиннее процедура, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с накладными расходами, которые удерживали от применения маленьких методов. Современные объектно-ориентированные языки в значительной мере устранили издержки вызовов внутри процесса.

Однако издержки сохраняются для того, кто читает код, поскольку приходится переключать переключать контекст, чтобы увидеть, чем занимается процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает устранить этот шаг, но главное, что способствует пониманию работы маленьких методов, это толковое присвоение им имен. Если правильно выбрать имя метода, нет необходимости изучать его тело. В итоге можно заключить, что следует активнее применять декомпозицию методов. Программисты придерживаются эвристического правила, гласящего, что, если ощущается необходимость что-то прокомментировать, надо написать метод.

В таком методе содержится код, который требовал комментариев, но его название отражает назначение кода, а не то, как он решает свою задачу. Такая процедура может применяться к группе строк или всего

лишь к одной строке кода. К ней прибегают даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода. Главным здесь является не длина метода, а семантическое расстояние между тем, что делает метод, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется лишь выделение метода. Нужно найти те части метода, которые кажутся согласованными друг с другом, и образовать новый метод. Когда в методе есть масса параметров и временных переменных, это мешает выделению нового метода. При попытке выделения метода в итоге приходится передавать столько параметров и временных переменных в качестве параметров, что результат оказывается ничуть не проще для чтения, чем оригинал.

Устранить временные переменные можно с помощью *Замены временной переменной вызовом метода* (Replace Temp with Query). Длинные списки параметров можно сократить с помощью приемов *Введение граничного объекта* (Introduce Parametr Object) и *Сохранение всего объекта* (Preserve Whole Object). Если даже после этого остается слишком много временных переменных и параметров, приходится выдвигать тяжелую артиллерию – необходима *Замена метода объектом метода* (Replace Method with Method Object).

Как определить те участки кода, которые должны быть выделены в отдельные методы? Хороший способ – поискать комментарии: они часто указывают на такого рода семантическое расстояние. Блок кода с комментариями говорит о том, что его действие можно заменить методом, имя которого основывается на комментарии. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях.

Условные операторы и циклы тоже свидетельствуют о возможности выделения. Для работы с условными выражениями подходит *Декомпозиция условных операторов* (Decompose Conditional). Если это цикл, следует выделить его и содержащийся в нем код в отдельный метод.

Большой класс. Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него атрибутов. А это может привести и к дублированию кода. Можно применить *Выделение класса* (Extract Class), чтобы связать некоторое количество атрибутов. Нужно так выбирать для компонента атрибуты, чтобы они имели смысл для каждого из них. Обычно одинаковые префиксы или суффиксы у некоторого подмножества переменных в классе наводят на мысль о создании компонента. Если разумно создание компонента как подкласса, то более простым оказывается *Выделение подкласса* (Extract Subclass).

Иногда класс не использует постоянно все свои переменные экземпляра. В таком случае оказывается возможным применить выделение класса или выделение подкласса несколько раз.

Как и класс с чрезмерным количеством атрибутов, класс, содержащий слишком много кода, создает хорошую среду для повторяющегося кода, хаоса и гибели. Простейшее решение – устранить избыточность в самом классе. Пять методов по сотне строк в длину иногда можно заменить пятью методами по десять строк плюс еще десять двухстрочных методов, выделенных из оригинала.

Как и для класса с большим числом атрибутов, обычное решение для класса с чрезмерным объемом кода состоит в том, чтобы применить выделение класса или выделение подкласса. Полезно установить, как клиенты используют класс, и применить *Выделение интерфейса* (Extract Interface) для каждого из этих вариантов. В результате может выясниться, как расчленить класс еще далее. Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. При этом может оказаться необходимым хранить копии некоторых данных в двух местах и обеспечить их согласованность. *Дублирование видимых данных* (Duplicate Observed Data) предлагает путь, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), можно в последующем удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров. Когда-то при обучении программированию рекомендовали все необходимые подпрограмме данные передавать в виде параметров. Это можно было понять, потому что альтернативой были глобальные переменные, а глобальные переменные часто пагубны и мучительны. Благодаря объектам ситуация изменилась, так как если какие-то данные отсутствуют, всегда можно попросить их у другого объекта. Поэтому, работая с объектами, следует передавать не все, что требуется методу, а столько, чтобы метод мог добраться до всех необходимых ему данных. Значительная часть того, что необходимо методу, есть в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

И это хорошо, потому что в длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании, а также потому, что их приходится изменять по мере того, как возникает необходимость в новых данных. Если передавать объекты, то изменений требуется мало, потому что для получения новых данных,

скорее всего, хватит пары запросов. *Замена параметра вызовом метода* (Replace Parameter with Method) уместна, когда можно получить данные в одном параметре путем вызова метода объекта, который уже переключать контекст, чтобы увидеть, чем занимается процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает устранить этот шаг, но главное, что способствует пониманию работы маленьких методов, это толковое присвоение им имен. Если правильно выбрать имя метода, нет необходимости изучать его тело. В итоге можно заключить, что следует активнее применять декомпозицию методов. Программисты придерживаются эвристического правила, гласящего, что, если ощущается необходимость что-то прокомментировать, надо написать метод.

В таком методе содержится код, который требовал комментариев, но его название отражает назначение кода, а не то, как он решает свою задачу. Такая процедура может применяться к группе строк или всего лишь к одной строке кода. К ней прибегают даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода. Главным здесь является не длина метода, а семантическое расстояние между тем, что делает метод, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется лишь выделение метода. Нужно найти те части метода, которые кажутся согласованными друг с другом, и образовать новый метод. Когда в методе есть масса параметров и временных переменных, это мешает выделению нового метода. При попытке выделения метода в итоге приходится передавать столько параметров и временных переменных в качестве параметров, что результат оказывается ничуть не проще для чтения, чем оригинал.

Устранить временные переменные можно с помощью *Замены временной переменной вызовом метода* (Replace Temp with Query). Длинные списки параметров можно сократить с помощью приемов *Введение граничного объекта* (Introduce Parametr Object) и *Сохранение всего объекта* (Preserve Whole Object). Если даже после этого остается слишком много временных переменных и параметров, приходится выдвигать тяжелую артиллерию – необходима *Замена метода объектом метода* (Replace Method with Method Object).

Как определить те участки кода, которые должны быть выделены в отдельные методы? Хороший способ – поискать комментарии: они часто указывают на такого рода семантическое расстояние. Блок кода с

комментариями говорит о том, что его действие можно заменить методом, имя которого основывается на комментарии. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях.

Условные операторы и циклы тоже свидетельствуют о возможности выделения. Для работы с условными выражениями подходит *Декомпозиция условных операторов* (Decompose Conditional). Если это цикл, следует выделить его и содержащийся в нем код в отдельный метод.

Большой класс. Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него атрибутов. А это может привести и к дублированию кода. Можно применить *Выделение класса* (Extract Class), чтобы связать некоторое количество атрибутов. Нужно так выбирать для компонента атрибуты, чтобы они имели смысл для каждого из них. Обычно одинаковые префиксы или суффиксы у некоторого подмножества переменных в классе наводят на мысль о создании компонента. Если разумно создание компонента как подкласса, то более простым оказывается *Выделение подкласса* (Extract Subclass).

Иногда класс не использует постоянно все свои переменные экземпляра. В таком случае оказывается возможным применить выделение класса или выделение подкласса несколько раз.

Как и класс с чрезмерным количеством атрибутов, класс, содержащий слишком много кода, создает хорошую среду для повторяющегося кода, хаоса и гибели. Простейшее решение – устранить избыточность в самом классе. Пять методов по сотне строк в длину иногда можно заменить пятью методами по десять строк плюс еще десять двухстрочных методов, выделенных из оригинала.

Как и для класса с большим числом атрибутов, обычное решение для класса с чрезмерным объемом кода состоит в том, чтобы применить выделение класса или выделение подкласса. Полезно установить, как клиенты используют класс, и применить *Выделение интерфейса* (Extract Interface) для каждого из этих вариантов. В результате может выясниться, как расчленить класс еще далее. Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. При этом может оказаться необходимым хранить копии некоторых данных в двух местах и обеспечить их согласованность. *Дублирование видимых данных* (Duplicate Observed Data) предлагает путь, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), можно в последующем удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров. Когда-то при обучении программированию рекомендовали все необходимые подпрограмме данные передавать в виде параметров. Это можно было понять, потому что альтернативой были глобальные переменные, а глобальные переменные часто пагубны и мучительны. Благодаря объектам ситуация изменилась, так как если какие-то данные отсутствуют, всегда можно попросить их у другого объекта. Поэтому, работая с объектами, следует передавать не все, что требуется методу, а столько, чтобы метод мог добраться до всех необходимых ему данных. Значительная часть того, что необходимо методу, есть в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

И это хорошо, потому что в длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании, а также потому, что их приходится изменять по мере того, как возникает необходимость в новых данных. Если передавать объекты, то изменений требуется мало, потому что для получения новых данных, скорее всего, хватит пары запросов. *Замена параметра вызовом метода* (Replace Parameter with Method) уместна, когда можно получить данные в одном параметре путем вызова метода объекта, который уже переключать контекст, чтобы увидеть, чем занимается процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает устранить этот шаг, но главное, что способствует пониманию работы маленьких методов, это толковое присвоение им имен. Если правильно выбрать имя метода, нет необходимости изучать его тело. В итоге можно заключить, что следует активнее применять декомпозицию методов. Программисты придерживаются эвристического правила, гласящего, что, если ощущается необходимость что-то прокомментировать, надо написать метод.

В таком методе содержится код, который требовал комментариев, но его название отражает назначение кода, а не то, как он решает свою задачу. Такая процедура может применяться к группе строк или всего лишь к одной строке кода. К ней прибегают даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода. Главным здесь является не длина метода, а семантическое расстояние между тем, что делает метод, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется лишь выделение метода. Нужно найти те части метода, которые кажутся согласованными друг с другом, и образовать новый метод. Когда в методе есть

масса параметров и временных переменных, это мешает выделению нового метода. При попытке выделения метода в итоге приходится передавать столько параметров и временных переменных в качестве параметров, что результат оказывается ничуть не проще для чтения, чем оригинал.

Устранить временные переменные можно с помощью *Замены временной переменной вызовом метода* (Replace Temp with Query). Длинные списки параметров можно сократить с помощью приемов *Введение граничного объекта* (Introduce Parametr Object) и *Сохранение всего объекта* (Preserve Whole Object). Если даже после этого остается слишком много временных переменных и параметров, приходится выдвигать тяжелую артиллерию – необходима *Замена метода объектом метода* (Replace Method with Method Object).

Как определить те участки кода, которые должны быть выделены в отдельные методы? Хороший способ – поискать комментарии: они часто указывают на такого рода семантическое расстояние. Блок кода с комментариями говорит о том, что его действие можно заменить методом, имя которого основывается на комментарии. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях.

Условные операторы и циклы тоже свидетельствуют о возможности выделения. Для работы с условными выражениями подходит *Декомпозиция условных операторов* (Decompose Conditional). Если это цикл, следует выделить его и содержащийся в нем код в отдельный метод.

Большой класс. Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него атрибутов. А это может привести и к дублированию кода. Можно применить *Выделение класса* (Extract Class), чтобы связать некоторое количество атрибутов. Нужно так выбирать для компонента атрибуты, чтобы они имели смысл для каждого из них. Обычно одинаковые префиксы или суффиксы у некоторого подмножества переменных в классе наводят на мысль о создании компонента. Если разумно создание компонента как подкласса, то более простым оказывается *Выделение подкласса* (Extract Subclass).

Иногда класс не использует постоянно все свои переменные экземпляра. В таком случае оказывается возможным применить выделение класса или выделение подкласса несколько раз.

Как и класс с чрезмерным количеством атрибутов, класс, содержащий слишком много кода, создает хорошую среду для повторяющегося кода, хаоса и гибели. Простейшее решение – устранить избыточность в самом классе. Пять методов по сотне строк в длину иногда

можно заменить пятью методами по десять строк плюс еще десять двухстрочных методов, выделенных из оригинала.

Как и для класса с большим числом атрибутов, обычное решение для класса с чрезмерным объемом кода состоит в том, чтобы применить выделение класса или выделение подкласса. Полезно установить, как клиенты используют класс, и применить *Выделение интерфейса* (Extract Interface) для каждого из этих вариантов. В результате может выясниться, как расчленить класс еще далее. Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. При этом может оказаться необходимым хранить копии некоторых данных в двух местах и обеспечить их согласованность. *Дублирование видимых данных* (Duplicate Observed Data) предлагает путь, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), можно в последующем удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров. Когда-то при обучении программированию рекомендовали все необходимые подпрограмме данные передавать в виде параметров. Это можно было понять, потому что альтернативой были глобальные переменные, а глобальные переменные часто пагубны и мучительны. Благодаря объектам ситуация изменилась, так как если какие-то данные отсутствуют, всегда можно попросить их у другого объекта. Поэтому, работая с объектами, следует передавать не все, что требуется методу, а столько, чтобы метод мог добраться до всех необходимых ему данных. Значительная часть того, что необходимо методу, есть в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

И это хорошо, потому что в длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании, а также потому, что их приходится изменять по мере того, как возникает необходимость в новых данных. Если передавать объекты, то изменений требуется мало, потому что для получения новых данных, скорее всего, хватит пары запросов. *Замена параметра вызовом метода* (Replace Parameter with Method) уместна, когда можно получить данные в одном параметре путем вызова метода объекта, который уже известен. Этот объект может быть полем или другим параметром. *Сохранение всего объекта* (Preserve Whole Object) позволяет взять группу данных, полученных от объекта, и заменить их самим объектом.

Если есть несколько элементов данных без логического объекта, выберите *Введение граничного объекта* (Introduce Parameter Object). Есть важное исключение, когда такие изменения не нужны. Оно касается ситуации, когда разработчик определенно не хочет создавать зависимость между вызываемым и более крупным объектами. В таких случаях разумно распаковать данные и передать их как параметры, но необходимо учесть, каких трудов это стоит. Если список параметров оказывается слишком длинным или модификации слишком частыми, следует пересмотреть структуру зависимостей.

Расходящиеся модификации. Программы структурируются, чтобы облегчить их модификацию. Программист хочет, чтобы при модификации можно было найти в системе одно определенное место и внести изменения именно туда. Если этого сделать не удается, то тут могут появиться две тесно связанные проблемы. Расходящиеся (divergent) модификации имеют место тогда, когда один класс часто модифицируется различными способами по разным причинам. Если, глядя на класс, разработчик отмечает для себя, что эти три метода придется модифицировать для каждой новой базы данных, а эти четыре метода придется модифицировать при каждом появлении нового финансового инструмента, это может означать, что вместо одного класса лучше иметь два.

Благодаря этому каждый класс будет иметь свою четкую зону ответственности и изменяться в соответствии с изменениями в этой зоне. Не исключено, что это обнаружится лишь после добавления нескольких баз данных или финансовых инструментов. При каждой модификации, вызванной новыми условиями, должен изменяться один класс, и вся типизация в новом классе должна выражать эти условия. Для того чтобы все это привести в порядок, определяется все, что изменяется по данной причине, а затем применяется выделение класса, чтобы объединить это все вместе.

Множественные изменения. Эту ситуацию М. Фаулер назвал «Стрельба дробью» [3]. Она похожа на расходящуюся модификацию, но является ее противоположностью. Увидеть ее можно, когда при выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов. Если изменения разбросаны повсюду, их трудно находить и можно пропустить важное изменение. В такой ситуации следует использовать *Перемещение метода* (Move Method) и *Перемещение поля* (Move Field), чтобы свести все изменения в один класс. Если среди имеющихся классов подходящего кандидата нет, нужно создать новый класс. Часто можно воспользоваться *Встраиванием класса* (Inline Class), чтобы поместить целую связку методов в

один класс. Возникнет какое-то число расходящихся модификаций, но с этим можно справиться.

Завистливые функции. Под такими функциями М. Фаулер понимает методы, которые больше интересуется не теми классами, в которых они находится, а какими-то другими. Чаще всего предметом зависти являются данные. Есть масса случаев, когда в программе сталкиваемся с методами, вызывающим полдюжины методов доступа к данным другого объекта. Изменение здесь очевидно: метод явно напрашивается на перевод в другое место, что и достигается *Перемещением метода* (Move Method). Иногда завистью страдает только часть метода; в таком случае к завистливому фрагменту применяется выделение метода.

Конечно, встречаются нестандартные ситуации. Иногда метод использует функции нескольких классов, так в который из них его лучше поместить? Нужно определить, в каком классе находится больше всего данных, и поместить метод вместе с этими данными. Иногда легче с помощью выделения метода разбить метод на несколько частей и поместить их в разные места. Есть несколько сложных схем, нарушающих это правило. Фундаментальное практическое правило гласит: то, что изменяется одновременно, надо хранить в одном месте. Данные и функции, использующие эти данные, обычно изменяются вместе, но бывают исключения. Наталкиваясь на такие исключения, мы перемещаем функции, чтобы изменения осуществлялись в одном месте.

Группы данных. Часто в программе одни и те же три-четыре элемента данных попадаются в множестве мест: поля в паре классов, параметры в нескольких сигнатурах методов. Связки данных, встречающихся совместно, надо превращать в самостоятельный класс. Сначала следует найти, где эти группы данных встречаются в качестве полей. Применяя к полям выделение метода, нужно преобразовать эти группы данных в класс. Затем обратить внимание на сигнатуры методов и применить *Введение граничного объекта* (Introduce Parameter Object) или *Сохранение всего объекта* (Preserve Whole Object), чтобы сократить их объем. В результате сразу удается укоротить многие списки параметров и упростить вызов методов.

Не стоит беспокоиться, что некоторые группы данных используют лишь часть полей нового объекта. Для проверки можно удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные. Если нет, это верный признак того, что данные напрашиваются на объединение их в объект. Сокращение списков полей и параметров несомненно улучшает код. После создания классов можно поискать

завистливые функции и обнаружить методы, которые желательно переместить в образованные классы.

Одержимость элементарными типами. В большинстве программных сред есть два типа данных. Тип «запись» позволяет структурировать данные в значимые группы. Элементарные типы данных служат стандартными конструктивными элементами. С записями всегда связаны некоторые накладные расходы. Они могут представлять таблицы в базах данных, но их создание может оказаться неудобным, если они нужны лишь в одном-двух случаях.

Один из ценных аспектов использования объектов заключается в том, что они затушевывают или вообще стирают границу между примитивными и большими классами. Нетрудно написать маленькие классы, неотличимые от встроенных типов языка. В Java есть примитивы для чисел, но строки и даты, являющиеся примитивами во многих других средах, суть классы.

Те, кто занимается ООП недавно, обычно неохотно используют маленькие объекты для маленьких задач. В качестве примера можно привести, например, денежные классы, соединяющие численное значение и валюту, специальные строки типа телефонных номеров и почтовых индексов. Выйти в мир объектов помогает рефакторинг *Замена значения данных объектом* (Replace Data Value with Object) для отдельных значений данных. Когда значение данного является кодом типа, можно обратиться к рефакторингу *Замена кода типа классом* (Replace Type Code with Class), если значение не воздействует на поведение.

Если есть условные операторы, зависящие от кода типа, может подойти *Замена кода типа подклассами* (Replace Type Code with Subclasses) или *Замена кода типа состоянием/ стратегией* (Replace Type Code with State / Strategy).

При наличии группы полей, которые должны находиться вместе, можно применить выделение класса. Увидев примитивы в списках параметров, можно воспользоваться *Введением граничного объекта* (Introduce Parameter Object). Если обнаружится разборка на части массива, можно попробовать *Замену массива объектом* (Replace Array with Object).

Операторы типа switch. Одним из признаков объектно-ориентированного кода служит сравнительная немногочисленность операторов типа switch (или case). Проблема, обусловленная применением switch, по существу, связана с дублированием. Часто один и тот же блок switch оказывается разбросанным по разным местам программы. При добавлении в переключатель нового варианта приходится искать все эти блоки

switch и модифицировать их. Понятие полиморфизма в ООП предоставляет элегантный способ справиться с этой проблемой.

Как правило, заметив блок switch, следует подумать о полиморфизме. Задача состоит в том, чтобы определить, где должен происходить полиморфизм. Часто переключатель работает в зависимости от кода типа. Необходим метод или класс, хранящий значение кода типа. Поэтому нужно воспользоваться выделением метода для выделения переключателя, а затем перемещением метода для вставки его в тот класс, где требуется полиморфизм. В этот момент следует решить, чем воспользоваться – заменой кода типа подклассами или заменой кода типа состоянием/стратегией. Определив структуру наследования, можно применить *Замену условного оператора полиморфизмом* (Replace Conditional with Polymorphism).

Если есть лишь несколько вариантов переключателя, управляющих одним методом, и не предполагается их изменение, то применение полиморфизма оказывается чрезмерным. В данном случае хорошим выбором будет *Замена параметра явными методами* (Replace Parameter with Explicit Method). Если одним из вариантов является null, можно попробовать прибегнуть к *Введению объекта Null* (Introduce Null Object).

Параллельные иерархии наследования. Параллельные иерархии наследования в действительности являются особым случаем «стрельбы дробью». В данном случае всякий раз при порождении подкласса одного из классов приходится создавать подкласс другого класса. Признаком этого служит совпадение префиксов имен классов в двух иерархиях классов. Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью перемещения метода и перемещения поля иерархия в ссылающемся классе исчезает.

Ленивый класс. Чтобы сопровождать каждый создаваемый класс и разобраться в нем, требуются определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданно в свое время, но уменьшившийся в результате рефакторинга. Либо это класс, добавленный для планировавшейся модификации, которая не была осуществлена. В любом случае следует дать классу возможность умереть. При наличии подклассов с недостаточными функциями можно попробовать *Свертывание иерархии* (Collapse Hierarchy). Почти бесполезные компоненты должны быть подвергнуты *Встраиванию класса* (Inline Class).

Теоретическая общность. Эффект такого кода возникает, когда говорят о том, что в будущем, наверное, потребуется возможность делать такие вещи, и хотят обеспечить набор механизмов для работы с вещами, которые не нужны. То, что получается в результате, труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданно, в противном случае они только мешают, поэтому лучше от них избавиться.

Если есть абстрактные классы, не приносящие большой пользы, от нужно них избавиться путем сворачивания иерархии. Ненужное делегирование можно устранить с помощью встраивания класса. Методы с неиспользуемыми параметрами должны быть подвергнуты *Удалению параметров* (Remove Parameter). Методы со странными абстрактными именами необходимо переименовать путем *Переименования метода* (Rename Method).

Теоретическая общность может быть обнаружена, когда единственными пользователями метода или класса являются контрольные примеры. Найдя такой метод или класс, нужно удалить его и контрольный пример, его проверяющий. Если есть вспомогательный метод или класс для контрольного примера, осуществляющий разумные функции, его, конечно, надо оставить.

Временное поле. Иногда обнаруживается, что в некотором объекте атрибут устанавливается только при определенных обстоятельствах. Такой код труден для понимания, поскольку естественно ожидать, что объекту нужны все его переменные. Трудно понять для чего существует некоторая переменная, когда не удается найти, где она используется. С помощью выделения класса можно создать класс и поместить туда весь код, работающий с этими переменными. Возможно, удастся удалить условно выполняемый код с помощью введения объекта Null для создания альтернативного компонента в случае недопустимости переменных.

Часто временные поля возникают, когда сложному алгоритму требуются несколько переменных. Тот, кто реализовывал алгоритм, не хотел пересылать большой список параметров, поэтому он разместил их в полях. Но поля действенны только во время работы алгоритма, а в другом контексте лишь вводят в заблуждение. Один из ценных аспектов использования объектов заключается в том, что они затушевывают или вообще стирают границу между примитивными и большими классами. Нетрудно написать маленькие классы, неотличимые от встроенных типов языка. В Java есть примитивы для чисел, но строки и даты, являющиеся примитивами во многих других средах, суть классы.

Те, кто занимается ООП недавно, обычно неохотно используют маленькие объекты для маленьких задач. В качестве примера можно привести, например, денежные классы, соединяющие численное значение и валюту, специальные строки типа телефонных номеров и почтовых индексов. Выйти в мир объектов помогает рефакторинг *Замена значения данных объектом* (Replace Data Value with Object) для отдельных значений данных. Когда значение данного является кодом типа, можно обратиться к рефакторингу *Замена кода типа классом* (Replace Type Code with Class), если значение не воздействует на поведение.

Если есть условные операторы, зависящие от кода типа, может подойти *Замена кода типа подклассами* (Replace Type Code with Subclasses) или *Замена кода типа состоянием/ стратегией* (Replace Type Code with State / Strategy).

При наличии группы полей, которые должны находиться вместе, можно применить выделение класса. Увидев примитивы в списках параметров, можно воспользоваться *Введением граничного объекта* (Introduce Parameter Object). Если обнаружится разборка на части массива, можно попробовать *Замену массива объектом* (Replace Array with Object).

Операторы типа switch. Одним из признаков объектно-ориентированного кода служит сравнительная немногочисленность операторов типа switch (или case). Проблема, обусловленная применением switch, по существу, связана с дублированием. Часто один и тот же блок switch оказывается разбросанным по разным местам программы. При добавлении в переключатель нового варианта приходится искать все эти блоки switch и модифицировать их. Понятие полиморфизма в ООП предоставляет элегантный способ справиться с этой проблемой.

Как правило, заметив блок switch, следует подумать о полиморфизме. Задача состоит в том, чтобы определить, где должен происходить полиморфизм. Часто переключатель работает в зависимости от кода типа. Необходим метод или класс, хранящий значение кода типа. Поэтому нужно воспользоваться выделением метода для выделения переключателя, а затем перемещением метода для вставки его в тот класс, где требуется полиморфизм. В этот момент следует решить, чем воспользоваться – заменой кода типа подклассами или заменой кода типа состоянием/стратегией. Определив структуру наследования, можно применить *Замену условного оператора полиморфизмом* (Replace Conditional with Polymorphism).

Если есть лишь несколько вариантов переключателя, управляющих одним методом, и не предполагается их изменение, то применение полиморфизма оказывается чрезмерным. В данном случае хорошим выбором будет *Замена параметра явными методами* (Replace Parameter with Explicit Method). Если одним из вариантов является null, можно попробовать прибегнуть к *Введению объекта Null* (Introduce Null Object).

Параллельные иерархии наследования. Параллельные иерархии наследования в действительности являются особым случаем «стрельбы дробью». В данном случае всякий раз при порождении подкласса одного из классов приходится создавать подкласс другого класса. Признаком этого служит совпадение префиксов имен классов в двух иерархиях классов. Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью перемещения метода и перемещения поля иерархия в ссылающемся классе исчезает.

Ленивый класс. Чтобы сопровождать каждый создаваемый класс и разобраться в нем, требуются определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданно в свое время, но уменьшившийся в результате рефакторинга. Либо это класс, добавленный для планировавшейся модификации, которая не была осуществлена. В любом случае следует дать классу возможность умереть. При наличии подклассов с недостаточными функциями можно попробовать *Свертывание иерархии* (Collapse Hierarchy). Почти бесполезные компоненты должны быть подвергнуты *Встраиванию класса* (Inline Class).

Теоретическая общность. Эффект такого кода возникает, когда говорят о том, что в будущем, наверное, потребуется возможность делать такие вещи, и хотят обеспечить набор механизмов для работы с вещами, которые не нужны. То, что получается в результате, труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданно, в противном случае они только мешают, поэтому лучше от них избавиться.

Если есть абстрактные классы, не приносящие большой пользы, от нужно них избавиться путем сворачивания иерархии. Ненужное делегирование можно устранить с помощью встраивания класса. Методы с неиспользуемыми параметрами должны быть подвергнуты *Удалению параметров* (Remove Parameter). Методы со странными абстрактными именами необходимо переименовать путем *Переименования метода* (Rename Method).

Теоретическая общность может быть обнаружена, когда единственными пользователями метода или класса являются контрольные примеры. Найдя такой метод или класс, нужно удалить его и контрольный пример, его проверяющий. Если есть вспомогательный метод или класс для контрольного примера, осуществляющий разумные функции, его, конечно, надо оставить.

Временное поле. Иногда обнаруживается, что в некотором объекте атрибут устанавливается только при определенных обстоятельствах. Такой код труден для понимания, поскольку естественно ожидать, что объекту нужны все его переменные. Трудно понять для чего существует некоторая переменная, когда не удается найти, где она используется. С помощью выделения класса можно создать класс и поместить туда весь код, работающий с этими переменными. Возможно, удастся удалить условно выполняемый код с помощью введения объекта Null для создания альтернативного компонента в случае недопустимости переменных.

Часто временные поля возникают, когда сложному алгоритму требуются несколько переменных. Тот, кто реализовывал алгоритм, не хотел пересылать большой список параметров, поэтому он разместил их в полях. Но поля действенны только во время работы алгоритма, а в другом контексте лишь вводят в заблуждение. В таком случае можно применить выделение класса к переменным и методам, в которых они требуются. Новый объект является объектом метода.

Цепочки сообщений. Цепочки сообщений появляются, когда клиент запрашивает у одного объекта другой, у которого клиент запрашивает еще один объект, у которого клиент запрашивает еще один объект и т. д. Это может выглядеть как длинный ряд методов getThis или последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей означают необходимость модификации клиента.

Здесь применяется прием *Сокрытие делегирования* (Hide Delegate). Это может быть сделано в различных местах цепочки. В принципе, можно делать это с каждым объектом цепочки, что часто превращает каждый промежуточный объект в посредника. Обычно лучше посмотреть, для чего используется конечный объект. Можно попробовать с помощью выделения метода взять использующий его фрагмент кода и путем перемещения метода передвинуть его вниз по цепочке. Если несколько клиентов одного из объектов цепочки желают пройти остальную часть пути, добавьте метод, позволяющий это сделать.

Посредник. Одной из главных характеристик объектов является инкапсуляция – сокрытие внутренних деталей от внешнего мира. Инкапсуляции часто сопутствует делегирование. Однако это может завести слишком далеко. Например, рассмотрев интерфейс класса, обнаруживаем, что половина методов делегирует обработку другому классу. Тут надо воспользоваться *Удалением посредника* (Remove Middle Man) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, не выполняющих большой работы, с помощью встраивания метода следует поместите их в вызывающий метод. Если есть дополнительное поведение, то с помощью *Замены делегирования наследованием* (Replace Delegation with Inheritance) можно преобразовать посредника в подкласс реального класса. Это позволит расширить поведение, не гонясь за всем этим делегированием.

Неуместная близость. Иногда классы оказываются в слишком близких отношениях и чаще, чем следовало бы, погружены в закрытые части друг друга. Классы должны следовать строгим правилам. Чрезмерно взаимосвязанные классы нужно разводить с помощью перемещения метода и перемещения поля, разделить части и уменьшить близость. Следует посмотреть, нельзя ли прибегнуть к *Замене двунаправленной связи однонаправленной* (Change Bidirectional Association to Unidirectional). Если у классов есть общие интересы, можно воспользоваться выделением класса, чтобы поместить общую часть в надежное место и превратить их в добропорядочные классы. Либо можно воспользоваться сокрытием делегирования, позволив выступить в качестве связующего звена другому классу.

К чрезмерной близости может приводить наследование. Подклассы всегда знают о своих родителях больше, чем последним хотелось бы. Если пришло время расстаться с домом, примените *Замену наследования делегированием* (Replace Inheritance with Delegation).

Альтернативные классы с разными интерфейсами. В этих случаях можно использовать переименование метода ко всем методам, выполняющим одинаковые действия, но различающимся сигнатурами. Часто этого оказывается недостаточно. В таких случаях классы еще недостаточно деятельны. Нужно продолжить применять перемещение метода для передачи поведения в классы, пока протоколы не станут одинаковыми. Если для этого приходится осуществить избыточное перемещение кода, можно попробовать компенсировать это *Выделением родительского класса* (Extract Superclass).

Неполнота библиотечного класса. Повторное использование кода часто рекламируется как цель применения объектов. Значение этого аспекта переоценивается (достаточно простого использования). Не следует отрицать, однако, что программирование во многом основывается на применении библиотечных классов. Разработчики библиотечных классов не всеведущи, и их не следует осуждать за это. Проблема в том, что часто считается дурным тоном и обычно оказывается невозможным модифицировать библиотечный класс, чтобы он выполнял какие-то желательные действия. Это означает, что испытанная тактика вроде перемещения метода оказывается бесполезной.

Для этой работы есть пара специализированных инструментов. Если в библиотечный класс надо включить всего лишь один-два новых метода, можно выбрать *Введение внешнего метода* (Introduce Foreign Method). Если дополнительных функций достаточно много, необходимо применить *Введение локального расширения* (Introduce Local Exten-sion).

Классы данных. Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Это – бессловесные хранилища данных, которыми другие классы наверняка манипулируют излишне обстоятельно. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока никто их не обнаружил, применить *Инкапсуляцию поля* (Incapsulate Field). При наличии полей коллекций нужно проверить, инкапсулированы ли они должным образом, и, если нет, применить *Инкапсуляцию коллекции* (Incapsulate Collection). Применить *Удаление метода установки значения* (Remove Setting Method) ко всем полям, значение которых не должно изменяться.

Посмотреть, как эти методы доступа к полям используются другими классами. Далее попробовать с помощью перемещения метода переместить методы доступа в класс данных. Если метод не удается переместить целиком, обратиться выделению метода, чтобы создать такой метод, который можно переместить. Через некоторое время можно начать применять сокрытие метода к методам получения и установки значений полей.

Отказ от наследования. Подклассам полагается наследовать методы и данные своих родителей. Но как быть, если наследование им не требуется? Обычная причина этого – неправильно задуманная иерархия. Необходимо создать новый класс на одном уровне с потомком и с помощью *Спуска метода* (Push Down Method), *Спуска поля* (Push Down Field) вытолкнуть в него все бездействующие методы. Благодаря этому

в родительском классе будет содержаться только то, что используется совместно.

Часто встречается совет делать все родительские классы абстрактными. М. Фаулер этого не советует, и замечает, по крайней мере, это годится не на все случаи жизни. С другой стороны, он считает нормальным стилем работы создание подклассов для повторного использования части функций. Также отмечает, что, если с не принятым наследством связаны какие-то проблемы, стоит следовать обычному совету. Однако не следует думать, что это надо делать всегда. Если подкласс повторно использует функции родительского класса, но не желает поддерживать его интерфейс, то нет возражений против отказа от реализаций, но от интерфейса отказываться не следует. В этом случае не надо возиться с иерархией; ее надо разрушить с помощью замены наследования делегированием.

Комментарии. Просто удивительно, как часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой. По мнению М. Фаулера, после рефакторинга комментарии часто оказываются ненужными. Если для объяснения действий блока требуется комментарий, он рекомендует применить выделение метода. Если метод уже выделен, но по-прежнему нужен комментарий для объяснения его действия, воспользуйтесь переименованием метода. А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените *Введение утверждения* (Introduce Assertion).

Почувствовав потребность написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали излишними. Комментарии полезны, когда программист не знает, как поступить. Помимо описания происходящего, комментарии могут отмечать те места, в которых программист не уверен. Правильным будет поместить в комментарии обоснование своих действий. Это пригодится тем, кто будет модифицировать код в будущем.

8.4. Уровни рефакторинга

По определению, рефакторинг – процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Иногда понятия рефакторинга заключается в определении регламентированного способа реструктуризации кода с применением небольших итерационных шагов. Прежде всего, рефакторинг обеспечивает постепенное развитие кода во вре-

мени, в результате чего реализуется эволюционный подход к программированию. Особенностью рефакторинга является то, что он сохраняет функциональную семантику базового кода. Многие разработчики, организующие свою работу на принципах адаптивного программирования, и, в частности, как отмечено выше, специалисты по экстремальному программированию полагают, что рефакторинг является ведущим подходом к разработке. На практике можно столь же часто встретить примеры применения рефакторинга небольших фрагментов кода, как и введения операторов if или циклов [15].

Первым уровнем рефакторинга можно считать такое изменение кода, которое не затрагивает структуру классов (количество и взаимосвязи, интерфейсы) объектно-ориентированной программной системы, т.е. рефакторингу подвергается программный код внутри классов. Это могут быть методы и алгоритмы, реализуемые методами, поля и т.п. Многие разработчики программного обеспечения считают, что при рефакторинге лучше полагаться на интуицию, основанную на опыте, но можно выделить наиболее очевидные причины, когда код нужно подвергнуть процессу рефакторинга. Сюда относится дублирование фрагментов кода, длинный метод реализации, использование достаточно длинного списка параметров, применение и использование избыточных временных переменных, изменение сигнатуры метода, инкапсуляция поля, замена условного оператора полиморфизмом и т.п.

Второй уровень рефакторинга относится к изменению структуры классов программной системы, добавлению новых классов, выделению и разбиению больших классов, встраиванию классов, переносу или добавлению новых методов, выделению интерфейсов, сокрытию делегирования и др. Основными стимулами его проведения являются следующие задачи:

- необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение программного модуля;
- необходимо исправить ошибку, причины возникновения которой не выделены четко структурированной базовой внешней формой;
- проблематика в командной разработке, которая обусловлена сложностью логики программного продукта.
- К третьему уровню рефакторинга (большому рефакторингу) можно отнести:
- разделение наследования (Tease Apart Inheritance) в ситуации, когда в программе запутанная иерархия наследования, в которой различные варианты представляются объединенными вместе так, что это сбивает с толку;

• преобразование процедурного проекта в объекты (Convert Procedural Design to Object), что содействует решению классической проблемы преобразования процедурного кода;

• отделение предметной области от представления (Separate Domain from Presentation), которое используется для разделения бизнес-логики и кода интерфейса пользователя. Опытные объектно-ориентированные разработчики поняли, что такое разделение жизненно важно для долго живущих систем;

• выделение иерархии (Extract Hierarchy), упрощающей чрезмерно сложные классы путем превращения их в группы подклассов.

Рефакторинг архитектуры программных систем является четвертым уровнем рефакторинга. Трудно представить сегодня серьезное коммерческое приложение, не взаимодействующее с базой данных (БД), что подразумевает уровень архитектуры, обеспечивающий взаимодействие с БД. Рефакторинг этого уровня – довольно трудоемкий процесс.

Операции рефакторинга архитектуры программной системы и БД концептуально являются более сложными, чем операции рефакторинга кода, поскольку при проведении операций рефакторинга кода необходимо заботиться лишь о сохранении функциональной семантики структуры фрагментации, а при осуществлении операций рефакторинга БД возникает процесс необходимости и сохранение информационной семантики. Достаточно важным фактором является то, что усложнение операций рефакторинга БД может быть обусловлено наличием большого количества связей, поддерживаемых архитектурой БД. Рефакторинг кода может быть связан не только с изначальной архитектурной проблематикой. Процесс необходимо осуществлять после комплексного анализа структуры и выявления «фрагментарных участков, необходимых для оптимизации со стороны команды разработчиков». В этом случае проводится рефакторинг «фрагментарного» участка программы. Профилировка поможет его определить.

При осуществлении процесса рефакторинга необходимо четко представлять, какой из методов более лучший и оптимальный для данной структуры, ведь нелогичное использование методов приведет к трудностям работы программы. Опытные разработчики четко представляют и делят процесс оптимизации кода и процесс рефакторинга. При процессе рефакторинга разработчик старается сделать так, чтобы код стал удобнее для понимания и его поддержки, а при оптимизации кода приходится делать процедуры, которые приводят к обратному эффекту, что приводит к проблемам читаемости кода, но за счет этого возрастает скорость его выполнения.

8.5. Методы рефакторинга

8.5.1. Основные методы

К наиболее часто употребляемым методам рефакторинга можно отнести [16]:

- изменение сигнатуры метода (Change Method Signature);
- инкапсуляция поля (Encapsulate Field);
- выделение класса (Extract Class);
- выделение интерфейса (Extract Interface);
- выделение локальной переменной (Extract Local Variable);
- выделение метода (Extract Method);
- генерализация типа (Generalize Type);
- встраивание (Inline);
- введение фабрики (Introduce Factory);
- введение параметра (Introduce Parameter);
- поля/метода (Pull Up);
- спуск поля/метода (Push Down);
- замена условного оператора полиморфизмом (Replace Conditional with Polymorphism).

Пожалуй, наилучшие рекомендации и практические советы по описанию и иллюстрации использования методов рефакторинга программного кода в объектно-ориентированных программах можно найти в монографии М. Фаулера [3].

Рефакторинг кода должен осуществляться до полного исчерпания его возможностей, поскольку наибольшая производительность может быть достигнута только в условиях работы с исходным кодом максимально высокого качества. При возникновении необходимости добавить к коду новые возможности следует оценить качество реплицируемого кода в аспектах данного проекта, что позволит успешно реализовать требуемые средства. Если ответ на этот вопрос является положительным, то можно приступать к добавлению новых функциональных средств.

При отрицательном решении данного аспектного внедрения код вначале должен быть подвергнут рефакторингу, для того чтобы он имел оптимальный вариант, и только после этого возможно осуществление процесса добавления новых функций. Применение указанного подхода приводит к значительному увеличению объема работы, но практика показывает, что если доработка начинается с высококачественного исходного кода, после чего постоянно осуществляется рефакторинг этого кода для поддержки его в том же состоянии, то все новые замыслы реализуются чрезвычайно показательно и эффективно.

Читателям, желающим получить более подробную информацию и рассмотреть конкретные примеры рефакторинга, рекомендуется обратиться к литературе, перечень которой дан в конце главы.

8.5.2. Формализация процесса рефакторинга на основе символьной записи структуры классов

В большом количестве опубликованных работ рассматривается использование UML-диаграмм для решения задачи синтеза и декларативные подходы к решению задачи анализа объектно-ориентирован-ных программных систем. Однако решение задачи анализа на основе UML не формализовано, а сами диаграммы дает небольшие возможности выполнять формальные преобразования над классами с целью рефакторинга проекта. Исключением в определенной части являются инструментальные средства IBM Rational. Один из возможных подходов к автоматизации – процессов рефакторинга предложен в работе [8].

Автор работы на основе ряда публикаций делает вывод, что класс – это триплет вида $C = \{P_v, P_t, P_b\}$, где P_v, P_t, P_b – множество частных, защищенных и публичных данных и методов класса. С другой стороны, основываясь на свойстве инкапсуляции, класс – это пара $C = \{F, M\}$, где F – множество полей, а M – множество методов класса.

Таким образом, $C = \{PvF \cup PvM, PtF \cup PtM, PbF \cup PbM\}$, (8.1)

где PvF, PtF, PbF – множество частных, защищенных и публичных полей класса, PvM, PtM, PbM - множество частных, защищенных и публичных методов класса.

Очевидно, что $PvF \in F$, $PtF \in F$, $PbF \in F$, $PvM \in M$, $PtM \in M$, $PbM \in M$, $PvF \cap PtF = \emptyset$, $PvF \cap PbF = \emptyset$, $PtF \cap PbF = \emptyset$,

$PvM \cap PtM = \emptyset$, $PvM \cap PbM = \emptyset$, $PvF \cup PtF \cup PbF = F$, $PvM \cup PtM \cup PbM = M$.

Для описания структуры класса в виде символической записи предлагается следующая нотация. Символ, обозначающий атрибут класса, записывается как N^{type}, где N - имя поля, а type – тип поля. Метод класса предлагается записывать как $Meth_{prim}^{type}(param)$, где *Meth* – идентификатор метода, *param* – перечень формальных параметров, а $prim \subseteq [virtual, abstract, ovrride]$. Допустимо, что *prim* = ∅ и/или *param* = ∅. Класс определяется как $Cname_{Cprim}$, где $Cprim = abstract|\emptyset$.

Синтаксис символьной записи структуры класса определяется грамматикой, представленной в табл. 8.1.

Таблица 8.1

Грамматика символьной записи структуры класса

<*ClassDare*>→<*CN*>=<*CNbase*>+{[<*PrvF*>,<*PrvM*>];[<*PrtF*>,<*PrtM*>]; [<*PubF*>,<*PubM*>]}	
<*CN*>→<*id*><*Cprim*>	<*FIELD*>→<*id*><*type*>
< PrvF>→<FIELDS>	<FIELD>→ Ø
< PrtF>→<FIELDS>	<METHODS>→<METHOD>
<PubF>→<FIELDS> <FIELDS>→ <FIELD>	<METHODS>→<METHOD>, <METHODS>
< PrvM >→<METHODS>	<METHOD>→ <id><type><prim><param>()
< PrtM >→<METHODS>	<Cprim>→abstarct|Ø
< PubM >→<METHODS>	<prim>→ virtual|abstract|override|Ø
<FIELDS>→ |<FIELD>,<FIELDS>	<CNbase>→<id>+|Ø

Таким образом, в символьном виде описание нового класса newClass, порожденного от базового класса *baseCalss* это выражение вида *newClass = baseClass* +{[*PvF*, *PvM*], [*PtF*, *PtM*], [*PbF*, *PbM*]}.

Рассмотрим пример символьной записи структуры классов, показанной в виде UML-диаграммы на рис. 8.1.

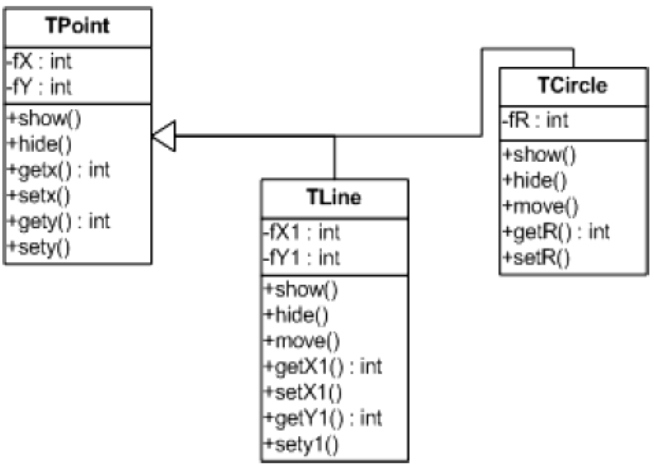


Рис. 8.1. UML-диаграмма представления классов геометрических

Фигур

$$TPoint = \begin{Bmatrix} [fX^{\text{int}}, fY^{\text{int}}],[],[show_{virtual}(), hide_{virtual}(), getX^{\text{int}}_{virtual}(), \\ setX_{virtual}(newX^{\text{int}}), getY^{\text{int}}_{virtual}(), setY_{virtual}(newY^{\text{int}})] \end{Bmatrix},$$

$$TLine = Tpoint + \begin{Bmatrix} [fX1^{\text{int}}, fY1^{\text{int}}],[],[show_{override}(), hide_{override}(), \\ getX1^{\text{int}}_{virtual}(), setX1_{virtual}(newX1^{\text{int}}), getY1^{\text{int}}_{virtual}(), \\ setY1_{virtual}(newY1^{\text{int}})] \end{Bmatrix},$$

$$TCircle = Tpoint + \begin{Bmatrix} [fR^{\text{int}}],[],[show_{override}(), hide_{override}(), getR^{\text{int}}_{virtual}(), \\ setR_{virtual}(newR^{\text{int}})] \end{Bmatrix}.$$

Очевидно, что символьная запись структуры классов компактнее соответствующей ей UML-диаграммы. Определим на основании предложенной символьной записи операции над классами для решения задачи анализа структуры проекта и его последующего рефакторинга. Определим для двух произвольных классов $C1$ и $C2$ операцию пересечения, которую будем записывать как $C1 \cap C2$. Введем в рассмотрение функцию $I(\chi_1, \chi_2) = [\chi_1 \backslash (\chi_1 \cap \chi_2), \chi_2 \backslash (\chi_1 \cap \chi_2)]$. Пусть операция $C1 \cap C2$ в соответствии со спецификацией (1) определяется как:

$$baseC = \begin{Bmatrix} [PvF_{C1} \cap PvF_{C2}, PvM_{C1} \cap PvM_{C2}], \\ [PtF_{C1} \cap PtF_{C2}, PtM_{C1} \cap PtM_{C2}], \\ [PbF_{C1} \cap PbF_{C2}, PbM_{C1} \cap PbM_{C2}] \end{Bmatrix}$$

$$C1 \cap C2 = baseC + \begin{Bmatrix} [I(PvF_{C1}, PvF_{C2}), I(PvM_{C1}, PvM_{C2})], \\ [I(PtF_{C1}, PtF_{C2}), I(PtM_{C1}, PtM_{C2})], \\ [I(PbF_{C1}, PbF_{C2}), I(PbM_{C1}, PbM_{C2})], \end{Bmatrix}$$

Базовый класс baseC – это ближайший общий предок в дереве иерархии или единый базовый класс BaseObject (NSObject, java.lang.Object, System.Object, TObject и т. п.), если такого не существует. Если реализация методов с одинаковым объявлением PvM, PtM, PbM различна для операндов операции пересечения, то в результате пересечения они получают дополнительный префикс abstract. Проиллюстрируем операцию

строгого пересечения на примере UML-диаграммы классов, представленных на рис. 8.2.

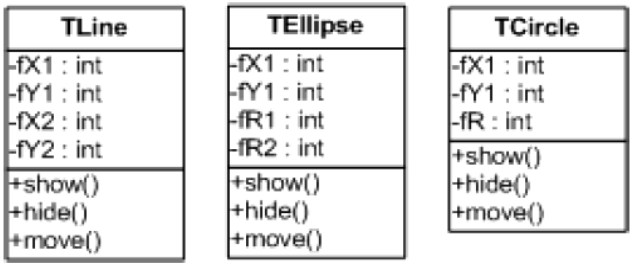


Рис. 8.2. UML-диаграмма классов без общего предка

$$TEllipse \cap TLin = BaseObject + \left\{ \begin{matrix} [fX1^{int}, fY1^{int}], [], \\ [show_{virtual,adstract}(),] \\ hide_{virtual,adstract}(), more()] \end{matrix} \right\}$$

$$TEllipse \cap TCircle = TEllipse \cap TLine,$$

$$TLine \cap TCircle = TEllipse \cap TLine.$$

Результатом пересечения классов, как правило, является новый класс или общий предок классов, заявленных в качестве операндов. Таким же образом определяем операцию вычитания классов C1 \ C2:

$$C1 \backslash C2 = \left\{ \begin{matrix} [PvF_{C1} \backslash PvF_{C2}, PvM_{C1} \backslash PvM_{C1} + \forall_{Mi \in C1} Mi_{abstract}] \\ [PtF_{C1} \backslash PtF_{C2}, PtM_{C1} \backslash PtM_{C2} + \forall_{Mi \in C1} Mi_{abstract}] \\ [PbF_{C1} \backslash PbF_{C2}, PbM_{C1} \backslash PbM_{C1} + \forall_{Mi \in C1} Mi_{abstract}] \end{matrix} \right\}$$

Операция вычитания есть традиционное вычитание множеств полей и методов класса с учетом необходимости перегрузки абстрактных мдов. Образование новых классов в результате выполнения операции пересечения является основой для процесса рефакторинга, описанного в [28]. При этом следует действовать по следующему формальному алгоритму вне зависимости от лингвистических сред и особенностей реализации.

Шаг 1. Выполнить преобразование в символьную запись всех объявлений классов программной системы.

Шаг 2. Для всех классов, имеющих общего предка в дереве иерархии выполняется операция строгого пересечения:

$$TParentObject = \cap^{i} C_{i}$$

Шаг 3. Если результат операции пересечения не равен $TParentObject \neq BaseObject$, определяем новый класс предка.

Шаг 4. Выполняем рефакторинг кода путем вычитания из каждого класса C_i класса *TParentObject*.

Рассмотрим пример рефакторинга программных классов, представленных на рисунке 8.2.

$$TFigure = BaseObject + \left\{ \begin{bmatrix} fX1^{\text{int}}, fY1^{\text{int}} \end{bmatrix}, [\,], \\ \left[show_{virtua,abstractl}(), hide_{virtual,abstract}(), move() \right] \right\},$$

$$TLine = TFigure + \{[fX2^{\text{int}}, fY2^{\text{int}}], [], [show_{override}(), hide_{override}()]\},$$

$$TEllipse = TFigure + \{[fR1^{\text{int}}, fR2^{\text{int}}], [], [show_{override}(), hide_{override}()]\},$$

$$TCircle = TFigure + \{[fR^{\text{int}}], [], [show_{override}(), hide_{override}()]\}.$$

Таким образом, формальный подход к решению задачи рефакторинга программных систем на основе символьной записи структуры классов является основой для построения программного продукта, позволяющего выполнять автоматизированный рефакторинг архитектуры программных систем практически без участия программиста.

8.6. Архитектурный рефакторинг. Архитектурные паттерны

8.6.1. Когда нужен архитектурный рефакторинг

Потребность в изменении существующей программной системы может возникнуть в ходе решения широкого круга задач по ее модернизации [14, 15]. В общем случае изменения существующей программной системы затрагивают не только программный код, но и все остальные артефакты, связанные с трансформируемой программной системой. Часто существенным является изменение архитектуры программной системы. В качестве примеров можно привести следующие сценарии, требующие изменения архитектуры существующей ПС.

1. Преобразования, обусловленные функциональными изменениями ПС. Желательно, чтобы внедрение новой функциональности не затронуло существующую логику системы. Также желательно, чтобы сложность внедрения новой функциональности в существующую систему не превышала существенным образом сложность реализации этой функциональности в рамках нового проекта. Изменение существующей архитектуры – хороший шаг на пути внедрения новой функциональности, облегчающий и дальнейшую эволюцию системы.

2. Смена платформы ПС. Смена платформы ПС (как аппаратной, так общего программного обеспечения, в частности операционной системы) должна минимально затрагивать существующий код. Желательно ограничиться изменениями только в узкой платформенно-зависимой прослойке системы. Выделение такой прослойки – архитектурная задача. Ее решение всегда сопряжено с необходимостью изменения архитектуры.

3. Обновление технологии разработки программного продукта, связанное, например, с переходом компонентное программирование, внедрением комплексной среды коллективной разработки с возможностями гибкого, формального и смешанного планирования и ведения отчетности, доступными на одной платформе, например, IBM Rational Team Concert .

4. Преобразования, связанные с реорганизацией компании, ведущей разработку. Преобразования, связанные с реорганизацией компании, ведущей разработку. Примером, такой реорганизации может стать аутсорсинг. Решение об использовании аутсорсинга – типичный шаг по оптимизации производства. К сожалению, этот шаг зачастую затрудняется проблемой выделения и передачи компонентов для внешней разработки. Изменение архитектуры программной системы способно облегчить решение этой задачи.

При описании методов рефакторинга принято использовать частично формализованный формат – шаблон или паттерн [1, 7]. Это повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться [15]. «Низко-

уровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные. На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы.

Любой паттерн описывает и именует типовую задачу, которая постоянно возникает в работе, а также принцип ее решения, причем таким образом, что это решение можно использовать потом снова и снова. Паттерн именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения. Помимо прочего, паттерны формируют словарь в проблемной области и позволяют двум специалистам в этой области именовать типовые решения и понимать друг друга, не объясняя каждый раз суть самих решений.

Например, в [3] паттерн имеет следующую структуру:

- название паттерна;
- краткая сводка ситуаций, в которых требуется данный метод;
- мотивировка применения;
- пошаговое описание применения метода рефакторинга.

Известно, что рефакторинг объектно-ориентированного кода зарекомендовал себя как эффективный способ решения задач эволюции и сопровождения программ. Однако, и это отмечается в статьях М. Ксензова [17,18], опубликованных в 2004 году, практически не существует исследований, освещающих рефакторинг на более высоком уровне абстракции – уровне архитектуры ПО. Ситуация мало изменилась и к настоящему времени. Поэтому вызывает интерес, возможен ли перенос данной методологии на более высокий уровень абстракции, с целью получения аналогичной методики систематического изменения архитектуры ПО?

Специфика исследования и трансформации архитектуры программного обеспечения заключается в том, что архитектура не имеет явного представления, за исключением, может быть, тех случаев, когда она явно задокументирована. Однако даже в последнем, случае трудно гарантировать соответствие задокументированной архитектуры фактической высокоуровневой логической структуре, которая на самом деле существует. Способом описания архитектуры и ее изменений могут стать структурные модели. В настоящее время существует большое количество нотаций и инструментов, поддерживающих структурное моделирование программных систем. В свете соответствия модели фактической струк-

туре существующего кода представляется исключительно важной возможность автоматического извлечения таких моделей из кода программных систем, поскольку в этом случае гарантирует их точность.

8.6.2. Построение архитектуры ПС по ее программному коду

Данной задаче посвящен ряд работ. Одной из них, наиболее интересной и универсальной по предлагаемому подходу является статья В. Миронова [16]. В этой работе рассматривается технология анализа программ большого размера на основе построения графов, что обеспечивает навигацию по исходным файлам с показом необходимых свойств (например, используемых классов, методов, функций и др.). Предварительно стоит рассмотреть возможные подходы и инструментальные средства анализа и формального представления текстов программ.

1. Язык UML, предназначенный для определения, визуализации, конструирования и документирования артефактов программных систем. Он позволяет в объектно-ориентированном виде описать систему практически со всех возможных точек зрения, в том числе и разные аспекты поведения системы. Основной недостаток – неточная семантика. Язык UML предназначен для графического представления системы на среднем и высоком уровнях, но малоэффективен для описания детальной логики программ нижнего уровня.

2. CASE-технологии – программные комплексы, автоматизирующие технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем. CASE-технологии использовались в реинжиниринге с момента их появления. Однако по своей природе CASE-средства малопригодны для низкоуровневого описания структуры ПС. Кроме того, они предполагают рассмотрение ПС со стороны организации бизнес-процессов, потоков и создания оптимальной ИС для предприятия. Несмотря на высокие потенциальные возможности CASE-технологии далеко не все разработчики ИС, использующие CASE-средства, достигают ожидаемых результатов.

3. Комплекс программ построения графов Graphviz [19], разработанный специалистами лаборатории AT&T, представляет собой пакет утилит по автоматической визуализации графов, заданных в виде описания на языке «dot». В пакет утилит Graphviz входит автоматический визуализатор «dot» для формирования ориентированных графов на основе входного текстового описания (на специальном языке описания объектов, связей и их характеристик) в виде графического, векторного или текстового файла. Данная технология является вспомогательным

средством, обеспечивающим поддержку оптимального расположения вершин графа.

4. Kscope – программа для исследования и редактирования исходных текстов больших проектов на языке C9 (сейчас KScope не поддерживается. Прошлые релизы и репозитарий исходного кода еще доступны на SourceForge.net). Программа позволяет построить граф-дерево вызовов, который наглядно показывает отношения между функциями. В Kscope используется свой подход к процессу визуализации: составление и вызов запроса. Недостатками данного подхода является малая информативность (очень сложно увидеть на графе всю программу целиком) и ограниченность синтаксисом языка C.

Рассмотрим принципы подхода к решению задачи построения архитектуры ПС по ее программному коду [16]. Одним из способов проведения анализа является формирование графов с различным уровнем отображения внутренних объектов и связей. Модель данных большой программной системы, реализованной на языках программирования C/C++, представляет собой наборы директорий, содержащих различные по использованию файлы: с исходными кодами, с описанием типов объектов и вспомогательные. Основные принципы построения архитектуры ПС:

- отображение графа связей файлов проекта. Показывает, каким образом исходные файлы проекта подключают друг друга;
- отображение иерархии наследования классов;
- отображение диаграммы вызовов функций. Диаграмма показывает, каким образом управление попадает в выбранную функцию, и куда оно передается из нее. Связь на диаграмме соответствует вызову функции;
- регулирование объема отображаемой информации путем выделения любого элемента программы с отображением только тех элементов, которые взаимодействуют с ним;
- для любого элемента программы возможность просмотреть участки кода, в которых этот элемент используется;
- сохранение исходных файлов исследуемой программы в хранилище данных;
- возможность выделения из всей программы только необходимой функциональной части и сборка только этой части.

Основные задачи анализа:

- выявление описанных, но неиспользуемых элементов;
- изучение исходного кода, на который отсутствует документация;

- исследование исходного кода для дальнейшей модернизации программного продукта;
- исследование исходного кода для дальнейшего переноса программного продукты на другую платформу;
- реструктуризация исходного кода через специально созданное хранилище данных.

Для решения задач было разработано средство анализа исходных текстов. Анализ включает в себя выделение из текстов программ необходимой разработчику информации: для поиска процедур и функций, для поиска имен описания типов объектов и самих объектов. На основе выделения формируются структуры различных типов – по функциям и процедурам, по объектам (их именам, их типам) с отображением адреса нахождения искомого объекта и связей его с другими объектами (например, список всех подчиненных функций одного модуля).

Основной подход при разработке средств анализа исходных текстов – создание кросс платформенного продукта. Использованы скриптовые языки Python, Lua. Это стабильные языки, использующиеся во многих проектах в качестве базовых или создания расширений. Для построения графического интерфейса пользователя использована библиотека wxWidgets [20], которая позволяет выглядеть приложению одинаково на всех платформах. Основной код wxWidgets вызывает элемент интерфейса платформы вместо того, чтобы повторно его реализовывать. Это предоставляет быстрый, естественно выглядящий интерфейс на каждой платформе. Для качественной визуализации используется комплекс Graphviz. Данные технологии поддерживают большое количество платформ и распространяются под свободными лицензиями.

Структура данных системы. При анализе исходного текста, программа загружает объекты в виртуальное хранилище данных и позволяет отобразить различные представления: функциональную связь (рис. 8.3), связи по глобальным объектам (рис. 8.4), связь по описателям объектов (рис. 8.5).

Схема функциональных связей предоставляет информацию о наличии описания различных функций в расположенных в разных директориях файлах и где именно эти функции вызываются (каждая связь на диаграмме соответствует вызову функции). На диаграмме показано как будет выглядеть рекурсия – замыкание функции 7 самой в себя. Каждая функция имеет адрес вида «Директория_1, Файл_1.c, Функция_1». Для построения графа связей конкретной функции можно сделать соответствующий запрос.

Схема связей по глобальным объектам похожа на предыдущую, но отображает использование глобальных переменных, констант, объектов. Схема описателей объектов отображает зависимость расположенных в разных файлах описаний классов, типов, структур. Связь вида «Тип1 -> Тип2» обозначает, что Тип2 является одним из составляющих Тип1. В классах это отношение «Потомок -> Родитель». Именно такой вид представлений позволяет получить исчерпывающую низкоуровневую информацию о рабочем проекте. При достаточно больших проектах происходит консолидация всей необходимой для разработчика информации в одном месте.

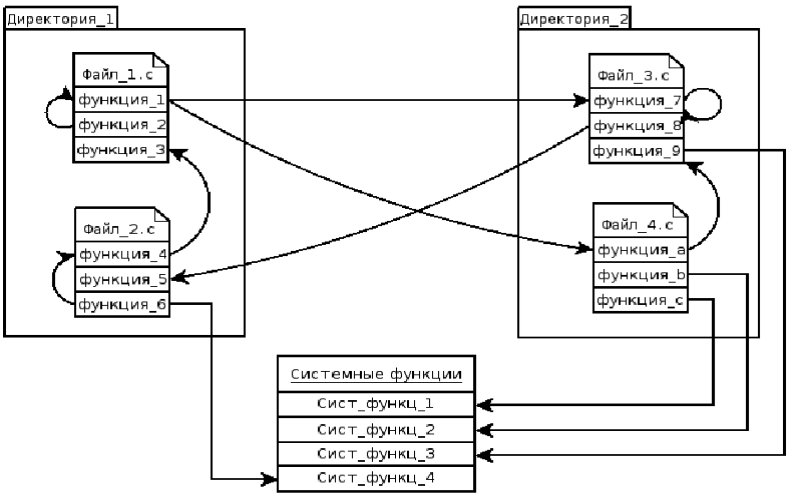


Рис. 8.3. Отображение функциональных связей

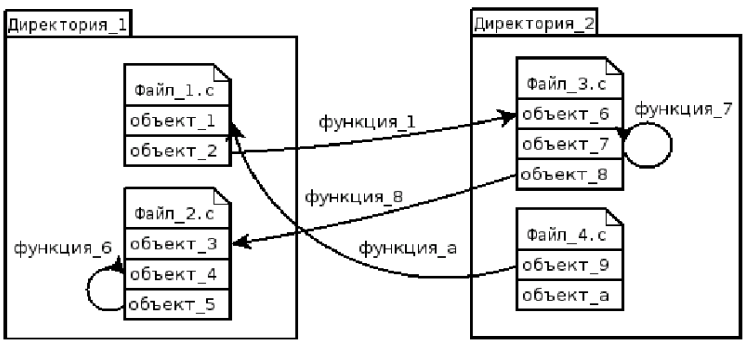


Рис. 8.4. Отображение связей между объектами

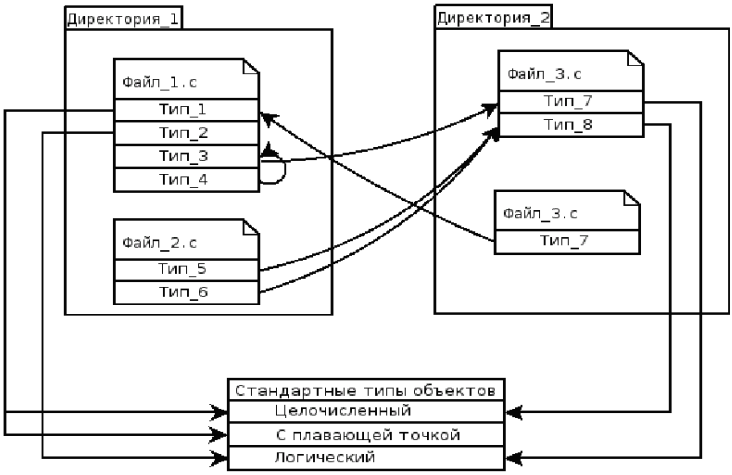


Рис. 8.5. Отображение связей по описателям объектов

Это позволяет разработчику объективно увидеть и оценить работу программы, найти неиспользуемые объекты (например, объекта на рис. 8.4), обнаружить рекурсии, увидеть родительские классы и их отличие от потомков.

Технологические цепочки. Технологические цепочки. Программа работает следующим образом: при выборе проекта для анализа, программа производит поиск исходных файлов и при обнаружении начинает загружать их в собственное хранилище данных, разбивая на простейшие объекты и сохраняя их зависимости. На втором этапе происходит работа непосредственно с хранилищем, а именно: в зависимости от целевого представления анализируются необходимые объекты данных и строятся зависимости, происходит обращение к Graphviz для получения графических координат элементов. Третий этап заключается в отображении на экране полученной информации.

В данной программе у программиста есть функционал сборки приложения, либо части приложения. При обычном способе сборки, который используется большинством популярных IDE, будет получен исполняемый файл и набор библиотек для работы программы. Библиотеки включают в себя все функции, и вычленить только часть не представляется возможным. Обычная сборка показана на рис. 8.6.

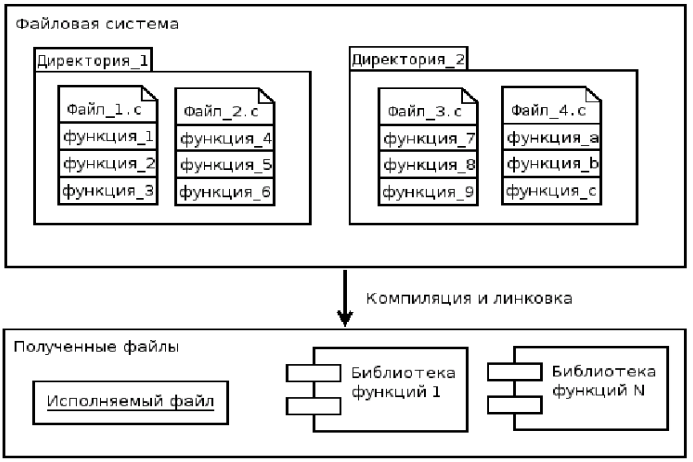


Рис. 8.6. Обычная сборка

Что делать, если требуется собрать программу только с одной или двумя основными функциями? Традиционный способ – переработка исходных текстов, ручной поиск зависимостей вызовов функций и включение в итоговый продукт только выбранных функций. Система позволяет автоматизировать этот процесс. На рис. 8.7 отображен процесс сборки только одной функции (функция_1).

Пользователь системы, выбрав нужную компоновку (требуемые функции) и расположение файлов целевого приложения, запускает процедуру обработки. Происходит обращение к хранилищу данных, подготовка данных, создание новой иерархии директорий и выгрузка необходимых объектов (классы, функции и т.д.) в микросреду для сборки – файлы нового проекта. Файлы имеют расположение, выбранное пользователем. Далее происходит компиляция и линковка. Результатом является исполняемый файл с требуемыми функциями и библиотека, которая содержит требуемые зависимости. Возможность выделения необходимых функций полезна при модификации приложения, при коллективной работе, при аудите исходных текстов сторонней организацией.

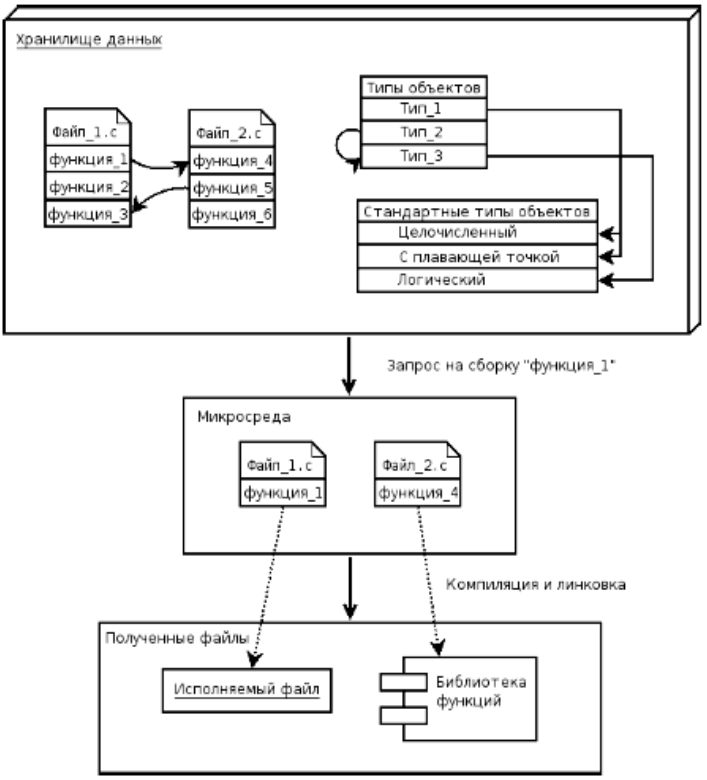


Рис. 8.7. Процесс сборки одной функции

8.6.3. Рефакторинг архитектуры многослойной иерархической ПС

Дальнейшее изложение материала этого раздела следует статьям М. Ксензова. Для исследования архитектуры программных систем используется нотация структурного моделирования, принятая в инструменте KLOCwork Architect. Этот инструмент предоставляет возможность автоматического извлечения моделей из программного кода и редактирования их. Далее рассматривается эта нотация.

Модели программных систем, используемые в KLOCwork Architect (в дальнейшем модель) [17], отдаленно напоминают модели типа сущность-отношение (Entity-Relation models). Основными единицами модели являются архитектурные блоки и отношения.

Архитектурный блок (Architecture Block). Архитектурные блоки – это основные элементы, составляющие модель. Архитектурные

блоки отображают структурные элементы программной системы вне зависимости от того уровня абстракции, на котором идет моделирование. Архитектурные блоки обладают, по меньшей мере, двумя основными атрибутами: имя и тип. Имена архитектурных блоков предопределяются именами тех структурных элементов системы, которые они представляют в модели. Типы архитектурных блоков существенно зависят от уровня абстракции, на котором проходит моделирование, и конкретной задачи, в рамках которой проводится исследование архитектуры.

При моделировании взаимодействия клиент-сервер – основной используемый тип архитектурных блоков – "подсистема". При моделировании систем, построенных в рамках компонентных технологий, – основной используемый тип – "компоненты". При моделировании системы сборки ПО – основные используемые типы - "папки" и "файлы". При объектно-ориентированном анализе – основной используемый тип - "класс".

Отношение (Relation). В модели KLOCwork Architect под отношением понимается некоторая односторонняя связь между парой архитектурных блоков. Между любой парой блоков в модели может быть произвольное количество разнонаправленных отношений, при этом их типы так же могут различаться. Так же, как и архитектурные блоки, отношения могут быть различных типов. В качестве примера можно привести следующие типы отношений:

- инстанция: A инстанциирует B (блок A – функция, блок B – класс);
- наследование: A наследует B (блоки A и B – классы);
- чтение данных: A читает данные из B (блок A – функция, блок B – класс, атрибут класса или функция);
- обращение: A вызывает B (блоки A и B – функции);
- A использует B: (блок A – класс или функция, блок B – класс или атрибут класса).

Модели обладают следующими свойствами:

1) иерархичность – каждый архитектурный блок может содержать другие архитектурные блоки, при этом связи между архитектурными блоками суммируются. Например, если в модели есть блок A, который содержит блок A1, и блок B, который содержит B1и B2, и между блоками A1 и B1 есть связь, а также между блоками A1 и B2 есть связь, то считается что между A и B есть две связи, поскольку они содержат два множества блоков, и количество связей между блоками из различных множеств равно двум;

2) точность – для каждого элемента модели можно указать «стоящий за ним» в моделируемой системе набор файлов и строк кода. Точность модели обеспечивается способностью инструмента KLOCwork Architect автоматически извлекать их из программного кода.

Над моделями в KLOCwork Architect определены операции редактирования:

- добавление блока, в модель допустимо добавлять новые блоки;
- удаление блока, из модели допустимо удалить произвольный блок;
- перенос блока, из модели допустимо вырезать блок и перенести его внутрь другого блока;
- переименование блока, в модели допустимо переименовывать блоки;
- объединение блоков в группу, модель позволяет объединять блоки в группы. При этом создается новый блок, и группируемые элементы переносятся внутрь него.

Важным свойством этих операций является то, что они сохраняют основные свойства модели – т. е. ее иерархичность и точность.

Инструмент KLOCwork Architect способен автоматически извлекать из программного кода базовые структурные модели, отражающие физическую структуру исследуемой программной системы. В состав таких моделей входят:

- архитектурные блоки, представляющие папки, в которых находятся файлы с исходным кодом системы;
- архитектурные блоки, представляющие собственно файлы. Такие архитектурные блоки находятся внутри соответствующих блоков, которые представляют папки, содержащие эти файлы;
- архитектурные блоки классов, переменных, функций, находящихся внутри соответствующих блоков файлов.

В [17,18] отмечается специфическая черта рефакторинга архитектуры: для достижения промежуточных целей, возникающих в ходе архитектурного рефакторинга, как правило, приходится выполнять более одного шага. Эти шаги относятся к различным фазам решения поставленных архитектурных задач. Можно условно выделить следующие фазы архитектурного рефакторинга: фаза "раскопки" архитектуры, фаза трансформации архитектуры, фаза семантического анализа подсистем и фаза проецирования изменений модели на программный код.

"Раскопка" архитектуры характеризуется тем, что соответствующие действия, применяемые к модели, не ориентированы на последующее проецирование на программный код. Они нужны только для понимания и структуризации модели.

Для шагов, относящихся к трансформации архитектуры, в отличие от шагов фазы раскопки, типично последующее проецирование их на реальный код программной системы. Шаги этой фазы четко связаны с реальной модификацией кода системы и, в конечном счете, ориентированы на его улучшение. Следует также отметить, что часть методов рефакторинга архитектуры не может быть строго отнесена к одной из названных категорий (раскопка и трансформация). На практике это означает, что решение о проецировании этих шагов на код принимает разработчик, руководствуясь поставленной задачей.

Как правило, между шагами описанных выше фаз архитектурного рефакторинга предпринимаются шаги, которые можно отнести к фазе семантического анализа подсистем: по ходу трансформаций часто встает задача выявления смысловой нагрузки подсистем. Для решения подобных задач, даже в первом приближении, зачастую приходится исследовать реальный программный код (здесь опять-таки помогает точность модели), анализировать сигнатуры функций и комментарии, а при отсутствии последних и сам код функций. Задача специалиста, вовлеченного в процесс архитектурного рефакторинга, – по возможности минимизировать объем семантического анализа (например, путем удаления вспомогательных блоков) и сделать его последовательным и направленным.

Задача специалиста, вовлеченного в процесс архитектурного рефакторинга, – по возможности минимизировать объем семантического анализа (например, путем удаления вспомогательных блоков) и сделать его последовательным и направленным.

Результат последнего шага – редактирование модели должен быть спроецирован на реальный программный код системы. Действительно, при проецировании удаления блоков из модели необходимо определить множество строк и файлов, которое соответствует удаленному блоку в программном коде. После этого необходимо удалить из программного проекта выявленные строки и файлы. При проецировании на код переноса блока в модели переносятся соответствующие строки и файлы в исходном коде программной системы и т.д. Производимые таким образом трансформации можно рассматривать как архитектурно-управляемый рефакторинг программного кода.

8.6.4. Слои в архитектуре ПС. Паттерн выделения слоев

Концепция слоев – одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части (см. глав 2 и 6). В архитектурах компьютерных систем, например, различают слои кода на языке программирования, функций операционной системы, драйверов устройств, наборов инструкций центрального процессора и внутренней логики микросхем. В гл. 2 отмечалось, что если система разбита на ряд слоев, то слой n – это набор компонентов системы, которые используют только компоненты слоя $n - 1$ и могут быть использованы только компонентами слоя $n + 1$.

Слой $n + 1$ использует слой n, следовательно, абстракция понятий слоя $n + 1$, по меньшей мере, не ниже чем у слоя n, а в идеале – если архитектура системы эффективна, его уровень абстракции должен быть выше. Соответственно, слой n скрывает (инкапсулирует) логику работы с понятиями, определенными на этом слое, позволяя, таким образом, слою $n + 1$ реализовать работу с более сложными понятиями, организовать более сложную логику, используя выразительные средства нижележащего слоя. Можно выбирать альтернативную реализацию базовых слоев – компоненты верхнего слоя способны работать без каких-либо изменений в нижележащих слоях, при условии сохранения интерфейсов. Зависимость между слоями, то есть, фактически, интерфейсы, предоставляемые нижними слоями верхним, можно свести к минимуму. Такая минимизация интерфейсов позволяет увеличивать гибкость системы.

Слои способны удачно инкапсулировать многое, но не все: модификация одного слоя подчас связана с необходимостью внесения каскадных изменений в остальные слои. Наличие избыточных слоев нередко снижает производительность системы. При переходе от слоя к слою данные обычно подвергаются преобразованиям из одного представления в другое. Несмотря на это, инкапсуляция нижележащих функций зачастую позволяет достичь весьма существенного преимущества. Например, оптимизация слоя транзакций обычно приводит к повышению производительности всех вышележащих слоев.

В статье [10] рассматривается один из паттернов выделения слоев – представитель целого семейства паттернов выделения слоев. Подвидов у этого паттерна существует достаточно много, и каждый из них обладает своей спецификой. Рассматриваемый паттерн имеет следующую структуру (следуя терминологии автора статей).

Имя: выделение слоев.

Ситуация: на диаграмме представлены элементы, для которых верны следующие условия:

- исходящие связи ведут только в последний выделенные слои (слой с номером n), или их нет, если ранее не был выделен ни один слой;
- “кандидаты на объединение в новый слой” с номером *n*+1 должны обладать общим смыслом и/или функциональностью. Простейшей проверкой на наличие общности является простой критерий: если для кандидатов можно подобрать "общее определение", то можно считать, что они обладают требуемой общностью.

Рецепт: Объединить блоки в новый слой *n*+1. Для двух произвольных слоев слой, обладающий большим порядковым номером, считается "вышележащим". Если в результате применения паттерна было выделено *n* слоев, и еще остались блоки, которые в силу ограничений не смогли быть отнесены ни к одному из выделенных слоев и формально не могут быть выделены в новый слой, то эти блоки по умолчанию считаются *n*+1 слоем, который в дальнейшем именуется "чердаком".

Различают строгие слои, которые не допускают никаких отклонений в строгой структуре, и потому встречающейся относительно редко, и нестрогие слои. Последние допускают связи вышележащего слоя с несколькими нижележащими слоями (потенциально – ко всем), а не только к непосредственному соседу снизу. Как отмечается в [17], архитектура с нестрогими слоями может быть, как результатом эрозии, так и осознанным решением. Возможным дефектом архитектуры с нестрогими слоями является нарушение абстракции. Это затрудняет анализ системы. Кроме того, изменения слоя в такой архитектуре значительно сложнее локализовать – волна изменений прокатится по всем слоям, работающим с изменяемым слоем.

Рассмотренные виды слоев можно модифицировать, позволив включать в произвольные слои сильносвязанные компоненты. При таком подходе сильносвязанные компоненты (СК) рассматривается, фактически, как атомарный элемент. Слои, содержащие СК в [17], названы поглощающими. Без подобного смягчения условий как сам СК, так и все блоки которые могли бы попасть в вышележащие слои, будут отправлены на "чердак". Заметим, что не всегда СК на структурных диаграммах свидетельствуют о плохой архитектуре системы. Возможным дефектом архитектуры с поглощающими слоями может стать эффект

"пропавшего слоя" – дефектная связь приводит к появлению СК, которые по смыслу должны находиться на разных слоях.

Паттерн выделения слоев может быть применен для чистого анализа (раскопки) архитектуры. Выделение слоев – это прием, который позволит сократить и сделать более направленным семантический анализ системы за счет структурного анализа. Это, несомненно, хорошо, поскольку семантический анализ – более ресурсоемкий процесс. При этом нет никакой необходимость отражения слоев на программный код и инфраструктуру его хранения. Например, при анализе архитектурного кода программной системы, написанной на Java, выделение слоев не предполагает обязательного выделения пакетов, соответствующих этим слоям. Если специалист, проводящий архитектурный рефакторинг, принимает решение все-таки не отображать слои в пакеты языка Java, то можно считать, что выделение слоев было применено для чистого анализа архитектуры.

Выделение слоев – хорошая основа для улучшения системы. Найти строгие слои в произвольной программной системе значительно труднее, чем найти слои с некоторыми допустимыми отклонениями, как, например, сильносвязанные компоненты. Тем не менее, для каждого из допустимых отклонений известны побочные эффекты, которые можно устранять, по возможности приводя слои к строгим.

В заключение этого раздела отметим, что существуют и другие архитектурные паттерны, часть из которых рассмотрена в [14]. В этой же работе предложены направления дальнейшего развития рефакторинга архитектуры:

1. Каталогизация. Направление, связанное с дальнейшим сбором, обобщением и классификацией паттернов рефакторинга.

2. Автоматизация. Представляет большой интерес возможность облегчения и автоматизации применения как существующих так вновь предлагаемых паттернов.

3. Верификация. Вызывает интерес возможность верификации сохранности поведения программной системы при архитектурном рефакторинге.

4. Направленность. Желательно вооружить разработчика, знающего паттерны архитектурного рефакторинга и имеющего соответствующие инструменты моделирования, процедурой, определяющей последовательность применения паттернов, чтобы сделать процесс эффективным. Создание подобных процедур представляет значительный исследовательский интерес.

8.7. Архитектурный рефакторинг для повышения производительности многослойных программных систем

8.7.1. Возможный подход к созданию программных систем

В свое время М. Фаулер высказал идею рефакторинга БД, однако об ахитектурном рефакторинге программных систем речи не было. Надо сказать, и в настоящее время вопросам архитектурного рефакторинга посвящено незначительное количество работ. В то же время эволюция сложных программных систем требует от разработчика повышенного внимания к выбору архитектуры. Практически всегда во время разработки, появляются новые требования со стороны заказчика, и приходиться пересматривать первоначальную архитектуру. Выделяют следующие фазы архитектурного рефакторинга: 1) "раскопки" архитектуры, 2) трансформация архитектуры, 3) семантический анализ подсистем и 4) проецирование изменений на программный код.

В настоящей работе рассматриваются и решаются вопросы рефакторинга многослойных программных систем (ПС), целью которого является повышение производительности системы.

Как правило, значительная часть программных систем (ПС) создается в срочном порядке. Требуется автоматизировать (создать поддерживающую ПС) для некоторой совокупность взаимодействующих бизнес-процессов. Часто наспех составленное техническое задание передается выбранной (возможно без предварительного анализа или на основе тендера) компьютерной фирме, которая обещает выполнить работу в требуемые (как правило, минимальные) сроки и за приемлемую стоимость.

Подобные фирмы обычно используют гибкие технологии создания программных систем, основанные на итерационном и инкрементном подходе к созданию ПО. Это может быть SCRAM или Agile-методология с элементами экстремального программирования. В этом случае можно избежать масштабного проектирования наперед и не тратить много сил на проектирование раньше времени. Такой поход к разработке ПС позволяет достаточно быстро создать совокупность программных модулей, автоматизирующих заданный набор бизнес-процессов *B*.

Однако зачастую эти модули часто создаются независимо друг от друга, и в этом случае могут быть пересечения по функциям, реализуемым модулями. Возможны (и это чаще) ситуации, когда один модуль

может обращаться к другому для выполнения некоторых функций, реализуемых этим модулем. Здесь нужно заметить, что под модулем понимается достаточно произвольный структурный элемент ПС (подсистема, компонент, отдельный программный модуль, группа классов, отдельный класс), который можно выделить, определив интерфейс взаимодействия между этим модулем и всем, что его окружает.

Часты ситуации, когда разрабатываемая ПС слабо документируется, и об архитектуре создаваемой системы и ее целесообразности разработчики особенно не задумываются. Однако, тем не менее, архитектура разрабатываемой системы существует, и она, собственно, создана ее авторами-разработчиками независимо от их желания. В первом приближении архитектуру ПС в этом случае можно представить некоторым множеством программных модулей:

$$M = \{m_{ij} \mid i = 1,2,\dots,N,\ \ j = 1,2,\dots,n_i\},$$

$$M = \bigcup_{i}^{N} M_i\ , \qquad |M| = K,$$

где N – количество бизнес-процессов;

K - количество модулей в программной системе;

i – номер бизнес-процесса;

j – номер модуля, реализующего j-функцию i-го бизнес-процесса,

n_i - количество функций, реализуемых i-м бизнес-процессом;

M_i – подмножество модулей, автоматизирующих i-й бизнес-процесс $b_i \in B$.

В общем случае справедливо соотношение

$$\bigcap_{i}^{N} M_i \neq \emptyset.$$

Каждый модуль m_{ij} можно представить следующими параметрами спецификации:

$$P_{ij} = \{Name, I_{ij}, O_{ij}, A_{ij}\},$$

где $Name$ – имя модуля m_{ij},

I_{ij}– параметры входного интерфейса модуля m_{ij},

O_{ij} – параметры выходного интерфейса модуля m_{ij},

A_{ij}– абстракция алгоритма, реализуемого модулем m_{ij},

Заметим, что абстракция через спецификацию позволяет абстрагироваться от алгоритма, описанного в теле модуля, до уровня знания лишь того, что данный модуль должен в итоге реализовать. Существует отображение вида $O: B \rightarrow M$, которое определяет подмножества модулей, автоматизирующих функции конкретных бизнес-процессов. Таким

образом, существуют отображения $O_i: b_i \rightarrow M_i \subset M,\ i = 1,2,\dots,N.$ При этом возможны непустые пересечения

$M_i \cap M_j \neq \emptyset, i,j = 1,2,\dots,N.$ Это свидетельствует о возможном дублировании некоторых функций бизнес-процессов в автоматизируемых их модулях ПС. Однако возможны ситуации, когда для некоторого отображения O_i $|M_i| \rightarrow |b_i|$, что говорит том, что ПС реализует не все функции бизнес-процесса b_i. Но даже если в этом плане нет претензий к разработанной программной системе, как отмечено выше, часто архитектура ПС не только предварительно не разрабатывается, но и недостаточно (или совсем) не документируется. Отсюда в интересах дальнейшей разработки системы или ее сопровождения возникает проблема "раскопки архитектуры", как ее часто называют в литературе [2, 12].

8.7.2. Представление созданной архитектуры ПС

Удобным и наглядным способом представления архитектуры программных систем является использование графов. В работе [19] строится модель ПС на основе исходного кода, когда может отсутствовать информация о составляющих систему блоках. В нашем случае рассматривается пример разработки ПС на основе гибкой технологии, когда в разрабатываемую систему последовательно добавляются новые модули. В этом случае каждый модуль m_{ij} системы можно представить именем *Name* и частью параметров O_{ij} из спецификации модуля – именами модулей, которые – могут быть вызваны из модуля m_{ij}. Для удобства и простоты дальнейших построений каждый модуль будем представлять в следующем виде:

$m_{ij} \rightarrow < namber1, namber2, namber3, \dots >,$

где $namber1$ – номер модуля m_{ij}, отождествляемый с его именем *Name;*

$namber2$ – номер 1-го модуля, к которому может обращаться модуль m_{ij} ;

$namber3$ – номер 2-го модуля, к которому может обращаться модуль m_{ij} и т.д.

Таким образом, в целом перечень всех модулей и их взаимосвязей можно представить списком следующего вида:

$S = S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_K,$

где $S_i,\ i = 1,2,\dots,K$ - элементы списка следующей структуры:

$S_1 = < 1, s, k, m, \dots >,$

$S_2 = < 2, l, t, f, \dots >,$

……..

$S_K < K, g, z, u, \ldots >$,

где $s, k, m, l, t, f, \ldots, g, z, u, \ldots$ – номера модулей.

На основе списка *S* можно построить граф *G*, отображающий структуру ПС. Однако в таком представлении трудно сделать вывод о типе архитектуры программной системы и ее качестве. Известно, что значительная часть современных программных систем имеют многослойную архитектуру. В таких архитектурах модули нижнего слоя для выполнения своих функций не обращаются к другим слоям. Организация вышележащих слоев может быть различной. Поэтому при анализе полученной архитектуры ПС первой задачей является выделение слоев модулей. Многослойная архитектура обеспечивает группировку связанной функциональности приложения в разных слоях, выстраиваемых вертикально, поверх друг друга. Функциональность каждого слоя объединена общей ролью или ответственностью. Слои слабо связаны, и между ними осуществляется явный обмен данными. Правильное разделение приложения на слои помогает поддерживать строгое разделение функциональности, что обеспечивает гибкость, а также удобство и простоту разработки.

Слои приложения могут размещаться физически на одном компьютере (на одном уровне) или быть распределены по разным компьютерам (n-уровней). Связь между компонентами разных уровней осуществляется через строго определенные интерфейсы. Далее будем рассматривать ПС одного определенного языкового уровня с хорошо определенными синтаксическими единицами в соответствии с хорошо определенными синтаксическими правилами и хорошо определенной семантикой элементарных операторов и синтаксических конструкций. Из рассмотрения исключим все вопросы, относящиеся к другим языковым уровням, таким, например, как интерпретация элементарных операторов в терминах более примитивных составляющих.

Элементарные операторы данного языкового уровня будем рассматривать как модули базового уровня, составляющие базовый слой. Это можно сделать потому, что все элементарные операторы ПС повсеместно доступны. Заметим, что здесь не учитываются привилегированные операторы машинного языка, которые не могут использоваться в прикладных программных системах. Модули, построенные из модулей базового уровня, могут рассматриваться только как модули нулевого уровня. При этом не требуется, чтобы все модули нулевого уровня были равнодоступны: например, в программах, написанных на языке, допускающем блочную структуру, некоторые модули нулевого уровня могут

быть локализованы в каком-либо блоке и, следовательно, доступны только внутри этого блока и его подблоков. С другой стороны, при конструировании модулей высших уровней в некоторых языках можно использовать модули разных уровней и даже того же самого уровня, что и конструируемый модуль (рекурсия, сопрограммы).

Введем в рассмотрение матрицу R размером $K \times K$, каждый элемент которой образуется по правилу:

$$r_{ij} = \begin{cases} 1, & \text{если } j \in S_i, \\ 0 - \text{в противном случае}. \end{cases}$$

Далее можно следовать алгоритму, который дается ниже.

0. Начало, I = 0.

1. Находим в матрице номера строк, все элементы которых равны нулю.

2. Фиксируем вершины с этими номерами, образующими I-слой.

3. I = I + 1.

4. Если остались столбцы с ненулевыми элементами, обнуляем столбцы с номерами найденных вершин. Переходим к п. 1.

5. Если все столбцы содержат только нулевые элементы, конец.

Надо заметить, что данный алгоритм позволяет построить послойную архитектуру ПС, которая удовлетворяет одному из вариантов, рассмотренных в [20]. Однако если в слоях ПС имеются горизонтальные связи или сильносвязанные модули, то полностью определить структуру ПС без дополнительного анализа не удастся.

8.7.3. Анализ на соответствие послойной архитектуре (выделение слоев)

Рассмотрим порядок проведения анализа на конкретном примере. Пусть задана списком *S* некоторая совокупность модулей ПС (9 модулей), которая представляется следующей матрицей *R*.

$$R = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Действуя по приведенному выше алгоритму, находим нулевые строки, и таким образом определяем модули нулевого слоя: $L_0 = \{8, 9\}$. Вычеркиваем 8 и 9 столбцы. Находим нулевую строку с номером 7, определяющую слой $L_1 = \{7\}$. Вычеркиваем седьмой столбец и определяем слой $L_2 = \{4, 5\}$. Вычеркиваем столбцы 4 и 5. По ставшим нулевыми строкам определяем модули третьего слоя $L_3 = \{2, 3, 6\}$. Вычеркиваем строки 2, 3 и 6. Вычеркиваем столбцы с этими номерами и определяем модули четвертого слоя $L_4 = \{1\}$. Получив распределение модулей по слоям ПС, можно построить граф *G* программной системы (рис. 7.8).

Анализируя полученный граф, следует отметить, что он не отвечает каноническим правилам многослойной структуры. В частности, модуль 6 не отвечает этим требованиям. Известно, что выделение слоев – хорошая основа для улучшения системы. Найти строгие слои в произвольной программной системе достаточно трудно, поскольку, как уже отмечалось, они могут содержать горизонтальные связи и сильносвязанные компоненты.

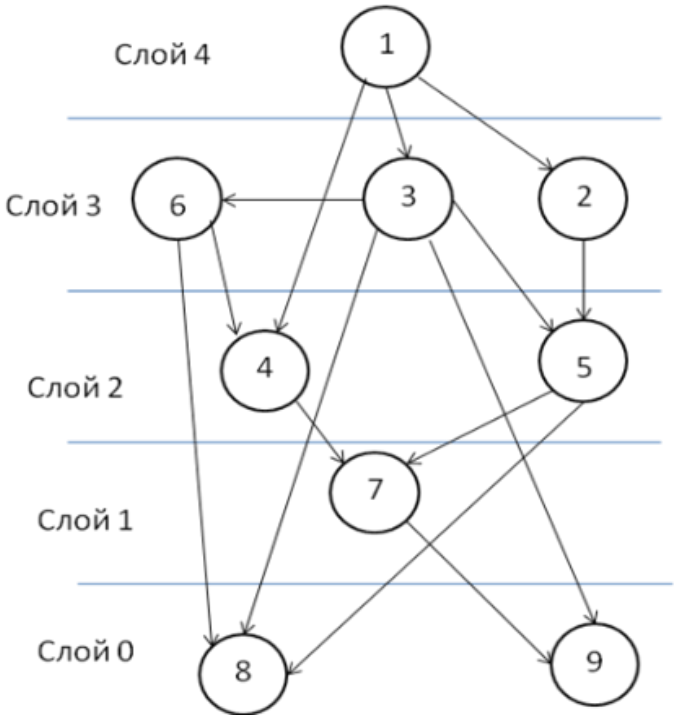


Рис. 7.8. Граф ПС

Поэтому целесообразно расширить понятие слоя, позволив включать в произвольные слои сильносвязанные компоненты. Эти компоненты при таком подходе можно рассматривать, как атомарные модули. Заметим, что не всегда сильносвязанные компоненты на структурных диаграммах свидетельствуют о плохой архитектуре системы. Возможным дефектом архитектуры с поглощающими слоями может стать эффект "пропавшего слоя" – дефектная связь приводит к появлению модулей, которые по смыслу должны находиться на разных слоях.

8.7.4. Коррекция (трансформация) архитектуры в интересах ее рефакторинга

В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований, сохраняющих функциональную семантику базового кода. По ходу трансформаций часто встает задача выявления смысловой нагрузки модулей. Задача специалиста, вовлеченного в процесс архитектурного рефакторинга, – по возможности минимизировать объем семантического анализа (например, путем удаления вспомогательных блоков) и сделать его последовательным и направленным. Первым уровнем рефакторинга можно считать такое изменение кода, которое не затрагивает структуру модулей, т.е. рефакторингу подвергается программный код внутри классов. Второй уровень рефакторинга относится к изменению структуры модулей или классов программной системы, добавлению новых классов, выделению и разбиению больших классов, переносу или добавлению новых методов, выделению интерфейсов и др.

Следующий, третий, уровень рефакторинга М. Фаулер называет крупным рефакторингом. В монографии идет речь о четырех рефакторингах третьего уровня. Это разделение наследования, преобразование процедурного проекта в объекты, отделение предметной области от представления и выделение иерархии. Рефакторинг архитектуры программных систем является четвертым уровнем рефакторинга. Необходимость в архитектурном рефакторинге может быть связана со следующими причинами:

1. Плохо структурированный код из-за часто вносимых изменений разработчиками, которые не до конца понимают архитектуру ПС.

2. Повышение производительности ПС. Рефакторинг первого и второго уровней, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более понятной и податливой для настройки производительности.

3. Потребность в функциональных изменениях ПС. Изменение существующей архитектуры может быть хорошим шагом на пути внедрения новой функциональности, облегчающим дальнейшую эволюцию системы.

4. Смена платформы ПС. Смена платформы ПС. Желательно ограничиться изменениями только в узкой платформенно-зависимой прослойке системы. Выделение такой прослойки всегда сопряжено с необходимостью изменения архитектуры.

5. Обновление технологии разработки программного продукта, связанное, например, с переходом на более совершенную технологию программирования, внедрением комплексной среды разработки.

Формализовать процесс коррекции архитектуры ПС или тем более построить алгоритм коррекции довольно затруднительно. Однако в ряде случаев, выделив отдельные фрагменты структуры ПС, можно их преобразовать, стремясь к получению наилучшей структуре, например, к дереву. Чаще всего это удается сделать путем объединения (поглощения) модулей. Некоторые примеры такой коррекции приведены в главах 1 и 2.

Результат коррекции архитектуры должен быть спроецирован на реальный программный код системы. При проецировании удаления модулей из модели необходимо определить множество строк и файлов, которое соответствует удаленному блоку в программном коде. После этого необходимо удалить из программного проекта выявленные строки и файлы. При проецировании на код переноса модуля в модели переносятся соответствующие строки и файлы в исходном коде программной системы и т.д. Производимые таким образом трансформации можно рассматривать как архитектурно-управляемый рефакторинг программного кода.

Заключение. Бытует высказывание: “архитектура – это то, за что увольняют системного архитектора и руководителя проекта”. Именно архитектор занимается проектированием архитектуры программной системы и разработкой архитектурного описания этой системы. Важнейшая обязанность архитектора заключается в разработке ключевых проектных решений относительно внутреннего устройства программной системы. Это начальный этап создания ПС и, как известно, ошибки, допущенные на этом этапе, исправляются многократно сложнее и дороже, чем позже они выявляются. В тоже время не существует хорошо разработанных формальных методов проектирования архитектуры программных систем. И если само программирование до сих пор во многом считается искусством, то в значительной большей мере это можно сказать относительно проектирования архитектуры программных систем.

Неоценима важность разработки архитектуры бортовой программной системы для класса летательных аппаратов, в которых ресурсы вычислительной системы существенно ограничены, особенно в

части памяти. Иногда бывает так, что программная система разрабатывается без предварительного архитектурного описания. Но как бы она не разрабатывалась, в ней уже заложена определенная архитектура. Далее начинается понимание важности и необходимости иметь ее архитектурное описание и начинается процесс “раскопки архитектуры” – представление созданной архитектуры, после чего начинается ее коррекция (рефакторинг). По мнению авторов, на этом этапе данная монограафия может оказать определенную помощь в процессах коррекции архитектуры создаваемой программной системы.

Литература к гл. 8

1. Эмблер С., Садаладж П. Рефакторинг баз данных: эволюционное проектирование. Пер. с англ. – М.: Издательский дом "Вильяме", 2007. – 368 с.

2. Smalltalk?! [Электронный ресурс]. URL: http://www.smalltalk.ru/articles/smalltalk-2.html

3. Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода. – Спб: Символ-Плюс, 2009. – С. 432.

4. Фаулер М. Архитектура корпоративных программных приложений. Пер. с англ. – М.: Издательский дом "Вильяме", 2006. – 544 с.

5. Деревянко В. Рефакторинг в Visual Studio. [Электронный ресурс]. URL: http://soft.sibnet.ru/review/?id=623

6. Методы улучшения качества кода: рефакторинг. [Электронный ресурс]. URL: http://social.msdn.microsoft.com/Forums/ru-ru/fordesktopru/thread/63d9ba19-491b-4e75-9796-6de5132e1a56

7. Технический долг [Электронный ресурс]. URL: http://habrahabr.ru/blogs/refactoring/119490/

8. Сергеев Г.Г. Формализация процессов рефакторинга на основе символьной записи структуры классов. Сборник научных трудов “Вестник НТУ “ХПИ”, Харьков, 2010, с. 100 – 104.

9. Эффект второй системы. [Электронный ресурс]. URL: http://habrahabr.ru/blogs/refactoring/121213/http://www.saasworld.ru/analytics/vend/

10. Краковецкий А. Как писать высококлассный код. Часть вторая. Возможности Visual Studio 2010. [Электронный ресурс]. URL: http://msug.vn.ua/Posts/Details/4165

11. Хант Э., Томас Д. Программист-прагматик. Путь от подмастерья к мастеру. Пер. с англ. Изд. Лори, 2009. – 270 с.

12. Refactoring (Рефакторинг) (комментарии). [Электронный ресурс]. URL: http://www.gamedev.ru/code/forum/?id=131858

13. Ворожко Я. Refactoring. [Электронный ресурс]. URL: http://pro100pro.com/category/refactoring

14. Гагарина Л.Г., Кокорева Е.В., Виснадул Б.Д. Технология разработки программного обеспечения: учебное пособие / под ред. Л.Г. Гагариной. – М.: ИД «ФОРУМ»: ИНФРА-М., 2008. – 400 с.

15. Черный С.Г. Оптимизация процесса структуризации кода. [Электронный ресурс]. URL: http://www.nbuv.gov.ua/portal/natural/Vejpt/2011_2_2/2011_2_2/31_34.pdf

16. Миронов В.О. Применение графов для анализа сложных систем на основе исходного кода программ. [Электронный ресурс]. URL: http://berestneva.am.tpu.ru/Papers/KONF2009/%

17. Ксензов М. Рефакторинг архитектуры программного обеспечения: выделение слоев. Труды РАН, препринт 4, 2004, с. 211 – 227

18. Welcome to Graphviz. [Электронный ресурс]. URL: http://graphviz.org/

С.В. Назаров

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Монография

Подписано в печать 25.04.2022.
Формат 60×90/16. Усл. печ. л. 13,5.
Тираж 1000 экз.

ООО «Русайнс».
117218, г. Москва, ул. Кедрова, д. 14, корп. 2.
Тел.: +7 (495) 741-46-28.
E-mail: autor@ru-science.com
http://ru-science.com