ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

3(139)
2008

ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с ноября 1995 г.

УЧРЕДИТЕЛЬ
Издательство "Новые технологии"

СОДЕРЖАНИЕ

Аннотации статей размещены на сайте журнала по адресу http://www.**informika.ru/text/magaz/it/** или **http://novtex.ru/IT.**

Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора наук.

УДК 004.231.3; 004.382.6;004.382.7; 007.384

С. Г. Бобков, канд. техн. наук,
Научно-исследовательский институт
системных исследований РАН

Методика проектирования микросхем для компьютеров серии "Багет"

Рассматривается маршрут проектирования индустриальных микросхем серии 1890 с технологическими нормами 0,5...0,18 мкм.

Введение

На рынке существует целый ряд компаний, таких как *Synopsys*, *Mentor Graphics*, *Cadence*, предлагающих как отдельные САПР, так и целые платформы, покрывающие весь маршрут проектирования микросхем [1—5]. Однако данные САПР являются универсальными, охватывают все области применения и не учитывают особенности проекта и некоторые особенности технологии производства микросхемы (например, тип и толщина подзатворного диэлектрика определяют предельные возможности ряда микроэлектронных структур по радиационной стойкости и температурным параметрам). Используя эти особенности, можно получить существенно лучшие показатели проекта по всем параметрам: быстродействию, потреблению питания, площади кристалла. Однако отход от имеющихся маршрутов проектирования приводит к заметному увеличению стоимости проекта, времени проектирования, риска и может доходить до десятков раз при глубокой модернизации маршрутов. Целью работы было создание маршрута проектирования микросхем, позволяющего создавать микросхемы с заданными характеристиками, в некоторых случаях превышающими до 3 раз характеристики микросхем, получаемых стандартными маршрутами проектирования, при увеличении времени проектирования не более чем в 2 раза. В созданном маршруте микросхемы должны изготовляться с технологическими нормами 0,5...0,18 мкм и удовлетворять всем требованиям микросхем, функционирующих в жестких условиях эксплуатации, характерных для промышленных ЭВМ.

Этапы разработки микросхем

Процесс разработки микросхем можно разбить на следующие основные этапы.

1. Формирование основных характеристик микросхемы (разработка технического задания — ТЗ).
2. Разработка архитектуры микросхемы, создание ее программной модели.
3. Разработка поведенческой модели.
4. Создание тестового программного обеспечения (ПО), включая системное программное обеспечение.
5. Разработка логической модели уровня RTL, верификация модели.
6. Уточнение технологии производства микросхем, выбор библиотек, IP-блоков (функционально законченных и полностью специфицированных блоков), оптимизация архитектуры микросхемы.
7. Создание принципиальной электрической схемы на уровне транзисторов (логический синтез, разработка заказных узлов).
8. Топологическое проектирование: размещение заказных блоков и библиотечных элементов, построение дерева синхронизаций, трассировка цепей, верификация проекта.
9. Тестирование изготовленных кристаллов и микросхем. Коррекция проекта при необходимости. Проведение испытаний микросхем, окончательное определение параметров микросхем.

Некоторые этапы проектирования из указанных выше могут отсутствовать. Например, если создается микросхема, функционально подобная известной, но с другими характеристиками, отсутствует этап 2, в ряде случаев не требуется этап 3. Оригинальность маршрута проектирования микросхем заключается, прежде всего, в специфике и порядке проектирования на каждом из выделенных этапов, цикличности повторения ряда процедур, включения некоторых процедур для достижения требуемых характеристик, выборе систем проектирования, включая разработку собственных.

Маршрут и методика проектирования микросхем

Рассмотрим поэтапно основные положения предлагаемого маршрута проектирования.

1. Разработка технического задания (ТЗ) включает: формирование основных характеристик микросхемы в соответствии с потребностями рынка или требованиями заказчика; условия эксплуатации, оценку сложности микросхемы, топологические нормы, требуемое тестовое программное обеспечение, оценку времени проектирования и маршрута проектирования, решение о необходимости покупки или разработки САПР, требуемые

аппаратные, людские ресурсы. Как показал опыт создания нескольких десятков сложных микросхем (с числом транзисторов сотни тысяч — десятки миллионов), успех создания микросхемы во многом зависит от продуманности маршрута проектирования и жесткого ему следования на протяжении всего проектирования. В маршруте проектирования должны быть определены не только требуемые программные пакеты проектирования, но и их версии. Изменение маршрута проектирования должно проходить установленным порядком, а не по ходу проектирования микросхем.

Как было сказано выше, эффективность маршрута проектирования может быть повышена его ориентацией под выделенный класс задач и используемые технологические нормы. Однако при охвате процесса разработки микросхем различной сложности, как правило, маршрут имеет несколько ветвей. Выбор же конкретной ветви проектирования происходит в процессе проектирования в зависимости от получаемых характеристик проекта. На этом же этапе необходимо определить по возможности ближайшие аналоги разрабатываемой микросхемы в целях использования имеющегося опыта работ, а также определить в существующих микросхемах наиболее удачные технические решения для их дальнейшего развития.

2. Разработка архитектуры микросхемы является сложным итерационным процессом. Например, разработка 64-разрядного суперскалярного микропроцессора с известной архитектурой на Западе оценивается суммой не более 100 млн долл., разработка современного высокопроизводительного микропроцессора с новой архитектурой — сотни миллионов и единицы миллиардов долларов.

На данном этапе в рассматриваемом маршруте проектирования происходит разработка базовой архитектуры, которая на последующих этапах может корректироваться и оптимизироваться. Оптимизация архитектуры выполняется на нескольких уровнях: на поведенческой модели; на логической модели; на системных и пользовательских программах; на этапах создания принципиальной электрической схемы и топологического проектирования.

Поведенческая модель позволяет оптимизировать базовые архитектурные решения.

Логическая модель (RTL-модель) определяет основные характеристики микросхемы и тем самым позволяет определить критические узлы и несбалансированность архитектурных решений. Для выявления недостатков архитектуры при компьютерном моделировании требуются многие часы функционирования высокопроизводительных компьютеров. Для микропроцессора 1890ВМ5Ф скорость выполнения инструкций на RTL-модели составляет примерно 300...400 инструкций/с, в то время как для поведенческой модели (С-модель, покомандная) — 1 000 000 инструкций/с. Данные получены на операции загрузки операционной системы (ОС) Linux на компьютере Pentium IV с частотой 3,2 ГГц. Такая скорость не позволяет в разумное время проектирования проверить различные варианты архитектурных решений на логической модели.

В созданном маршруте проектирования оптимизация архитектуры на уровне логической модели осуществляется на тестовых платах, функционально подобных платам с разрабатываемой микросхемой, где вместо разрабатываемой микросхемы устанавливаются микросхемы FPGA — микросхемы, программируемые пользователем. Такой подход позволяет оперативно не только отлаживать логическую модель, но и проверить и оптимизировать различные архитектурные решения. При этом частота функционирования микросхем FPGA не намного уступает частоте функционирования разрабатываемой микросхемы, а в ряде случаев эти частоты совпадают. Так, микросхема FPGA на тестовой плате для разработки контроллера Ethernet 10/100 Мбит/с 1890ВГ3Т функционировала на той же частоте, что и сама микросхема, микросхема FPGA тестовой платы для создания 350-мегагерцевого микропроцессора 1890ВМ5Ф функционировала на частоте 24 МГц.

Эффективная разработка микросхем с новой архитектурой невозможна без моделирования с соответствующим системным и прикладным программным обеспечением. Например, для разработки микросхем графического контроллера 1890В-Г10Т и 1890ВГ14Т [6—9] был создан специализированный стенд, позволяющий выполнять реальные пользовательские программы на текущей версии модели микросхемы, что позволило дополнительно оптимизировать архитектуру микросхем и поднять производительность. За счет оптимизации архитектуры на этом этапе производительность микросхемы 1890ВГ10Т была поднята почти в 2 раза, а микросхемы 1890ВГ14Т — в 3 раза. Производительность микросхемы 1890ВГ14Т, изготовленной с технологическими нормами 0,35 мкм, соответствует производительности западных микросхем, изготовленных с технологическими нормами 0,18...0,25 мкм. На этапе разработки архитектуры также определяются функциональные узлы, которые должны быть разработаны заказным путем. Перечень заказных узлов может быть уточнен при создании логической модели микросхемы.

На этапах создания принципиальной электрической схемы и топологического проектирования может оказаться, что некоторые цепи имеют слишком большую нагрузку или некоторые операции требуют слишком большое число логических элементов. Оптимизация микроархитектуры на этих этапах позволяет поднять быстродействие до десятков процентов, потребление питания и площадь кристалла снизить на единицы процентов. Для микропроцессора 1890ВМ5Ф на данном этапе удалось поднять быстродействие почти в 1,5 раза за счет незначительной оптимизации логической модели, позволившей уменьшить число логических элементов на критических функциях и уменьшить длину трасс на кристалле на критических путях. Было проведено несколько десятков итерационных процессов для данной микросхемы.

3. Разработка поведенческой модели имеет несколько задач. На первом этапе разработки микросхемы поведенческая модель необходима для оптимизации архитектуры микросхемы. Особую значимость модель имеет для параллельной с разработкой микросхемы разработки программного обеспечения.

В созданной методике проектирования поведенческая модель также важна для тестирования логической модели микросхемы. На всем пути проектирования микросхемы происходит постоянное сравнение функционирования логической и поведенческой моделей. Модели проектируются независимыми коллективами разработчиков и, следовательно, вероятность одинаковых ошибок мала. Созданные средства проектирования позволяют на каждом такте процесса моделирования сравнивать состояния всех регистров поведенческой и логической моделей. При наличии различий состояний регистров делается соответствующее оповещение для поиска ошибки. Особую эффективность такая методика показала при создании сложнофункциональных микросхем. При создании суперскалярного 64-разрядного микропроцессора 1890ВМ5Ф такой подход позволил выявить свыше 30 % ошибок, сделанных в логической модели.

4. Тестовое программное обеспечение создается для каждого этапа проектирования микросхем. Тесты, необходимые для разработки микросхемы, можно разбить на четыре группы: тесты функционального контроля, сертификационные тесты, тесты для отбраковки кристаллов и микросхем и тесты для испытаний и периодического контроля. Наиболее сложными являются тесты функционального контроля. Во всех тестах должны быть предусмотрены: автоматический контроль выполнения тестов с записью полного состояния микросхемы в случае сбоя; возможность пошагового выполнения тестов; выдача (сохранение на диске) состояний микросхемы в запрашиваемые моменты времени.

5. Разработка логической модели уровня RTL в маршруте проектирования микросхем серии 1890 ведется на языке *Verilog*, хотя мог бы быть и язык VHDL. Разработка модели предполагает следование созданным на предыдущих этапах документам — архитектуре микросхемы, и техническому заданию. Для достижения требований ТЗ в ряде случаев приходится делать итерационный процесс по разработке архитектуры, т. е., если ТЗ выполнить не удается по каким-либо параметрам, приходится возвращаться на этап разработки архитектуры. Другая возможность улучшения показателей микросхемы — написание моделей отдельных узлов на уровне Gate. Однако это может привести к увеличению времени проектирования этих узлов в несколько раз. На этом же этапе должны быть разработаны также логические модели всех заказных узлов. При разработке заказных узлов необходимо достичь полного соответствия функционирования этих двух моделей. На этапе создания логической модели возможно также определение дополнительных узлов, которые должны быть разработаны заказным путем. Верификация модели происходит с использованием технологических плат на базе микросхем FPGA и тестового ПО.

6. В разработанной методике на этапе уточнения технологии производства, выбора библиотек и IP-блоков происходит определение дальнейшего маршрута проектирования и оптимизация микроархитектуры проектируемой микросхемы. Особое место на этом этапе занимает проектирование сложнофункциональных микросхем. Одним из основных вопросов при создании таких микросхем является разработка блоков предельного быстродействия, взаимодействие их с блоками меньшего быстродействия и аналоговыми блоками. В настоящее время для решения задач такого класса применяют два основных подхода:

- использование заказного проектирования при разработке блоков предельного быстродействия;
- разработка специализированных САПР для синтеза и оптимизации проекта на основе специально разработанных библиотек элементов.

В обоих этих подходах используется оптимизация на уровне архитектуры всей системы и архитектуры отдельных блоков. Первый подход отличается тем, что в нем используется весь спектр доступных способов по оптимизации проекта на всех стадиях — от разработки архитектуры до получения топологии. Существенными недостатками такого подхода являются: высокая стоимость, большое время разработки, сложность быстрого перехода на новую технологию. Второй подход характеризуется разработкой и использованием набора САПР и технологических библиотек элементов для проектирования микросхем выделенного класса. Такой подход позволяет приблизиться к быстродействию полностью заказных схем с проигрышем (в пределах 10—50 %) по току потребления и занимаемой площади. Недостатком данного подхода является то, что необходимы большие вложения в разработку специализированных САПР и технологических библиотек. Однако такие библиотеки и САПР являются универсальными для создания микросхем выделенного класса.

В основе созданной методики лежит использование коммерчески доступных САПР совместно с заказным проектированием отдельных блоков. Для достижения эффекта, сопоставимого с применением специализированных САПР и заказного проектирования, разработана методика, позволяющая выделить блоки разного быстродействия и оценить возможности автоматического синтеза для каждого блока. Разработаны методика синтеза, методика сокращения цепочки буферов для большого числа нагрузок, установленных на выходе одного триггера, методика определения наиболее критичных параметров дерева синхронизации и выделения элементов, требующих ручной расстановки [9].

Выделение блоков предельного быстродействия, которые можно получить средствами синтеза, проводится на основе коэффициента количества элементов $K_Э$:

$$K_Э = N_С/N_п,$$

где $N_С$ — оценочное число двухвходовых элементов типа И-НЕ (ИЛИ-НЕ), получаемых средством

синтеза, необходимых для реализации самой длинной цепочки обработки данных в разрабатываемой модели; $N_{\text{п}}$ — максимальное число таких же двухвходовых элементов, которое может быть использовано для построения цепочки обработки данных, работающей на требуемой максимальной частоте передачи сигнала. Данный коэффициент оценивается для всех блоков проекта. Это позволяет разделить проект на блоки, получаемые средствами синтеза, и блоки, получаемые с использованием заказного проектирования.

7. Создание принципиальной электрической схемы на уровне транзисторов является итерационным процессом. Сначала проводится предварительный синтез. Этот этап позволяет получить более точные, по сравнению с этапом разработки архитектуры, оценки производительности, площади и энергопотребления. На этом этапе выделяются высокоскоростные цифровые блоки, требуемого быстродействия которых возможно достичь за счет специализированного маршрута проектирования, такого как: разделение блоков предельного быстродействия на подблоки; ручная расстановка; специализированный маршрут синтеза.

Для блоков, имеющих нарушение быстродействия, необходимо провести анализ архитектуры в целях выделения подблоков, имеющих нарушения по быстродействию. На основании этих данных проводится разделение проекта на подблоки. Использование итерационного синтеза подблоков позволяет значительно сократить сроки выбора оптимального решения в целях достижения требуемого быстродействия. Применение специализированного маршрута разработки для подблоков позволяет добиться предельного быстродействия без использования заказного проектирования.

Специальная методика синтеза основана на использовании ряда подходов, позволяющих по сравнению с заказным проектированием добиться сокращения сроков достижения требуемого быстродействия. Основное внимание в этой методике уделяется повышению частоты работы за счет следующих подходов:

- поиск путей распространения сигналов, не требующих срабатывания за один период синхросигнала;
- выделение подблоков, работающих в режимах, в которых частота синхросигнала снижена в несколько раз по сравнению с максимальной;
- сокращение числа триггеров, находящихся в конце цепочек комбинаторных элементов, нагруженных на один выход триггера;
- сокращение числа элементов в цепочке между триггерами;
- минимизация расфазировки синхросигналов и минимизация фронтов синхросигнала.

Сокращение цепочки буферов, установленной на выходе одного триггера, проводится следующим образом. Создается несколько управляющих триггеров и проводится перераспределение нагрузки между ними. Это позволяет при разработке топологии получить оптимальное распределение элементов по площади кристалла и добиться требуемого быстродействия.

Если предпринятые действия не приводят к получению требуемых показателей быстродействия, проводится изменение архитектуры блоков и принципов передачи данных между блоками.

После окончания синтеза микросхемы необходимо провести верификацию списка соединений на соответствие логической модели уровня RTL.

8. Топологическое проектирование разделяется на заказное и автоматическое.

Если в синтезируемом проекте имеются отдельные заказные узлы, сначала размещаются заказные узлы, а затем проводится топологическое проектирование всего кристалла. Этот процесс, как правило, итерационный, нужен для достижения требуемых параметров. Трассировка цепей проводится в два этапа: сначала выполняется трассировка шин "земли" и "питания", а затем трассировка сигнальных цепей.

Если в процессе разработки топологии требуемого быстродействия подблоков не достигают, необходимо определить значение расхождения и в зависимости от него предпринять соответствующие действия. Если расхождение составляет единицы процентов, то его, как правило, удается устранить изменением параметров, задаваемых при топологическом проектировании критичных подблоков. Если расхождение составляет порядка 10 %, необходимо определить перечень критичных цепей, длину цепочек и нагрузочные характеристики буферов в этих цепочках. Дальше необходимо провести оптимизацию этих цепочек. Эффективным способом оптимизации является подгонка нагрузочных способностей выходных буферных элементов под реальные нагрузки, что позволяет в ряде случаев также сократить число элементов в цепочке. Следующим шагом является ручная расстановка следующих элементов:

- подблоков формирования сигналов выходных шин;
- подблоков приема данных от входных шин;
- элементов, используемых для формирования синхросигналов с кратными частотами.

Для таких блоков используются следующие рекомендации по расстановке. Для подблоков, формирующих выходные сигналы шин, элементы формирования выходных разрядов необходимо устанавливать рядом с площадочными элементами соответствующих разрядов. Для подблоков, принимающих входные сигналы шин, элементы приема входных разрядов необходимо устанавливать также рядом с площадочными элементами соответствующих разрядов. Для шин, у которых критичным параметром является задержка от входного синхросигнала до выходных данных и не используется ФАПЧ (фазовая автоподстройка частоты), необходимо строить отдельное дерево синхросигнала для выходных триггеров. Элементы, формирующие синхросигналы с кратными частотами, необходимо устанавливать рядом с источником синхросигнала и на минимальном расстоянии друг от друга.

Это позволяет:

- сократить задержку переключения выходных сигналов и минимизировать ее разброс;
- повысить запас по параметрам *setup* и *hold* для входных данных относительно фронта синхросигнала и также минимизировать его разброс;
- минимизировать разброс дерева синхронизации триггеров, используемых для приема входных данных с шин и для формирования выходных данных шин.
- минимизировать расфазировку синхросигналов с кратными частотами.

Если принимаемые шаги не приводят к достижению требуемого быстродействия, происходит возврат на этап синтеза или этап создания логической модели.

После проведения топологического проектирования необходимо выполнить следующие действия:

- верифицировать топологию на соответствие проектно-конструкторским нормам;
- экстрагировать паразитные элементы;
- верифицировать принципиальную схему с учетом реальных задержек в топологии.

Как показывает практический опыт, перед отправкой на завод топологии кристаллов важны не только проверки DRC (*Design Rule Check*), LVS (*Layout Ver sus Schematic*), ANTENNA, плотность металлов, обычно предоставляемые заводом-изготовителем, но и проверка на закоротки — так называемая ERC (*Electrical Rule Check*) проверка. Причем по своему составу ни одна из перечисленных выше проверок не перекрывает ERC. Эта проверка особенно актуальна, когда в одном кристалле присутствуют нескольких шин "питание" и "земля". Обычно такая ситуация бывает в проектах, где есть заказные аналоговые блоки, которым необходимы незашумленные "земля" и "питание", и цифровые блоки на библиотечных элементах, либо в проектах, содержащих высоковольтные и низковольтные транзисторы. Также положительный результат эта проверка дает в больших цифровых проектах, где невозможен надежный визуальный контроль.

В маршруте проектирования написаны программы для четырех ERC проверок.

1. Проверка наличия закороток между "питанием" и "землей" по металлам на уровне элементов. Проводимость объявляется только по металлическим межсоединениям элементов.

2. Проверка наличия закороток между "питанием" и "землей" по металлам и диффузиям на транзисторном уровне. Проводимость объявлена по металлам и диффузиям.

3. Проверка наличия закороток между "питанием" и "землей" по *N*-карману и подложке. Проводимость объявлена по металлам, диффузиям, *N*-карману и подложке. При этой проверке часто встречаются следующие ошибки: разрыв по металлу между контактной площадкой "питания" и *N*-карманом; разрыв по металлу между контактной площадкой "земли" и истоками транзисторов *n*-типов.

4. В этой проверке объявлены транзисторы, диоды, резисторы, емкости. Присутствует проводимость по всем проводящим слоям, включая все слои поликремния. Проводятся проверки на неподсоединенные:

- затворы, стоки, истоки, подложки транзисторов к схеме;
- узлы к "земле" и "питанию" через один или более транзистор, резистор, диод.

Несмотря на простоту проверок, после них в ряде проектов находились ошибки.

На данном этапе созданного маршрута проектирования в кристалл включают так называемые "ремонтные" элементы. "Ремонтные" элементы — это библиотечные логические ячейки, равномерно размещенные на кристалле и не задействованные в схеме. Число таких элементов определяется размером кристалла и обычно составляет порядка 100 штук. Первоначально затворы транзисторов таких ячеек подсоединяют к шинам "земли" или "питания". Если необходимо сделать небольшую коррекцию электрической схемы после изготовления микросхемы, то можно задействовать "ремонтные" элементы, сделав изменения только в слоях металлов и тем самым сократить число изменяемых масок. С учетом того что стоимость изготовления кристаллов при небольших партиях (6...12 пластин) для технологических норм 0,25...0,18 мкм определяется в основном стоимостью фотошаблонов, данный способ позволяет существенно сократить материальные затраты на изготовление исправленной версии СБИС. Такой подход позволяет также ускорить подготовку документации для изготовления кристаллов и их изготовление на 1—2 месяца.

9. Тестирование изготовленных кристаллов и микросхем разделяется на два этапа — исследовательский и производственный. Исследовательский этап направлен на поиск возможных ошибок, производственный — на выявление дефектов при изготовлении. На исследовательском этапе используется комплект тестов, созданный при разработке модели. Особую значимость приобретает выполнение системного и прикладного программного обеспечения. Весьма эффективным тестированием является тестирование микросхем сторонними организациями. Такое тестирование можно разделить на сертификационное, т. е. соответствие стандартам, и на уровне пользовательских задач. Для производственного тестирования в созданном маршруте проектирования во все проекты вводятся средства тестирования через порт JTAG, сканирующие цепочки и встраиваемые средства самотестирования [9]. Проведение испытаний микросхем и окончательное определение параметров микросхем осуществляются в соответствии с существующими ГОСТ.

Заключение

Созданы маршрут и методики проектирования КМОП-микросхем с технологическими нормами 0,18...0,5 мкм. С использованием этой методики разработано свыше 20 различных микросхем объемом от нескольких сот тысяч до де-

сятков миллионов транзисторов. Микросхемы изготовлялись как на отечественных, так и на западных фабриках (всего пять фабрик). Одним из положений данной методики является проектирование быстродействующих микросхем со сложной структурой. В рамках процедуры разработки таких схем проводится выделение блоков предельного быстродействия, требующих заказного или смешанного проектирования, и организация их взаимодействия с блоками меньшего быстродействия, создаваемых средствами синтеза. Было разработано два типа микросхем со сложной структурой: микросхемы графического контроллера (1890ВГ10Т и 1890ВГ14Т) и 64-разрядный суперскалярный микропроцессор 1890ВМ5Ф. В части разработки микросхем графических контроллеров методика дала возможность сократить сроки разработки в 3 раза и снизить стоимость проектирования в 3 раза по сравнению с полностью заказным проектированием. По сравнению с синтезируемым проектом частота функционирования микросхем графического контроллера увеличена более чем в 2 раза при увеличении времени проектирования в 2 раза. По своим параметрам микросхема 1890ВГ14Т, созданная с технологическими нормами 0,35 мкм, соответствует по производительности западным микросхемам, производящимся по нормам 0,18...0,25 мкм [9]. Использование методики при проектировании микропроцессора 1890ВМ5Ф позволило увеличить частоту функционирования более чем в 3 раза по сравнению с синтезируемым проектом.

Список литературы

1. **Евтушенко Н., Немудров В., Сырцов И.** Методология проектирования систем на кристалле, основные принципы, методы, программные средства // Электроника. 2003. № 6.
2. **Galaxy** Design Platform. http://www.synopsys.com/products/solutions/galaxy_platform.html, 8 мая, 2006.
3. **Discovery** Verification Platform. http://www.synopsys.com/products/solutions/discovery_platform.html, 18 июня, 2006.
4. **ENCOUNTER** DIGITAL IC DESIGN PLATFORM. Data Sheet, http://www.cadence.com/brochures/encounter_brochure.pdf, 2005.
5. **INCISIVE** FUNCTIONAL VERIFICATION PLATFORM. Data Sheet. http://www.cadence.com/brochures/incisive_platform.pdf, 2005.
6. **Бобков С. Г., Евлампиев Б. Е.** Схема тестирования высокочастотных блоков однокристального графического контроллера // Научная сессия МИФИ-2003. Сборник научных трудов. В 14 т. Т. 1. Автоматика. Электроника. Электронные измерительные системы. М.: Изд-во МИФИ, 2003. С. 173—174.
7. **Аряшев С. И., Бобков С. Г., Грузинова Е. В., Евлампиев Б. Е., Корниленко А. В., Сивакова Т. В.** Разработка микросхемы графического ускорителя с использованием ПЛИС FPGA Altera // 3-я научно-техническая конференция, г. Пушкинские горы, 2001 г. Электроника, микро- и наноэлектроника. Сборник научных трудов / Под ред. В. Я. Стенина. М.: Изд-во МИФИ, 2001. С. 28—33.
8. **Бобков С. Г.** Методика тестирования микросхем для компьютеров серии "Багет" // Программные продукты и системы. 2007. № 3. С. 2—5.
9. **Бобков С. Г., Евлампиев Б. Е.** Разработка методик проектирования быстродействующих СБИС со сложной структурой класса микросхем графического контроллера // Информационные технологии и вычислительные системы. 2005. № 1. С. 88—104.
10. **Бобков С. Г., Евлампиев Б. Е., Сидоров А. Ю.** Блок самотестирования внутренней памяти // 1-я Всероссийская научно-техническая конференция, Подмосковье, 2005 г. Проблемы разработки перспективных микроэлектронных систем. Сб. науч. тр. / Под. общ. ред. А. Л. Стемпковского. М.: Изд. ИППМ РАН, 2005. С. 222—228.

УДК 681.5:004.414.2

П. Н. Бибило, д-р техн. наук, проф.,
Объединенный институт проблем информатики
Национальной академии наук Беларуси,
г. Минск

Совместное использование синтезаторов Leonardo и XST при проектировании цифровых схем на FPGA

Описываются результаты экспериментального исследования реализаций цифровых схем на FPGA. Показывается, что более эффективную реализацию комбинационных схем на FPGA во многих случаях можно получить путем совместного использования двух синтезаторов — Leonardo и XST.

Введение

Эффективной элементной базой реализации цифровых схем являются FPGA (*Field-Programmable Gate Arrays*). Синтез логических схем FPGA может быть осуществлен в различных системах проектирования (синтезаторах), среди которых наиболее распространенными являются синтезатор XST [1] системы проектирования WebPack ISE, Synplify и Leonardo Spectrum (далее Leonardo) [2]. Кроме схем FPGA и CPLD (*Complex Programmable Logic Devices*) синтезатор Leonardo позволяет вести синтез схем в библиотеке синтеза, заданной непосредственно проектировщиком [2]. Такие схемы могут быть реализованы в составе заказных СБИС. Синтезатор Leonardo может оптимизировать схемы по различным критериям (площадь — *area*, быстродействие — *delay*) и учитывать разнообразные технологические ограничения. Синтез схемы от алгоритмических описаний на языках VHDL и Verilog

в Leonardo разбит на два этапа — *высокоуровневый синтез*, результатом которого является так называемое промежуточное RTL0-описание (RTL — *Register Transfer Level*), и *технологическое отображение* (*technology mapping*) [2]. Далее речь будет идти об исходных описаниях и различных RTL-описаниях, представленных только на языке VHDL.

RTL-описание может быть получено повторно в Leonardo (этим он выгодно отличается от других синтезаторов) из описания синтезированной логической схемы с помощью специальной команды unmap. Назовем это описание RTL1. Естественно, описание RTL1 функционально эквивалентно исходному алгоритмическому VHDL-описанию и описанию RTL0. По описанию RTL1 может быть проведен ***повторный*** синтез схемы FPGA как в синтезаторе Leonardo, так и в других синтезаторах, получающих схемы по VHDL-описаниям. В практике проектирования было замечено, что повторно построенная схема часто обладает лучшими характеристиками, чем схема, построенная по исходному VHDL-описанию. Таким образом, синтезатор Leonardo позволяет путем повторного (итеративного) синтеза уменьшать сложность схемы. Однако такое уменьшение сложности схем происходит не всегда.

В связи с этим предлагается выбирать лучшие из реализаций в Leonardo, полученные путем итеративного повторного синтеза, и подавать их на вход другого синтезатора — XST. В этом случае может быть получена схема меньшей сложности, чем схема, получаемая каждым из синтезаторов Leonardo, XST в отдельности. Результаты соответствующего эксперимента над различного вида исходными описаниями цифровых схем представлены в данной статье. Под ***сложностью*** схемы FPGA на этапе логического синтеза будем понимать число программируемых элементов FPGA, выражаемое в числе Slices [3]. Две одинаковые секции (*Slice*) образуют основной логический блок FPGA, называемый конфигурируемым логическим блоком. В одну секцию входят два четырехвходовых функциональных генератора LUT (*Look Up Table*), логика ускоренного переноса и запоминающий элемент. Один LUT может реализовать любую булеву функцию от четырех аргументов, заданную таблицей истинности. Кроме того, LUT может быть использован как синхронное ОЗУ размерностью 16 × 1 бит. Конфигурируемые логические блоки в FPGA располагаются в виде матрицы, например, в микросхеме XCTS100 семейства SPARTAN II содержится 600 конфигурируемых логических блоков, организованных в виде матрицы 20 × 30.

Синтез схем FPGA в синтезаторе XST системы WebPack

После создания проекта в системе WebPack ISE 8.1i синтез осуществлялся единообразно (при всех настройках синтезатора XST по умолчанию) — это соответствует распространенному варианту практического использования синтезатора. Сложность полученных логических схем подсчитывается в числе Slices. Это число выдается в результирующий протокол синтеза.

Синтез схем FPGA в синтезаторе Leonardo

Во всех случаях синтез в Leonardo также проводился стандартным образом — критерием оптимизации была площадь (*area*) схемы. Из результирующих протоколов синтеза бралось значение сложности — число Slices.

Примечание. Число Slices в результирующих протоколах Leonardo в 2 раза меньше числа LUT. В синтезаторе XST все аппаратурные затраты пересчитываются в число Slices, отдельно выдается число LUT.

Управление синтезом схем в Leonardo осуществлялось с помощью скриптов. Пример скрипта для синтеза схемы `add6` (файл с VHDL-описанием имеет имя `add6.vhd`) дан в листинге 1. Познакомиться с подобными скриптами для управления процессом синтеза в Leonardo можно по книге [2].

Листинг 1. Скрипт для управления синтезом схемы в Leonardo

```
clean_all;
set encoding Gray;
set modgen_select Smallest;
set asic_auto_dissolve_limit 500;
set auto_dissolve_limit 500;
read add6.vhd;
load_library xis2.syn;
set -hierarchy flatten
set effort standard
optimize -target xis2 -macro -area -ef-
fort standard -hierarchy flatten
report_area -cell_usage
```

Если требовалось получить RTL-описание для повторного синтеза, то в скрипт (листинг 1) добавлялись две команды:

```
unmap
auto_write add6.vhd
```

Синтез в библиотеке xis2 соответствует синтезу для микросхемы XC2S100 семейства SPARTAN II.

Организация экспериментов

Целевая микросхема FPGA. Во всех экспериментах в качестве целевой микросхемы была выбрана микросхема XC2S100 семейства SPARTAN П.

Исходные данные. Эксперимент были проведен на 40 примерах четырех видов исходных функциональных VHDL-описаний, взятых из практики проектирования цифровых схем.

Вид 1. Программируемые логические матрицы (ПЛМ), реализующие системы ДНФ полностью определенных функций, описывающие комбинационную логику в матричной форме (20 примеров). В качестве исходных данных были отобраны примеры из библиотеки схем [4].

Вид 2. Логические уравнения, описывающие многоуровневую комбинационную логику в базисе И, ИЛИ, НЕ (15 примеров). В качестве исходных данных были взяты примеры из известной библиотеки MCNC benchmark.

Вид 3. Системы слабоопределенных булевых функций, описывающие таблицы микрокоманд отечественных микроконтроллеров (два примера). Таблица представляет собой кодовый преобразователь: для каждого булева (двоичного) набора значений входных переменных задавался соответствующий булев набор значений выходных переменных (функций). На остальных наборах, не вошедших в таблицу, значения всех функций полагаются неопределенными. Таким образом, каждая таблица микрокоманд интерпретировалась как система не полностью определенных (частичных) булевых функций. В языке VHDL для типа данных std_logic и std_logic_vector неопределенное значение (*don't care*) обозначается через "-".

Рассмотрим **пример** VHDL-описания системы частичных функций. Описание `umn` (листинг 2) представляет собой умножитель пары чисел, представленных в двоичном коде и выбираемых из множества {`0, 1, 2`}, т. е. неполный умножитель. Данное устройство описывается системой частичных функций, заданной в табл. 1. Если одно из умножаемых чисел представляет собой число 3, то значения выходов не определены (равны неопределенному значению "-"). Для кодирования каждого из умножаемых чисел нужны две булевы переменные. Первое число задается в виде вектора `a = (a(1), a(0))`, второе — в виде вектора `b = (b(1), b(0))`, старшие разряды — `a(1), b(1)`; старший разряд произведения — `s(2)`, для представления произведения `s = (s(2), s(1), s(0))` достаточно трех булевых переменных.

Таблица 1

Система частичных функций

a (1)	a (0)	b (1)	b (0)	s (2)	s (1)	s (0)
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	—	—	—
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	1	0
0	1	1	1	—	—	—
1	0	0	0	0	0	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	—	—	—
1	1	0	0	—	—	—
1	1	0	1	—	—	—
1	1	1	0	—	—	—
1	1	1	1	—	—	—

Листинг 2. VHDL-описание системы частичных функций

```
library ieee;
use ieee.std_logic_1164.a11;
entity umn is
  port (a, b : in std_logic_vector (1 downto 0);
        s : out std_logic_vector (2 downto 0));
end umn;
architecture BEHAVIOR of umn is begin
  s <=
    "000" when a & b = "0000" else
    "000" when a & b = "0001" else
    "000" when a & b = "0010" else
    "000" when a & b = "0100" else
    "001" when a & b = "0101" else
    "010" when a & b = "0110" else
    "000" when a & b = "1000" else
    "010" when a & b = "1001" else
    "100" when a & b = "1010" else
    "---";
end BEHAVIOR;
```

В подобной форме задавались описания verg1, verg2 (табл. 2) систем частичных функций, описывающих таблицы микрокоманд. Например, для схемы verg1 число входных переменных равно 17, число $k = 2003$ (строк таблицы) значительно меньше числа 2^{17} всех наборов булева пространства размерности $n = 17$.

Вид 4. Алгоритмические (функциональные) описания: uart — универсальный асинхронный приемопередатчик, осуществляющий прием и передачу информации, представленной последовательным кодом; watchdog — сторожевой таймер; timer2 — таймер-счетчик (3 примера).

Этапы эксперимента. Для каждого из 40 исходных VHDL-описаний в эксперименте выполнялись следующие действия, разбитые на шесть этапов.

Этап 1. Синтез схемы в Leonardo по исходному VHDL-описанию; полученное решение называется далее базовым решением Leonardo и обозначается Base_Leo. Получение RTL0-описания схемы применением команды unmap.

Этап 2. Повторный синтез в Leonardo по RTL-описаниям, начиная с RTL1-описания (пять итераций синтеза — пять схем).

Этап 3. Нахождение среди пяти RTL-описаний (RTL1, ..., RTL5) того описания, по которому получена схема наименьшей сложности. Сложность такой схемы приведена в табл. 2 в столбце

Таблица 2

Результаты выполнения этапов 1—3 эксперимента

Вид исходного описания	Имя схемы	n	m	k	$S_{ПЛМ}$	Leonardo S_{FPGA} (slices)						
						Base_Leo	RTL1	RTL2	RTL3	RTL4	RTL5	best_RTL
ПЛМ	add6	12	7	1092	33852	33	19	16	16	*13*	13	**13**
	addm4	9	8	480	12480	48	62	*50*	51	51	51	50
	b2	16	17	110	5390	238	223	223	218	210	*207*	**207**
	b9	16	5	123	4551	20	23	23	25	26	24	23
	bc0	26	11	479	30177	179	167	163	168	*162*	164	**162**
	chkn	29	7	153	9945	71	72	73	*70*	72	72	**70**
	dk48	15	17	42	1974	32	30	29	30	29	30	**29**
	ibm	48	17	173	19549	47	39	39	38	38	*34*	**34**
	in0	15	11	135	5535	70	72	*71*	75	71	71	71
	in2	19	10	137	6576	97	96	96	*95*	96	95	**95**
	intb	15	7	664	24568	212	211	*208*	208	209	208	**208**
	m2	8	16	96	3072	30	30	30	30	30	31	30
	m3	8	16	128	4096	35	35	35	35	35	35	35
	misg	56	23	75	10125	24	23	23	23	*22*	22	**22**
	prom2	9	21	287	11193	212	*211*	214	212	212	212	**211**
	shift	19	16	100	5400	27	27	27	27	27	27	27
	signet	39	8	124	10664	72	68	65	63	62	*61*	**61**
	sym10	10	1	837	17577	35	35	35	36	44	41	35
	tial	14	8	640	23040	327	*211*	340	211	263	247	**211**
	ts10	22	16	128	7680	56	*24*	24	24	24	24	**24**
Многоуровневые схемы	C1355	41	32			40	38	38	38	38	38	**39**
	C432	36	7			35	30	29	29	29	29	**29**
	C499	41	32			35	36	36	36	36	36	36
	C880	60	26			25	20	20	20	20	20	**20**
	Cht	47	36			23	23	23	23	23	23	23
	Count	35	16			16	*16*	16	16	16	16	16
	Dalu	75	16			48	43	*40*	41	44	43	**40**
	Frg1	28	3			31	32	29	29	28	*26*	**26**
	Frg2	143	139			217	202	198	*191*	194	191	**191**
	I6	138	67			16	*16*	16	16	16	16	16
	I8	133	81			199	199	*170*	199	170	199	**170**
	Too_large	38	3			99	100	94	*87*	94	92	**87**
	X1	51	35			83	73	68	70	*67*	69	**67**
	X3	135	99			148	151	*148*	151	148	151	148
	X4	94	71			83	82	79	*78*	79	79	**78**
Частичные функции	Verg1	17	61	2003		1009	754	715	689	674	*665*	**665**
	Verg2	18	63	2129		1371	1162	1148	1155	1121	*1111*	**1111**

best_RTL. RTL-описание, по которому получено решение best_RTL, назовем before_unmap. Для каждого RTL-описания before_unmap бралось также описание after_unmap, полученное в результате применения команды unmap к полученной логической схеме. Таким образом, по выбранному RTL-описанию формировались два RTL-примера, соответствующие потоки примеров названы before_unmap и after_unmap.

Этап 4. Синтез в XST схемы по исходному VHDL-описанию. Полученное решение называется базовым решением XST и обозначается Base_XST.

Этап 5. Синтез в XST схемы из потока примеров before_unmap.

Этап 6. Синтез в XST схемы из потока примеров after_unmap.

Таким образом, для каждого исходного VHDL-описания синтезировалось девять схем FPGA.

Для видов 1—3 описаний результаты эксперимента представлены в табл. 2, 3. Для вида 4 результаты эксперимента даны в табл. 4, 5. В табл. 2, 4 даны результаты синтеза в синтезаторе Leonardo, в табл. 3, 5 — в синтезаторе XST. Во всех таблицах жирным шрифтом выделены решения, **улучшающие** базовые решения, полученные соответствующим синтезатором. Курсивом в табл. 2 выделены решения, по которым формируется поток примеров after_unmap. В табл. 2, 4 используются следующие обозначения:

n — число входов схемы;

m — число выходов схемы;

k — число промежуточных шин ПЛМ (число элементарных конъюнкций в реализуемой системе ДНФ булевых функций). Для вида 3 примеров

Таблица 3

Результаты этапов 4—6 эксперимента

Вид	Имя схемы	XST Base_XST S_{FPGA} (slices)	XST before_unmap S_{FPGA} (slices)	XST after_unmap S_{FPGA} (slices)
ПЛМ	add6	7	12	**6**
	addm4	50	**48**	60
	b2	239	**237**	253
	b9	23	**22**	**20**
	bc0	602	**217**	**211**
	chkn	80	91	90
	dk48	29	33	33
	ibm	39	40	39
	in0	94	99	96
	in2	101	**81**	**80**
	intb	246	256	256
	m2	32	32	**30**
	m3	60	**58**	**56**
	misg	18	18	18
	prom2	346	325	348
	shift	29	**27**	29
	signet	55	55	65
	sym10	13	14	14
	tial	233	320	237
	ts10	28	28	28
Многоуровневые схемы	C1355	41	41	41
	C432	33	**31**	**31**
	C499	40	43	43
	C880	26	26	**22**
	Cht	26	26	26
	Count	19	19	19
	Dalu	44	45	**43**
	Frg1	23	26	28
	Frg2	204	209	208
	I6	18	18	18
	I8	164	**146**	**137**
	Too_large	91	103	*
	X1	69	*	*
	X3	152	179	165
	X4	81	84	84
Частичные функции	Verg1	1062	**575**	**762**
	Verg2	1393	**527**	**595**

* — синтезаторы несовместимы по данным.

схем число k означает число строк в таблице микрокоманд;

$S_{ПЛМ}$ — площадь схемы ПЛМ, вычисляемая по формуле $S_{ПЛМ} = (2n + m) \times k$(бит);

S_{FPGA} — сложность схемы FPGA (в числе Slices).

Для вида 1 описаний эксперимент можно интерпретировать как замену схем ПЛМ схемами FPGA.

Обсуждение результатов экспериментов

Вид 1. В 13 примерах из 20 ПЛМ итеративный повторный синтез позволил улучшить базовое решение Base_Leo, получаемое синтезатором Leonardo по исходному VHDL-описанию схем ПЛМ. Совместное использование сначала синтезатора Leonardo, затем XST позволяет улучшить решение (см. табл. 3), получаемое синтезатором XST, в восьми случаях из 20 примеров схем ПЛМ.

Вид 2. Для многоуровневой комбинационной логики в 10 примерах из 15 итеративный повторный синтез позволил улучшить базовое решение Base_Leo, получаемое синтезатором Leonardo по исходному VHDL-описанию. Совместное использование сначала синтезатора Leonardo, затем XST позволяет улучшить решение, получаемое синтезатором XST, в четырех случаях из 15 примеров схем, синтезированных по уравнениям, описывающим многоуровневую логику (см. табл. 3).

Вид 3. В обоих примерах verg1, verg2 систем частичных функций итеративный повторный синтез позволил улучшить базовое решение Base_Leo. Совместное использование сначала синтезатора Leonardo, затем XST позволяет значительно улучшить решение, получаемое синтезатором XST, в обоих примерах verg1, verg2.

Вид 4. Для трех примеров uart, watchdog, timer2 алгоритмических описаний итеративный синтез в Leonardo не позволяет улучшить базовое решение Base_Leo (табл. 4). Результаты по итерациям RTL3, RTL4, RTL5 в табл. 3 не приводятся — они хуже результатов, полученных на итерациях RTL1, RTL2. В табл. 5 приведены результаты синтеза в синтезаторе XST для описаний вида 4. В алгоритмических описаниях uart, watchdog, timer2 имеется много

Таблица 4

Синтез схем по алгоритмическим описаниям в Leonardo, результаты этапов 1—3 эксперимента

Вид	Имя схемы	$n + m$	Leonardo S_{FPGA} (slices)								
			Base_Leo			RTL1			RTL2		
			slices	triggers	mux	slices	triggers	mux	slices	triggers	mux
Алгоритмические описания	uart	26	61	97	24	68	97	13	82	97	25
	watchdog	20	16	26	22	25	26	2	25	26	2
	timer2	31	114	112	74	145	112	38	152	112	37

Таблица 5

Синтез схем по алгоритмическим описаниям в XST, результаты этапа 4 эксперимента

Вид	Имя схемы	$n + m$	XST Base_XST	
			slices	triggers
Алгоритмические описания	uart	26	79	97
	watchdog	20	18	21
	timer2	31	127	112

триггеров, комбинационная логика не выделена отдельно, а "размыта" по всей схеме. Для получения более простых логических схем можно выделить отдельно комбинационную часть и провести ее оптимизацию так, как предлагается в работе [5].

Заключение

Для алгоритмических описаний отдельно выделенной комбинационной логики итеративный повторный синтез в Leonardo во многих случаях позволяет уменьшить сложность схемы FPGA, особенно это может быть успешным для систем не полностью определенных булевых функций. Совместное использование двух синтезаторов может быть успешным при реализации комбинационной логики большой размерности. Для алгоритмических описаний, приводящих к схемам с большим числом элементов памяти и "распределенной" комбинационной логикой, совместное использование двух синтезаторов, по-видимому, будет не столь успешным, хотя попытка улучшить решение не является безнадежной.

В общем можно заключить, что для улучшения схемных решений можно воспользоваться предложенной в статье методикой совместного использования нескольких синтезаторов, строящих схемы по VHDL-описаниям. Например, вместо синтезатора XST промежуточное RTL-описание может быть подано на вход синтезатора Synplify.

Список литературы

1. **Зотов Ю. В.** Проектирование цифровых устройств на основе ПЛИС фирмы XILINX в САПР WebPack ISE. М.: Горячая линия-Телеком, 2003. 624 с.
2. **Бибило П. Н.** Системы проектирования интегральных схем на основе языка VHDL. StateCAD, ModelSim, LeonardoSpectrum. М.: СОЛОН-Пресс, 2005. 384 с.
3. **Кузелин О. М., Кнышев Д. А., Зотов Ю. В.** Современные семейства ПЛИС фирмы Xilinx: Справочное пос. М.: Горячая линия-Телеком, 2004. 440 с.
4. http://wwwl.cs.columbia.edu/~cs4861/sis/espresso-examples/ex/
5. **Бибило П. Н., Кочанов Д. А.** Оптимизационные преобразования VHDL-моделей цифровых систем. // Современная электроника. 2006. N 5. С. 64 —66.

УДК 621.3.049.771.14

А. Б. Аюпов, Интел,
А. М. Марченко, д-р техн. наук,
ИППМ РАН

Метод оптимизации трассируемости в аналитическом алгоритме размещения стандартных ячеек

Предложен новый метод явной оптимизации трассируемости в аналитических алгоритмах размещения. Показан способ использования деревьев Штейнера в качестве модели цепи в алгоритме размещения. Результаты экспериментов показывают улучшение трассируемости на 15 % в сравнении с алгоритмом APlace.

Введение

В последние годы с переходом на субмикронные технологии постоянно возрастает сложность интегральных схем (ИС) и требования к алгоритмам САПР. В задаче размещения стандартных ячеек возникла потребность в новых алгоритмах, которые обеспечивали бы высокое качество и возможность оптимизации реальных характеристик СБИС, таких как производительность, трассируемость, потребляемая мощность и т. д. Это послужило причиной появления за последние несколько лет множества новых алгоритмов размещения. Для сравнения качества результатов новых алгоритмов одна из международных конференций по физическому проектированию [1] объявила в 2005 году конкурс алгоритмов размещения, в котором суммарная длина проводников после размещения являлась критерием качества. Аналитические алгоритмы размещения, использующие нелинейное программирование, такие как APlace [2], MPL5 [3], показали лучшие результаты.

В алгоритме APlace задача размещения формулируется как задача нелинейной оптимизации без ограничений со следующей целевой функцией:

$$Cost = C1 + C2 \rightarrow \text{минимум}, \qquad (1)$$

где $C1$ — оценка суммарной длины проводников (проводники представляют собой физическую

реализацию цепей); $C2$ — штрафная функция перекрытий между ячейками.

Эффективные методы решения задачи нелинейной оптимизации, такие как метод сопряженного градиента или квазиньютоновский метод, требуют, чтобы целевая функция (1) была гладкой. Поэтому длина проводников для функции $C1$ оценивается в работе [2] по следующей формуле:

$$C1 = \sum_{\substack{k \in \\ \text{набор} \\ \text{цепей}}} \left\{ \begin{array}{l} \max(x_1^k, ..., x_{n_k}^k) - \min(x_1^k, ..., x_{n_k}^k) + \\ \max(y_1^k, ..., y_{n_k}^k) - \min(y_1^k, ..., y_{n_k}^k) \end{array} \right\}, \quad (2)$$

где (x_i^k, y_i^k) — координаты i-го терминала k-й цепи, длина проводников цепи оценивается через полупериметр охватывающего цепь прямоугольника.

Негладкие функции max и min сглажены согласно следующим формулам:

$$\max(x_1^k, ..., x_{n_k}^k) = \eta \log\left(\sum_{i \in [1..n_k]} \exp\left(\frac{x_i^k}{\eta}\right) \right); \quad (3)$$

$$\min(x_1^k, ..., x_{n_k}^k) = -\eta \log\left(\sum_{i \in [1..n_k]} \exp\left(-\frac{x_i^k}{\eta}\right) \right). \quad (4)$$

Чем меньше η, тем лучше сглаженные формулы (3) и (4) приближают негладкие функции max и min.

Из формул (3) и (4) можно вывести сглаженную функцию для абсолютной величины:

$$|x_i^k| = \eta \log\left(1 + \exp\left(\frac{x_i^k}{\eta}\right)\right) + \eta \log\left(1 + \exp\left(-\frac{x_i^k}{\eta}\right)\right). \quad (5)$$

Штрафная функция перекрытий между ячейками $C2$ введена в целевую функцию для получения размещения с минимальным перекрытием ячеек. Детали по реализации функции $C2$ могут быть найдены в работе [2].

Суммарная длина проводников является классической функцией для оптимизации в алгоритмах размещения, так как может быть легко вычислена и коррелирует со многими параметрами схемы: производительностью, потребляемой мощностью и трассируемостью. Под трассируемостью понимается возможность проведения всех необходимых соединений между стандартными ячейками после этапа размещения. В условиях жестких ограничений на трассировочные ресурсы корреляции между трассируемостью и суммарной длиной проводников недостаточно, поэтому появились алгоритмы размещения, которые учитывают трассируемость явно в целевой функции. В аналитическом методе авторы [2] предлагают уменьшать плотность размещаемых ячеек в регионах с повышенной плотностью трасс. В других алгоритмах подобный эффект достигается с помощью "раздувания" (увеличения площади) ячеек [5, 6]. Такие методы используют результат работы алгоритма глобальной трассировки для вычисления перегруженных областей во время размещения и затем учитывают эту информацию в размещении. Данные алгоритмы имеют следующие недостатки: они не учитывают изменение трассировки, связанное с перемещением ячеек, а также не принимают во внимание множество трасс, проходящих над перегруженным регионом и не связанных с ячейками данного региона, но вносящих вклад в загруженность региона. В работе [6] используется линейное программирование для оптимизации трассируемости уже после того, как ячейки размещены. В этом случае изменения позиций ячеек для оптимизации трассируемости могут быть только локальными, так как во время оптимизации по-прежнему необходимо избегать перекрытий между ячейками.

В этой работе предложен алгоритм размещения стандартных ячеек, в котором оптимизируется топология трасс и трассируемость вместе с размещением ячеек, что не было изучено в предыдущих работах.

1. Постановка задачи

Задача размещения представлена как задача нелинейной оптимизации без ограничений, со следующей целевой функцией:

$$Cost = C1^* + C2 + C3 \rightarrow \text{минимум}, \quad (6)$$

где $C1^*$ — функция оценки суммарной длины трасс, построенных алгоритмом глобальной трассировки; $C2$ — штрафная функция перекрытий между ячейками [2]; $C3$ — функция трассируемости, которая следит за превышением трассировочной пропускной способности. Реализация новых функций $C1^*$ и $C3$ описана в разд. 2 и 3 соответственно.

2. Штейнеровская модель цепи и длина трасс

В качестве геометрической модели цепи в предложенном алгоритме размещения используется дерево Штейнера, построенное алгоритмом глобальной трассировки. На рис. 1, *а* представлен пример цепи, полученной алгоритмом глобальной трассировки. Ключевая идея предложенного метода — использовать дополнительные (штейнеровские) точки трассировки в размещении наряду с размещаемыми ячейками. Перемещения штейнеровских точек определяются оптимизацией функции трассируемости (рис. 1, *а*, *б*). После значительного перемещения ячеек и штейнеровских точек возникает необ-

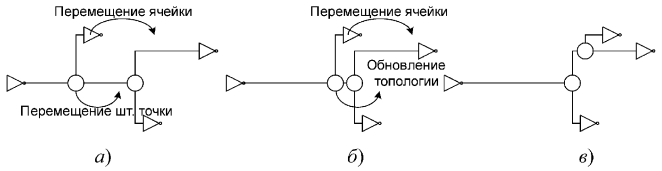


Рис. 1. Эволюция штейнеровского дерева:
а — незначительное перемещение ячеек и штейнеровских точек; *б* — значительное перемещение ячеек и штейнеровских точек; *в* — обновленное дерево

ходимость перестроения дерева с помощью алгоритма трассировки повторно (рис. 1, *в*).

Таким образом, каждая цепь, представленная деревом Штейнера, состоит из нескольких двухконцевых проводников (сегментов), соединяющих либо ячейку со штейнеровской точкой, либо две ячейки, либо две штейнеровские точки.

Использование дерева Штейнера в качестве модели цепи позволяет оптимизировать длину трассы цепи. Таким образом, оптимизируется сумма полупериметров всех составных сегментов, входящих в штейнеровское дерево данной цепи. Тогда, используя выражение (2) для двухконцевых сегментов, можно определить $C1^*$:

$$C1^* = \sum_{\substack{k \in \\ \text{набор} \\ \text{цепей}}} \sum_{\substack{s \in \\ \text{сег-ты} \\ \text{цепи } k}} \{\max(x_1^{s,k}, x_2^{s,k}) - \min(x_1^{s,k}, x_2^{s,k}) + \max(y_1^{s,k}, y_2^{s,k}) - \min(y_1^{s,k}, y_2^{s,k})\}, \quad (7)$$

где $(x_1^{s,k}, y_1^{s,k})$ и $(x_2^{s,k}, y_2^{s,k})$ — координаты терминалов сегмента s цепи k, которыми являются либо ячейка, либо точка Штейнера.

3. Трассируемость

Для того чтобы вычислить трассируемость в некоторой области, данная область разбивается горизонтальными и вертикальными разрезами на прямоугольные клетки одинакового размера. Для каждой клетки вычисляется локальная характеристика трассируемости как превышение плотности трасс над пропускной способностью клетки отдельно по вертикальным и горизонтальным направлениям. Трассируемость для всей области вычисляется как сумма локальных составляющих для каждой клетки. Эта характеристика используется в задаче оптимизации в виде следующей штрафной функции:

$$C3 = \sum_k \begin{bmatrix} \left(\sum_i RD_h(s_i, b_k) - Cap_h(b_k)\right)^2 + \\ \left(\sum_i RD_v(s_i, b_k) - Cap_v(b_k)\right)^2 \end{bmatrix}, \quad (8)$$

где $RD_h(s_i, b_k)$ и $RD_v(s_i, b_k)$ — занятость клетки b_k сегментом цепи s_i, или вклад сегмента цепи s_i в плотность трассировки над клеткой b_k по вертикальному и горизонтальному направлениям; $Cap_h(b_k)$ и $Cap_v(b_k)$ — горизонтальная и вертикальная пропускная способность клетки b_k соответственно.

Вероятностная оценка занятости клетки двухконцевой цепью s_i была впервые использована в работе [6]:

$$RD(s_i) = \frac{l(s_i)}{bbox\_area(s_i)},$$

где $l(s_i)$ — относительная длина цепи s_i, выраженная числом клеток, которые она занимает; $bbox\_area(s_i)$ — число клеток, попадающих в охватывающий сегмент s_i прямоугольник. Такая формулировка подразумевает, что каждая клетка, находящаяся в охватывающем сегмент прямоугольнике, имеет одно и то же значение занятости $RD(s_i)$. Для применения такой оценки к задаче оптимизации формула для $RD(s_i)$ была обобщена на всю площадь трассировки с помощью функции $SAB(s_i, b_k)$:

$$SAB(s_i, b_k) = \begin{cases} 1, \text{ охватывающий сегмент } s_i \\ \text{прямоугольник покрывает } b_k, \\ 0 \text{ — иначе.} \end{cases} \quad (9)$$

Используя проекции на горизонтальное и вертикальное направления, можно получить следующие выражения для оценки занятости клетки цепью:

$$\begin{aligned} RD_h(s_i, b_k) &= SAB(s_i, b_k) RD_h(s_i) = \\ &= SAB(s_i, b_k) \frac{\frac{|x_2 - x_1|}{w}}{\left(\frac{x_2 - x_1}{w} + 1\right)\left(\frac{y_2 - y_1}{h} + 1\right)}; \\ RD_v(s_i, b_k) &= SAB(s_i, b_k) RD_v(s_i) = \\ &= SAB(s_i, b_k) \frac{\frac{|y_2 - y_1|}{h}}{\left(\frac{x_2 - x_1}{w} + 1\right)\left(\frac{y_2 - y_1}{h} + 1\right)}, \end{aligned} \quad (10)$$

где (x_1, y_1) и (x_2, y_2) — координаты терминалов двухконцевого сегмента s_i, w и h ширина и высота клетки соответственно.

Заметим, что вычислять трассируемость для разных направлений трассировки необходимо потому, что пропускная способность металлов различается для разных направлений трасс.

Из определения функции (9) следует ее разрывность, и для задачи оптимизации требуется ее сглаживание. Используя координаты терминалов сег-

мента s_i и координаты (x_t, y_t) центра клетки b_k, функцию *SAB* можно определить следующим образом:

$$SAB(x_1, y_1, x_2, y_2, x_t, y_t) = \\ = Coin(x_1, x_2, x_t)\,Coin(y_1, y_2, y_t),$$

где функция *Coin* определена так:

$$Coin(x_1, x_2, x_t) = \begin{cases} 1, \text{ если } \min(x_1, x_2) < x_t < \max(x_1, x_2), \\ [0 \text{ — иначе.} \end{cases} \quad (12)$$

Приближая функцию (12) непрерывной функцией, было получено следующее выражение для функции *Coin*:

$$Coin(x_1, x_2, x_t) = \left(\frac{|x_t - (x_1 - \Delta)|}{\Delta} - \frac{|x_t - (x_1 + \Delta)|}{\Delta} \right) - \\ - \left(\frac{|x_t - (x_2 - \Delta)|}{\Delta} - \frac{|x_t - (x_2 + \Delta)|}{\Delta} \right). \quad (13)$$

Параметр Δ используется для контроля над точностью приближения. Рис. 2 иллюстрирует график функции $f(x_t) = Coin(0.1, 0.2, x_t)$ с параметром $\Delta = 0{,}02$.

"Разрывная" функция на рис. 2 представляет собой график функции (12), "негладкая" — график функции (13); "гладкая" — график функции (13) со сглаженной функцией $|f(x)|$, которая вычисляется по формуле (5).

4. Экспериментальные результаты

Блок-схема эксперимента. Блок-схема маршрута физического проектирования, используемого в эксперименте, представлена на рис. 3.

На вход алгоритма поступает список стандартных ячеек и список соединений между ними. Алгоритм состоит из трех основных этапов.

1. Начальное размещение, целевая функция для которого выглядит следующим образом:

$$Cost = C1 + C2. \quad (14)$$

Функция *C*1 в данной целевой функции оптимизирует сумму полупериметров охватывающих прямоугольников для цепей согласно формуле (2). Оптимизация трассируемости не включена в целевую функцию (14), так как на этапе начального размещения недостаточно информации о топологии проводников. Начальное размещение заканчивается, когда среднее значение перекрытия ячеек *cop* достигнет заранее установленного пользователем значения *copthresh*.

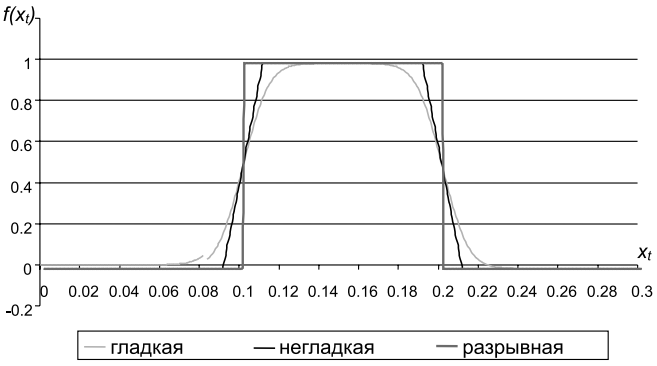


Рис. 2. График функции Coin

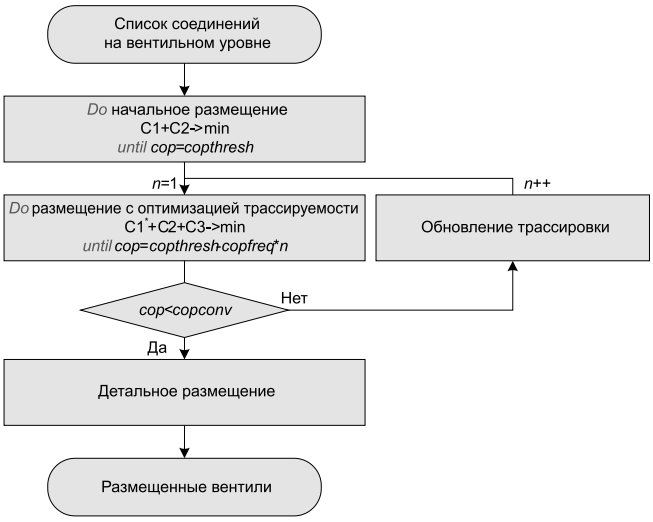


Рис. 3. Блок-схема эксперимента

2. Размещение с оптимизацией трассируемости с целевой функцией:

$$Cost = C1^* + C2 + C3. \quad (15)$$

После начального размещения вызывается алгоритм глобальной трассировки создания трасс для цепей. Каждая следующая итерация алгоритма глобальной трассировки происходит через заранее заданное пользователем значение перекрытия ячеек *copfreq* относительно значения на предыдущей остановке. Второй этап алгоритма размещения останавливается, когда среднее значение перекрытия ячеек достигнет заранее заданного значения *copconv*.

3. Этап детального размещения (или легализации) для полного устранения перекрытий, а также установки ячеек в ряды.

Методы ускорения сходимости. Перед каждой функцией *C*1, *C*1*, *C*2, *C*3 из формул (14), (15) существуют коэффициенты или веса. Эти веса и их динамические изменения очень важны для сходимости алгоритма размещения. В предложенном алгоритме все члены в целевой функции разделены на две группы: "жесткое ограничение" и "функция минимизации". К "жестким ограничениям" относятся штрафные функции, оптимизация которых необходима для получения удовлетворительного результата. В случае функций (14) и (15) *C*2 — функция с "жестким ограничением". К "функциям минимизации" относятся *C*1 и *C*1*, или штрафные функции, которые представляют собой нежесткие ограничения, т. е. ограничения,

Результаты работы алгоритмов PBPlace и APlace для тестовых примеров LGSynth

Bench	Nodes	#PIs	#POs	HRPM		RWL		TBO		MBO	
				APlace	PBPlace	APlace	PBPlace	APlace	PBPlace	APlace	PBPlace
alu2	590	10	6	3123,0	0,99	3454,2	1,00	4	1	1	1
alu4	1091	14	8	6783,8	1,00	7736,6	0,98	53	31	7	3
apex6	774	135	99	6087,5	1,01	6331,8	1,00	293	261	9	8
apex7	292	49	37	1518,0	0,99	1537,8	0,99	3	0	1	0
C1355	947	41	32	5287,7	1,00	5609,5	1,03	15	1	2	1
C1908	1146	33	25	7999,4	1,02	9037,7	1,02	7	2	2	1
C2670	1727	233	140	14555,4	1,03	15685,9	1,05	564	535	17	10
C3540	1825	50	22	12658,5	1,00	13923,0	1,01	20	4	2	2
C499	451	41	32	3426,7	1,02	3655,0	1,04	5	3	3	2
C6288	4771	32	32	18819,1	1,24	23927,9	1,18	76	13	10	3
C7552	5303	207	108	53585,7	1,00	57701,7	0,99	2362	1996	11	10
C880	297	60	26	2403,6	1,04	2615,0	1,07	1	2	1	1
count	165	35	16	673,4	1,00	667,0	1,00	0	0	0	0
dalu	2921	75	16	16801,9	1,01	18640,3	1,02	219	110	7	4
example2	354	85	66	2016,0	1,06	2145,2	1,08	13	15	3	2
i2	208	201	1	1748,7	0,97	1583,5	0,95	157	145	31	20
i3	102	132	6	1357,9	1,04	1244,6	1,04	43	37	10	7
i4	172	192	6	2032,8	1,02	1921,8	1,02	149	120	18	14
i5	312	133	66	2420,5	0,98	2294,3	0,97	175	134	15	13
i6	872	138	67	5977,0	1,05	6606,3	1,06	97	80	10	8
i7	1020	199	67	9029,9	1,02	9592,7	1,01	178	138	10	7
i9	1834	88	63	11186,0	1,03	12493,0	1,02	57	43	3	3
la	157	26	19	638,0	1,00	655,1	1,00	0	0	0	0
pair	1733	173	137	14700,7	1,03	15748,2	1,01	534	561	10	7
rot	758	135	107	5708,6	1,02	6013,1	1,02	208	217	9	8
term1	565	34	10	2659,1	1,03	3041,0	1,04	5	1	3	1
too_large	1398	38	3	6174,8	1,01	7092,0	1,02	54	48	4	5
vda	1984	17	39	15145,7	1,00	16721,4	1,00	238	169	6	5
x1	428	51	35	2319,7	0,99	2526,1	0,99	27	7	4	2
x3	1328	135	99	7724,5	1,01	8085,9	1,02	282	262	17	9
x4	858	94	71	3828,0	1,03	4196,2	1,03	43	26	5	3
average	1136,97	90,19	45,66	8012,6	1,02	7962,5	1,02	183,6	154,7	7,2	5,0
%	—	—	—	100,0 %	102,2 %	100,0 %	105,5 %	100,0 %	84,3 %	100,0 %	69,4 %

которые могут быть не выполнены; $C3$ в этом случае — также "функция минимизации".

В начале процесса размещения веса слагаемых, относящихся к "функциям минимизации", намного больше, чем веса, относящиеся к "жестким ограничениям", в то же время веса внутри каждой группы нормализованы таким образом, чтобы каждое слагаемое вносило одинаковый вклад в целевую функцию. Затем на каждой итерации решения оптимизационной задачи алгоритм размещения проверяет, оптимизируется ли каждый член в группе "жесткое ограничение". В случае необходимости алгоритм увеличивает вклад соответствующей функции с "жестким ограничением" в 2 раза.

Результаты. Для экспериментов был выбран набор стандартных тестовых примеров LGSynth [10]. Для получения схем на вентильном уровне тестовые примеры прошли этап логического синтеза и технологического отображения с помощью пакета логического проектирования MVSIS [11]. Позиции внешних выводов и геометрия области размещения назначены случайным образом.

В таблице приводится сводный результат работы предложенного алгоритма PBPlace по сравнению с результатами алгоритма APlace [2], построенного на базе блок-схемы маршрута (см. рис. 3) с целевыми функциями (14) и (15), но без участия функции $C3$.

Сравнивались следующие характеристики размещения:

HRPM — сумма полупериметров охватывающих прямоугольников для всех цепей; RWL — суммарная длина трасс (проводников); MBO — максимальное значение превышения плотности трасс над пропускной способностью клетки среди всех клеток; TBO — общее значение превышения плотности трасс над пропускной способностью во всех клетках (трассируемость).

Значения HRPM и RWL для PBPlace даны как отношения к соответствующим значениям для APlace.

Экспериментальные данные показывают, что, оптимизируя трассируемость явно, PBPlace улучшает ее (TBO, см. таблицу) в среднем на 15 % по сравнению с алгоритмом APlace.

Заключение

В данной работе был предложен метод аналитического размещения, который, по сравнению с алгоритмом APlace, включает следующие новые особенности:

- явное моделирование трассируемости в размещении;
- использование штейнеровских деревьев для моделирования трассировки цепей;
- взаимодействие с алгоритмом глобальной трассировки.

Экспериментальные результаты показали, что трассируемость (TBO) была улучшена с помощью этого метода в среднем на 15 % по сравнению с алгоритмом APlace.

Список литературы

1. **ISPD** placement contest testcases and results. http://www.sigda.org/ispd2006/contest.html, http://www.sigda.org/ispd2005/contest.html.

2. **Kahng, Wang Q.** Implementation and extensibility of an analytical placement // Proc. ISPD'04. April 2004.

3. **Chan T., Cong J., Sze K.** Multilevel generalized force-directed method for circuit placement // Proc. ISPD'05. April 2005.

4. **Naylor W.** et al. Non-Linear Optimization System and Method for Wire Length and Delay Optimization for an Automatic Electric Circuit Placer. US Patent 6301693, Oct. 2001.

5. **Hou W., Yu H., Hong X., Cai Y., Wu W., Gu J., Kao W. H.** A new congestion-driven placement algorithm based on cell inflation // Proc. of Asia South Pacific Design Automation Conference. 2001. P. 605—608.

6. **Yang X., Kastner R., Sarrafzadeh M.** Congestion reduction during placement based on integer programming // Proc. Int. Conf. on Computer-Aided Design. 2001. P. 573—576.

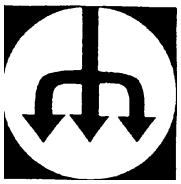
7. **Ruehli P., Wolff Sr., Goetzel G.** Analytical power timing optimization techniques for digital systems // Proc. of the 14-th ACM/IEEE Design Automation Conference. 1977. P. 142—146.

8. **Young Sham E.** Congestion prediction in early stages // roc. ALIP'05. April 2005.

9. **Eisenmann H., Johannes F. M.** Generic global placement and floorplanning // Proc. DAC'98. June 1998.

10. **LGSynth** suite. Http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cbl.ncsu.edu/CBL_Docs/lgs93.html.

11. **MVSIS** reference. Http://embedded.eecs.berkeley.edu/Respep/Research/mvsis..



ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ И СЕТИ

УДК 519.6

А. П. Карпенко, д-р физ.-мат. наук, проф.,
В. Г. Федорук, канд. техн. наук, **Е. В. Федорук**
МГТУ им. Н. Э. Баумана

Балансировка загрузки многопроцессорной системы при распараллеливании одного класса вычислительных задач

Исследуются статические и динамические методы балансировки загрузки многопроцессорных вычислительных систем при распараллеливании одного класса вычислительных задач.

Введение

Схема решения достаточно широкого класса вычислительных задач (далее — F-задач) неформально может быть представлена в следующем виде:

- область решения задачи покрывается некоторой сеткой;
- в узлах сетки проводятся вычисления значений функций, ассоциированных с этими узлами;
- на основе вычисленных значений формируется решение задачи.

В виде такой схемы может быть представлено, например, решение задачи глобальной условной оптимизации комбинацией метода случайного поиска с каким-либо методом локальной оптимизации.

В случае, когда суммарная вычислительная сложность функций, ассоциированных с узлами сетки, велика, целесообразно решение F-задач на многопроцессорных вычислительных системах (МВС).

На первый взгляд, F-задачи вкладываются в более широкий класс задач, в котором программы, обрабатывающие различные узлы сетки, информационно связаны. К задачам этого класса относятся, например, задачи математической физики, решаемые различными конечно-разностными и конечно-элементными методами. Вследствие исключительной практической важности таких задач имеется большое число работ, посвященных их распараллеливанию (см., например, [1]). Однако в задачах указанного класса полностью неизвестными являются вычислительные сложности функций, ассоциированных с узлами сетки, только на первой итерации. На последующих итерациях в качестве оценок вычислительных сложностей узлов можно использовать их значения на предыдущей итерации. Такая возможность отсутствует в F-задачах.

Вместе с тем при построении операционных систем МВС одной из важнейших является задача планирования вычислений при выполнении множества информационно и логически не связанных заданий. Этой проблеме также посвящено большое число публикаций (см., например, [2]). Однако в F-задачах формирование решения всей задачи и, быть может, построение сетки информационно и логически связаны с функциями, ассоциированными с узлами сетки. Поэтому прямое использование методов планирования вычислений, используемых в операционных системах МВС, в F-задачах не удается.

Работа посвящена исследованию эффективности следующих методов балансировки загрузки МВС при решении F-задачи [3, 4]:

- статическая балансировка;
- динамическая централизованная балансировка методом равномерной декомпозиции узлов — равномерная балансировка;
- динамическая централизованная балансировка методом экспоненциальной декомпозиции узлов — экспоненциальная балансировка;
- динамическая децентрализованная балансировка диффузным методом с перераспределением загрузки по инициативе получателя — диффузная балансировка.

В последнем случае рассматривается перераспределение загрузки только по инициативе получателя, поскольку для F-задачи такой алгоритм более эффективен, чем перераспределение загрузки по инициативе отправителя. При этом, как известно [5], возможно использование детерминированных и стохастических алгоритмов поиска отправителя. С точки зрения используемой в работе методики оценки эффективности балансировки эти алгоритмы эквивалентны. Поэтому алгоритм поиска отправителя при рассмотрении диффузной балансировки не фиксируется.

1. Постановка задачи

Пусть X — n-мерный вектор параметров задачи. Положим, что $X \in R^n$, где R^n — n-мерное арифметическое пространство. Параллелепипедом допустимых значений вектора параметров назовем не пустой параллелепипед $\Pi = \{X \,|\, x_i^- \leqslant x_i \leqslant x_i^+, i \in [1, n]\}$, где x_i^-, $x_i^+ +$ — заданные константы. На вектор X дополнительно может быть наложено некоторое число функциональных ограничений, формирующих множество

$$D = \{X | g_i(X) \geqslant 0,\ i = 1, 2, ...\},$$

где $g_i(X)$ — непрерывные ограничивающие функции.

На множестве $D_X = \Pi \cap D$ определена вектор-функция $F(X) \in R^m$. Ставится задача поиска значения некоторого функционала $\Phi = \Phi(F(X))$.

Приближенное решение поставленной F-задачи, полагается, может быть найдено по следующей схеме.

Шаг 1. Покрываем параллелепипед Π некоторой сеткой Ω с узлами $X_1, X_2, ..., X_Z$.

Шаг 2. В тех узлах сетки Ω, которые принадлежат множеству D_X, вычисляем значения вектор-функции $F(X)$.

Шаг 3. На основе вычисленных значений $F(X)$ находим приближенное значение функционала $\Phi(F(X))$.

Суммарное число арифметических операций, необходимых для однократного определения принадлежности вектора X множеству D_X (т. е. суммарную вычислительную сложность ограничений $x_i^- \leqslant x_i \leqslant x_i^+$ и ограничивающих функций $g_i(X)$), обозначим $C_g \geqslant 0$. Вообще говоря, величина C_g зависит от вектора X. Мы, однако, пренебрежем этой зависимостью и будем полагать, что $C_g = \text{const}$. Заметим, что до начала вычислений величина C_g, как правило, неизвестна. Однако в процессе первого же определения принадлежности узла сетки Ω множеству D_X эту величину можно легко найти (с учетом предположения о независимости этой величины от вектора X). Поэтому будем полагать величину C_g известной.

Неизвестную вычислительную сложность вектор-функции $F(X)$ обозначим $C_f(X)$. Подчеркнем зависимость величины C_f от вектора X. Величина $C_f(X)$ удовлетворяет очевидному ограничению $C_f(X) \geqslant 0$. Вычислительную сложность $C_f(X_i)$ назовем вычислительной сложностью узла X_i.

Вычислительную сложность конечномерной аппроксимации функционала $\Phi(F(X))$ положим равной ZC_Φ, а вычислительную сложность генерации сетки Ω — равной ZC_Ω, где при данных n, m величины C_Φ, C_Ω — известные константы.

В качестве вычислительной системы рассмотрим однородную МВС с распределенной памятью, состоящую из процессоров P_1, P_2, ..., P_N и *host*-процессора, имеющих следующие параметры: t — время выполнения одной арифметической операции с плавающей запятой; $d = d(N)$ — диаметр коммуникационной сети; l — длина вещественного числа в байтах; t_s [с] — латентность коммуникационной сети; t_c [с] — время передачи байта данных между двумя соседними процессорами системы без учета времени t_s [с]. Для простоты записи положим, что число узлов сетки Ω кратно числу процессоров в системе N, так что $\frac{Z}{N} = z > 1$.

В качестве меры эффективности параллельных вычислений используем ускорение

$$S_i(N) = \frac{T(1)}{T_i(N)},$$

где $T(1)$ — время последовательного решения задачи на одном процессоре системы; $T_i(N)$ — время параллельного решения той же задачи на N процессорах; $i \in [1, 4]$ — номер метода балансировки.

Кроме того, для оценки эффективности балансировки используем следующие величины: оценку ускорения сверху $S_i^+(N)$; оценку ускорения снизу $S_i^-(N)$; относительную оценку ускорения сверху $\eta_i^+(N) = \dfrac{S_i^+(N) - S_1^+(N)}{S_1^+(N)}$ и аналогичную относительную оценку ускорения снизу $\eta_i^-(N)$.

Оценка ускорения сверху $S_i^+(N)$ достигается в ситуации, когда вычислительные сложности $C_f(X)$ всех узлов сетки Ω одинаковы и равны C_f. В этом случае легко обеспечить равномерную загрузку всех процессоров, что и является основанием для использования достигнутого ускорения в качестве оценки ускорения сверху. Данная ситуация моделирует случай, когда вычислительная сложность $C_f(X)$ является случайной величиной с малой дисперсией.

Оценка ускорения снизу $S_i^-(N)$ достигается в ситуации, когда вычислительная сложность $C_f(X)$ одного из узлов X^* сетки Ω многократно превышает вычислительные сложности $C_f(X)$ любых других узлов этой сетки, т. е. $C_f = C_f(X^*) \gg \gg C_f(X_j) = \mu C_f$ для любых $X_j \neq X^*$, $0 \leqslant \mu \ll 1$. В этих условиях неизбежна сильная несбалансированность загрузки процессоров. В пределе при $C_f \to \infty$ все процессоры, кроме процессора, которому назначен для обработки узел X^*, по сути, простаивают в течение всего времени решения задачи и $S_i^-(N) \to 1$. Это обстоятельство и является основанием для использования достигнутого ускорения в качестве оценки ускорения снизу. Ситуация, близкая к данной, может иметь место в случае, когда дисперсия вычислительной сложности $C_f(X)$ велика, а также когда объем множества D_X значительно меньше объема параллелепипеда П.

Основные результаты в работе получены для случая, когда множество D_X совпадает со всем параллелепипедом П, т. е. когда $D_X = \Pi$. Это допущение не является слишком ограничительным, поскольку непринадлежность некоторого узла X_j множеству D_X можно интерпретировать как равенство нулю вычислительной сложности этого узла $C_f(X_j)$, что включает в себя ограничение $0 \leqslant C_f(X)$.

2. Статическая балансировка

Для F-задачи статическую балансировку загрузки естественно организовать с использованием геометрической схемы распараллеливания [5], при которой узлы сетки Ω разбиваются на $\kappa = N$ множеств Ω_i по z узлов в каждой. Число узлов множества Ω_i, принадлежащих множеству D_X, обозначим ξ_i, а сумму этих величин обозначим ξ. В сделанных предположениях схема параллельных вычислений при решении F-задачи с использованием статической балансировки загрузки имеет следующий вид.

Шаг 1. Host-процессор строит сетку Ω и разбивает ее узлы на множества Ω_i, $i \in [1, N]$.

Шаг 2. Процессор P_i выполняет следующие действия:

- принимает от *host*-процессора координаты узлов множества Ω_i;
- последовательно для всех узлов этого множества определяет их принадлежность множеству D_X;
- вычисляет в каждом из ξ_i узлов значение вектор-функции $F(X)$;
- передает *host*-процессору вычисленные значения и заканчивает вычисления.

Шаг 3. Host-процессор на основе полученных значений $F(X)$ вычисляет приближенное значение функционала $\Phi(F(X))$.

В соответствии с изложенной схемой время решения F-задачи на процессоре P_i можно оценить величиной

$$\tau_i = 2t_s + znldt_c + \xi_i mldt_c + zC_g t + t\sum_j C_f(X_{i,j}),$$

где $X_{i,j} \in \Omega_i$, $j \in [1, \xi_i]$; $zC_g t + t\sum_j C_f(X_{i,j})$ — вычислительная загрузка процессора P_i; $2t_s + znldt_c + \xi_i mldt_c$ — его коммуникационная загрузка. При этом время параллельного решения всей F-задачи оценивается величиной

$$T(N) = \max_{i \in [1, N]} \tau_i + ZC_\Omega t + \xi C_\Phi t.$$

Аналогично оценка времени решения задачи на одном процессоре

$$T(1) = ZC_g t + t \sum_{j=1}^{\xi} C_f(X_j) + ZC_\Omega t + \xi C_\Phi t.$$

Если множество D_X совпадает со всем параллелепипедом П, то, очевидно, $\xi_i = z$, $\xi = Z$ и

$$T(N) = 2t_s + zT_c + zC_g t + t \max_{i \in [1, N]} \sum_{j=1}^{z} C_f(X_{i,j}) + Z(C_\Omega + C_\Phi)t, \quad (1)$$

где $T_c = T_c(n + m) = (n + m)ldt_c$;

$$T(1) = Z(C_\Omega + C_g + C_\Phi)t + t\sum_{j=1}^{z} C_f(X_j). \qquad (2)$$

Из (1), (2) следует, что для времени параллельного решения F-задачи имеет место оценка сверху

$$T_1^+(N) = 2t_s + z(T_c + C_g ft) + ZC_{\Omega\Phi},$$

а для времени последовательного решения — оценка

$$T_1^+(1) = ZCt,$$

где $C_{gf} = C_g + C_f$, $C_{\Omega\Phi} = C_\Omega + C_\Phi$; $C = C_\Omega + C_g + C_f + C_\Phi$.

Аналогично из (1), (2) следует, что время параллельного решения F-задачи может быть оценено снизу величиной

$$T_1^-(N) = 2t_s + zT_c + z(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t,$$

а время решения задачи на одном процессоре — величиной

$$T_1^-(1) = Z(C_g + \mu C_f + C_{\Omega\Phi})t + (1 - \mu)C_f t.$$

Утверждение 1. Оценка ускорения сверху при статической балансировке

$$S_1^+(N) = \frac{ZCt}{2t_s + z(T_c + C_{gf}t) + ZC_{\Omega\Phi}},$$

а оценка ускорения снизу

$$S_1^-(N) = \frac{Z(C_g + \mu C_f + C_{\Omega\Phi})t + (1 - \mu)C_f t}{2t_s + zT_c + z(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t}.$$

Как и следовало ожидать, при $C_g f \to \infty$ оценка $S_1^+(N)$ стремится к максимально возможной, равной числу процессоров в системе N, а оценка $S_1^-(N)$ — к единице.

3. Динамическая централизованная балансировка методом равномерной декомпозиции узлов

Разобьем узлы сетки Ω на $K = \kappa N$ множеств ϖ_k $\kappa \in [1, z]$, и для простоты записи примем, что величина Z кратна κN, так что каждое из множеств ϖ_k содержит по $s = \frac{Z}{\kappa N} = \frac{z}{\kappa}$ узлов (рис. 1). Тогда схема параллельных вычислений при решении F-задачи с использованием равномерной балансировки загрузки имеет следующий вид.

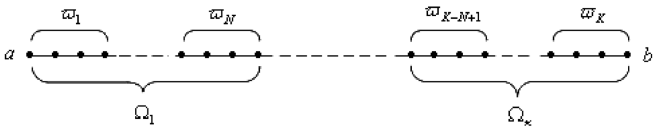


Рис. 1. Схема равномерного разбиения узлов для случая $n = 1$ (параллелепипед П представляет собой отрезок $[a, b]$): $s = 4$, точками обозначены узлы сетки Ω

Шаг 1. Host-процессор строит сетку Ω и разбивает ее узлы на множества ϖ_k.

Шаг 2. Процессор P_i принимает от *host*-процессора координаты первого из нераспределенных множеств узлов ϖ_k.

Шаг 3. Процессор P_i для точек назначенного ему текущего множества узлов ϖ_k выполняет следующие действия:

- определяет принадлежность каждого из этих узлов множеству D_X;
- если узел принадлежит множеству D_X, то вычисляет в этом узле значение вектор-функции $F(X)$;
- после завершения обработки текущего множества узлов ϖ_k посылает *host*-процессору вычисленные значения $F(X)$.

Шаг 4. Если исчерпаны не все множества ϖ_k, то *host*-процессор посылает, а процессор P_i принимает координаты следующего множества узлов ϖ_k, которое обрабатывается процессором P_i аналогично шагу 3 и т. д.

Шаг 5. Если исчерпаны все множества ϖ_k, то *host*-процессор:

- посылает освободившемуся процессору P_i сообщение об окончании решения задачи;
- после получения всех вычисленных значений вектор-функции $F(X)$ от всех процессоров вычисляет приближенное значение функционала $\Phi(F(X))$.

Заметим, что при $\kappa = 1$, когда число множеств ϖ_k равно числу процессоров в системе, динамическая равномерная балансировка вырождается в статическую балансировку.

В случае, когда вычислительные сложности всех узлов сетки Ω одинаковы, величина κ представляет собой число множеств ϖ_k, назначенных каждому из процессоров P_i. Поэтому аналогично п. 2 в качестве оценки сверху времени параллельного решения F-задачи имеем

$$T_2^+(\kappa, N) = 2\kappa t_s + z(T_c + C_g ft) + ZC_{\Omega\Phi}t. \qquad (3)$$

Объединим множества узлов ϖ_k в κ совокупностей $\Omega_1, \Omega_2, \ldots, \Omega_\kappa$ по N множеств ϖ_k в каждой (рис. 2). Если узел X^* принадлежит совокупности Ω_i (вероятность этого равна κ^{-1}), то оценка снизу времени параллельного решения F-задачи есть

$$T_2^-(i, \kappa, N) = 2it_s + isT_c + is(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t. \qquad (4)$$

Из (3), (4) имеем следующее утверждение.

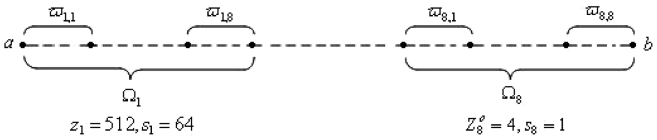


Рис. 2. Схема экспоненциального разбиения узлов для случая $n = 1$: $Z = Z_1 = 1024$, $\rho = 2$, $N = 8$; точками обозначены узлы сетки Ω

Утверждение 2. Оценка ускорения сверху при динамической равномерной балансировке

$$S_2^+(\kappa, N) = \frac{ZCt}{2\kappa t_s + z(T_c + C_{gf}t) + ZC_{\Omega\Phi}t};$$

оценка ускорения снизу

$$S_2^-(i, \kappa, N) = \frac{Z(C_{\Omega\Phi} + C_g + \mu C_f)t + (1-\mu)C_f t}{2it_s + isT_c + is(C_g + \mu C_f)t + (1-\mu)C_f t + ZC_{\Omega\Phi}t};$$

средняя по $i \in [1, \kappa]$ оценка ускорения снизу

$$M[S_2^-(i, \kappa, N)] = \frac{Z(C_{\Omega\Phi} + C_g + \mu C_f)t + (1-\mu)C_f t}{b\kappa}\left(\Psi\left(\frac{c}{b}\right) - \Psi\left(\frac{c}{b} + \kappa\right)\right),$$

где $\Psi(*)$ — символ полигамма функции и $b = 2t_s + sT_c + s(C_g + \mu C_f)t$, $c = (1-\mu)C_f t + ZC_{\Omega\Phi}t$ •

Относительная оценка ускорения сверху при динамической равномерной балансировке удовлетворяет неравенству $\min\eta_2^+(N) \leqslant \eta_2^+(\kappa, N) \leqslant 0$, где нижняя граница $\min\eta_2^+(N)$ соответствует $S_2^+(z, N)$. Потери ускорения обусловлены тем фактом, что при динамической равномерной балансировке число актов обмена каждого из процессоров P_i с *host*-процессором больше, чем при статической балансировке, и поэтому выше коммуникационные расходы, обусловленные латентностью коммуникационной сети.

Легко показать, что при $C_f \to \infty$ значение $\min\eta_2^+(N)$ стремится к нулю.

Относительная оценка ускорения снизу при динамической равномерной балансировке удовлетворяет неравенству $\min\eta_2^-(N) \leqslant \eta_2^-(i, \kappa, N) \leqslant \max\eta_2^-(N)$, где нижняя граница этой оценки $\min\eta_2^-(N)$ соответствует величине $S_2^-(z, z, N)$, а ее верхняя граница $\max\eta_2^-(N)$ — величине $S_2^-(1, z, N)$. Граница $\min\eta_2^-(N)$ является отрицательной величиной и имеет, таким образом, смысл потери ускорения по сравнению со статической балансировкой. Аналогично граница $\max\eta_2^-(N)$ имеет смысл приобретения ускорения.

Легко показать, что при $C_f \to \infty$ значение $\min\eta_2^-(N)$ стремится к нулю, а значение $\max\eta_2^-(N)$ стремится к $\mu(z-1)$.

4. Динамическая централизованная балансировка загрузки методом экспоненциальной декомпозиции узлов

На k-й итерации экспоненциальной декомпозиции узлов среди оставшихся Z_k узлов сетки Ω выделяется множество узлов Ω_k, содержащее $z_k = \frac{Z_k}{\rho}$ узлов, где целая положительная величина ρ — коэффициент декомпозиции. Для простоты записи положим, что величины Z_k кратны ρ, а величины z_k — числу процессоров в системе N, так что подмножества $\varpi_{k,j}$, $j \in [1, N]$, содержат по $s_k = \frac{z}{\rho^k}$ узлов (рис. 2). Тогда схема параллельных вычислений при решении F-задачи с использованием экспоненциальной балансировки имеет следующий вид.

Шаг 1. Host-процессор строит сетку Ω и присваивает счетчику числа итераций k значение 1 (принято, что $Z_1 = Z$).

Шаг 2. Если исчерпаны не все узлы сетки Ω, то *host*-процессор выделяет среди Z_k узлов этой сетки множество узлов Ω_k.

Шаг 3. Если множество узлов Ω_k не исчерпано, то *host*-процессор передает, а процессор P_i принимает от него координаты узлов подмножества $\varpi_{k,j}$.

Шаг 4. Процессор P_i для узлов текущего подмножества $\varpi_{k,j}$ выполняет следующие действия:

- определяет принадлежность каждого из этих узлов множеству D_X;
- если узел принадлежит множеству D_X, то вычисляет в этом узле значение вектор-функции $F(X)$;
- после завершения обработки текущего подмножества узлов $\varpi_{k,j}$ посылает *host*-процессору найденные им значения $F(X)$;

Шаг 5. Если множество узлов Ω_k исчерпано, то *host*-процессор выполняет присваивания $k = k + 1$, $Z_k = Z_k(1 - \rho^{-1})$ и переходит к шагу 2.

Шаг 6. Если исчерпаны все узлы сетки Ω, то *host*-процессор:

- посылает освободившемуся процессору P_i сообщение об окончании решения задачи;
- после получения всех вычисленных значений вектор-функций $F(X)$ от всех процессоров вычисляет приближенное значение функционала $\Phi(F(X))$.

Ограничимся далее случаем $\rho = 2$. При этом общее число множеств Ω_k равно $\kappa + 1$, где

$\kappa = \log_2 z$, $z_k = \frac{Z}{2^k}$, $k \in [1, \kappa]$, $z_{\kappa+1} = \frac{Z}{2^\kappa} = N$ и $s_{\kappa+1} = 1$ (рис. 2). Подчеркнем, что в данном случае, в отличие от п. 3, величина κ не является независимой, а определяется величинами Z, N.

Аналогично п. 2 оценка сверху времени параллельного решения F-задачи определяется выражением

$$T_3^+(N) = 2(\kappa + 1)t_s + zT_c + zC_{gf}t + ZC_{\Omega\Phi}t. \quad (5)$$

Если узел X^* принадлежит множеству Ω_k, $k \in [1, \kappa + 1]$, то время параллельного решения F-задачи оценивается снизу величиной

$$T_3^-(k, N) = 2kt_s + z\sigma_k T_c + z\sigma_k(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t. \quad (6)$$

Здесь величина $z\sigma_k$ — общее число узлов, назначенных для обработки процессору, которому выпал узел X^*. Если $k \in [1, \kappa]$, то $\sigma_k = \left(1 - \frac{1}{2^k}\right)$; если $k = \kappa + 1$, то $\sigma_{k+1} = 1$.

Вероятность того, что узел X^* принадлежит множеству узлов Ω_k, равна $p_k = \frac{1}{2^k}$ при $k \in [1, \kappa]$ и $p_{\kappa+1} = \frac{1}{2^\kappa} = z^{-1}$ при $k = \kappa + 1$.

Из (5), (6) и последнего замечания имеем следующее утверждение.

Утверждение 3. Оценка ускорения сверху при динамической экспоненциальной балансировке

$$S_3^+(N) = \frac{ZCt}{2(\kappa + 1)t_s + z(T_c + C_{gf}t) + ZC_{\Omega\Phi}t};$$

оценка ускорения снизу

$$S_3^-(k, N) = \frac{Z(C_{\Omega\Phi} + C_g + \mu C_f)t + (1 - \mu)C_f t}{2kt_s + z\sigma_k T_c + z\sigma_k(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t},$$

где $\sigma_k = \left(1 - \frac{1}{2^k}\right)$, если $k \in [1, \kappa]$, и $\sigma_{\kappa+1} = 1$;

средняя по $k \in [1, \kappa + 1]$ оценка ускорения снизу

$$M[S_3^-(k, N)] = \sum_{k=1}^{\kappa} \frac{1}{2^k} \frac{T^-(1)}{T_3^-(k, N)} + \frac{1}{z} \frac{T^-(1)}{T_3^-(\kappa + 1, N)}.$$

Относительная оценка ускорения сверху $\eta_3^+(N)$ удовлетворяет неравенству $\eta_3^+(N) \leqslant 0$, т. е. при динамической экспоненциальной балансировке по сравнению со статической балансировкой могут иметь место потери ускорения. Так же, как при динамической равномерной балансировке, эти потери обусловлены латентностью коммуникационной сети.

Легко показать, что при $C_f \to \infty$ величина $\eta_3^+(N)$ стремится к нулю.

Относительная оценка ускорения снизу при динамической экспоненциальной балансировке удовлетворяет неравенству $\min\eta_3^-(N) \leqslant \eta_3^-(k, N) \leqslant \max\eta_3^-(N)$. Здесь нижняя граница $\min\eta_3^-(N)$ соответствует величине $S_3^-(\kappa + 1, N)$, а верхняя граница $\max\eta_3^-(N)$ — величине е. $S_3^-(1, N)$.

Легко показать, что при $C_f \to \infty$ оценка $\eta_3^-(k, N)$ стремится к величине $\frac{\mu z(1 - \sigma_k)}{\mu(z\sigma_k - 1) + 1}$. Так как $\sigma_k \leqslant 1$ при всех допустимых k, то оценка $\eta_3^-(k, N)$ неотрицательна. То есть оценка ускорения снизу при динамической экспоненциальной балансировке не хуже такой же оценки при статической балансировке. Отсюда же следует, что при $C_f \to \infty$ нижняя граница $\min\eta_3^-(N)$ стремится к нулю, а верхняя граница $\max\eta_3^-(N)$ стремится к величине $\frac{\mu z}{\mu(z - 2) + 2}$.

5. Динамическая децентрализованная балансировка загрузки диффузным методом

Схема параллельных вычислений при решении F-задачи с использованием метода диффузной балансировки загрузки имеет следующий вид.

Шаг 1. Host-процессор строит сетку Ω и разбивает ее узлы на N множеств Ω_i, $i \in [1, N]$, по z узлов в каждом.

Шаг 2. Процессор P_i:

- принимает от *host*-процессора координаты узлов множества Ω_i;
- последовательно для всех узлов этого множества определяет их принадлежность множеству D_X;
- если узел принадлежит множеству D_X, то вычисляет в этом узле значение вектор-функции $F(X)$;
- передает *host*-процессору вычисленные значения $F(X)$.

Шаг 3. Если процессор P_i закончил вычисления и передал *host*-процессору вычисленные значения вектор-функции $F(X)$, то этот процессор:

- посылает запрос некоторому процессору P_j;
- если процессор P_j имеет необработанные узлы, то процессор P_i принимает от процессора P_j координаты $1/\rho$ части из них;
- посылает сообщение *host*-процессору об этом факте (для того, чтобы *host*-процессор имел возможность определить момент завершения решения задачи).

Шаг 4. Процессор P_j:

- в каждом из принятых узлов вычисляет значение вектор-функции $F(X)$;
- передает *host*-процессору вычисленные значения $F(X)$.

Шаг 5. Host-процессор после получения от всех процессоров сообщений о завершении вычислений посылает всем им сообщение о завершении решения задачи и на основе полученных значений $F(X)$ вычисляет приближенное значение функционала $\Phi(F(X))$.

Здесь ρ — коэффициент декомпозиции остатка узлов. Ограничимся случаем, когда $\rho = 2$.

Легко видеть, что если вычислительные сложности $C_f(X)$ одинаковы и равны C_f, то при диффузной балансировке перераспределения узлов не проводится и диффузная балансировка превращается в статическую балансировку. Поэтому оценка ускорения сверху совпадает с такой же оценкой при статической балансировке.

Положим, что узлом X^* является узел $X_{i,\,k} \in \Omega_i$, $k \in [1, z]$. Тогда аналогично п. 2 время параллельного решения F-задачи может быть оценено снизу величиной

$$T_4^-(k, N) = 2t_s + zT_c(n) + kT_c(m) + k(C_g + \mu C_f)t + \\ + (1 - \mu)C_f t + T_a + ZC_{\Omega\Phi}t, \qquad (7)$$

где $T_c(n) = nldt_c$; $T_c(m) = mldt_c$; $T_a = T_a(k, N)$ — расходы процессора, которому выпал для обработки узел X^*, обусловленные перераспределением между другими процессорами системы необработанных узлов $X_{i,\,j}$, $j \in [k + 1, z]$ (такие узлы имеются, очевидно, в случае, когда $k < z$). Пренебрежем вычислительными расходами на декомпозицию необработанных узлов. Тогда величина T_a включает в себя только коммуникационные расходы и ее оценка при $k < z$ равна $T_a = qt_s + (z - k)T_c(n)$, а при $k = z$ — равна $T_a(k, N) = 0$. Здесь $q \approx \log_2(z - k) + 1$ и точное равенство имеет место, если величина $(z - k)$ есть целая степень двух.

Вероятность того, что узел X^* является k-м узлом одного из множеств узлов Ω_i, равна $p_k = \frac{1}{z}$.

Утверждение 4. Оценка ускорения сверху при диффузной балансировке равна оценке ускорения сверху при статической балансировке, т. е.

$$S_4^+(N) = S_1^+(N);$$

оценка ускорения снизу

$$S_4^-(k, N) \approx \\ \approx \frac{Z(C_{\Omega\Phi} + C_g + \mu C_f)t + }{((2 + q)t_s + 2zT_c(n) - k(T_c(n) - T_c(m)) + } \rightarrow \\ \rightarrow \frac{+ (1 - \mu)C_f t}{+ k(C_g + \mu C_f)t + (1 - \mu)C_f t + ZC_{\Omega\Phi}t},$$

где $q \approx \log_2(z - k) + 1$;

средняя по $k \in [1, z]$ оценка ускорения снизу

$$M[S_4^-(k, N)] = \frac{1}{z}\sum_{k=1}^{z}\frac{T^-(1)}{T_4^-(k, N)} \bullet$$

Легко показать, что поведение оценки $S_4^-(k, N)$, как функции величины k, зависит от вычислительной сложности C_f. При выполнении неравенства

$$C_f \leqslant \frac{t_s(z-1)^{-1}\log_2\mathbf{e} + (T_c(n) - T_c(m)) - C_g t}{\mu t} \qquad (8)$$

оценка $S_4^-(k, N)$ является возрастающей функцией величины k, а при выполнении неравенства

$$C_f \geqslant \frac{t_s\log_2\mathbf{e} + (T_c(n) - T_c(m)) - C_g t}{\mu t} \qquad (9)$$

— убывающей функцией этой же величины.

Относительная оценка ускорения снизу при диффузной балансировке $\eta_4^-(k, N)$ удовлетворяет неравенству $\min\eta_4^-(N) \leqslant \eta_4^-(k, N) \leqslant \max\eta_4^-(N)$. При выполнении условия (8) нижняя граница $\min\eta_4^-(N)$ соответствует величине $S_4^-(1, N)$, а верхняя граница $\max\eta_4^-(N)$ — величине $S_4^-(z, N)$. Аналогично при выполнении условия (9) нижняя граница соответствует величине $S_4^-(z, N)$, а верхняя граница — величине $S_4^-(1, N)$.

При $k = z$ диффузная балансировка вырождается в статическую балансировку и при выполнении условия (8) $\max\eta_4^-(N) = 0$, а при выполнении условия (9) — $\min\eta_4^-(N) = 0$.

Поскольку при выполнении условия (8) верхняя граница относительной оценки ускорения снизу $\max\eta_4^-(N)$ равна нулю, величина $\min\eta_4^-(N)$ отрицательна. Таким образом, при выполнении условия (8) имеют место потери ускорения по сравнению со статической балансировкой. Потери ускорения обусловлены расходами на перераспределение между другими процессорами системы необработанных узлов, т. е. компонентой $T_a(k, N)$ в выражении (7).

Напротив, при выполнении условия (9) нижняя граница относительной оценки ускорения снизу $\min\eta_4^-(N)$ равна нулю. Поэтому величина $\max\eta_4^-(N)$ положительна, т. е. при выполнении условия (9) имеет место приобретение ускорения по сравнению со статической балансировкой.

Легко показать, что при $C_f \to \infty$ величина $\eta_4^-(k, N)$ стремится к величине $\frac{\mu(z-k)}{\mu(k-1)+1} \geqslant 0$. Отсюда следует, что в пределе аналогично динамической

равномерной балансировке верхняя граница $\max\eta_4^-(N)$ стремится к величине $\mu(z-1)$.

Заключение

Результаты работы позволяют выполнить сравнительный анализ эффективности статической, динамической равномерной, динамической экспоненциальной и диффузной балансировок загрузки при решении F-задачи в зависимости от ее параметров и параметров используемой МВС. Это, в конечном счете, позволяет сделать обоснованный выбор метода балансировки загрузки или совокупности таких методов для решения конкретной F-задачи.

Выполненное исследование не закрывает всех вопросов, связанных с эффективностью указанных методов балансировки. Например, не удается аналитическое определение оптимального числа множеств ϖ_k, $k \in [1, K]$, для динамической равномерной балансировки при промежуточных значениях дисперсии вычислительной сложности $C_f(X)$. Поэтому предполагаются дополнительные исследования с помощью имитационного моделирования и натурных вычислительных экспериментов.

Список литературы

1. **Карпенко А. П., Шурайц Ю. М.** Параллелизм методов интегрирования дифференциальных уравнений в частных производных // Автоматика и телемеханика. 1992. № 10. С. 3—20.
2. **Танненбаум Э.** Современные операционные системы. 2-е изд. 2007. СПб.: Питер. 1038 с.
3. **Волков К. Н.** Применение средств параллельного программирования для решения задач механики жидкостей и газа на многопроцессорных вычислительных системах // Вычислительные методы и программирование. 2006. Т. 7. С. 69—84.
4. **Gubenko G.** Dynamic load Balancing for Distributed Memory Multiprocessors // Journal of parallel and distributed computing. 1989. N 7. P. 279—301.
5. **Карпенко А. П., Пупков К. А.** Моделирование динамических систем на транспьютерных сетях. М.: Биоинформ, 1995. 173 с.
6. **Бейтман Г., Эрдейи Л.** Высшие трансцендентные функции. Гипергеометрические функции. Функции Лежандра. М.: Наука, 1965. 296 с.

УДК 004.032.24

А. В. Климов,
ИТМиВТ им. С. А. Лебедева РАН

Умножение плотных матриц на неоднородных высокопараллельных вычислительных системах (анализ коммуникационной нагрузки)

При распараллеливании задач на системе из большого числа процессоров важное значение имеет коммуникационная нагрузка на сеть и способность сети справляться с этой нагрузкой. На примере задачи умножения плотных матриц изучаются требования, предъявляемые к коммуникационному обеспечению вычислительной системы. В качестве эталона системы рассматривается проект суперкомпьютера Merrimac [1]. Его развитая многоуровневая коммуникационная сеть обеспечивает хорошие пропускные способности на разных уровнях локальности. Обосновывается тезис о важности снижения коммуникационных затрат за счет выбора оптимального распределения вычислений по процессорным элементам. Демонстрируется возможность выбора оптимального распределения, не зависящего от параметров сети.

Введение

При анализе эффективности параллельных программ обычно рассуждают о параллелизме выполнения операций. Дополнительно иногда принимают в расчет задержки на передачу данных между процессорами, считая, что они для всех пар процессоров одинаковы или, по крайней мере, ограничены некоторой константой (см., например, [6]). И лишь очень редко встречаются оценки на основе учета конфликтов в коммуникационной сети, или, говоря иначе, учета ограничений по ее пропускной способности.

В данной работе исследуется влияние пропускной способности коммуникационной среды на производительность параллельной вычислительной системы на примере задачи умножения матриц.

Обычно в качестве меры пропускной способности используют понятие бисекционной пропускной способности. Мы увидим, что этот подход недостаточен, поскольку в нем не принимается в расчет сложная многоуровневая организация сети. Фактически учитываются только свойства сети самого верхнего уровня организации системы.

Нас интересуют очень большие системы, состоящие из многих тысяч процессоров. Они неизбежно будут иметь иерархическое строение, связанное с технологией их разработки и производства. Например, если в одном кристалле (чипе) могут находиться десятки и сотни вычислительных устройств, то взаимодействие между уст-

ройствами одного кристалла будет, скорее всего, обеспечено гораздо лучше, чем взаимодействие между устройствами из разных кристаллов.

В качестве модельной вычислительной системы будем рассматривать проект системы Merrimac, разработанный в Станфордском университете [1]. И хотя он так и не был воплощен в "железо", его идейный потенциал далеко не исчерпан. Его можно считать образцом хорошо сбалансированной высокопроизводительной вычислительной системы. Для нас важным является его коммуникационное обеспечение, которое "заточено" на эффективную передачу интенсивного потока мелких сообщений (до нескольких байтов). В разделе 1 даются основные существенные для нас сведения об этой системе. Мы используем ее лишь в качестве примера для иллюстрации нашего подхода к оценке производительности, но с равным успехом его можно применить и для других систем.

Затем, в разделе 2, проводится анализ алгоритма умножения плотных матриц с точки зрения возникающей при его выполнении коммуникационной нагрузки. При этом мы не ориентируемся ни на какую конкретную модель вычислений или язык программирования и не используем никакого конкретного программного кода. Будем исходить просто из постановки математической задачи и математического же представления о вычислении как о некоем вычислительном графе, где в вершинах находятся скалярные операции типа сложения или умножения, а по ребрам передаются данные от результатов одних операций к аргументам других. Мы вводим понятие коммуникационной сложности алгоритма как наименьшей коммуникационной нагрузки на сеть, соединяющую K процессорных элементов, среди которых равномерно распределяется вычислительный граф. Для умножения матриц порядка N дается оценка коммуникационной сложности снизу в виде формулы $O(N^2\sqrt[3]{K})$.

Наши рассуждения применимы к любой реализации математической схемы алгоритма и любым уровням организации системы (FPU, чип, плата и т. п.). Это и делается в разделе 3 для различных уровней системы Merrimac. Показывается, что производительность существенно зависит от распределения вычислений по элементам вычислительной системы.

Наконец, в разделе 4 описывается конструктивно метод распределения, обеспечивающий минимум коммуникационной нагрузки на всех уровнях системы.

В качестве дополнительной иллюстрации подхода в разделе 5 даются аналогичные оценки для другой гипотетической системы — HTMT. Обосновывается тезис, что при оценивании и сравнении производительности на основе теста по умножению плотных матриц важен такой показатель, как минимальный размер матриц, при котором все еще достигается так называемая проектная производительность.

1. Структура системы Merrimac

Система Merrimac с проектной производительностью 2 Пфлопс имеет иерархическую структуру, в которой насчитывается пять уровней (шесть вместе с уровнем всей системы). На первом уровне находится функциональное арифметическое устройство с плавающей точкой (FPU), способное вычислять одно сложение и одно умножение на каждом такте при частоте 1 ГГц. У каждого FPU имеется своя небольшая регистровая память (по 64 регистра длиной 64 бит на операнды и результат). FPU объединяются в кластеры по четыре. В одном чипе находятся 16 таких кластеров. Далее чипы (nodes) размещаются по 16 штук на плате, 32 платы объединяются в стойку, 16 или 32 стойки образуют систему.

На каждом уровне объединения используется коммуникационная среда, обеспечивающая взаи-

Таблица 1

Параметры коммуникационной среды системы Merrimac

Компонент уровня	Число компонентов предыдущего уровня	Число минимальных элементов (FPU)	Производительность, Гфлоп/с	Пропускная способность канала взаимодействия данного компонента с его внешней средой, Гбайт/с	Пропускная способность сети, объединяющей компоненты данного уровня, в пересчете на 1 узел, Гбайт/с
FPU	—	1	2	8	512 (внутрикластерный коммутатор)
Кластер	4 FPU	4	8	4	64 (межкластерный коммутатор)
Узел (чип)	16 кластеров	64	128	20	20
Плата	16 узлов	1024	2048	80	5
Стойка	32 платы	32K	64000	1280	2,5
Система	16 стоек[1]	512K	1 000 000		—

[1] Для удобства сопоставления с другими системами используется модель половинного размера с проектной пиковой производительностью ровно в 1 Пфлопс.

модействие между объединяемыми компонентами, а также канал обмена между ними и вышестоящей средой. В табл. 1 приведены параметры коммуникационной среды на каждом уровне структурной иерархии. Для компонентов каждого уровня указаны:

- число компонентов предыдущего уровня, из которых состоит данный компонент;
- число FPU, содержащихся в данном компоненте;
- пиковая производительность (миллиарды операций в секунду) данной единицы;
- ширина (пропускная способность) каналов связи данной единицы с ее внешней средой (Гбайт/с);
- удельная пропускная способность, исчисленная как отношение пропускной способности к объему данной единицы (здесь — к числу содержащихся в ней узлов).

На некоторых уровнях (FPU, кластер, узел) имеется память (регистровая, стримовая, кэш + + DRAM). Каждое FPU использует свою локальную регистровую память объемом 192 64-разрядных слова. Каждый кластер через внутрикластерный коммутатор пользуется своим банком стримовой памяти емкостью 8 Кслов. Все кластеры чипа через межкластерный коммутатор используют общий кэш, выходящий на внешние блоки DRAM. Память каждого вида приписана тому структурному уровню, на котором к ней имеется непосредственный доступ, а доступ из других этажей нагружает соответствующую коммуникационную среду. Здесь нас будет интересовать только пропускная способность коммуникационных сред разного уровня как основной лимитирующий фактор.

Из табл. 1 видно, что на разных уровнях имеются разные пропускные способности в пересчете на одну единицу (узел или ФАУ), причем с ростом уровня эти пропускные способности убывают. По-видимому, это связано с возрастанием стоимости связей по мере повышения уровня в связи с увеличением их протяженности.

Есть два пути решения этой проблемы. Можно стремиться обеспечивать равную пропускную способность на всех уровнях, и тогда программисты будут практически избавлены от необходимости думать об оптимальном распределении вычислений и данных. Проблема будет оставаться только в некотором различии задержек, но оно вполне может скрываться избыточным параллелизмом задач (при наличии достаточного места для сохранения состояний ожидающих процессов). Но цена этого пути может оказаться высокой, и основная доля стоимости системы будет приходиться на коммуникационное оборудование самого верхнего уровня.

Другой путь состоит в том, чтобы посредством надлежащего программирования обеспечивать значительное преобладание ближних передач данных над дальними. Для одних задач это проще, для других труднее. Зато будет заметно дешевле оборудование — в ущерб производительности некоторых "трудных" задач.

Возможность снижения удельного веса дальних передач для конкретной задачи может иметь свои пределы. Ниже мы проанализируем на предмет этих пределов задачу умножения матриц. На основании полученных оценок будут выведены ограничения на производительность, связанные с передачами данных между компонентами системы.

2. Задача умножения матриц

Результатом умножение двух матриц A и B размерностью соответственно $N_1 \times N_0$ и $N_0 \times N_2$ является матрица C размерностью $N_1 \times N_2$ с элементами

$$C_{ij} = \sum_{k=1}^{N_0} A_{ik} B_{kj}. \qquad (1)$$

Данную задачу (обычно при $N_0 = N_1 = N_2 = N$) часто используют для оценки потенциальных возможностей высокопроизводительных вычислительных систем (суперкомпьютеров). Это связано с тем, что в ней:

- число операций заметно больше, чем объем вводимых данных (N_3 против N_2);
- граф связей достаточно тесный, и задача плохо разбивается на слабо связанные подзадачи;
- структура задачи достаточно проста для проведения анализа эффективности;
- у нее нет простых альтернативных, заметно более эффективных решений.

Для формализации второго свойства введем понятие *коммуникационной сложности* задачи. Это функция $S(N, K)$ от размерности задачи N и числа процессоров K. Ее значение показывает минимально возможный объем пересылаемой между процессорами информации при равномерном распределении вычислительной нагрузки. Можно показать (и ниже это будет сделано), что коммуникационная сложность умножения матриц размерности $N \times N$ равна $O(N^2 \sqrt[3]{K})$.

Используя понятие коммуникационной сложности, можно дать оценку снизу времени выполнения задачи при ее распределенном выполнении на системе из K элементов. В роли элементов могут выступать любые структурные единицы (плата, процессор и т. д.). Соответственно, коммуникационную сложность, вычисленную исходя из

общего числа этих единиц, следует сопоставить с общей пропускной способностью.

При умножении двух матриц по формуле (1) необходимо выполнить $N_0N_1N_2$ скалярных умножений и почти столько же сложений. Несмотря на то, что существуют теоретически более эффективные (в смысле числа умножений) решения [4], мы будем исходить из прямого вычисления данной формулы. Будем допускать только вариации, связанные с ассоциативностью и коммутативностью сложений.

Пусть имеется K процессорных модулей, соединенных коммуникационной сетью. Множество операций скалярного умножения, выполняемых в алгоритме, представляет собой параллелепипед размера $N_0 \times N_1 \times N_2$. Он должен быть как-то разбит на части, распределяемые среди K процессорных модулей.

Не вдаваясь в доказательство, примем достаточно правдоподобное допущение, что оптимальное (в смысле экономии пересылок) распределение устроено следующим образом. Каждое измерение параллелепипеда разбивается соответственно на K_0, K_1 и K_2 равных частей, где $K_0K_1K_2 = K$. В каждом процессоре исполняется один из блоков, ограниченный плоскостями, проведенными через точки деления (рис. 1).

В модуле номер $\langle k_0, k_1, k_2\rangle$ (где k_1 — номер части соответствующего направления) блок номер $\langle k_1, k_0\rangle$ матрицы A умножается на блок номер $\langle k_0, k_2\rangle$ матрицы B. Результатом является слагаемое для блока номер $\langle k_1 \cdot k_2\rangle$ матрицы C. В каждый модуль приходят два входных блока и выходит один выходной. Их размеры — $d_1 \times d_0$, $d_0 \times d_2$ и $d_1 \times d_2$ соответственно, где $d_l = N_l/K_l$.

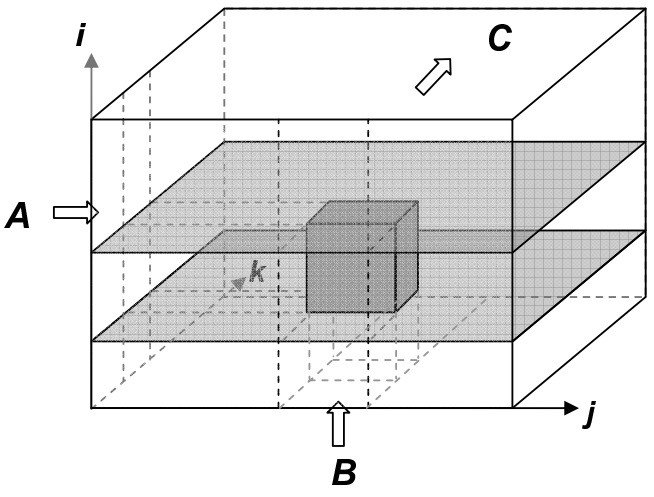


Рис. 1. Пространство узлов вычислительного графа с операцией умножения. Внутри выделена область, выполняемая в отдельном процессорном элементе. Элементы исходной матрицы A_{ik} распространяются вдоль направления j, матрицы B_{kj} — вдоль направления i. Результат C_{ij} снимается вдоль направления k

Общая нагрузка (в словах) на один процессорный модуль составляет

$$S_{PROC} = d_1d_0 + d_0d_2 + d_1d_2 = \\ = d_0d_1d_2(d_0^{-1} + d_1^{-1} + d_2^{-1}). \quad (2)$$

Тогда общая коммуникационная сложность

$$S_{TOTAL} = S_{PROC}K_0K_1K_2 = N_0N_1N_2(d_0^{-1} + \\ + d_1^{-1} + d_2^{-1}) = N_0N_1N_2\Big(\frac{K_0}{N_0} + \frac{K_1}{N_1} + \frac{K_2}{N_2}\Big). \quad (3)$$

Если считать размерности всех матриц и общее число процессоров K заданными, то в скобках будет стоять сумма трех чисел с фиксированным произведением. Ее минимум будет достигаться при равных слагаемых, т. е. все блоки должны быть как можно ближе к квадратным. В этом случае коммуникационная сложность задачи составит

$$S_{OPT} = 3N_0N_1N_2\sqrt[3]{\frac{K}{N_0N_1N_2}}. \quad (4)$$

В частности, при подстановке $N_0 = N_1 = N_2 = N$ получаем

$$S_{TOTAL} = N^2(K_0 + K_1 + K_2); \quad (5)$$

$$S_{OPT} = 3N^2\sqrt[3]{K}. \quad (6)$$

Мы пока не приняли в расчет операции межблочного сложения и соответствующие передачи данных. Но в этом нет необходимости, поскольку сложения не меняют количества передаваемой информации. Если требуется выдать сумму K чисел, хранящихся в K разных процессорах, то, как ни крути, придется передать через коммутатор не менее K чисел.

Полученные оценки позволяют оценить снизу время выполнения задачи, если известна общая пропускная способность сети BW или удельная пропускная способность $BW_{PROC} = BW/K$:

$$T_{COMM} \geqslant S_{OPT}/BW = S_{OPT}/(K \cdot BW_{PROC}). \quad (7)$$

Интересно сопоставить коммуникационную сложность с нагрузкой сети при далеком от оптимального распределении вычислительных операций по процессорам. В частности, если использовать равномерно-случайное разбрасывание, то величина трафика составит

$$S_{RAN} = 3N^3\frac{K-1}{K}. \quad (8)$$

При больших K сомножителем $\frac{K-1}{K}$ можно пренебречь. Получаем

$$\frac{S_{RAN}}{S_{OPT}} \approx \frac{N}{\sqrt[3]{K}}. \quad (9)$$

При $K \approx N^3$ формула (9) утверждает, что случайное разбрасывание будет также оптимальным. Не удивительно, поскольку в этом случае каждая операция назначается на свой отдельный процессор, а стоимости пересылок между разными процессорами считаем не зависящими от номеров процессоров.

Если $K \ll N^3$, то коммуникационная нагрузка в оптимальном разбиении будет в $\sqrt[3]{\frac{N^3}{K}}$ раз лучше, чем в случайном. Соответственным будет и выигрыш, если сеть загружена на пределе.

3. Анализ производительности суперкомпьютера Merrimac на задаче умножения матриц

При изучении эффективности параллельной системы обычно рассматривают два аспекта: загрузку вычислительного оборудования и загрузку коммуникационных каналов. Если предположить, что параллелизм задачи более чем достаточен для того, чтобы нагрузить на 100 % все арифметические устройства, то производительность может сдерживаться либо со стороны FPU, либо со стороны коммуникационной сети. Зная число K и производительность PR вычислительных компонентов, а также общее число операций в задаче N_{OP}, легко рассчитать предельное вычислительное время $T_{COMP} = \frac{N_{OP}}{K \cdot PR}$. Вместе с тем, зная пропускную способность межкомпонентного обмена BW (на компонент) и нагрузку S, можно рассчитать предельное коммуникационное время $T_{COMM} = \frac{S}{K \cdot BW}$. Далее, предполагая возможность совмещения обменов с вычислениями, имеем следующую оценку снизу для времени выполнения алгоритма:

$$T = \min(T_{COMP}, T_{COMM}). \qquad (10)$$

Система Merrimac имеет несколько уровней иерархии, и для разных уровней будут разные значения T_{COMM}. Вычислим оценки для задачи умножения матриц размерностью 1024 × 1024. На каждом уровне иерархии будет свое оптимальное разбиение параллелепипеда скалярных операций по компонентам. Для удобства будем выбирать его так, чтобы все K_l были степенями 2. Тогда условием оптимальности будет не равенство всех K_l, а то, что они должны отличаться друг от друга не более чем в 2 раза.

В табл. 2 приведены данные по значениям K, PR и BW для компонентов разных уровней и вычисленные значения T_{COMM} для случая $N_0 = N_1 = N_2 = N = 1024$. Для вычислительного времени имеем

$$T_{COMP} = \frac{2N^3}{1PFlops} = 2 \text{ мкс}. \qquad (11)$$

Для каждого уровня получилось свое время T_{COMM}. Наибольшее из них (4,8 мкс) и будет определять нижнюю оценку. Поскольку она получилась больше, чем T_{COMP}, приходится признать, что на данной задаче производительность 1Пфлопс недостижима. В лучшем случае будет $1\text{Пфлопс}\frac{T_{COMM}}{T_{COMP}} = 0{,}42\text{Пфлопс}$.

Можно улучшить показатель производительности, если увеличить размерность матриц. Дело в том, что число операций растет пропорционально N^3, а нагрузка на коммуникационные сети — пропорционально N^2. При любом N наихудшее коммуникационное время будет на уровне взаимодействия между платами. В табл. 3 показаны времена T_{COMM} и T_{COMP}, а также максимально возможная производительность для различных N.

Таким образом, на данной системе в силу ограничений ее коммуникационной сети предельная производительность в Пфлопс достижима на задаче умножения матриц размерностью не ниже 4096 × 4096. Сохранение производительности при понижении размерности матриц вдвое требует соответствующего удвоения пропускной способности коммуникационных сред, в первую оче-

Таблица 2

Расчет коммуникационного времени для $N = 1024$, $T_{COMP} = 2$ мкс

Уровень	Число элементов K	Пропускная способность BW, Гслов/с	Наилучшее разбиение $K_0 + K_1 + K_2$	Коммуникационная нагрузка на сеть $S = N^2(K_0 + K_1 + K_2)$, Мслов	$T_{COMM} = \frac{S}{K \cdot BW}$, мкс
FPU	512K	1	64 + 64 + 128 = 256	256	0,5
Кластер	128K	0,5	32 + 64 + 64 = 160	160	2,5
Узел	8K	2,5	16 + 16 + 32 = 64	64	3,2
Плата	512	10	8 + 8 + 8 = 24	24	4,8
Стойка	16	160	2 + 2 + 4 = 8	8	3,2
Система	1				

Таблица 3

Времена вычислений и коммуникаций для различных N

N	T_{COMP}, мкс	T_{COMM}, мкс	Пфлопс
256	0,03	0,3	0,10
512	0,25	1,2	0,21
1024	2	4,8	0,42
2048	16	19,2	0,84
4096	128	76,8	1,0
8192	1024	307,2	1,0

редь тех, которые работают на пределе. Основным ограничителем в данном случае является сеть между платами. Было бы разумно увеличить ее пропускную способность хотя бы в 1,5 раза. Тогда пропускные способности трех верхних уровней были бы более сбалансированными.

4. Достижимость оценок

Нагрузки S (см. табл. 2), вычисленные в предыдущем разделе, являются нижними оценками. Они получены из рассмотрения некоторого класса отображений вычислительных операций на аппаратную среду. Выведенные общие формулы были независимо применены к различным структурным уровням, причем оценки, получаемые для каждого уровня, реализуются на своем отображении. Возникает вопрос, а достижимы ли все эти оценки одновременно, т. е. возможно ли такое отображение вычислительных операций на совокупность из всех вычислительных устройств, при котором достигаются все полученные оценки?

Ответ утвердительный. Действительно, существует отображение, которое обеспечивает оптимальность распределения на всех уровнях. Чтобы его описать, введем единую нумерацию всех арифметических устройств (FPU). Полный номер FPU на рассмотренной конфигурации системы кодируется 19-разрядным двоичным числом, составленным из полей, кодирующих номер FPU в кластере, номер кластера в узле, номер узла в плате и т. д., расположенных справа налево (рис. 2).

Операции умножения, подлежащие отображению в FPU, кодируются тройкой индексов: $\langle i, k, j\rangle$,

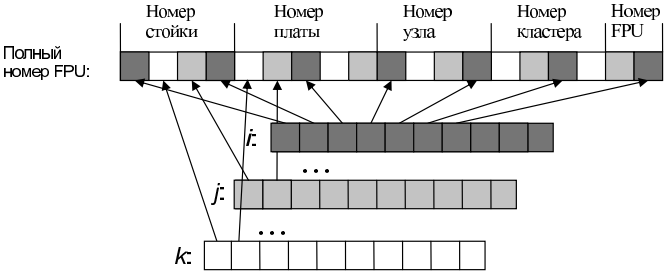


Рис. 2. Отображение двоичных разрядов индексов *i, j, k* в двоичные разряды полного номера FPU

где $\langle i, k\rangle$ — индексы элемента первой матрицы, $\langle k, j\rangle$ — индексы элемента второй матрицы. Каждый индекс имеет десять двоичных разрядов (для $N = 1024$). Для получения полного номера FPU по индексам $\langle i, k, j\rangle$ двоичные разряды последних скрещиваются, как показано на рис. 2.

Рассмотрим какой-либо уровень системы, например, уровень, где компонентами являются платы. Полный номер платы состоит из номера стойки и номера платы внутри стойки и содержит девять двоичных разрядов. Среди этих разрядов на каждый из трех индексов приходится ровно по три разряда. Это разряды старшие. Следовательно, на конкретную плату попадают тройки индексов, у которых в старших трех битах всех индексов определенные общие значения. Отсюда видно, что блок операций конкретной платы имеет размеры, равные 1/8 от размеров полного параллелепипеда (в данном случае — куба). Поэтому для плат $K_0 = K_1 = K_2 = 8$, что в точности совпадает с оптимальными параметрами, указанными в табл. 2. Легко проверить, что для остальных уровней тройки $\langle K_0, K_1, K_2\rangle$ также соответствуют указанным в табл. 2. Таким образом, построенное отображение действительно является оптимальным одновременно для всех уровней системы.

Математически это отображение можно выразить с помощью функции zip3(a, b, c), которая выдает целое число, вычисленное по следующей рекурсивной схеме:

$$\mathrm{zip3}(0,0,0) = 0;$$

$$\mathrm{zip3}(2\cdot a + d, 2\cdot b + e, 2\cdot c + f) = (((\mathrm{zip}(a, b, c)\cdot 2 + d)\cdot 2 + e)\cdot 2 + f,$$

где d, e и f могут принимать значения только из {0, 1}. Тогда отображение запишется в виде формулы

$$N_{FPU}(i, k, j) = \mathrm{trunc}(K_{FPU}\cdot \mathrm{zip}(i, k, j)/N^3)). \quad (12)$$

В зависимости от языка программирования это отображение в той или иной форме должно быть частью дизайна параллельного алгоритма.

Заметим, что указанное отображение не зависит от числа уровней и особенностей каждого уровня, хотя зависит от общего числа элементарных вычислительных устройств K_{FPU}. От последней зависимости тоже можно избавиться, если кодировать номер FPU не целым числом, а дробным, лежащим в диапазоне 0...1. Тогда формула для отображения упростится до вида

$$N(i, k, j) = \mathrm{zip}(i, k, j)/N, \quad (13)$$

где "/" понимается как деление с вещественным результатом. Эта формула определяет оптимальное отображение при любом выборе элементарной единицы (FPU, узел, и т. п.). И есть основания полагать, что алгоритм с таким отображением является оптимальным для систем с любой ком-

муникационной структурой. Главное, чтобы нумерация аппаратных единиц соответствовала аппаратной связности, когда коммуникационно близкие единицы имеют близкие номера.

5. Аналогичные работы

Для сравнения упомянем работу [5], где на примере этой же задачи обсуждается производительность другой гипотетической системы — HTMT (*Hybrid Technology Multi-Threaded architecture*), основанной на криогенной технологии одноквантовой логики RSFQ. Для получения предельной производительности в 1,1 Пфлопс авторы используют умножение матриц с $N = 208\,000$, выполняемое за 16,2 с. Как следует из наших рассуждений, такой тест не раскрывает достоинства системы в должной мере. Интересно было бы найти наименьшую размерность матриц, которые умножались бы с такой же производительностью.

В самой статье [5] размерность матриц выбрана как наибольшая, при которой она помещается в памятях разных уровней. Реализован трехуровневый блочный алгоритм, где на среднем уровне применяется систолический алгоритм Кэннона на двумерной решетке процессорных элементов SPELL размера 64 × 64. На ней умножаются подматрицы размерности 8000 × 8000 за 0,75 мс, что равнозначно производительности чуть более 1Пфлопс. Нас интересует возможность уменьшить размерность матриц с сохранением петафлопсной производительности.

Рассмотрим умножение матриц размерности 1024 × 1024. На каждый из 4096 SPELL приходится по блоку размера 16 × 16. Каждый SPELL имеет 5 FPU с циклом 15пс. Поэтому умножение блоков (при полной загрузке FPU) потребует время (обозначения взяты из [5])

$$M_s = 2 \cdot 16^3 \frac{15\text{пс}}{5} = 25 \text{ нс}. \qquad (14)$$

Передача блока между соседними SPELL потребует время

$$D_s = 16^2\text{слов} \frac{1}{16.7\text{Гслов/с}} = 15 \text{ нс}. \qquad (15)$$

Поскольку $M_s > D_s$, время пересылок может быть скрыто их совмещением с вычислениями (хотя это потребует дополнительных ухищрений), и потому общее время работы алгоритма составит $64M_s = 1{,}6$ мкс. Дополнительно следует учесть время загрузки и выгрузки результатов, которое исходя из скорости работы внешней памяти SRAM можно оценить в 0,3 мкс. Тем самым на этом размере производительность 1Пфлопс достигается, хотя и на пределе. А на размере 512 × 512 это будет уже не так, поскольку время пересылок будет на 20 % превосходить время вычислений. Таким образом, минимальная (из выражающихся степенью двойки) размерность матриц, при умножении которых достижим (теоретически) петафлопсный уровень, есть 1024.

Из этого мы можем также сделать вывод, что коммуникационное обеспечение системы HTMT в целом примерно в 3—4 раза лучше, чем у Merrimac (по крайней мере, для задач с коммуникационной сложностью, подобной той, которая свойственна умножению матриц). Не так много, если учесть, что оно основано на гораздо более продвинутой технологии RSFQ.

Выводы

Показателем "распараллеливаемости" задачи является не только ширина ее вычислительного графа (число операций, которые могут выполняться параллельно), но и объемы возникающих информационных потоков. Пределы в реализации вычислений могут проистекать не только из ограничений по производительности арифметических устройств и латентности (задержек) при передаче данных, но и из пропускной способности коммуникационной сети. Важно уметь оценивать свойство алгоритмов оказывать давление на сеть и оптимизировать их с этой точки зрения.

При равной стоимости сетей разного уровня пропускные способности сетей более высоких уровней обычно получаются меньше (в пересчете на один процессорный модуль). Поэтому при распределении вычислительного графа по процессорным модулям следует стремиться к тому, чтобы большая часть связей падала на сети более низких уровней. Если этого не делать, полагаясь, например, на случайное разбрасывание узлов графа по модулям, то большая часть коммуникационной нагрузки будет падать на сеть самого верхнего уровня (с самыми дальними связями), обладающей самой низкой удельной пропускной способностью, со всеми вытекающими из этого последствиями для производительности системы.

Обычно в качестве меры пропускной способности сети используют *бисекционную пропускную способность*. Из наших рассуждений следует, что этого недостаточно. Бисекционная пропускная способность характеризует пределы потока "самых дальних" передач. Но если задача обладает локальностью, то близкие передачи преобладают над дальними и, как следствие, могут стать более существенными пределы для потока на близкие расстояния. Так, при умножении матриц размерности 1024 × 1024 на системе Merrimac основным сдерживающим фактором является сеть между платами. А сеть между стойками заполнена на 2/3.

Поэтому снижение бисекционной пропускной способности на 30 % на нашей задаче пройдет незамеченным, т. е. как показатель пропускной способности сети в целом она не достаточна. Необходимо обобщение. Например, можно определить k-секционную пропускную способность. Разбиваем систему на k примерно равных частей, выбираем произвольно одну из них и определяем максимальный поток из этой части в направлении остальных. Минимум этого потока по всем разбиениям и будет значением k-секционной пропускной способности. При $k = 2$ получается обычная бисекционная. И теперь пропускная способность сети выражается не одним числом, а некоторой функцией от k, где аргумент k изменяется от 2 до N, где N — число минимальных функциональных вычислительных устройств.

Задачи должны программироваться и распределяться так, чтобы коммуникационная нагрузка на сеть была оптимальной. При этом желательно, чтобы не приходилось подстраиваться под особенности коммуникационной аппаратуры. Для задачи умножения матриц нам это удалось. Достижение этой цели можно сравнить с разработкой так называемых кэш-независимых (*cash-oblivious*) алгоритмов [3].

Выражаю благодарность В. Н. Балину за привлечение меня к данной работе, Д. С. Северову — за подчеркивание важности проблемы коммуникаций, В. А. Александрову — за побуждение к работе по написанию статьи, А. М. Степанову — за ценные замечания, А. С. Окуневу — за моральную поддержку.

Список литературы

1. **Dally W.** et al. Merrimac: Supercomputing with streams. http:/merrimac.stanford.edu/publications/sc03_merrimac.pdf
2. **Танненбаум Э.** Архитектура компьютера: Пер. с англ, 4-е изд. СПб.: Питер, 2002.
3. **Frigo M., Leiserson C. E., Prokop H. and Ramachandran S.** Cache Oblivious Algorithms // In 40th Annual Symposium on Foundations of Computer Science. New York. Oct. 1999. P. 285—298.
4. **Strassen V.** Gaussian Elimination is Not Optimal // Numerische Matematik. 1969. Vol. 13. P. 353—356.
5. **Amaral J. N.** et al., An HTMT Performance Prediction Case Study: Implementing Cannon's Dense Matrix Multiply Algorithm // CAPSL Tech. Memo 26, University of Delaware. February, 17, 1999, USA.
6. **Wilkinson B., Allen M.** Parallel Programming. 2d ed. Pearson Prentice Hall, 2005.

УДК 004.272

А. С. Оленин, д-р физ.-мат. наук,
Институт проблем проектирования
в микроэлектронике РАН

Проблемы приближенного конструирования задач

Рассматриваются проблемы приближения произвольных вычислительных структур совокупностью базовых элементов, вопросы создания условий, при которых архитектура ЭВМ могла бы на ранних стадиях разработки функционировать с учетом специфики больших задач, отмечено важное значение для этого концепции подобия, предложен семантический метод построения приближенных моделей реальных задач.

Практика применения параллельных вычислительных систем на широком спектре больших научно-технических задач показывает значительное снижение реальной производительности системы по сравнению с ее пиковой производительностью. Главная причина этого состоит в том, что структура параллельности задач, которая весьма индивидуальна для каждой из них, оказывается неадекватной параллельной структуре системы. Современные суперкомпьютеры традиционной фоннеймановской архитектуры предоставляют довольно ограниченный набор средств параллельного структурирования программ и поэтому часто не в состоянии полностью раскрыть внутренний потенциал параллельных свойств задачи, а значит, не позволяют достичь наиболее высокой реальной производительности.

В значительной степени делу помогают усилия пользователей, тратящих немало времени, чтобы извлечь максимальный эффект из тех вычислительных средств, которые имеются в их распоряжении. Однако принципиально это проблему не решает. Нужны средства, которые, несмотря на многообразие форм параллельных вычислительных графов [1], присущих различным задачам, универсально отрабатывали бы их общее свойство — наличие независимых операций, выполняемых по мере готовности к исполнению. В этом случае внутренний параллелизм задачи автоматически проявляется при любой сложности вычислительных графов.

Анализ множества вычислительных алгоритмов [2] показывает, что в них широко представлены вычисления с простейшими упорядоченны-

ми графами, которые могут быть тщательно изучены и реализованы с максимальной эффективностью специальными средствами. В этой связи представляется полезным условно выделить некоторые характерные типы вычислений.

Последовательные вычисления. Здесь вычислительный процесс состоит из последовательности шагов вида

$$c = a.f.b,$$

где a, b, c — скалярные операнды; f — знак операции (например, двухместной). Этот тип вычислений весьма распространен и имеет место, когда параллельность в программе отсутствует или не поддерживается архитектурой компьютера.

Векторные вычисления представляются совокупностью шагов вида

$$\boldsymbol{C} = \boldsymbol{A}.f.\boldsymbol{B},$$

где $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ — операнды, являющиеся векторами; f — знак операции, единый для всех элементов вектора. Естественно, что размеры векторов могут изменяться от шага к шагу. В общем случае в качестве операндов выступают матрицы, массивы и т. п.

Параллельные вычисления отвечают записи

$$\boldsymbol{F} = \boldsymbol{A}.\boldsymbol{F}.\boldsymbol{B},$$

здесь каждому элементу операндов может соответствовать свой знак операции, т. е. $\boldsymbol{F}$ — это вектор знаков операций, а в общем случае — матрица, массив и т. п.

Можно видеть, что из приведенных типов вычислений наиболее обобщенную форму имеют параллельные вычисления, которые включают, как частный случай, последовательный и векторный типы. Отметим, что векторный тип представляет собой простейший, но весьма распространенный вид параллелизма, который в большинстве случаев может быть явно выделен в программах.

Обобщенная форма параллельных вычислений описывает все многообразие графов вычислений. Однако идентифицировать ее в явном виде далеко не всегда просто. Именно этой проблемой с помощью разного рода систем параллельного программирования занимаются пользователи, пытаясь достичь высокой реальной производительности на традиционных параллельных системах, которые требуют как раз явного представления графа для последующего его распараллеливания. Какова бы ни была параллельная архитектура традиционной фон-неймановской машины, она предполагает в основном явную реализацию параллелизма со всеми вытекающими отсюда трудностями для ее пользовательской поддержки. Но, например, архитектуры потокового типа с автоматическим распределением ресурсов [3] не нуждаются в явной форме вычислительного графа, а реализуют ее неявно, что является существенным преимуществом в случае динамически сложных нерегулярных вычислений.

Как известно, структурную среду для вычислительных задач образуют в основном следующие компоненты:

- структуры данных (скаляры, векторы, матрицы, массивы, битовые поля, логические переменные и т. д.);
- структуры программ (операторы, циклы, подпрограммы и т. д.);
- структуры (графы) вычислений, соответствующие алгоритмическому строению задачи.

Причем в больших задачах наиболее существенно проявляются разнообразие, сильная связанность всех структурных элементов между собой и зависимость их функционирования от динамически формируемых данных. Все эти и многие другие объективные особенности сложных вычислений должны учитываться при поиске и принятии оптимальных архитектурных решений, так как в противном случае потенциальные возможности ускорения счета, заложенные в структуре задач, будут реализованы неэффективно.

В связи с такой ситуацией поставим следующий вопрос. Как создавать условия, при которых архитектура уже на ранних стадиях разработки (например, в виде имитационной модели с сильными ограничениями на объем вычислений) могла бы функционировать с учетом специфики больших задач? Если этого сделать нельзя, мы практически до завершения проекта не можем иметь объективных данных о функционировании машины на задачах крупномасштабной информационной и структурной емкости. При этом исправление слишком поздно выявленных недостатков становится зачастую либо невозможным, либо чрезвычайно дорогостоящим. Отсюда ясно, что поставленный вопрос существенно затрагивает и экономическую сторону проекта, особенно на фоне финансовых ограничений и нарастающей сложности проблем.

Уместно вспомнить, что аналогичная ситуация имела место еще в докомпьютерную эпоху в ряде областей науки и техники. Например, при конструировании летательных аппаратов сначала проводились исследования на малых моделях в аэродинамических трубах. В результате продувок оценивались многие важные факторы, влияющие на конструкцию реальных аппаратов. Благодаря таким исследованиям технические и экономические качества разработок существенно повышались. Для нашего дальнейшего рас-

смотрения важно отметить, что в этих работах основанием для переноса данных, полученных в условиях маломасштабных моделей, на масштаб реальных объектов служило подобие физических процессов при соблюдении известных критериев подобия. Появление компьютеров позволило существенным образом развернуть эту проблему в сторону машинного моделирования, так как в принципе стало возможным исследовать любую крупномасштабную задачу (доступную ресурсам ЭВМ) без привлечения дорогих натурных экспериментов. Однако точность и достоверность компьютерного моделирования всецело зависят от качества математических и численных моделей, что является основным предметом внимания специалистов по постановке больших задач.

В области разработки архитектуры ЭВМ ситуация принципиально отлична от той, что имеет место в областях приложений вычислительных средств. Это, прежде всего, связано с тем, что никакая модель проектируемой машины не может быть сравнима по производительности с реально действующей ЭВМ. Поэтому применение даже наиболее мощных суперкомпьютеров для функционирования такой модели не в состоянии обеспечить ее обкатку на реальных крупномасштабных задачах, для которых, в первую очередь, и предназначена разработка. Для того чтобы преодолеть этот замкнутый круг, принципиальное значение приобретает концепция подобия. Необходимо иметь ответ на следующий вопрос: при каких условиях результаты исследований, полученные на малых масштабах с помощью модели, можно переносить на масштабы больших задач для будущей реальной машины. Опыт исследований в этом направлении показывает, что главным критерием подобия в данном случае является вычислительный граф, описывающий содержание задачи в структурной форме. Иными словами, если структура вычислений на малых и больших масштабах сохраняется постоянной, то, несмотря на возможный разброс, результаты моделирования в обоих случаях обязательно должны коррелировать между собой, т. е. нести информацию друг о друге.

Таким образом, чтобы эффективно использовать принцип подобия при разработке архитектуры ЭВМ, необходимо: во-первых, уметь выполнять расчеты на малых и больших масштабах с неизменным графом вычислений, а во-вторых, уметь делать это для сложных структур, присущих большим задачам. Остановимся подробнее на этих условиях.

Для достижения первого условия автором на принципах комплексной типизации и анализа всего процесса постановки задач разработана экспериментальная версия пакета представительных вычислений [4], ориентированная на физико-математические приложения. Особенность пакета состоит в том, что он содержит алгоритмы с разнообразной структурой вычислений. Причем уже в эти алгоритмы по возможности заложены такие важные свойства больших задач, как линейность и нелинейность, стационарность и нестационарность, непрерывность и дискретность, детерминированность и случайность, многомерность и т. д. Каждая программа пакета плавно масштабируется по объему вычислений за счет параметрически настраиваемых размеров массивов. Это позволяет исследовать программы как на малых, так и на больших масштабах времени при неизменном графе вычислений.

Переходя ко второму условию, отметим, что каждая отдельная большая задача имеет достаточно жесткую структурную организацию, поэтому результаты ее моделирования далеко не всегда применимы к каким-либо другим задачам. Однако интересы разработчиков архитектур ЭВМ требуют, чтобы предоставленные в их распоряжение задачи максимальным образом отражали характерные свойства тех вычислений, которые, хотя непосредственно и не участвуют в процессе проектирования, вполне реальны на практике. В этом случае появляются основания для более объективной оценки проекта на широком спектре приложений.

Изложенные соображения подводят нас к проблеме приближения произвольной вычислительной структуры совокупностью базовых структур. Математические аналоги различного рода приближений давно известны. Это, например, приближение произвольной функции набором базисных функций, приближение произвольного вектора набором базисных векторов и так далее. Применение аппарата приближений играет огромную роль в развитии многих областей научных исследований. Однако проблема приближения вычислительных структур выглядит, на наш взгляд, значительно сложнее и многограннее проблем математических приближений. Здесь встают чрезвычайно трудные с точки зрения формализации проблемы: во-первых, как определять характерный базис вычислительных структур, а во-вторых, как формировать произвольную структуру на основе базисного набора. По мере решения этих вопросов возможности синтеза гибких и сложных образований с заданными структурными свойствами

должны расширяться. При этом универсальный характер конструируемых структур обеспечивается общностью базиса, а их конкретная специфика определяется степенью участия базисных составляющих.

В силу отмеченной сложности формализованного подхода к проблеме приближения структур автором на первом этапе предлагается семантический метод ее решения, опирающийся на смысловой структурный анализ процесса решения задач. Возьмем, например, в качестве базиса набор представительных вычислений, о котором говорилось ранее, и на его основе построим некоторую приближенную модель реальной вычислительной задачи.

Рассмотрим вычислительную структуру решения двумерной задачи Навье—Стокса для несжимаемой вязкой жидкости. В одной из формулировок эта задача состоит из двух нестационарных уравнений в частных производных для скоростей течения и уравнения Пуассона для давления. Мы не будем выписывать эти хорошо известные уравнения, а остановимся на вычислительной стороне их решения. Алгоритм решения сводится к поочередному расчету полей скорости и давления в каждый момент времени с использованием информации об этих полях в предыдущие моменты. Во многих случаях нестационарные уравнения решаются с помощью какой-либо явной схемы, а для решения уравнения Пуассона применяется какой-нибудь прямой метод.

Акцентируя внимание на структуре вычислений, важно отметить следующее обстоятельство. В основе разностных схем лежит шаблон сетки, устанавливающий тип связи данного узла с его ближайшими соседями. Использование такого шаблона в явной форме определяет структуру вычислений любой явной схемы, различия же в конфигурации шаблонов, часто существенные в математическом смысле, практически не добавляют ничего качественно нового в характер вычислительного процесса. Поэтому в структурном смысле приближением множества явных схем может служить любая структурно подобная явная схема, в которой заложены характерные свойства данного множества.

Таким образом, имея в составе базисного набора программы, реализующие двумерную явную схему и, например, такие прямые методы решения уравнения Пуассона, как быстрое преобразование Фурье или циклическая редукция, можно осуществить приближенное моделирование реальной задачи Навье—Стокса в ее структурном подобии. Если соответствующие программы имеют масштабируемые параметры: размер сетки и число шагов по времени, то в соответствии с общей схемой решения интерпретирующая программа будет действовать так, как это делалось бы в реальной задаче. Три базисных программы поочередно работают на заданном размере сетки, выполняя общее для всех шагов заданное число итераций по времени. Несмотря на простоту, описанная приближенная модель, очевидно, отражает структурные свойства реальной задачи. Для нее не являются существенными конкретный вид коэффициентов уравнений, их правых частей и численные значения величин. В широком диапазоне этих факторов структура вычислений остается качественно неизменной, что позволяет отделить внутреннее содержание задачи от ее количественных многообразий.

Итак, на примере задачи Навье—Стокса мы сконструировали приближенную модель, которая более сложна, чем исходные базисные программы, близка по структуре к реальной задаче и может работать как на малых, так и на больших масштабах. Эти условия необходимы для того, чтобы изучать большие задачи в их структурном подобии на малопроизводительных моделях архитектуры.

Аналогичным образом на основе базисного набора элементов автором построены приближенные модели реальных задач с неоднородными полями и сложной геометрией, многосеточных схем, сильнодеформируемых сред, кинетических взаимодействий и других. Эти модели, несмотря на приближенный характер в плане реально существующих больших задач, в определенной степени интерпретируют реальные семантические и структурные особенности таких задач. Возможности функционирования на малых и больших масштабах с сохранением структуры вычислительного графа обеспечивают минимально необходимые условия для применения приближенных аналогов больших задач в исследованиях архитектуры на разных уровнях проектирования.

Список литературы

1. **Воеводин В. В., Воеводин Вл. В.** Параллельные вычисления. СПб.: БХВ-Петербург, 2002.

2. **Ахо А., Хопкрофт Дж., Ульман Дж.** Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

3. **Бурцев В. С.** Система массового параллелизма с автоматическим распределением аппаратных средств суперЭВМ в процессе решения задачи // Юбилейный сборник трудов институтов Отделения информатики, вычислительной техники и автоматизации РАН. Т. 2. М.: 1993. С. 5—27.

4. **Оленин А. С.** Принципы типизации вычислений // Системы и средства информатики. Вып. 16. М.: Наука, 2006. С. 386—392.

УДК 004.4'416

Галазин А. Б., Грабежной А. В.,
Нейман-заде М. И., канд. физ.-мат. наук,
ЗАО "МЦСТ"
galazin@mcst.ru, grab_av@mcst.ru, muradnz@mcst.ru

Оптимизация размещения данных для эффективного исполнения программ для архитектур с многобанковой кэш-памятью данных

Различные исследования показывают, что блокировки кэш-памяти являются важной проблемой достижения высокой производительности. В данной статье рассматривается структура блокировок многобанкового кэша и предлагается новый метод уменьшения их числа. Рассматриваемый метод был реализован в оптимизирующем компиляторе для микропроцессора "Эльбрус" и продемонстрировал свою эффективность на задачах пакетов Spec95 и Spec2000.

Введение

Критичным фактором обеспечения производительности современных микропроцессоров является эффективная загрузка их вычислительных устройств. Наиболее известным способом обеспечения непрерывной работы микропроцессора и снижения числа блокировок конвейера является организация иерархической системы памяти с использованием кэш-памятей разных уровней [1—3]. Метод кэширования подтвердил свою эффективность для различных классов приложений. Объем кэш-памяти ограничен физическими размерами микропроцессоров и необходимостью обеспечивать быстрый доступ к находящимся в ней данным. Поэтому несмотря на увеличение размеров кэш-памяти ее эффективное использование является важным условием достижения высокой производительности на современных вычислительных системах.

Основной проблемой, препятствующей максимально эффективному использованию кэш-памяти, является возможное вытеснение данных из кэш-памяти до их использования.

Для решения этой проблемы разработаны оптимизирующие преобразования размещения данных. Эти преобразования позволяют изменять адреса глобальных массивов или других глобальных структур данных с тем, чтобы уменьшить число конфликтов, ведущих к вытеснению данных. Одним из широко используемых методов преобразования размещения данных является Padding [4] — изменение начального адреса глобальных массивов или размерности массивов с помощью добавления неиспользуемых элементов (смещений). Преобразование Padding довольно широко описано в литературе, и его применение позволяет значительно уменьшить число промахов, тем самым повысив производительность. В то же время данное преобразование описано только для кэш-памятей, организованных в виде одного банка.

Для случая кэш-памяти, состоящей из нескольких банков, в известной авторам литературе данное преобразование не исследовалось и не описывалось. При одинаковой общей задаче повышения производительности методы оптимизации работы однобанковых и многобанковых кэш-памятей должны различаться, поскольку для однобанковых одной из основных проблем является вытеснение полезных данных из кэша, а для многобанковых — внутренние блокировки, связанные с неоптимальной работой банков кэш-памяти.

В данной статье описывается разработанное авторами преобразование Multibank Padding для многобанковой кэш-памяти второго уровня микропроцессора "Эльбрус" [5, 6]. Его реализация позволила снизить число блокировок кэш-памяти и повысить производительность на ряде задач пакетов Spec95, Spec2000 [7]. Все приведенные результаты получены на вычислительном комплексе с микропроцессором "Эльбрус".

Аппаратные особенности платформы

Микропроцессор "Эльбрус" реализует архитектуру с широким командным словом (VLIW) [8, 9] и характеризуется следующими свойствами:

- широкой командой;
- наличием шести универсальных логических устройств, из них четыре устройства предназначены для плавающих вычислений;
- большим регистровым файлом (256 универсальных регистров);
- 64 Кбайт кэш-памяти данных первого уровня (пропускная способность по считыванию — 9,6 Гбайт/с, записи — 4,8 Гбайт/с);
- 256 Кбайт кэш-памяти данных второго уровня (пропускная способность по считыванию — 9,6 Гбайт/с, записи — 4,8 Гбайт/с).

Для дальнейшего изложения остановимся подробнее на устройстве кэш-памяти второго уровня. Кэш-память второго уровня в микропроцессоре "Эльбрус" является четырехассоциативной, ее объем составляет 256 Кбайт, и в отличие от многих других архитектур она является четырехканальной, что потребовало расслоения на четыре банка. Объем каждого банка составляет 64 Кбайт, разбитых на четыре столбца по 256 элементов

(блоков данных) в каждом. Таким образом, размер блока данных (ширина банка) составляет 64 байт. Помимо "Эльбруса" многобанковые кэш-памяти реализованы, например, в архитектурах Sun OpenSPARC Tl [10] и Intel Itanium 2 [11]. Для управления запросами от микропроцессора реализован арбитр, который принимает запросы, коммутирует их в четыре банка и через схему приоритетов выдает в банки кэш-памяти второго уровня. При невозможности немедленного обслуживания запросы буферизуются и формируются очереди к банкам. Размер каждой очереди ограничен запросами. Таким образом, полная пропускная способность кэш-памяти складывается из пропускной способности каждого банка. Вследствие такого устройства кэш-памяти второго уровня в микропроцессоре "Эльбрус" проблемы при исполнении приложений с интенсивной обработкой больших массивов данных возникают не только из-за вытеснения полезных данных из памяти, но и по причине неравномерной загрузки очередей арбитра. Неравномерная загрузка очередей арбитра возникает из-за произвольного определения адресов глобальных структур данных на этапе редактирования связей (линковки).

Блокировки кэш-памяти второго уровня

Во многих научных приложениях данные организованы таким образом, что они хранятся в глобальных массивах большой размерности. Во время компиляции программы на этапе редактирования связей адреса глобальных массивов определяются произвольно. В результате во время исполнения возникают конфликты по доступу в ячейку кэш-памяти. В случае многобанковой кэш-памяти запросы распределяются между банками неравномерно, вследствие чего возникает переполнение очередей запросов в одном или нескольких банках на фоне малоиспользуемых устройств других банков. Подобный эффект особенно хорошо наблюдается при исполнении задач с плавающей точкой из пакетов Spec95, Spec2000 на микропроцессоре "Эльбрус". В ходе исследования были собраны данные о числе внутренних блокировок кэш-памяти второго уровня, связанных с неоптимальной загрузкой банков и работой устройства доступа в память. Эти данные представлены на рис. 1 (см. четвертую сторону обложки).

На рис. 1 через L2$ обозначены внутренние блокировки кэш-памяти второго уровня, связанные с неравномерной загрузкой банков (блокировки 1-го типа), а через MAU — блокировки, возникшие из-за устройства доступа в память (блокировки 2-го типа). Здесь и далее под числом блокировок понимается именно число тактов блокировок на 1000 исполненных инструкций. Для наглядности данные представлены на логарифмической шкале.

Как видно на рисунке, на задачах 101.tomcatv, 146.wave5 наблюдается значительное число блокировок кэш-памяти второго уровня, связанных с неравномерной загрузкой банков, особенно отличается задача 102.swim, в которой число блокировок 1-го типа превышает число исполненных инструкций более чем в 3 раза. Из представленных данных также видно, что основная часть блокировок возникает из-за устройства доступа в память. На задачах из пакета Spec2000 степень потери производительности из-за блокировок 1-го типа не столь заметна, однако ниже будет показано, что преобразование Multibank Padding также полезно и для них.

Оптимизация размещения массивов

В оптимизирующем компиляторе для микропроцессора "Эльбрус" было разработано и реализовано преобразование Multibank Padding, которое определяет более оптимальное с точки зрения использования кэш-памяти взаимное расположение глобальных массивов программы и в соответствии с выбранными параметрами модифицирует начальные адреса глобальных массивов. Данное преобразование учитывает архитектурные особенности целевой платформы, структуру и профиль исполняемой программы, что позволяет создавать эффективный код.

Задача обеспечения равномерной загрузки банков кэш-памяти второго уровня предполагает нахождение такого взаимного расположения глобальных массивов программы, при котором во всех циклах распределение запросов к разным банкам в любой момент времени будет близко к равномерному. При этом взаимное расположение по банкам адресов двух обращений к памяти будет сохраняться только в случае одинакового темпа изменения адреса. Необходимого распределения запросов можно добиться смещением начальных адресов массивов (padding). Для определения оптимальных смещений для каждого из массивов решается задача минимизации следующего функционала:

$$\sum_{\substack{loop\_in\_loops \\ speed\_in\_speeds}} Counter(loop)\Big(\max_{j=0..3}(\#oper_{j,\,speed,\,loop}(\alpha)) + \frac{(NumOfMaxSums(\alpha)-1)}{4}\Big) \to \min_{\alpha}, \qquad (1)$$

где *loops* — все циклы задачи; *speeds* — все темпы обработки массивов (по байтам, по словам, по байтам через 10 и т. д.); *Counter* (*loop*) — счетчик тела цикла по профилю; *j* — номер банка кэш-памяти второго уровня; *oper* — инструкция чтения/записи ($\#oper_j$ — среднее число таких инст-

```
int A[1024], B[1024], C[1024];

for (i=0; i<300; i++)
   A[i]=B[i];

for (i=0; i<200; i++)
{
   B[i]=A[i]+C[i+48]+B[i+16];
   C[i]=B[i+1];
}
for (i=1; i<101; i++)
{
   A[i]=C[i-1];
   B[2*i]=C[2*i-1];
}
```

Рис. 2. Фрагмент программы

рукций в одной итерации цикла, использующих *j*-й банк кэша); *NumOfMaxSums* — число столбцов с одинаковым максимальным значением (1..4); α — взаимное расположение массивов.

Необходимо отметить, что минимизация функционала (1) не обеспечивает равномерную нагрузку банков кэш-памяти во всех циклах программы. Фактически определяются такие параметры, чтобы нагрузка на банки стала равномерной в циклах с большим счетчиком по профилю, это может произойти за счет того, что в циклах с небольшим счетчиком нагрузка станет более неравномерной. Поэтому важным условием эффективной работы преобразования является наличие адекватного профиля исполнения задачи.

Для иллюстрации решения задачи повышения равномерности распределения запросов по банкам кэш-памяти второго уровня воспользуемся примером, представленным на рис. 2.

Для удобства расчетов будем полагать, что изначально адреса глобальных массивов выровнены на размер строки кэш-памяти данных второго уровня, т. е. в случае микропроцессора "Эльбрус" на 256 байт.

Исходя из формальной постановки задачи каждой тройке (массив, цикл, темп обработки массива в цикле) сопоставим вектор из четырех элементов, причем элементом вектора с номером *i* будет число запросов в *i*-й банк кэш-памяти первой итерации цикла. Номер банка вычисляется в соответствии с конкретной реализацией кэш-памяти данных второго уровня, в случае "Эльбруса" — это 6-й и 7-й биты адреса запроса в память. В итоге для примера, представленного на рис. 2, из получившихся векторов можно составить следующую таблицу (рис. 3). Как видно, в данном примере при стандартном определении адресов массивов (последовательно с выравниванием на 256) практически не используются три из четырех банков кэш-памяти, что уменьшает ее пропускную способность и, как следствие, может значительно увеличить время исполнения.

Зафиксируем порядок массивов *A*, *B*, *C* и с учетом счетчика цикла по профилю найдем оптимальное с точки зрения нашей задачи взаимное расположение массивов. Для этого воспользуемся методом покоординатного спуска.

1. Просуммируем столбцы и получим рис. 4.

2. Для каждого вектора, полученного на шаге 1, выберем максимальный элемент и сложим эти максимумы, умножив их на счетчик соответствующего цикла. В итоге получим:

$$tmp = 2 \cdot 300 + 4 \cdot 200 + 2 \cdot 100 + 2 \cdot 100 = 1800.$$

3. Для каждого вектора, соответствующего массиву *B*, подберем такой циклический сдвиг элементов (очевидно, что всего сдвигов три, поскольку сдвиг на 4 возвращает нас в исходное состояние), чтобы характеристика, вычисленная в п. 2, стала минимальной. Получим, что при сдвиге на 1 распределение будет следующим (рис. 5):

$$tmp = 1 \cdot 300 + 2 \cdot 200 + 2 \cdot 100 + 1 \cdot 100 = 1000.$$

	loop1				loop2				loop3_speed1				loop3_speed2			
A	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
B	1	0	0	0	2	1	0	0	0	0	0	0	1	0	0	0
C	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0

Рис. 3. Распределение запросов в память по банкам кэш-памяти второго уровня

loop1				loop2				loop3_speed1				loop3_speed2			
2	0	0	0	4	1	0	1	2	0	0	0	2	0	0	0

Рис. 4. Результат первого шага алгоритма

		loop1				loop2				loop3_speed1				loop3_speed2			
A		1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
B	(1)	0	1	0	0	0	2	1	0	0	0	0	0	0	1	0	0
C		0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0

Рис. 5. Сдвиг векторов массива *B* вправо на 1

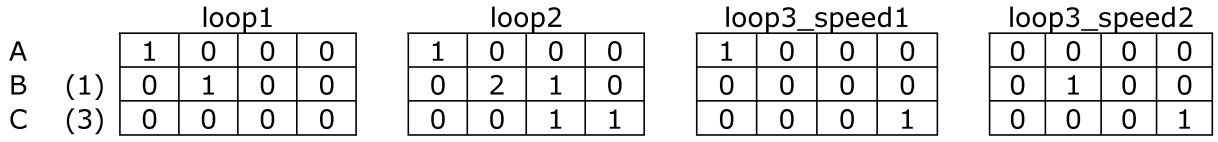


Рис. 6. Сдвиг векторов массива *C* вправо на 3

4. Аналогично поступим с массивом *C* и получим, что минимальное значение получается при сдвиге на 3 (рис. 6):

$$tmp = 1 \cdot 300 + 2 \cdot 200 + 100 + 1 \cdot 100 = 900.$$

Дальнейшие сдвиги ситуацию не улучшают, следовательно, найден координатно-локальный минимум.

На рис. 7 представлен результат применения преобразования к исходному примеру. Смещение определяется как ширина банка (64 байт), умноженное на сдвиг.

В общем случае в функционале (1) появляется дополнительное слагаемое ($NumOfMaxSums(\alpha)$—1)/4), необходимое для того, чтобы при одинаковой максимальной для разных α загрузке минимум функционала достигался при более равномерной загрузке банков.

Точное решение задачи минимизации функционала (1) предполагает исследование всех возможных сдвигов для всех возможных порядков массивов. Реализация такого анализа в оптимизирующем компиляторе невозможна в силу технологических требований к времени и используемым ресурсам. Предложенный алгоритм покоординатного спуска не гарантирует нахождения точного решения задачи, однако позволяет быстро найти локальный минимум и его использование позволяет получить более эффективный код.

```
int A[1024];
char padding_B[64];
B[1024];
char padding_C[3*64];
C[1024];

for (i=0; i<300; i++)
   A[i]=B[i];

for (i=0; i<200; i++)
{
   B[i]=A[i]+C[i+48]+B[i+16];
   C[i]=B[i+1];
}

for (i=1; i<101; i++)
{
   A[i]=C[i-1];
   B[2*i]=C[2*i-1];
}
```

Рис. 7. Пример после преобразования Multibank Padding

Результаты оптимизации размещения массивов

В результате применения оптимизации Multibank Padding получаем изменения числа блокировок, представленные на рис. 8 (см. четвертую сторону обложки).

На рис. 8 через L2$_orig обозначены блокировки 1-го типа, а через MAU_orig — блокировки 2-го типа на исходных неоптимизированных задачах. Через L2$_pad, MAU_pad обозначены блокировки кэш-памяти второго уровня, аналогичные блокировкам L2$_orig, MAU_orig соответственно на задачах после применения оптимизации Multibank Padding. Значения приведены для блокировок L2$_pad и MAU_pad. Для наглядности данные представлены на логарифмической шкале.

Как видно из представленных данных, благодаря преобразованию удалось значительно снизить число внутренних блокировок 1-го типа на задачах 101.tomcatv, 102.swim, 146.wave5; кроме того, незначительно снизились те же блокировки на задачах пакета Spec2000. В то же время увеличилось число блокировок, возникающих из-за работы устройства доступа в память, что объясняется взаимосвязью блокировок этих двух типов и тем фактом, что блокировки 1-го типа маскируют блокировки 2-го типа. Однако суммарное число блокировок 1-го и 2-го типов уменьшилось, вследствие чего эффект от преобразования оказался в основном положительным.

В результате применения преобразования на представленных задачах пакета Spec95 число блокировок 1-го типа в среднем[1] снижается на 59 %, на задачах пакета Spec2000 — на 13 %. В то же время число блокировок 2-го типа на задачах Spec95 в среднем увеличивается на 11 %, а на задачах пакета Spec2000 в среднем снижается на 4 %. На рис. 9 представлены отношения времен исполнения исследуемых программ без использования оптимизации к временам исполнения с использованием оптимизации, то есть фактически представлен коэффициент ускорения (если значение больше 1), замедления (если значение меньше 1) задачи. Среднее ускорение задач пакета Spec95 составило 9 %, а на пакете Spec2000 — 5 %. Как видно на рис. 9, эти результаты хорошо коррелируют с ранее представленными данными об

[1] Здесь и далее под средним понимается среднее геометрическое.

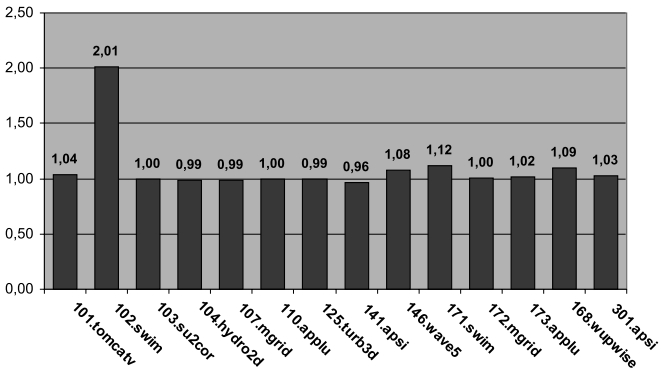


Рис. 9. Ускорение/замедление задач

изменении числа блокировок 1-го типа, что свидетельствует об эффективности описанного преобразования.

Заключение

В статье описано влияние блокировок многобанковой кэш-памяти на производительность, предложен метод снижения числа таких блокировок, основанный на сборе и анализе информации о среднем числе обращений к банкам кэш-памяти в циклах задачи. Приведена математическая формализация задачи и быстрый алгоритм нахождения локально-оптимального решения. Описанный метод был реализован в оптимизирующем компиляторе для микропроцессора "Эльбрус" и доказал свою эффективность на задачах с плавающей точкой из пакетов Spec95 и Spec2000.

Список литературы

1. **Smith A. J.** Cache memories // ACM Computing Surveys. September 1982. Vol. 14, N 3. P. 473—530.
2. **Hennessy J., Patterson D.** Computer Architecture: A Quantitative Approach. Morgan Kauffman, 1990.
3. **Столярский Е. З., Шилов В. В.** Организация и работа кэш-памяти. // Информационные технологии. 2000. № 7. С. 2—8.
4. **Rivera G., Tseng C.-W.** Data Transformations for Eliminating Conflict Misses // Proc. of the 1998 ACM SIGPLAN Conference on Programming Languages Design and Implementation. May 1998. P. 38—49.
5. **Babayan B.** E2K Technology and Implementation // Proc. of the 6th International Conf. Euro-Par 2000 — Parallel Processing. January 2000. Vol. 1900/2000. P. 18—21.
6. **Dieffendorf K.** The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join × 86/IA-64 Melee. Microprocessor Report. February 1999. Vol. 13. N 2. P. 1—7.
7. **Standard** Performance Evaluation Corporation. http://www.spec.org.
8. **Ellis J. R.** Bulldog. A Compiler for VLIW Architectures. Cambridge: MIT Press, 1986.
9. **Colwell R. P., Nix R. P., O'Donnel J. J., Papworth D. B., Rodman P. K.** A VLIW Architecture for a Trace Scheduling Compiler // Proc. of the 2nd International Conf. on Architectural Support for Programming Languages and Operating Systems. October 1987. P. 180—192.
10. **OpenSPARC** T1 Micro architecture Specification. http://opensparc-tl.sunsource.net/
11. **Intel** Itanuim 2 Processor Reference Manual for Software Development and Optimization. June 2002.

УДК 004.076; 004.272.26

Ю. Н. Ильин,
Московский физико-технический институт
(государственный университет)

Анализ эффективности энергопотребления микропроцессора с упорядоченным выполнением команд в многопотоковом режиме исполнения

Одной из важнейших задач при проектировании современных микропроцессоров является сокращение их энергопотребления. В работе рассмотрены основные составляющие энергопотребления, критерии эффективности энергосберегающих микроархитектурных решений. Применение параллелизма на уровне потоков исполнения является эффективным средством повышения производительности и полезной загрузки устройств микропроцессора. Предлагается модель системы оценки производительности и потребляемой мощности для этого способа организации вычислений, а также проводится анализ эффективности энергопотребления микропроцессора с упорядоченным исполнением команд в многопотоковом режиме.

Введение

До последнего времени (с выходом каждого нового поколения) микропроцессоры потребляли все большую и большую мощность. Pentium 4 Cedarmill позволяет достичь в ~ 8 раз большей скалярной производительности (2.5 × IPC на 3 × frequency), но потребляет в 38 раз большую мощность по сравнению с микропроцессором i486, выпущенным на той же технологии [1]. Дело в том, что рост производительности микропроцессоров достигался исключительно экстенсивным способом: за счет увеличения тактовой частоты и числа транзисторов на кристалле. Согласно эмпирическому закону Мура число транзисторов на кристалле удваивается каждые полтора года. Увеличение степени интеграции полупроводниковых элементов на кристалле позволяет разработчикам микропроцессора размещать больше оборудования на кристалле, увеличивая как число логических устройств, так и объемы памяти. К сожалению, экстенсивный подход к проектированию микропроцессоров привел, наряду с ростом производительности, к катастрофическому падению эффективности вычислений и росту потребляе-

мой микропроцессором мощности. Мощность, рассеиваемая некоторыми современными микропроцессорами (например, Pentium 4 Prescott), уже превысила 100 Вт и стала препятствием на пути дальнейшего повышения их производительности. Рост потребляемой мощности осложняет также создание микропроцессоров, предназначенных для переносных устройств, в которых время автономной работы ограничивается емкостью аккумулятора, и поэтому энергоэффективность работы микропроцессора, наряду с производительностью, становится ключевым фактором для его коммерческого успеха.

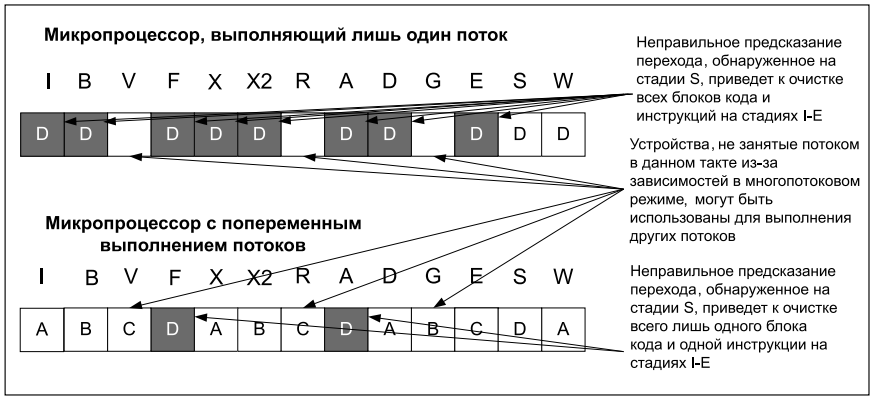


Рис. 1. Конвейер микропроцессора и возможные потери при однопотоковом и многопотоковом исполнении команд

Для снижения энергопотребления микропроцессора могут применяться различные способы: отключение неработающих устройств или их частей; реализация некритичных по времени цепей с помощью минимальных по размеру транзисторов; минимизация переключений длинных шин; создание энергоэффективных устройств кэш-памяти [2, 3]. Большинство из этих оптимизаций являются низкоуровневыми. Поэтому возможность и успешность их применения во многом зависят от выбранной микроархитектуры процессора. Таким образом, учет и оптимизация энергопотребления еще на стадии проектирования становятся важными как при разработке процессоров для переносных устройств, так и при создании высокопроизводительных микропроцессоров.

Одним из главных решений, принимаемых при проектировании микроархитектуры микропроцессора, является выбор способа организации вычислений. Использование параллелизма на уровне потоков исполнения как способа организации вычислений оказалось очень эффективным средством повышения полезной загрузки устройств микропроцессора как для сложных микропроцессоров, использующих неупорядоченное выполнение команд (Pentium 4, Alpha 21164) [3], так и для сравнительно простых микропроцессоров с упорядоченным выполнением команд (VIA C3 Ezra) [4]. При многопотоковом исполнении в конвейере микропроцессора одновременно присутствуют команды сразу нескольких, частично или полностью независимых потоков (программ). Когда один из потоков не может использовать ресурсы микропроцессора из-за зависимостей между инструкциями или промаха в кэш-память, эти ресурсы могут быть задействованы другим потоком. Обычно потоки выбираются по кругу, из готовых к выполнению потоков, это позволяет достичь их чередования и сокращения возможных потерь, вызванных спекулятивным выполнением инструкций (рис. 1).

Однако повышение загрузки устройств процессора не обязательно приводит к повышению энергоэффективности вычислений. Исследования, проведенные для микропроцессора с неупорядоченным выполнением инструкций, показывают значительное возрастание энергоэффективности вычислений с увеличением числа потоков. Однако микропроцессор с упорядоченным выполнением команд сильно отличается от высокопроизводительного суперскаляра, рассмотренного в работе [6], поэтому для него анализ эффективности энергопотребления в многопотоковом режиме должен быть проведен отдельно. Проведению этого исследования и посвящена данная работа.

Подходы к повышению энергоэффективности микроархитектурных решений

Принятие микроархитектурных решений невозможно без учета их влияния на производительность микропроцессора, его площадь и потребляемую им мощность. К сожалению, в большинстве случаев решения, увеличивающие производительность, ведут также к росту площади и потребляемой мощности. Поэтому оптимальное решение всегда является компромиссом между этими характеристиками микропроцессора. Для оценки энергоэффективности микроархитектурных решений применяются различные критерии.

Общепризнанным подходом к повышению энергоэффективности является **максимизация производительности** на 1 Вт энергозатрат [6, 7]. Микроархитектурные решения, удовлетворяющие ему, ведут к минимизации затрат энергии на выполнение задачи, что делает его применение оправданным при создании микропроцессоров для переносных устройств. В автономных устройствах на первый план выходит количество работы,

которое может быть выполнено устройством без подзарядки:

$$E_{\text{Task}} = P_{\text{Task}} T_{\text{Task}} = \left(\frac{P_{\text{Task}}}{\Pi_{\text{Task}}}\right) N_{\text{Instr}},$$

где E_{Task} — затраты энергии; P_{Task} — средняя мощность; T_{Task} — время выполнения; П — производительность; N_{Instr} — число инструкций в задаче.

Другим часто используемым подходом к повышению энергоэффективности является **максимизация производительности при заданной рассеиваемой мощности** [7]. Этот подход наиболее применим при создании систем, ориентированных на достижение высокой производительности. Он основан на идее, что микроархитектурная оптимизация, увеличивающая производительность или сохраняющая мощность, должна быть реализована тогда и только тогда, когда она дает результаты лучшие, чем простое изменение частоты/напряжения.

Мощность, потребляемая микропроцессором, включает две составляющие: статическую и динамическую:

$$P_{\text{total}} = P_{\text{stat}} + P_{\text{dyn}}.$$

Статическая мощность P_{stat} — возникает в основном вследствие токов утечки и потребляется все время, пока на микропроцессор подано питание. Таким образом, энергия, рассеиваемая током утечки, тем меньше, чем меньше время работы программы. В данной работе статическая мощность рассматриваться не будет.

Динамическая мощность P_{dyn} — потребляется в процессе переключения цепей внутри микропроцессора. Для КМОП-устройств, использующих полный перепад, в рабочей точке с частотой F_0 и напряжением V_0 можно приближенно считать, что

$$P_{\text{dyn}} \approx \alpha C_0 V_0^2 F_0,$$

где α — коэффициент активности устройства (доля переключающихся цепей устройства), C_0 — эффективная емкость цепей устройства. Для конкретного КМОП-устройства в первом приближении можно считать, что рабочая частота пропорциональна напряжению с коэффициентом пропорциональности K_f:

$$F_0 \approx K_f V_0.$$

Производительность Π_0 при данных рабочих условиях описывается формулой

$$\Pi_0 = IPC_0 \cdot F_0,$$

где IPC_0 — число инструкций, выполняемых за такт процессором при частоте F_0. Следовательно, получаем:

$$\begin{cases} P_{dyn} = \alpha C_0 V_0^2 F_0 = a C_0 K_f V_0^3; \\ \Pi_0 = IPC_0 F_0 = IPC_0 K_f V_0. \end{cases}$$

Приведенные выше выражения показывают, как изменяются рассеиваемая мощность и производительность при масштабировании частоты/напряжения. С помощью дифференцирования этих формул можно показать, что простое увеличение напряжения на 1 % ведет к увеличения производительности на 1 %, при этом потребляемая мощность увеличится на 3 %. Таким образом, согласно этому критерию микроархитектурная оптимизация, которая ведет к увеличению потребляемой мощности более чем на 3 % при росте производительности на 1 %, должна быть отвергнута [6].

В данной работе будут использованы оба критерия для анализа эффективности энергопотребления микропроцессора в многопотоковом режиме исполнения.

Модель многопотокового процессора и система оценки производительности

Для анализа эффективности энергопотребления микропроцессора в многопотоковом режиме исполнения была создана система оценки производительности, которая описывает микроархитектуру микропроцессора VIA C3 Ezra [8], модифицированную для поддержки одновременного выполнения нескольких потоков. Микроархитектурная модель включает в себя: конвейер, подсистему памяти, кэши инструкций и данных, общий кэш второго уровня (кэш-память L2), механизм предсказания переходов. Модель позволяет легко добавить стадию в любом месте конвейера, изменить размеры и ассоциативность кэшей, изменить число одновременно моделируемых потоков. В табл. 1 приведена конфигурация микропроцессора, а на рис. 2 — диаграмма его конвейера.

Таблица 1

Параметры микропроцессора

Параметр	Значение
Кэш-инструкций (ICache)	64 Кбайт, 4 колонки, 32-байтные строки, 1 запрос/такт
Кэш-данных (DCache)	64 Кбайт, 4 колонки, 32-байтные строки, 1 запрос/такт
Кэш L2 (L2 Cache)	256 Кбайт, 8 колонок, 32-байтные строки, 1 запрос за 2 такта
ITLB, DTLB	128 адресов, 8 колонок
Задержка загрузки данных (до CPU)	из кэша L2 — 12 тактов, из памяти — 132 такта
Задержка получения трансляции адреса при промахе в ITLB (DTLB)	20 тактов
Конвейер	13 стадий, неблокируемый
Потери при неправильном предсказании перехода	12 тактов
Предсказатель переходов	Комбинированный (бимодальный 1 Кбайт, GShare 2 Кбайт, селектор 2 Кбайт)
Размеры очередей (для каждого потока)	FQ — 4 блока кода по 16 байт, FIQ — 4 декодированные команды, IQ — 12 микроинструкций

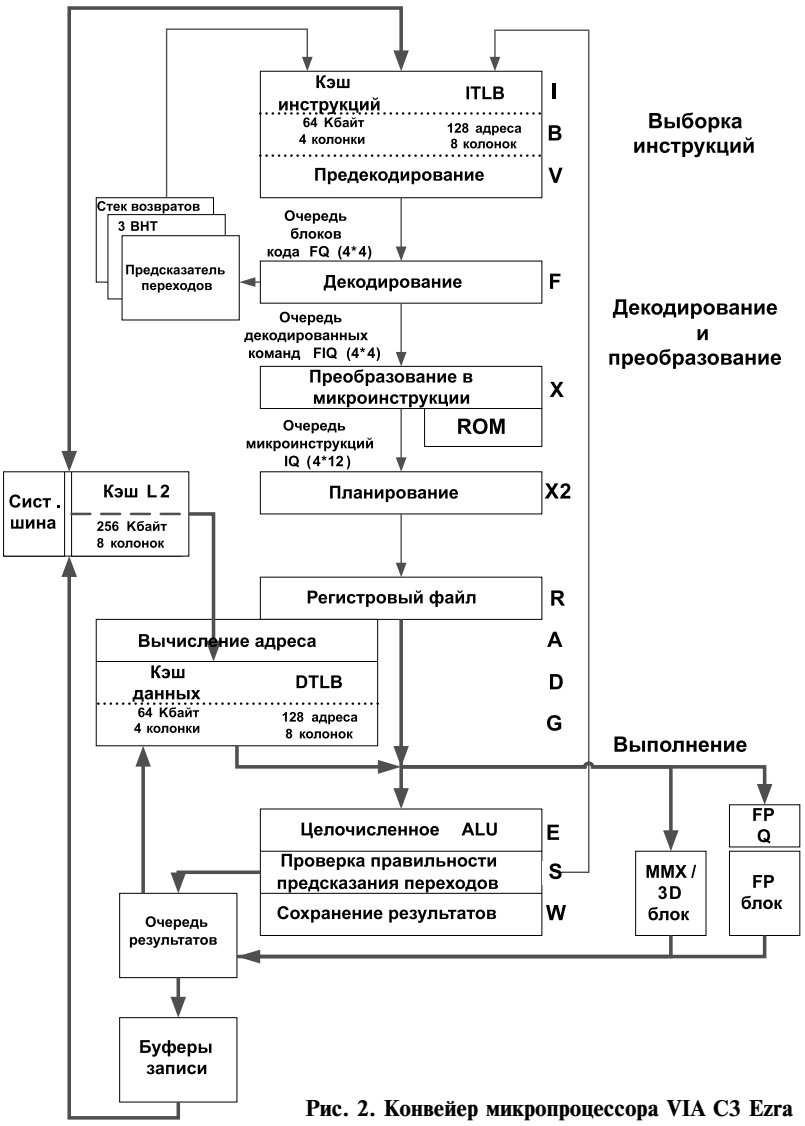


Рис. 2. Конвейер микропроцессора VIA C3 Ezra

Для сокращения времени моделирования изучаемых решений вместо интерпретации тестовой программы при каждом запуске теста использовались трассы выполняемых тестом инструкций. Этот подход позволяет ускорить моделирование, так как все команды теста декодируются и интерпретируются всего лишь один раз — в процессе создания трассы выполняемых инструкций. Для создания трассы был использован открытый симулятор архитектуры x86 архитектуры — Bochs. Для каждой выполняемой инструкции в трассе сохраняется вся необходимая информация о ней: ее адрес; является ли она командой перехода или нет; физический адрес выполняемой инструкции; адреса обращений в память и др. Эта трасса загружается системой оценки производительности и используется для моделирования поведения процессора. Моделирование поведения микропроцессора в случае ошибочного предсказания переходов проводилось с использованием словаря статических инструкций (из программы выделяются все встречающиеся в ней инструкции и помещаются в словарь, в который можно обращаться, используя адрес инструкции).

Для создания трасс использовался пакет тестов SPEC INT 2000. Из каждого теста, входящего в этот пакет, был выделен фрагмент длиной в 300 миллионов инструкций. Этот фрагмент сохранялся после выполнения инициализационной части теста. Использование такого подхода позволяет сократить время, затрачиваемое на моделирование изучаемых решений. Табл. 2 содержит некоторую информацию по тестам, используемым при снятии трасс для системы оценки производительности.

На рис. 3 и 4 приведены данные по производительности и эффективности использования кэш-памяти данных на различных тестах из пакета SPEC INT 2000 для микропроцессора VIA C3 Ezra в однопотоковом режиме. Тест 181.mcf сильно отличается от остальных тестов крайне низкой производительностью (*IPC* ~ 0,05) и очень большим числом промахов в кэш-память данных ~ 26 %. Этот тест обращается к данным таким образом, что они практически не переиспользуются, что и приводит к снижению эффективности кэш-памяти данных, а также значительному росту энергопотребления за счет частых обращений в кэш-память L2 и в системную память. Ввиду столь особого поведения этот тест исключен нами из дальнейшего рассмотрения, а на рис. 3 и 4 приведены результаты усреднения с учетом этого теста (среднее) и без него (среднее без 181.mcf).

На рис. 3 видно, что архитектурная производительность микропроцессора в однопотоковом ре-

Таблица 2

Сводная информация об использованных тестах

Название	Набор входных данных	Выполнено перед сохранением трассы, млрд инструкций
164.gzip	graphic	68,1
175.vpr	place	2,1
176.gcc	166.i	15,0
181.mcf	inp.in	43,5
186.crafty	crafty.in	74,7
197.parser	ref.in	83,1
254.gap	ref.in	79,8
255.vortex	lendian1.raw	58,2
256.bzip2	inp.program	51,3
300.twolf	ref	130,0

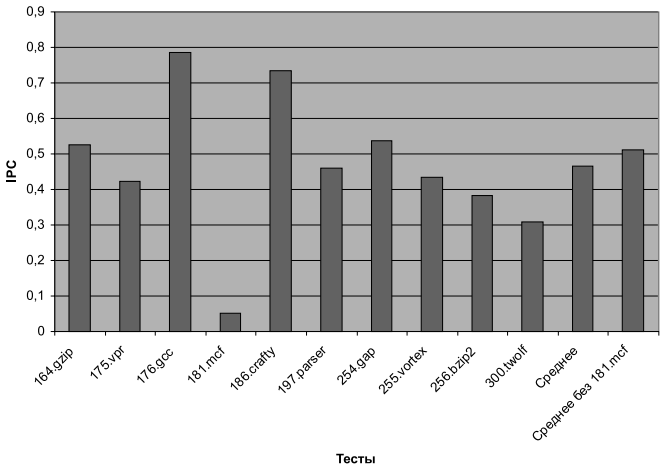


Рис. 3. Производительность (IPC)

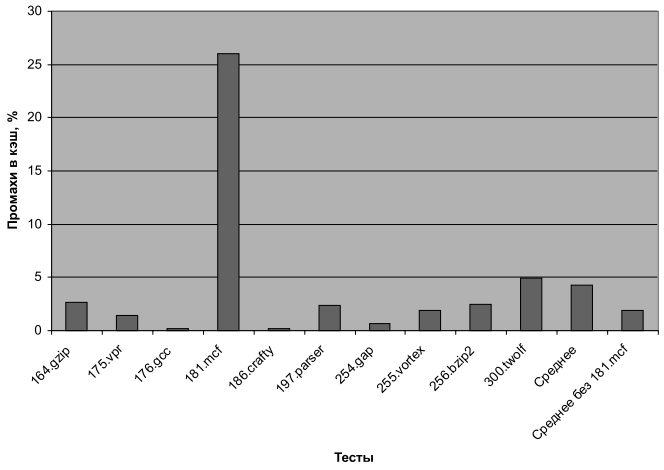


Рис. 4. Доля промахов в кэш-данных

жиме сильно отличается на различных тестах. Это усложняет сравнение последовательного исполнения тестов с их одновременным исполнением. При одновременном выполнении поток с более быстрым тестом исполняет за такт больше инструкций, чем поток с медленным тестом, и поэтому вносит больший относительный вклад как в производительность, так и в энергопотребление микропроцессора, если заканчивать моделирование по окончании самого медленного потока. Для проведения справедливого сравнения необходимо, чтобы все потоки многопотокового теста выполнили одинаковое число инструкций. Поэтому в многопотоковом режиме все потоки начинают работать одновременно. После выполнения 300 миллионов инструкций поток приостанавливается и освобождает занятые им ресурсы, а оставшиеся потоки, еще не достигшие этой границы, продолжают исполнение. Когда все потоки достигнут 300 млн инструкций и остановятся, многопотоковый тест считается законченным.

Методика оценки энергопотребления

Для проведения оценок энергопотребления была использована модель энергопотребления, встроенная в систему оценки производительности, что позволило моделировать как полезное, так и бесполезное (основанное на неправильном спекулятивном выполнении) использование всех ресурсов микропроцессора.

Модель энергопотребления, примененная в этом исследовании, основана на площади и аналогична приведенной в работе [1]. В этой модели микропроцессор разделен на несколько высокоуровневых блоков. Полученные с помощью системы оценки производительности и активности каждого из блоков используются затем при вычислении энергопотребления всего кристалла. Системой моделировались следующие основные блоки микропроцессора: кэш инструкций L1, кэш данных L1, общий кэш L2, блок выборки команд, очереди выбранного кода и декодированных инструкций (FQ, FIQ, IQ), блок предсказания переходов (BPU), TLB инструкций, TLB данных, ALU, регистровый файл.

Энергопотребление всего микропроцессора вычисляется, как сумма энергопотребления всех его блоков. А энергопотребление каждого блока находится по формуле:

$$E_{Unit} = A_{Unit} S_{Unit} \varepsilon,$$

где A_{Unit} — активность блока микропроцессора; S_{Unit} — относительная площадь блока; ε — плотность энергопотребления, определяемая типом логики (статическая логика, память, динамическая логика), используемым в этом блоке.

Активность каждого из блоков определяется подсчетом для этого блока числа микроархитектурных событий, порожденных на конкретном тесте или программе. Например, активность регистрового файла определяется подсчетом числа операций записи и чтения, выполненных им. Поскольку активности являются высокоуровневыми, то они не зависят от конкретных данных, обрабатываемых устройством. Модель энергопотребления включает в себя 35 активностей. Каждому блоку присвоена относительная площадь, которая является оценкой его размера при заданных микроархитектурных параметрах. Эти относительные площади были оценены путем масштабирования относительных площадей соответствующих блоков нескольких предыдущих микропроцессоров (PowerPC, MIPS, Intel). Помимо площади блока был учтен также тип его логики, так как различные типы логики потребляют различную мощность. Данные о плотности энергопотребления ε и используемом в блоке типе логики были получены путем консультаций с инженерами, которые занимались реализацией соответствующих блоков микропроцессоров.

Энергопотребление многопотокового микропроцессора

Для сравнения однопотокового и многопотокового выполнения тестов мы будем использовать в этой главе следующие характеристики: производительность (*IPC*), эффективность энергопотребления E/N_{Instr} (отношение затраченной энергии E к числу выполненных полезных инструкций N_{Instr}), мощность (среднюю мощность, потребляемую процессором при запуске теста), а также эффективность достижения производительности $E/(N_{Instr} \cdot IPC)$ (отношение эффективности энергопотребления E/N_{Instr} к достигаемой производительности *IPC*).

На рис. 5 (см. третью сторону обложки) представлены результаты однопотокового выполнения тестов из пакета SPEC INT 2000. Столбцы E/N_{Instr} и $E/(N_{Instr} \cdot IPC)$ нормированы к базовому значению (в каждом случае наименьшее однопотоковое значение). Это сделано по двум причинам: во-первых, модель энергопотребления не строилась для расчета абсолютного энергопотребления или рассеиваемой мощности, поэтому позволяет сосредоточиться на относительных значениях; во-вторых, это позволяет увидеть относительные отклонения различных тестов от средних значений на этом же графике. Для нормировки столбцов *Ideal* E/N_{Instr}, *Ideal* $E/(N_{Instr} \cdot IPC)$ использованы те же базовые значения, что и для столбцов E/N_{Instr}, $E/(N_{Instr} \cdot IPC)$ соответственно.

Как можно видеть на рис. 5, для тестов наблюдается сильный разброс практически по всем метрикам. Производительность и потребляемая мощность выглядят положительно коррелированными: чем больше процессор загружен, тем больше потребляемая им мощность. Однако производительность *IPC* и энергия на выполнение полезной инструкции E/N_{Instr} выглядят отрицательно коррелированными; чем меньше неиспользуемых ресурсов, тем меньше их расходуется напрасно, тем более эффективным становится процессор.

Столбец *Ideal* E/N_{Instr} на рис. 5 (см. третью сторону обложки) показывает эффективность энергопотребления для каждого теста при идеальном предсказании переходов, позволяя нам оценить потери, вызванные неправильной спекулятивностью (путем сравнения со столбцом E/N_{Instr}). Современные процессоры используют спекулятивное выполнение инструкций (например, предсказание переходов и выборку кода по предсказанному направлению перехода) для повышения производительности. Ошибки спекулятивного исполнения приводят к выполнению инструкций, результаты которых не будут использованы, и, тем самым, к снижению эффективности вычислений. Как можно видеть, потери от неправильной спекулятивности невелики. Лишь на 164.gzip они достигают 7 %. Этот результат может быть объяснен упорядоченным исполнением инструкций в исследуемом микропроцессоре и сильной разреженностью конвейера, вызванной малым *IPC*.

Как можно видеть на рис. 6 (см. третью сторону обложки), изменение числа одновременно выполняемых потоков не оказывает заметного влияния на эффективность работы микропроцессора (E/N_{Instr}). Этот результат сильно отличается от наблюдаемого в работе [7] значительного возрастания энергоэффективности вычислений в микропроцессоре с неупорядоченным выполнением инструкций при увеличении числа потоков. Такое расхождение может быть объяснено малой спекулятивностью выполнения команд в VIA C3 Ezra (упорядоченное выполнение команд), а также значительным числом конфликтов между потоками за разделяемую кэш-память. Увеличение числа промахов в кэш-память, вызванное этими конфликтами и наблюдаемое на рис. 7 (см. третью сторону обложки), приводит к дополнительным затратам энергии на их обработку. Это не только нейтрализует выигрыш от снижения спекулятивности выполнения команд, но даже приводит к незначительному росту затрат энергии на выполнение полезной инструкции. Такой результат подтверждается при моделировании многопотокового выполнения с кэш-памятями инструкций и данных размером 64 Кбайт, выделенными каждому потоку.

Хотя увеличение числа потоков не позволяет минимизировать энергию, требуемую на выполнение задачи, оно приводит к более эффективному достижению производительности, что подтверждается столбцом $E/(N_{Instr} \cdot IPC)$ на рис. 6. Этот факт позволяет применять параллелизм на уровне потоков исполнения для повышения энергоэффективности вычислений как в высокопроизводительных серверных процессорах, так и в микропроцессорах, предназначенных для переносных устройств. Такой результат может быть получен совместным применением многопотокового исполнения вместе с масштабированием частоты/напряжения (рис. 8)

На рис. 8 представлены точки, соответствующие производительности и потребляемой мощности микропроцессора, работающего в многопотоковом режиме исполнения. Увеличение числа потоков с 1 до 4 ведет к соответствующему, почти линейному, росту рассеиваемой мощности. Через точку, отвечающую работе в четырехпотоковом режиме, проведена теоретическая кривая (масштабирование частоты/напряжения), показывающая производительность и потребляемую мощность, получаемые масштабированием частоты/напряжения.

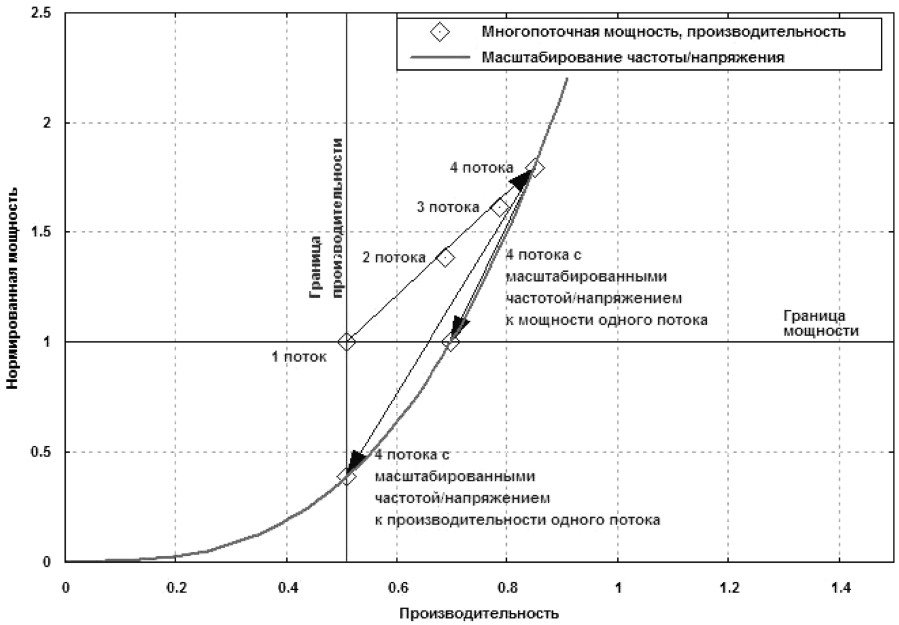


Рис. 8. Использование масштабирования частоты/напряжения для снижения потребляемой микропроцессором мощности

В серверных микропроцессорах производительность ограничивается допустимой рассеиваемой мощностью. Точка пересечения кривой масштабирования частоты/напряжения с границей однопотоковой мощности показывает возможность увеличения производительности на 40 % при той же потребляемой мощности.

В микропроцессорах для переносных устройств важно сокращение энергопотребления при сохранении требуемого уровня производительности (например, на задачах декодирования видео). Точка пересечения кривой масштабирования частоты/напряжения с границей, отвечающей однопотоковой производительности, показывает возможность обеспечения однопотоковой производительности при сокращении рассеиваемой мощности почти в 2,5 раза.

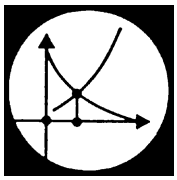
В обоих случаях многопотоковое исполнение позволило поднять архитектурную производительность (*IPC*) микропроцессора практически при сохранении энергоэффективности вычислений. При фиксированной частоте это ведет к соответствующему росту производительности процессора. Наличие резерва производительности позволяет использовать масштабирование частоты/напряжения, чтобы снизить производительность или потребляемую мощность к желаемому уровню и при этом значительно повысить энергоэффективность микропроцессора.

Заключение

Из полученных результатов можно сделать вывод, что в микропроцессоре с упорядоченным выполнением команд применение параллелизма на уровне потоков исполнения является перспективным средством повышения энергоэффективности вычислений. И хотя само по себе многопотоковое исполнение не ведет к сокращению потребления энергии на задачу, оно позволяет добиться увеличения производительности гораздо более эффективным способом по сравнению с масштабированием частоты/напряжения. Это позволяет использовать уменьшение частоты/напряжения, чтобы снизить потребляемую мощность к желаемому уровню и при этом достичь значительного повышения энергоэффективности вычислений. В мобильных микропроцессорах таким образом можно добиться сокращения в 2,5 раза потребляемой мощности и затрачиваемой энергии на задачу при сохранении заданной производительности. В серверных процессорах можно достигнуть увеличения производительности на 40 % (в 1,4 раза) при сохранении заданной рассеиваемой мощности, а следовательно, добиться увеличения энергоэффективности на 40 %.

Список литературы

1. **Grochowski E., Annavaram M.** Energy per Instruction Trends in Intel Microprocessors // Technology @ Intel Magazine. March 2006.
2. **Ильин Ю. Н.** Выборочное использование многоколоночной кэш-памяти, как кэш-памяти прямого отображения для сокращения энергопотребления // Труды XLVII Научной Конференции МФТИ "Современные проблемы фундаментальных и прикладных наук". Ч. I. Радиотехника и Кибернетика. 26—27 ноября 2004 г. С. 36—37.
3. **Супрун А.** Снижение потребляемой мощности частично-ассоциативной кэш-памяти // Труды XLVII Научной Конференции МФТИ "Современные проблемы фундаментальных и прикладных наук". Ч. I. Радиотехника и Кибернетика. 26—27 ноября 2004 г. С. 42—43.
4. **Tulsen D., Eggers S., Levy H.** Simultaneous Multithreading: Maximizing On-Chip Parallelism // Proc. 22nd Annual International Symposium on Computer Architecture, 22—24 June 1995. P. 392—403.
5. **Seng J., Tullsen D., Cai G.** Power-Sensitive Multithreaded Architecture / Published in Proceedings of the 2000 International Conference on Computer Design. 17—20 Sept. 2000. P. 199—206.
6. **Gochman S., Ronen R., etc.** The Intel® Pentium® M Processor: Microarchitecture and Performance // Technology @ Intel Magazine. May 2003.
7. **Burd T., Brodersen R.** Energy efficient CMOS microprocessor design // Proceedings of the 28th Annual Hawaii International Conference on System Sciences 1995. P. 288—297.
8. **VIA C3TM** Ezra Processor Datasheet. Version 1.10.



УДК 004.81

В. М. Картак., канд. физ.-мат. наук, доц.,
Э. А. Мухачева, д-р техн. наук, проф.,
Л. И. Васильева, канд. техн. наук,
Уфимский государственный авиационный технический университет,
А. А. Петунин, канд. техн. наук, доц.,
Уральский государственный технический университет, г. Екатеринбург

Задача размещения ортогональных многоугольников: модели и алгоритм покоординатной упаковки

Рассматривается задача размещения ортогональных многоугольников. Она встречается как индивидуальная проблема, так и в качестве аппроксимационной для многих задач размещения геометрических объектов сложной формы. Рассматривается постановка задачи в условиях массового производства. Предложен метод ее решения, основанный на покоординатной упаковке ориентированных многоугольников.

Введение

При решении задач раскроя-упаковки (Cutting&Packing, C&P) возникает широкий ряд проблем. Общим для этих задач является наличие двух групп объектов, между элементами устанавливается и оценивается соответствие. Различие задач определяется, прежде всего, фактором геометрии: задачи прямолинейного раскроя-упаковки и так называемые задачи фигурного раскроя, касающиеся размещения геометрических объектов сложных форм. Оба подкласса задач принадлежат к NP-трудным задачам, и точные методы полиномиальной сложности для их решения неизвестны. Тем не менее, подкласс задач прямолинейной упаковки быстро развивается и к настоящему времени известны многие эффективные алгоритмы локального поиска и оптимизационные алгоритмы переборного типа. Для задач фигурного раскроя-упаковки возникают дополнительные трудности, связанные с трудоемкостью формализации условий взаимного непересечения объектов и условий их размещения в заданных областях C&P [1]. Поэтому среди таких задач перспективно выделение индивидуальных классов более простых задач, для решения которых применяются модифицированные методы прямолинейного C&P.

На практике часто возникает необходимость решения задачи C&P заготовок, представляющих собой объединение конечного числа прямоугольников, чаще всего связанных друг с другом (например, развертка коробки). Впервые подобные фигуры, названные "гофрами", рассматривались Корницкой М. Н. и Липовецким А. И. [2]. Гофром в [2] назван многоугольник, ребра которого можно разбить на две ветви, составленные из ступенчатых ломаных с вершинами, координаты которых монотонно возрастают по оси абсцисс и монотонно убывают по оси ординат. В связи с трудностями, возникающими в процессе решения задач фигурного раскроя, этими авторами был предложен подход, связанный с упрощением формы раскраиваемых заготовок путем их аппроксимации фигурами определенных классов и решением указанной задачи для этих классов фигур. Предложенный аппроксимационный подход был разработан для проектирования раскройных карт в единичном производстве путем обобщения эффективных алгоритмов негильотинного раскроя на подкласс фигур, названных гофрами.

В нашем случае рассматривается более широкий класс геометрических объектов сложных форм — геометрические фигуры, представляющие собой произвольные совокупности прямоугольников.

1. Постановка задачи размещения ортогональных многоугольников

Ортогональным многоугольником (*ОМ*) называется многоугольник, у которого все стороны горизонтальны или вертикальны [3]. Ортогональный многоугольник можно представить как плоскую фигуру, состоящую из конечного числа неперекрывающихся прямоугольников, ребра которых параллельны осям координат, с фиксированным положением относительно друг друга. На рис. 1 приведены примеры ортогональных многоугольников.

Ортогональный многоугольник является частным случаем геометрических объектов сложной формы и относится к фигурным элементам с пря-

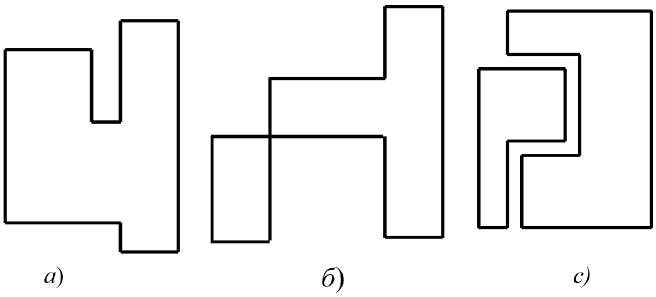


Рис. 1. Примеры ортогональных многоугольников

молинейными ортогональными составляющими. В связи с прямоугольной структурой ортогональных многоугольников рассмотрение таких заготовок не вызывает трудностей, связанных с представлением исходной информации и моделированием условий взаимного неперекрытия.

Задача двумерной упаковки ортогональных многоугольников заключается в следующем: имеются листы прямоугольной формы заданной ширины W и длины L и m видов ортогональных многоугольников (заготовок) заданной потребности b_i, $i = 1, ..., m$ (комплектности). Требуется разместить ортогональные многоугольники на однонаправленных с ними листах таким образом, чтобы суммарное число затраченных листов было минимально.

Очевидно, что любой ОМ может быть задан набором прямоугольников.

Начальной точкой каждого прямоугольника, составляющего ортогональный многоугольник, назовем его левую нижнюю вершину; начальной точкой ортогонального многоугольника — левую нижнюю вершину описывающего его прямоугольника.

Ортогональный многоугольник зададим:

- числом k составляющих его прямоугольников;
- размерами каждого прямоугольника — длиной и шириной (p_i, q_i), $i = 1, ..., k$;
- координатами (x_i, y_i), $i = 1, ..., k$, их начальной точки относительно начальной точки ОМ.

На рис. 2 представлен ортогональный многоугольник, в котором: (0,0) — координаты его начальной точки, (0,1), (1,0), (2,0) — координаты начальных точек составляющих прямоугольников.

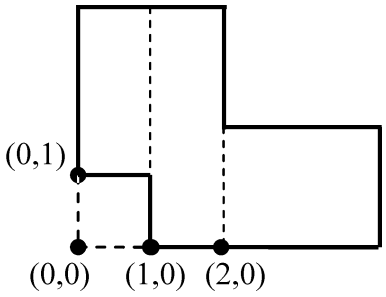


Рис. 2. Разбиение ортогонального многоугольника на прямоугольники с указанием координат начальных точек

Размещение ортогональных многоугольников на листе будем считать допустимым, если выполнены следующие условия:

- грани ортогонального многоугольника параллельны сторонам листа;
- размещенные на листе ортогональные многоугольники не перекрывают друг друга;
- ортогональные многоугольники не перекрываются со сторонами листа.

Поставленная задача относится к классу задач раскроя-упаковки. В условиях массового производства она описывается непрерывной моделью линейного программирования [4]. Подробно описание этой модели приведено в разделе 3. Сложным моментом при решении поставленной задачи является заполнение листов ортогональными многоугольниками конструирования упаковки.

План рассматриваемой задачи упаковки представляет собой совокупность карт (способов) упаковки с указанием интенсивностей их применения.

2. Упаковка ортогональных многоугольников на листах

Определения и операции с кортежами. Для размещения ортогонального многоугольника на листе представим его в виде кортежей.

Кортеж строится по каждому из направлений координатных осей (Ox и Oy) и определяется как последовательность прямоугольных блоков.

Построение кортежа по оси Ox (x-кортежа) происходит следующим образом: левую нижнюю вершину прямоугольника, описывающего ОМ, примем за начало координат; через координаты оси OX, соответствующие левой или правой грани каждого составляющего ОМ прямоугольника, мысленно проводятся сечения (рис. 3). Они разделяют ОМ на прямоугольные полосы. Длина полученных полос — расстояние между соседними отмеченными координатами. Ширину полос определим как величину, равную суммарной ширине прямоугольников, пересекающих соответствующую полосу. Тогда для x-кортежа ОМ, изображенного на рис. 3, *а*, имеем: $V_x^1 = y_2$, $V_x^2 = y_1$, $V_x^3 = y_1 + (y_3 - y_2)$, $V_x^4 = y_3$. Таким образом, длина блока x-кортежа равна расстоянию между соседними координатами, а ширина — размеру соответствующего сечения V_x.

Аналогично строится кортеж по направлению оси Oy (y-кортеж), который представлен на рис. 3, *б*. Здесь $V_y^1 = x_4$, $V_y^2 = x_1 + (x_4 - x_3)$, $V_y^3 = x_2$.

Кортеж записывается в виде списка блоков с указанием их длины и ширины; размер списка $(2 \times \nu)$, где ν — число блоков.

Далее упаковка ортогонального многоугольника на лист сводится к решению двух задач размещения кортежей, представляющих ОМ, по соот-

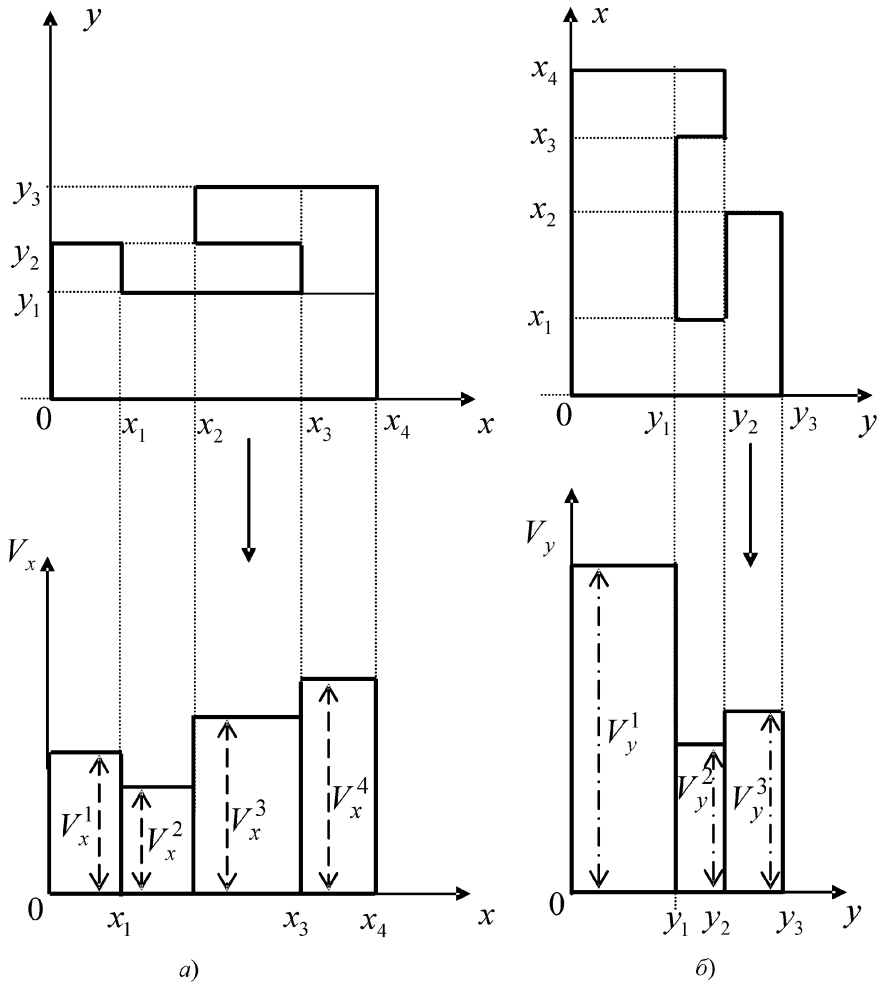


Рис. 3. Представление ортогонального многоугольника в виде двух кортежей: *а* — по оси *Ox*; *б* — по оси *Oy*

ветствующей координатной оси с выполнением условий допустимости размещения. Размещение кортежей происходит следующим образом. Левую нижнюю вершину листа считаем за начало координат. Соответствующий первому ОМ *x*-кортеж размещается с начала координатной оси *Ox*: на эту ось наносятся координаты, отвечающие началам и концам блоков *x*-кортежа. Кроме того, каждому участку оси отвечает суммарное значение сечения на этом промежутке, равное ширине блока.

Аналогично по направлению оси *Oy* размещается *y*-кортеж первого ОМ.

После завершения процедуры размещения первого ОМ, т. е. размещения соответствующих ему *x*(*y*)-кортежей, его начальной точке присваиваются координаты расположения на листе. Причем размещение *x*-кортежа работает на определение абсциссы координаты расположения начальной точки ОМ на листе, *y*-кортежа — ординаты.

Для размещения следующих заготовок определим операцию сложения *x*(*y*)-кортежей как сложение отвечающих им блоков на каждом отрезке оси *Ox* (*Oy*).

Процедура сложения кортежей: на ось *Ox* наносятся координаты, соответствующие началу или окончанию блоков складываемых *x*-кортежей. Далее на каждом полученном интервале оси вычисляется ширина блока, равная суммарной ширине блоков складываемых *x*-кортежей и имеющая смысл суммарного сечения на данном отрезке. На рис. 4 показано сложение *x*-кортежей *A* и *B*, где V_x^i — ширина блоков складываемых кортежей ($i = 1, \ldots, 7$). Результат сложения — *x*-кортеж *C*, H_j — ширина его блоков ($j = 1, \ldots, 6$).):

$$H_1 = V_x^1 + V_x^4;\ H_4 = V_x^2 + V_x^6;$$

$$H_2 = V_x^3 + V_x^5;\ H_5 = V_x^2 + V_x^7;$$

$$H_3 = V_x^3 + V_x^6;\ H_6 = V_x^7.$$

Однако на рис. 4 не учитывалась ширина листа. Если ширина получившихся блоков на каком-либо интервале оси *Ox* превышает ширину листа, то в получаемой карте упаковки ОМ перекрываются.

На рис. 5 показано сложение *x*-кортежей с учетом ширины листа. На рис. 5, *а*, видно, что при сложении двух *x*-кортежей, выполненном без сдвига, ширина, соответствующая четвертому блоку результирующего *x*-кортежа, превышает ширину листа (V_x^0). В таких случаях, если это возможно, необходимо провести сдвиг добавляемого *x*-кортежа вдоль оси на величину, равную ширине блока, превышающего V_x^0 (рис. 5, *б*). Если сдвиг невозможен (т. е. при размещении *x*-кортежа со сдвигом на δ его правая координата по оси *Ox* превышает длину листа), то ищется другое расположение *x*-кортежа данной заготовки.

Аналогично выполняется сложение *y*-кортежей.

Учет необходимых и допустимых условий размещения. С начала размещения кортежей формируется два списка для учета размера листа по данному направлению и размера сечения листа, соответствующего каждому отрезку конкретной координатной оси.

Первый блок размещаемого *x*(*y*)-кортежа может быть расположен либо с координаты, определяющей "начало" координатной оси *Ox* (*Oy*), либо с координаты, соответствующей началу или концу уже

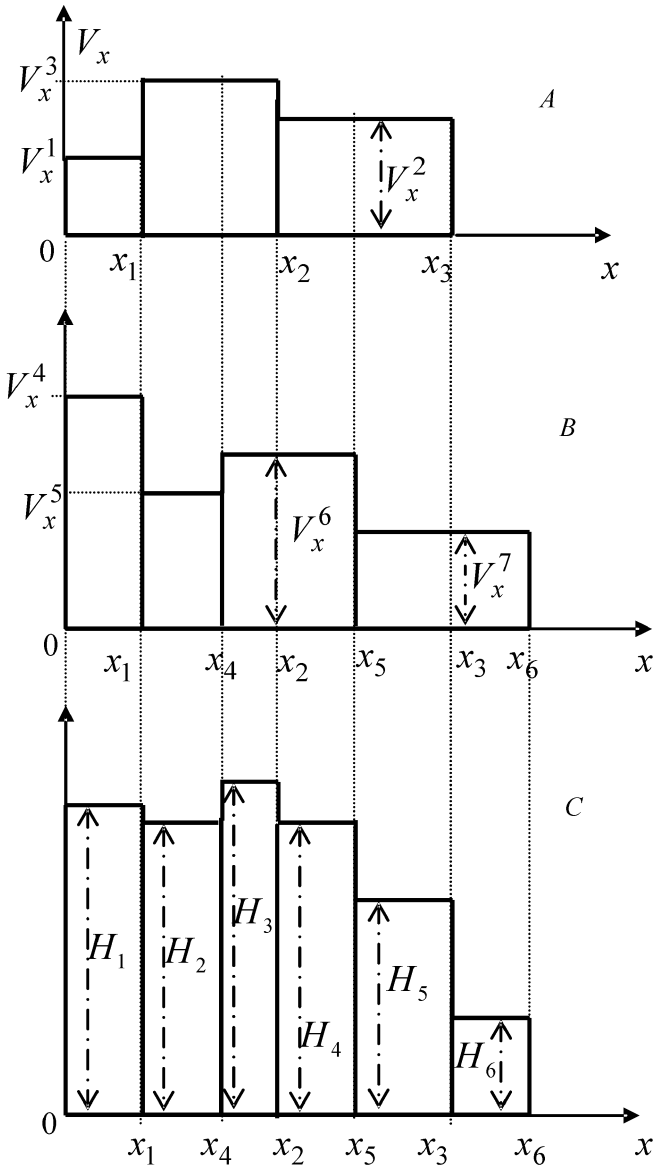


Рис. 4. Сложение кортежей без ограничения на ширину блоков

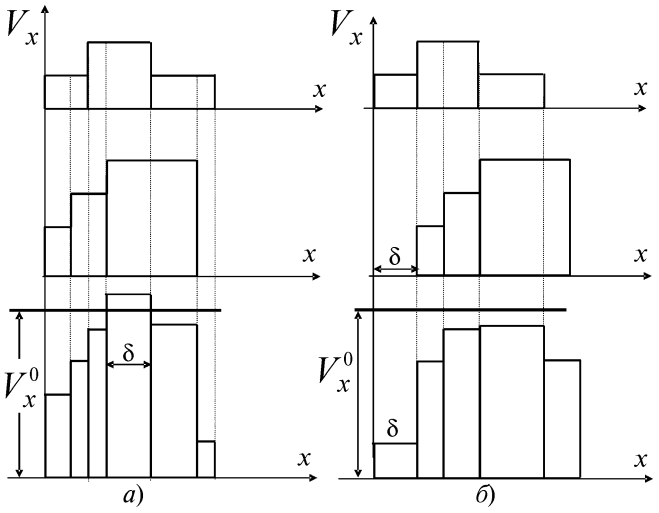


Рис. 5. Сложение *x*-кортежей с учетом ограничения на ширину результирующих блоков:
а — превышение ширины листа; *б* — сдвиг второго *x*-кортежа на δ

расположенного блока $x(y)$-кортежа. Если при размещении очередного $x(y)$-кортежа он выходит за границы листа, то ищется другое его размещение.

Утверждение 1. Для того чтобы размещение ортогональных многоугольников на листе было допустимым, необходимо выполнение следующих условий:

- правая координата расположения добавляемого $x(y)$-кортежа не должна превышать длину (ширину) листа;
- ширина блока результирующего $x(y)$-кортежа при добавлении $x(y)$-кортежа очередного ОМ не должна превышать ширину (длину) листа.

Сформулированные в утверждении 1 условия следуют из определения допустимости размещения. Указанные условия соответствуют необходимым условиям допустимости размещения, но не являются достаточными. Однако проверка данных условий в процессе укладки является своеобразным отсечением полной проверки допустимости размещения, так как при несоблюдении этих условий размещение заведомо будет недопустимым.

Утверждение 2. Для того чтобы размещение было допустимым, достаточно выполнение условия взаимного неперекрытия размещаемых ортогональных многоугольников.

Таким образом, если размещены оба кортежа очередного ортогонального многоугольника, проверяется достаточное условие допустимости размещения — условие неперекрытия этого многоугольника с уже размещенными на листе заготовками. Это означает, что в любом сечении листа суммарное значение сечений, проходящих через размещенные ОМ, не может превышать размер листа по данному направлению. Следует вспомнить, что нахождение ОМ на листе означает, что его начальной точке присвоены координаты расположения; тогда для всех начальных точек прямоугольников, составляющих ОМ, также определены координаты.

Проверка достаточного условия допустимости размещения следует из условий неперекрытия каждых двух прямоугольников, составляющих размещаемые ОМ. Пусть (x_i, y_i) — координаты начальной точки i-го прямоугольника на листе, а (p_i, q_i) — его размеры, $i = 1,2$. Тогда неперекрытие двух прямоугольников на плоскости означает выполнение хотя бы одного из условий:

$$\left[\begin{array}{l} x_2 \geqslant x_1 + p_1 \text{ или } x_1 \geqslant x_2 + p_2; \\ y_2 \geqslant y_1 + q_1 \text{ или } y_1 \geqslant y_2 + q_2. \end{array}\right.$$

Если при упаковке очередного ОМ не выполняются указанные в утверждениях 1, 2 условия, то ищется другое его размещение. Если не находится удовлетворяющего условиям размещения, то ОМ

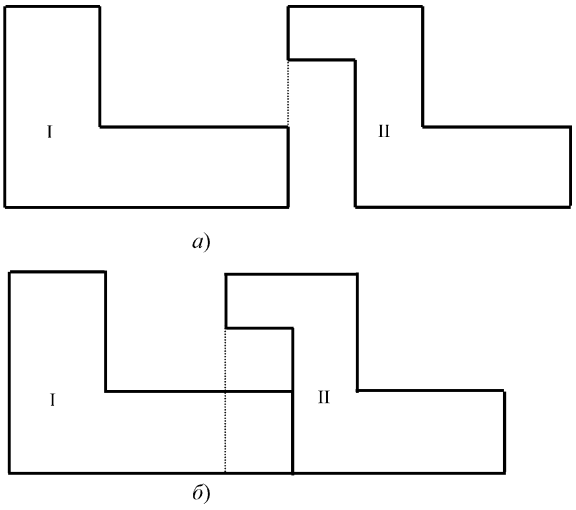


Рис. 6. Пример расположения ортогональных многоугольников: *а* — без процедуры уплотнения; *б* — с процедурой уплотнения

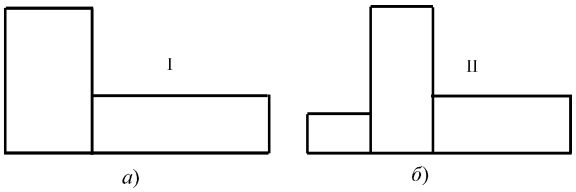


Рис. 7. *X*-кортежи ортогональных многоугольников: *а* — *x*-кортеж I; *б* — *x*-кортеж II

далее в формировании данной карты упаковки не участвует.

Процедура уплотнения. Предложенное правило размещения очередного ОМ обладает тем недостатком, что получаемая таким образом упаковка не всегда является плотной. На рис. 6, *а* показано расположение ортогональных многоугольников, на котором видна необходимость "подвинуть" второй ОМ влево, т. е. вдоль координатной оси *Ox*. В связи с этим к описанному правилу предлагается дополнительная процедура уплотнения.

Процедура уплотнения представляет собой корректировку правила укладки кортежа очередной заготовки и состоит в следующем: кроме координат, отвечающих началу или концу блоков кортежа, предлагается рассматривать также координаты, соответствующие так называемым "псевдорезам" — сечениям по дополнительным координатам. Тогда при размещении кортежа очередного ОМ в качестве дополнительных координат рассматриваются координаты, отвечающие сдвигам от координат блоков размещенного кортежа на длины блоков размещаемого кортежа.

Пусть размещен некоторый x-кортеж ортогонального многоугольника и $x_1, x_2, ..., x_k$ — координаты расположения его блоков по оси *Ox*. Тогда для следующего размещаемого x-кортежа с длинами блоков $\alpha_1, \alpha_2, ..., \alpha_n$ в качестве псевдорезов будут рассматриваться координаты вида $x' = x_i - (\alpha_n - \alpha_j)$, $i = 1, 2, ..., k, j = 1, 2, ..., n$, при условии, что $x_i - (\alpha_n - \alpha_j) \geqslant 0$, $i = 1, 2, ..., k, j = 1, 2, ..., n$.

При использовании процедуры уплотнения по данному направлению очередного кортежа ОМ рассматриваются следующие варианты начальной координаты размещения:

- координата, определяющая начало координатной оси по данному направлению;
- координата, соответствующая началу или окончанию какого-либо блока уже расположенной заготовки;
- координата, соответствующая псевдорезам.

На рис. 7 для рассматриваемых ортогональных многоугольников изображены соответствующие им x-кортежи.

Рассмотрим их укладку на листе, ширина которого совпадает с максимальной шириной блоков (обозначим ее V_x^0).

Пусть x-кортеж I размещен на листе; тогда для того, чтобы разместить кортеж II, в качестве начальной координаты могут рассматриваться x_1, x_2 или x_3. Как видно на рис. 8, *а*, начальной координатой для расположения x-кортежа II может быть только x_3 (по правилу сложения кортежей), в результате чего конечной будет упаковка, изображенная на рис. 6, *а*.

Использование при укладке x-кортежей процедуры уплотнения, показанной на рис. 7, *б*, предполагает проведение псевдореза по координате $x_3' = x_3 - p_3$. (При этом для любой координаты $x < x_3'$ проведение псевдорезов не имеет смысла, так как при этом превышается ширина листа.) В результате будет получена упаковка, представленная на рис. 6, *б*.

3. Методы математического программирования для решения задачи планирования упаковки ортогональных многоугольников

Математическая модель задачи на базе целочисленного линейного программирования. Задача размещения ортогональных многоугольников может быть представлена в следующем виде. Пусть каждая упаковка r характеризуется вектором $\alpha_r = (\alpha_{r1}, \alpha_{r2}, ..., \alpha_{rm})$, компоненты α_{ri} которого указывают число ОМ i-го типа ($i = 1, ..., m$), получаемых по карте r, и h — число всевозможных карт упаковок. Необходимо найти совокупность реализуемых упаковок $r_1, r_2, ..., r_h$ и вектор интенсивностей их применения $x = (x_1, x_2, ..., x_h)$, удовлетворяющие следующим условиям:

$$x_t \geqslant 0,\ t = 1, ..., h; \tag{1}$$

$$\{x_t\} = 0,\ t = 1, ..., h; \tag{2}$$

$$\sum_{t=1}^{h} x_t \alpha_{rt} = B = (b_1, b_2, ..., b_m)^{\mathrm{T}}; \tag{3}$$

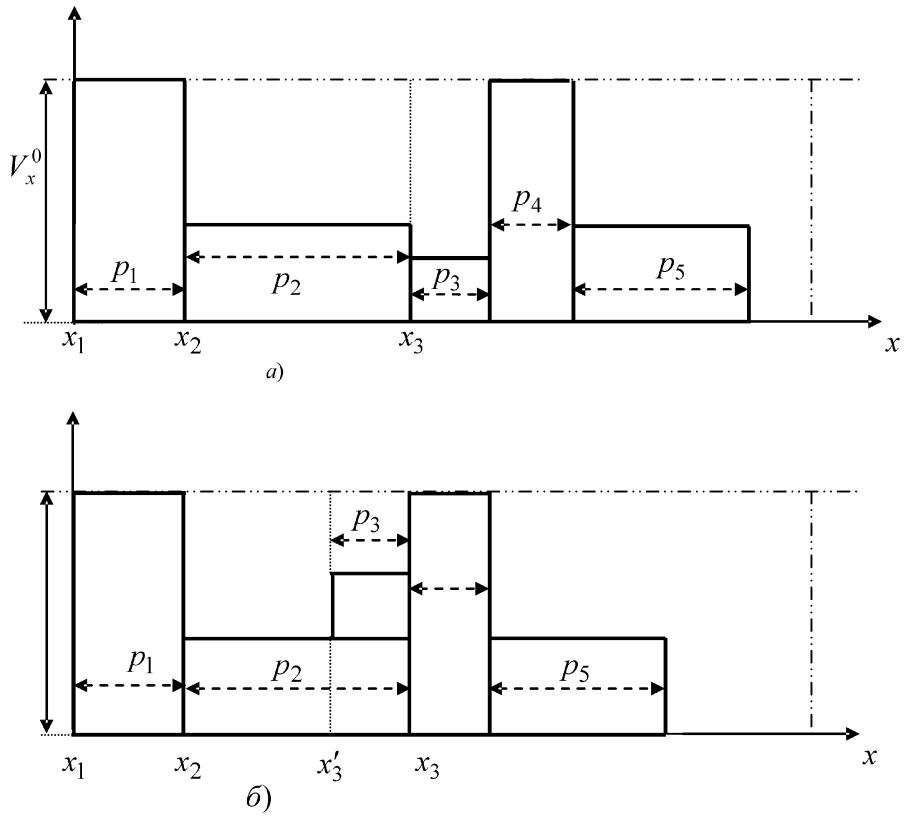


Рис. 8. Расположение *x*-кортежей на листе:
а — без процедуры уплотнения; *б* — с процедурой уплотнения

$$\mu(r, x) = \sum_{t=1}^{h} x_t \to \min. \qquad (4)$$

Здесь условия (1) и (2) означают соответственно требования неотрицательности и целочисленности переменных x_t, $t = l, ..., h$, а условие (3) — выполнение плана по заданному ассортименту заготовок.

План упаковки $(r, x) = (x_1, x_2, ..., r_h; x)$ является допустимым, если упаковки $r_1, r_2, ..., r_h$ и вектор $x = (x_1, x_2, ..., x_h)$ удовлетворяют условиям (1)—(3). Допустимый план является оптимальным, если величина (4) достигает минимума.

Приведенная задача является задачей целочисленного линейного программирования. Воспользуемся ее релаксацией линейным программированием (ЛП). Для этого при решении данной задачи основной проблемой является нахождение множества векторов $\{\alpha(r_t), t = 1, ..., h]$, характеризующих карту упаковки и определяющих матрицу ограничений.

Для формирования матрицы ограничений предлагается рассматривать не все множество возможных способов упаковки, а лишь способы, входящие в базисное множество, а далее по мере необходимости генерировать улучшающие способы, заменяя ими некоторые из базисных. В качестве начального базисного множества можно рассматривать однородные упаковки, т. е. упаковки одинаковых ортогональных многоугольников. Такой подход был предложен Л. В. Канторовичем и В. А. Залгаллером [5].

В условиях непрерывного производства при решении задачи упаковки методами ЛП с недетерминированной матрицей на каждом шаге итерационного процесса генерируется улучшающий способ, имеющий максимальную оценку. В качестве оценки принимается сумма объективно обусловленных оценок, используемых в раскрое заготовок (двойственных переменных). Возможность построения такого способа оказывается весьма принципиальной, так как именно он будет служить признаком оптимальности всего раскройного плана. В задачах линейного и гильотинного раскроя удается построить способ с максимальной оценкой, при этом алгоритмы генерирования улучшающего способа представляют собой реализацию идей динамического программирования.

В связи с тем, что в рассматриваемой задаче заготовки имеют сложную форму, применение методов динамического программирования для построения такого способа оказывается затруднительным. Поэтому для получения допустимого решения поставленной задачи предлагается рассматривать лишь некоторое подмножество $\{\alpha_{r_t}, t = 1, ..., k\}$, $k \leqslant h$, всевозможных упаковок, карты которых предварительно сгенерированы с помощью эвристического алгоритма случайной выборки. Указанный алгоритм основан на генерации наиболее вероятного подмножества допустимых карт раскроя.

Теперь если предположить, что некоторое подмножество из множества всевозможных упаковок известно и b_i, $i = 1, ..., m$, настолько велики, что условием целочисленности интенсивностей применения упаковок можно пренебречь, то рассматриваемая проблема транспонируется в классическую задачу линейного программирования.

Математическая модель задачи планирования на базе линейного программирования. Пусть задано множество заготовок P, их требуемое число $B = (b_1, b_2, ..., b_m)$. Кроме того, пусть сгенерировано некоторое множество карт упаковок $\{\alpha(r_t), t = 1, ..., k\}$, $(k \leqslant h$, h — число всевозможных карт упаковки). Требуется найти совокупность реали-

зуемых упаковок r_1, r_2, ..., r_k и вектор интенсивностей их применения $x = (x_1, x_2, ..., x_k)$, удовлетворяющие следующим условиям:

$$x_t \geqslant 0,\ t = 1, ..., k; \tag{5}$$

$$\sum_{t=1}^{k} x_t \alpha_{r_t} = B = (b_1, b_2, ..., b_m)^{\mathrm{T}}; \tag{6}$$

$$\mu(r, x) = \sum_{t=1}^{k} x_t \to \min. \tag{7}$$

Целевая функция (7) имеет смысл суммарного числа листов, затраченных в процессе упаковки заданного множества заготовок. Условия (5), (6) являются условиями реализуемости плана раскроя-упаковки.

Алгоритм решения задачи базируется на методах линейного программирования, а именно — симплекс-методе с матрицей ограничений, состоящей из векторов, характеризующих допустимые способы упаковки. В качестве исходного допустимого плана упаковки используется план, состоящий из однородных способов упаковки. Каждый такой способ представляет собой упаковку листа заготовками одного фиксированного типа.

Алгоритм генерации последовательностей заготовок и случайной выборки для конструирования упаковки. Пусть каждому ОМ сопоставлены два кортежа. Как уже говорилось, для решения задачи планирования упаковок необходимо предварительно сгенерировать некоторое множество карт упаковки.

В связи с тем, что комплектность b_i каждого типа заготовок различна, для построения карт упаковки предварительно предлагается сгенерировать последовательности из номеров ОМ в целях более равномерного их распределения по листам. Каждая такая последовательность будет представлять номера тех типов заготовок, которые мы хотим упаковать на один лист.

Пусть $S = \sum_{i=1}^{m} b_i$ — общее число ОМ. Сопоставим каждому типу ОМ полуоткрытый интервал $[c_i, d_i)$, где $c_1 = 0$, $d_1 = b_1/S$, $c_2 = d_1$, $d_2 = c_2 + b_2/S$, ..., $c_m = d_{m-1}$, $d_m = c_m + b_m/S$. Очевидно, что $d_m = 1$, т. е. отрезок [0,1) мы разбили на m интервалов. Длина i-го интервала ($i = 1, ..., m$) пропорциональна доле ОМ i-го типа от общего числа заготовок.

Случайным образом из отрезка [0,1) выбираются числа p_j. Если $c_i \leqslant p_j \leqslant d_i$, то ОМ i-го типа включается в последовательность. Таким образом, ОМ с большей комплектностью будут встречаться в последовательностях чаще, что обеспечит близкое к равномерному распределение каждого типа заготовок по листам. Формирование каждой последовательности продолжается до тех пор, пока суммарная площадь ортогональных многоугольников, входящих в эту последовательность, не превысит площади листа. Общее число последовательностей (и, соответственно, генерируемых карт упаковки) задается произвольно; в нашем описании оно равно k.

Далее из каждой сгенерированной последовательности, указывающей, какие типы заготовок будут располагаться на рассматриваемом листе, с использованием алгоритма случайной выборки будут формироваться карты упаковки. Результат генерации карты — вектор $\alpha_r = (\alpha_{r1}, \alpha_{r2}, ..., \alpha_{rm})$, характеризующий данную упаковку r, и координаты положения входящих в нее заготовок на листе.

Согласно предлагаемому алгоритму случайной выборки конструирование способа упаковки происходит путем поочередного размещения заготовок на листе в том порядке, который указан в последовательности. Процедура размещения ортогональных многоугольников каждой последовательности выполняется заданное число раз: случайным образом заготовки из данной последовательности переупорядочиваются и вновь размещаются на листе. В результате запоминается та карта упаковки, для которой суммарная площадь размещенных ортогональных многоугольников максимальна.

Алгоритм конструирования карты упаковки из каждой последовательности заготовок состоит из выполнения следующих шагов:

Шаг 1. *Подготовительный шаг:* представление каждого ортогонального многоугольника в виде совокупности кортежей, построенных по каждому направлению.

Шаг 2. *Размещение x-кортежа и y-кортежа первого ОМ*; после размещения обоих кортежей начальной точке этого многоугольника присваиваются координаты его расположения на листе.

Шаг 3. *Размещение следующего ОМ*: процесс размещения каждого его кортежа с проверкой выполнения необходимых условий допустимости (при нарушении необходимых условий ищется новое размещение $x(y)$-кортежа).

Шаг 4. *Проверка достаточного условия*: если удалось размещение обоих кортежей ортогонального многоугольника, удовлетворяющее необходимым условиям допустимости, проверяется выполнение условия неперекрытия этого ОМ с ранее размещенными. При нарушении условия неперекрытия делается шаг назад; при выполнении достаточного условия начальной точке размещенного на листе ОМ присваиваются координаты.

Шаг 5. *Формирование вектора α, отвечающего карте упаковки.*

4. Результаты вычислительного эксперимента

Вычислительный эксперимент был проведен в целях определения зависимости эффективности плана упаковки и времени работы алгоритмов, с процедурой уплотнения и без нее от числа конструируемых карт упаковки и числа шагов генератора карты.

Для определения эффективности полученных результатов использован коэффициент упаковки

$$К\_У = PP/PL,$$

где PP — суммарная площадь размещенных на листе заготовок; PL — лошадь листа.

В табл. 1 приведены средние показатели эффективности плана упаковки для 10 видов заготовок заданных комплектностей (от 500 до 1500), среди которых три — большие (составляют от 30 до 40 % от площади листа), три — малые (составляют от 3 до 6 % от площади листа) и четыре — средние.

По результатам эксперимента можно сделать следующие выводы.

- Зависимость получаемого плана упаковки от числа шагов генератора (числа попыток уложить все ортогональные многоугольники из выбранной последовательности на листе) не линейна, так как если в размещаемую последовательность попадают заготовки больших размеров или сильно искривленные, то не всегда удается найти их удачную упаковку, даже при большом числе попыток. Это лишь увеличивает время работы программы, а средний коэффициент упаковки получается невысоким из-за наличия хотя бы одной карты с показателем эффективности 0,5—0,6.
- При использовании алгоритма с процедурой уплотнения его время работы значительно возрастает, при этом значение показателя эффективности увеличивается.
- Число генерируемых карт упаковки также влияет на показатели эффективности плана и на время работы программы (табл. 2).

По результатам вычислительного эксперимента было замечено, что выгоднее генерировать 15—20 карт упаковки. Дальнейшее увеличение числа генерируемых карт приводит к незначительному улучшению коэффициента упаковки по сравнению с затратами времени на работу программы.

Одна из полученных карт упаковки приведена на рис. 9 (см. четвертую сторону обложки).

Таблица 1

Зависимость коэффициента упаковки от числа шагов генератора

Число типов заготовок	Число генерируемых карт	Число шагов генератора	Процедура уплотнения	Время, с	К_У
10	15	5	—	11	0,662
10	15	15	—	25	0,667
10	15	25	—	56	0,666
10	15	5	+	95	0,698
10	15	15	+	240	0,7067
10	15	25	+	270	0,7074

Таблица 2

Зависимость коэффициента упаковки от числа генерируемых карт

Число заготовок	Число шагов генератора	Число генерируемых карт	Процедура уплотнения	Время, мин	К_У
10	15	40	+	16	0,7367
10	15	40	—	4	0,7008
10	15	20	+	12	0,7331
10	15	20	—	1	0,694

Заключение

Приведена постановка задач размещения ортогональных многоугольников в прямоугольных областях. Эта задача имеет самостоятельное прикладное значение, а также может применяться в качестве аппроксимационной при решении проблем фигурного раскроя. Для ее решения предложен метод конструирования размещения ОМ на листе, т. е. алгоритм-декодер, с помощью которого генерируется множество различных размещений. Далее предлагается решать задачу методом линейного программирования на этом множестве. Приведены результаты численного эксперимента с различными значениями параметров алгоритма. В перспективе разрабатывается метод с использованием эволюционного наивного алгоритма и генетического алгоритма [6].

Список литературы

1. **Верхотуров М. А.** Задача нерегулярного раскроя плоских геометрических объектов: моделирование и расчет рационального раскроя // Информационные технологии. 2000. № 5. С. 37—42.

2. **Корницкая М. Н.** Автоматизация проектирования карт фигурного нерегулярного раскроя в условиях единичного производства на основе аппроксимационного подхода. Дисс. на соискание ученой степени кандидата технических наук. Свердловск: Политехнический институт, 1990. 130 с.

3. **Covering** Rectilinear Polygons with Axis-Parallel Rectangles / V. S. Anil Kumar, H. Ramesh // Information and Control. 1999. V. 63. N 3. P. 164—189.

4. **Мухачева Э. А., Рубинштейн Г. Ш.** Математическое программирование // Новосибирск: Наука СО. 1986. 236 с.

5. **Канторович Л. В., Залгаллер В. А.** Рациональный раскрой промышленных материалов. Новосибирск: Наука СО, 1971. 320 с.

6. **Филиппова А. С.** Моделирование эволюционных алгоритмов решения задач прямоугольной упаковки на базе технологии блочных структур // Информационные технологии. 2006. № 11. 32 с.

УДК 004.421.6

В. М. Захаров, д-р техн. наук, **Б. Ф. Эминов,**
КГТУ им. А. Н. Туполева

Анализ алгоритмов разложения двоично-рациональных стохастических матриц на комбинацию булевых матриц

Рассматриваются алгоритмы разложения стохастических матриц цепей Маркова с двоично-рациональными элементами на имплицирующий вектор и множество стохастических булевых матриц. Для предлагаемых алгоритмов разложения оцениваются вычислительная сложность и размер имплицирующих векторов. Данные алгоритмы позволяют снизить вычислительную сложность разложения и получить более точную оценку размера имплицирующего вектора для заданной матрицы.

Введение

Автономный конечный вероятностный автомат (ВА) с детерминированной функцией выхода и со стохастической матрицей **P** переходов состояний можно преобразовать в двухблочную схему [1], состоящую из источника случайности Бернулли, выдающего с вероятностью p_i букву x_i из некоторого алфавита X, и детерминированного автомата. Методы синтеза [1] ВА базируются на разложении стохастической матрицы **P** цепи Маркова на множество стохастических булевых матриц (СБМ) и имплицирующий вектор (ИВ) [2—5], размер l которого зависит от алгоритма разложения. Величина l определяет комбинационную сложность соответствующего автомата, представленного в некотором логическом базисе, например, при полиномиальном представлении автоматных моделей марковских функций и конечно-автоматных случайных последовательностей над полем Галуа [6, 7].

В подходе разложения стохастических матриц, представленном в работах [3—5, 8, 9] (подход 1), l и вычислительная сложность (ВС) разложения определяются, в основном, размерами и энтропией матриц **P**. В другом подходе учитывается точность представления элементов стохастических матриц [3—5, 8, 9] (подход 2). Двоично-рациональное представление элементов матрицы **P**, используемое в алгоритмах подхода 2, обосновывается выбором вычислительной базы.

Целью данной работы является определение оценок параметра l для подхода 2 и вычислительной сложности известных и предлагаемых алгоритмов разложения обоих подходов. Предполагается, что описываемые алгоритмы уменьшат значения l и ВС за счет снижения точности ε представления элементов матрицы **P** и использования эвристик.

Зададим простую цепь Маркова (ЦМ) системой [10] $(S, \mathbf{P}, \boldsymbol{\pi}_0)$, где $S = \{s_i\}$, $i = \overline{0, m-1}$ — конечное множество состояний ЦМ; $\mathbf{P} = (p_{ij})$, $i, j = \overline{0, m-1}$ — квадратная стохастическая матрица размера $m \times m$; $\boldsymbol{\pi}_0$ — стохастический вектор размера m, определяющий начальное распределение вероятностей состояний ЦМ. Известна формула разложения [4] матрицы **P** на сумму стохастических булевых матриц:

$$\mathbf{P} = \sum_{a=0}^{l-1} q_a B^{(a)}, \qquad (1)$$

где q_a, $a = \overline{0, l-1}$ — элементы стохастического ИВ $\tilde{\boldsymbol{q}}$; $\mathbf{B}^{(a)}$ — стохастическая булева матрица, $a = \overline{0, l-1}$. СБМ $\mathbf{B}^{(a)} = (b_{ij})$, $i, j = \overline{0, m-1}$ — это квадратная матрица размера $m \times m$, у которой элементы $b_{ij} \in \{0; 1\}$, $i, j = \overline{0, m-1}$, и на каждой i-й строке существует только один элемент $b_{ij} = 1$. Обозначим СБМ $\mathbf{B}^{(a)}$ как **B**. Под *формированием СБМ* будем понимать процесс определения координат элементов b_{ij} СБМ, где $b_{ij} = 1$. СБМ **B** будем называть *уникальной*, если в разложении (1) не существует других СБМ, равных **B**.

В работе рассматриваются следующие четыре алгоритма, выполняющие разложение (1) стохастической матрицы **P** цепи Маркова:

- алгоритм [5], использующий метод минимакса [4] (алгоритм 1);
- предлагаемый алгоритм, основанный на методе [3] представления элементов матриц **P** двоично-рациональными дробями (алгоритм 2);
- два предлагаемых алгоритма, также базирующихся на двоично-рациональном представлении элементов матрицы **P** (алгоритмы 3 и 4).

Введем в рассмотрение следующие переменные. Пусть $\tilde{\mathbf{P}}(m, b)$ — класс стохастических матриц с двоично-рациональными элементами, представленными с точностью $\varepsilon = 1/2^b$. Число $b = -\log_2 \varepsilon$, $b \in N$ (N — множество натуральных чисел) — число двоичных разрядов, необходимых для хранения элемента p_{ij} или числителя a_{ij} элемента p_{ij}, $i, j = \overline{0, m-1}$; $c_{ijk} \in \{0; 1\}$ — значение k-го двоичного разряда элемента p_{ij} или a_{ij}, $i, j = \overline{0, m-1}$, $k = \overline{0, b-1}$.

Будем использовать следующие способы хранения СБМ:

- в виде квадратной бинарной матрицы $\mathbf{B} = (b_{ij})$ размера $m \times m$, $i, j = \overline{0, m-1}$, $b_{ij} \in \{0; 1\}$ (способ 1);
- в виде числового вектора $\tilde{\mathbf{B}} = \{b_i\}$ размера m, i, $b_i = \overline{0, m-1}$, где позиции элементов p_{ib_i} в мат-

рице **P** задают позиции единичных (равных единице) элементов СБМ (способ 2).

При втором способе необходим меньший объем памяти для хранения СБМ и не требуется выполнять вычисления для определения позиции единичного элемента СБМ.

Замечание 1. В рассматриваемых алгоритмах будем считать, что операции сравнения чисел, присвоения значения одной переменной другой, вычитания и сложения чисел, двоичного сдвига чисел являются равнозначными — имеют одинаковую ВС.

Подход разложения матриц, основанный на методе минимакса

Приведем описание **алгоритма 1** [5]. На каждом цикле алгоритма 1, до тех пор пока матрица **P** не станет нулевой, осуществляется выбор по строкам максимальных элементов p_{ij} матрицы **P**, $i, j = \overline{0, m-1}$, задающих координаты текущей СБМ. Минимальное значение из выбранных элементов p_{ij}, вычитаемое из всех p_{ij}, задает элемент ИВ, соответствующий текущей СБМ. Для алгоритма 1 известны следующие оценки размера l ИВ:

$$1)\ 1 \leqslant l \leqslant m^2 - m + 1 \quad [4, 5]; \qquad (2)$$

$$2)\ \max_{0 \le i \le m-1} d_i \leqslant l \leqslant \sum_{i=0}^{m-1} d_i - m + 1 \quad [8, 9], \qquad (3)$$

если $\exists p_{ij} = 0$, $i, j = \overline{0, m-1}$, где d_i — число ненулевых элементов i-й строки матрицы **P**.

Предложение 1. Вычислительная сложность алгоритма 1 равна $O(m^4)$.

Схема доказательства следующая. На каждом цикле алгоритма (максимальное число циклов совпадает с оценкой (2) — $m^2 - m + 1$) из каждой строки (m строк) выбираются максимальные элементы (m сравнений) матрицы **P**, т. е. ВС алгоритма равна $O(m^4)$. Предложение доказано.

Замечание 2. В разложении (1) матрицы **P** по алгоритму 1 каждая из СБМ уникальна.

Из данного замечания можно сделать вывод о том, что для алгоритма 1 размер l ИВ не сокращается за счет объединения одинаковых СБМ.

Двоично-рациональный подход разложения матриц

Для выполнения алгоритмов второго подхода элементы матрицы **P** необходимо привести к одному знаменателю. Для этого:

- вначале элементы $p_{ij} > 0$, $i, j = \overline{0, m-1}$, матрицы **P** приводятся к обыкновенной дроби с заданной погрешностью $\varepsilon = 1/2^b$;
- затем находится общий знаменатель элементов как наименьшее общее кратное от всех элементов $p_{ij} > 0$ матрицы **P**.

Элементы $p_{ij} > 0$ матрицы $\mathbf{P} = (p_{ij})$, $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$, $i, j = \overline{0, m-1}$, приведенные к виду $p_{ij} = a_{ij}/2^b$, $a_{ij} = \overline{0, 2^b}$, можно хранить не в виде вещественных чисел диапазона от 0 до 1, а в виде целых положительных чисел a_{ij} диапазона от 0 до 2^b — числителей элементов исходной матрицы **P**.

Формализуем метод, описанный в работе [3], в виде ***алгоритма 2.*** Координаты единичных элементов СБМ задаются выбранными по одному из каждой строки псевдослучайным образом элементами p_{ij} матрицы $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$, $i, j = \overline{0, m-1}$, для которых выполняется условие $c_{ijk} = 1$, где $k = \overline{0, b-1}$ — текущий рассматриваемый двоичный разряд элементов матрицы **P**. Новому элементу ИВ, соответствующему сформированной СБМ, присваивается значение 2^{-k}, а переменные c_{ijk}, соответствующие выбранным элементам p_{ij}, обнуляются. Формирование СБМ продолжается до тех пор, пока не будут обнулены все элементы p_{ij} матрицы **P**.

Предложение 2. Вычислительная сложность алгоритма 2 равна $O(m^4 b^3)$.

Доказательство предложения 2 опускается, так как ниже приводится доказательство модификации данного алгоритма. Основной вклад в ВС алгоритма вносят операции случайного выбора элементов формируемых СБМ и проверки данных СБМ на уникальность.

В дальнейшем двоичные разряды чисел с меньшими весами будем называть *младшими разрядами*. Также примем, что разряд с большим весом имеет нулевой порядковый номер, а с уменьшением веса порядковый номер разряда увеличивается.

В целях уменьшения ВС введем следующие изменения в алгоритме 2 (модификацию алгоритма 2 будем называть *алгоритмом 2м*).

1. Представим СБМ способом 2 — в виде вектора $\boldsymbol{B} = \{b_i\}$, $i = \overline{0, m-1}$.

2. Исключим операцию сравнения полученной СБМ с ранее найденными.

Если текущая СБМ может быть получена не только на k-м, $k = \overline{0, b-1}$, но и на младших относительно k разрядах элементов матрицы **P**, то на текущем цикле алгоритма будем выполнять разложение (1) матрицы **P** и для младших разрядов элементов матрицы **P**.

Теорема 1. В разложении (1) матрицы **P** по алгоритму 2м каждая из СБМ уникальна.

Доказательство. В алгоритме 2м СБМ $\boldsymbol{B} = \{b_i\}$, $i = \overline{0, m-1}$, может быть получена как для текущего k-го разряда элементов матрицы **P**, $k = \overline{0, b-1}$, так и для младших, еще не рассмотренных, разрядов элементов матрицы **P**. Если существует j-й разряд, где $j < k$, для которого при любом b_i выполняется равенство $c_{ib_ij} = 1$ (то есть СБМ $\boldsymbol{B}$ можно

сформировать на k-м и j-м разрядах элементов p_{ib_i} матрицы **P**), то текущий элемент q_l ИВ увеличивается на значение 2^{-j} и $c_{ib_ij} = 0$. Поэтому в последующих циклах алгоритма 2м повторяющиеся СБМ сформированы не будут. Теорема доказана.

Для исключения промахов при случайном (псевдослучайном) выборе из строки матрицы **P** ненулевых элементов (например, для создания СБМ) будем осуществлять розыгрыш элементов каждой строки матрицы **P** с помощью вектора с уменьшающейся длиной и состоящего из ненулевых элементов. ВС операций выбора элементов из данного вектора определяется суммой арифметической прогрессии: $(1 + m) \times m/2$, т. е. ВС этой операции равна $O(m^2/2)$.

ВС выбора ненулевых элементов p_{ij} из i-й строки матрицы **P** последовательным образом, $i, j = \overline{0, m-1}$, например, слева направо (в сторону увеличения номеров столбцов **P**), меньше ($O(m)$), чем при псевдослучайном ($O(m^2/2)$), или минимаксном ($O(m^2)$) выборе элементов.

Предложение 3. Вычислительная сложность алгоритма 2м равна $O(m^3b^2)$.

Схема доказательства. Максимально может быть сформировано $m^2 - m + 1$ [3] различных СБМ. На каждом разряде $k = \overline{0, b-1}$ выполняется проверка уникальности (m операций сравнения) текущей СБМ в $b - k - 1$ младших разрядах матрицы **P**. Тогда $(m^2 - m + 1)m\sum\limits_{k=0}^{b-1}(b - k - 1) \approx m^3b^2$. Предложение доказано.

В отличие от алгоритма 2, имеющего ВС, равную $O(m^4b^3)$ (предложение 2), ВС алгоритма 2м снижается до значения $O(m^3b^2)$. ВС алгоритма 2м будет меньше ВС алгоритма 1 при выполнении неравенства $b < \sqrt{m}$.

Теорема 2. Алгоритм 2м для класса стохастических матриц $\tilde{\mathbf{P}}(m, b)$ не результативен, если матрица $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$, $\mathbf{P} = (p_{ij})$, $i, j = \overline{0, m-1}$, обладает одним из следующих свойств:

1) $\exists k = \overline{0, b-1}$, для которого не выполняется условие:

$$\sum_{j=0}^{m-1} c_{0jk} = \sum_{j=0}^{m-1} c_{1jk} = \ldots = \sum_{j=0}^{m-1} c_{(m-1)jk},$$ где

c_{ijk} — значение k-го разряда p_{ij}, $i = \overline{0, m-1}$;

2) $\exists p_{ij} = 1$, $i, j = \overline{0, m-1}$.

Следствие. При результативности алгоритма 2м, для каждой i-й строки матрицы $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$, $i = \overline{0, m-1}$, выполняется условие

$$\sum_{j=0}^{m-1}\sum_{k=0}^{b-1} c_{ijk} = \text{const}.$$

Задача вычисления размера l ИВ для алгоритма 2м сводится к задаче нахождения экстремума числа единиц в двоичных разрядах элементов по строкам матрицы **P**. Условия этой задачи следующие:

- сумма двоичных единиц на каждой i-й строке матрицы **P** равна константе $t = \sum\limits_{j=0}^{m-1}\sum\limits_{k=0}^{b-1} c_{ijk}$, $i = \overline{0, m-1}$;
- условие стохастичности по строкам матрицы **P**: $\sum\limits_{j=0}^{m-1}\sum\limits_{k=0}^{b-1} c_{ijk}2^k = 2^b$;
- число двоичных элементов по строкам находится в диапазоне $2 \leqslant \sum\limits_{j=0}^{m-1}\sum\limits_{k=0}^{b-1} c_{ijk} \leqslant 2^b$.

Теорема 3. Размер l ИВ, получаемого при разложении (1) матрицы **P** по алгоритму 2м, определяется из условий:

$$l \leqslant t, \text{ где } t = \sum_{j=0}^{m-1}\sum_{k=0}^{b-1} c_{ijk},\ i = \overline{0, m-1},$$

$$\text{и } \begin{cases} 2 \leqslant l \leqslant 2^b \text{ при } m \geqslant 2^b, \\ 2 \leqslant l \leqslant m^2 - m + 1 \text{ при } m < 2^b \end{cases}. \quad (4)$$

Доказательство. По следствию из теоремы 2 $t = \sum\limits_{j=0}^{m-1}\sum\limits_{k=0}^{b-1} c_{ijk} = \text{const}$ для каждой i-й строки матрицы **P**, $i = \overline{0, m-1}$. Так как для создания СБМ требуется наличие на каждой i-й строке, на некотором k-м разряде элементов p_{ij} хотя бы по одному $c_{ijk} = 1$, то t задает общее число СБМ в разложении. Тогда размер ИВ ограничен неравенством $l \leqslant \sum\limits_{j=0}^{m-1}\sum\limits_{k=0}^{b-1} c_{ijk}$. Знак неравенства определяет случаи, когда в результате разложения (1) на разных разрядах элементов матрицы **P** получаются одинаковые СБМ.

Рассмотрим нижнюю границу второго условия теоремы: $2 \leqslant l$. Для получения случая, когда в результате разложения получается только одна СБМ, необходимо, чтобы на каждой i-й строке матрицы **P** $\exists p_{ij} = 1$, $i, j = \overline{0, m-1}$. Но по второму условию теоремы 2 данная матрица не разложима в (1) по алгоритму 2м. Случай $l = 2$ будет получен при существовании в каждой i-й строке матрицы **P** двух элементов, равных 1/2, т. е. $2 \leqslant l$.

Теперь рассмотрим верхнюю границу условия теоремы. Для выполнения условия стохастичности по строкам матрицы **P** должно выполняться равенство: $\sum\limits_{j=1}^{m} p_{ij} = 1 = 2^b/2^b$, $i = \overline{0, m-1}$. Пусть

вначале $m \geqslant 2^b$. Условие стохастичности будет выполняться, если значение $\forall p_{ij} > 0$, $i, j = \overline{0, m-1}$, будет равно минимальному значению вероятности $1/2^b$ (у $\forall p_{ij} > 0$ будет установлена единица в младшем разряде). То есть при разложении такой матрицы получится 2^b СБМ.

Перейдем к рассмотрению условия $m < 2^b$. Если будут существовать элементы p_{ij} с двоичными единицами в младшем разряде, то в сумме не будет получено значение 2^b (ввиду малого числа столбцов матрицы **P**). В отличие от случая $m \geqslant 2^b$ остающаяся сумма единиц $2^b - m$ перераспределяется в m элементах каждой i-й строки матрицы **P**, $i = \overline{0, m-1}$. В соответствии с теоремой 4.3, приведенной в работе [3], стохастическую матрицу **P** можно разложить максимум на $m^2 - m + 1$ различных СБМ. Теорема доказана.

Оценка l алгоритма 2м, равная $2 \leqslant l \leqslant 2^b$ при $m \geqslant 2^b$, эквивалентна оценке (3) размера ИВ алгоритма 1 и применима для случая, когда в матрице **P** существуют нулевые элементы. Оценка l алгоритма 2м, равная $2 \leqslant l \leqslant m^2 - m + 1$ при $m < 2^b$, эквивалентна оценке (2) размера ИВ алгоритма 1 для общего случая.

Для случаев, когда ввиду условий 1 или 2 теоремы 2 невозможно разложить в выражении (1) стохастическую матрицу по алгоритму 2м, можно предложить следующий алгоритм (***алгоритм 3***), также учитывающий точность двоично-рационального представления элементов стохастической матрицы.

Пусть дана матрица $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$. Координаты единичных элементов СБМ задаются выбранными по одному из каждой строки элементами p_{ij} матрицы **P**, $i, j = \overline{0, m-1}$, удовлетворяющими условию $p_{ij} > 0$ при минимальном значении j. Новому элементу ИВ, соответствующему сформированной СБМ, присваивается значение 2^{-b}, а из выбранных элементов p_{ij} вычитаются числа 2^{-b}. Алгоритм 3 завершается тогда, когда по строкам, слева направо, будут пройдены и обнулены все элементы матрицы **P**.

В целях уменьшения ВС алгоритма 3 введем следующие изменения:

1) после формирования вектора **B** будем выполнять вычитание из выбранных элементов p_{ib_i} не значения 2^{-b}, а значения $\min\limits_{0 \leqslant i \leqslant m-1} p_{ib_i}$;

2) исключим операцию проверки на повторение появления найденных СБМ, представленных способом 2 (в виде числового вектора ***B***).

Теорема 4. В разложении (1) матрицы **P** по алгоритму 3 каждая из СБМ уникальна.

Теорема 5. Размер l ИВ, получаемого при разложении (1) матрицы **P** по алгоритму 3, определяется из условия $1 \leqslant l \leqslant m^2 - m + 1$.

Доказательство. Рассмотрим нижнюю границу условия теоремы: $1 \leqslant l$. Данное неравенство станет равенством в случае, если матрица **P** является СБМ. Случай $l = 2$ будет получен при существовании в каждой строке матрицы **P** двух элементов, значения которых равны 1/2.

Рассмотрим верхнюю границу условия теоремы: $l \leqslant m^2 - m + 1$. Пусть в матрице **P** все элементы $p_{ij} > 0$, $i, j = \overline{0, m-1}$. На каждом цикле выполняется следующее условие: хотя бы один элемент p_{ij} становится равным нулю. Тогда последовательно можно обнулить все элементы матрицы **P** за m^2 циклов. Но на последнем цикле в каждой строке обнуляется один элемент, так как иначе не будет получена СБМ. То есть максимально может быть выполнено $m^2 - m + 1$ циклов алгоритма 3 и, следовательно, $l \leqslant m^2 - m + 1$. Теорема доказана.

Лемма 1. Алгоритм 3 завершается, если $\exists i$-я строка матрицы $\mathbf{P} = (p_{ij})$, $i, j = \overline{0, m-1}$, для которой выполняется равенство $\sum\limits_{j=1}^{m} p_{ij} = 0$.

Теорема 6. Вычислительная сложность алгоритма 3 равна $O(m^3)$.

Схема доказательства. На каждом цикле алгоритма (максимальное число которых совпадает с размером l ИВ (теорема 5) — $m^2 - m + 1$) из каждой строки (m строк) выбираются элементы $p_{ij} > 0$ матрицы **P**. То есть сложность алгоритма равна $O(m^3)$. Теорема доказана.

В алгоритме 3, как и в алгоритме 1, после получения текущей СБМ, выполняется вычитание числа $\min\limits_{0 \leqslant i \leqslant m-1} p_{ib_i}$ из элементов p_{ij} матрицы **P**, $i, j = \overline{0, m-1}$, выбранных на текущем цикле алгоритма. Но между алгоритмами есть следующие отличия.

1. Для уменьшения ВС на каждой i-й строке выбирается не $\max\limits_{0 \leqslant j \leqslant m-1} p_{ij}$, $i = \overline{0, m-1}$, а элемент $p_{ij} > 0$ с минимальным значением j.

2. В алгоритме 3 используются не дроби, а целые числа a_{ij}, $a_{ij} = 0, 2^b$, числители элементов p_{ij} исходной матрицы **P**, что позволяет исключить округления.

Рассмотрим ***алгоритм 4,*** также выполняющий разложение (1) матриц $\mathbf{P} \in \tilde{\mathbf{P}}(m, b)$, применимый к подклассам матриц, описанных в теореме 2 для алгоритма 2м, снижающий ВС подхода 2 получения разложения (1) и дающий точную оценку параметра l для конкретной матрицы **P**.

Пусть $\mathbf{W}^{(k)} = (w_{ij}^{(k)})$, $w_{ij}^{(k)} = \overline{0, 2^{k+1}}$, $i, j = \overline{0, m-1}$, $k = \overline{0, b-1}$ — матрица, элементы которой содержат значения текущего k-го разряда элементов p_{ij} матрицы $\mathbf{P}$. Если матрица обладает свойством 1 или 2 теоремы 2, сформулированной для алгоритма 2м, то в алгоритме 4 значения элементов $w_{ij}^{(k)}$ матрицы $\mathbf{W}^{(k)}$, на основе которых не удалось сформировать СБМ, с удвоением переносятся в элементы $w_{ij}^{(k+1)}$ матрицы $\mathbf{W}^{(k+1)}$. Для хранения матриц $\mathbf{W}^{(k)}$ требуется $m^2b(b+1)/2$ бит.

Алгоритм 4 состоит из следующих трех блоков.

1. Из исходной матрицы $\mathbf{P}$ формируются матрицы $\mathbf{W}^{(k)} = (w_{ij}^{(k)})$, $i, j = \overline{0, m-1}$, $k = \overline{0, b-1}$, по следующему правилу: если $p_{ij} = 1$, то $w_{ij}^{(0)} = 2$; иначе $w_{ij}^{(k)} = c_{ijk}$.

2. При несоблюдении условия $\sum_{j=0}^{m-1} w_{0j}^{(k)} = \sum_{j=0}^{m-1} w_{1j}^{(k)} = \ldots = \sum_{j=0}^{m-1} w_{(m-1)j}^{(k)}$, $k = \overline{0, b-2}$, значения элементов $w_{ij}^{(k)}$, соответствующие k-му разряду элементов p_{ij} матрицы $\mathbf{P}$, с удвоением переносятся в соответствующие элементы $w_{ij}^{(k+1)}$ матрицы $\mathbf{W}^{(k+1)}$, так как справедливо равенство $1/2^a = 2^k/2^{a+k}$, где a — натуральное число.

3. Текущая СБМ $\mathbf{B}^{(l)}$ для $\forall k = \overline{0, b-1}$ формируется из элементов $w_{ij}^{(k)} > 0$ матрицы $\mathbf{W}^{(k)}$ с минимальным значением j. Осуществляется проверка повторения матрицы $\mathbf{B}^{(l)}$ в матрицах $\mathbf{W}^{(j)}$, $j = \overline{k+1, b-1}$. При получении СБМ $\mathbf{B}^{(l)}$ в матрице $\mathbf{W}^{(k)}$ в ИВ $\tilde{\boldsymbol{q}}$ добавляется новый элемент $q_l = \min\limits_{0 \le i \le m-1} w_{ib_i^{(l)}}^{(k)} 2^{-(k+1)}$, а при нахождении $\mathbf{B}^{(l)}$ в матрице $\mathbf{W}^{(j)}$ к q_l добавляется $\min\limits_{0 \le i \le m-1} w_{ib_i^{(l)}}^{(j)} 2^{-(j+1)}$.

Лемма 2. Алгоритм 4 конечен.

Теорема 7. Вычислительная сложность алгоритма 4 равна $O(m^2b^2)$.

Схема доказательства. Для каждой матрицы $\mathbf{W}^{(k)}$, $k = \overline{0, b-1}$ (число матриц равно b), выполняется обход ее элементов (m^2 операций) и проверка полученной СБМ $\mathbf{B}^{(l)}$ (для каждого k может быть m СБМ) на уникальность в матрицах $\mathbf{W}^{(j)}$, $j = \overline{k+1, b-1}$ (число матриц $\mathbf{W}^{(j)}$ равно $b-k-1$, а операция проверки на уникальность требует m сравнений чисел). Тогда получаем, что $bm^2 + m^2 \sum_{k=0}^{b-1} (b-k-1) \approx bm^2 + b^2m^2$. То есть ВС алгоритма равна $O(m^2b^2)$. Теорема доказана.

Получим, что при значениях b, меньших m, ВС алгоритма 4 будет меньше, чем ВС других рассмотренных алгоритмов.

Лемма 3. В разложении (1) матрицы $\mathbf{P}$ по алгоритму 4 каждая из СБМ уникальна.

Теорема 8. Размер l ИВ, получаемого при разложении (1) матрицы $\mathbf{P}$ по алгоритму 4, определяется из условий

$$l \le \sum_{j=0}^{m-1} \sum_{k=0}^{b-1} w_{ij}^{(k)}, \sum_{j=0}^{m-1} \sum_{k=0}^{b-1} w_{ij}^{(k)} = \text{const}, \; i = \overline{0, m-1},$$

$$\text{и} \begin{cases} 2 \le l \le 2^b \text{ при } m \ge 2^b, \\ 2 \le l \le m^2 - m + 1 \text{ при } m < 2^b \end{cases}, \qquad (5)$$

где $w_{ij}^{(k)}$, $i, j = \overline{0, m-1}$, $k = \overline{0, b-1}$ — элементы матриц $\mathbf{W}^{(k)}$ после выполнения шага 2 алгоритма 4.

Доказательство данной теоремы аналогично доказательству теоремы 3, сформулированной для алгоритма 2м. Отличие заключается в том, что вместо переменных $c_{ijk} \in \{0; 1\}$ используются элементы $w_{ij}^{(k)} = \overline{0, 2^{k+1}}$ матриц $\mathbf{W}^k$.

Полученная в алгоритме 4 оценка размера l ИВ $\tilde{\boldsymbol{q}}$ не меньше аналогичной оценки алгоритма 1, но

Алгоритм	Оценка l	Оценка ВС
1	$1 \le l \le m^2 - m + 1$ [4, 5] и $\max\limits_{0 \le i \le m-1} d_i \le l \le \sum\limits_{i=0}^{m-1} d_i - m + 1$ [8, 9]	$O(m^4)$
2	—	$O(m^4b^3)$
2м	$l \le \sum\limits_{j=0}^{m-1} \sum\limits_{k=0}^{b-1} c_{ijk}, \sum\limits_{j=0}^{m-1} \sum\limits_{k=0}^{b-1} c_{ijk} = \text{const}$, $i = \overline{0, m-1}$, и $\begin{cases} 2 \le l \le 2^b \text{ при } m \ge 2^b \\ 2 \le l \le m^2 - m + 1 \text{ при } m < 2^b \end{cases}$	$O(m^3b^2)$
3	$1 \le l \le m^2 - m + 1$	$O(m^3)$
4	$l \le \sum\limits_{j=0}^{m-1} \sum\limits_{k=0}^{b-1} w_{ij}^k, \sum\limits_{j=0}^{m-1} \sum\limits_{k=0}^{b-1} w_{ij}^k = \text{const}$, $i = \overline{0, m-1}$, и $\begin{cases} 2 \le l \le 2^b \text{ при } m \ge 2^b \\ 2 \le l \le m^2 - m + 1 \text{ при } m < 2^b \end{cases}$	$O(m^2b^2)$

позволяет для конкретной матрицы **P** получить следующую более точную оценку l по формуле (5):

$$l \leqslant \sum_{j=0}^{m-1} \sum_{k=0}^{b-1} w_{ij}^{(k)}.$$

Заключение

Рассмотрены алгоритмы двух подходов разложения стохастических матриц в вид (1): подход, основанный на методе минимакса, и двоично-рациональный подход. Для алгоритмов даны оценки вычислительной сложности и размера l имплицирующего вектора (см. таблицу), показаны конечность алгоритмов и уникальность формируемых стохастических булевых матриц. Предложенные алгоритмы позволяют снизить сложность вычисления разложения (1), определить методики оценки параметра l. Предложено решение (алгоритмы 3 и 4) по устранению выявленных ограничений метода [3] на вид стохастических матриц, представленных двоично-рациональными дробями.

Уменьшение ВС алгоритмов осуществляется за счет:

- последовательного выбора элементов из строк;
- представления СБМ в виде вектора чисел (способ 2);
- снижения точности представления элементов матрицы **P** (значения b).

Параметры алгоритмов второго подхода — ВС и l — зависят не только от размера m и энтропии $H(\mathbf{P})$ стохастических матриц **P**, но и от числа разрядов b элементов матриц **P**. ВС данных алгоритмов снижаются при уменьшении значения b, т. е. при понижении точности представления элементов матрицы **P**. Минимальную ВС среди представленных алгоритмов имеет алгоритм 4. Из выражений (4) и (5) видно, что уменьшение параметра b (уменьшение диапазона переменной k в знаке суммы) приводит к уменьшению значения l.

Список литературы

1. **Davis A. C.** Markov chains as random input automata // Ann. Amer. Math. Monthly. 1961. V. 68. N 3. P. 264—267.
2. **Бухараев Р. Г.** Основы теории вероятностных автоматов. М.: Наука, 1985. 287 с.
3. **Бухараев Р. Г., Захаров В. М.** Управляемые генераторы случайных кодов. Казань: Изд-во Казан. ун-та, 1978. 160 с.
4. **Ченцов В. М.** Об одном методе синтеза автономного стохастического автомата // Кибернетика. — Киев, 1968. № 3. С. 32—35.
5. **Поспелов Д. А.** Вероятностные автоматы. М.: Энергия, 1970. 88 с.
6. **Захаров В. М., Нурутдинов Ш. Р., Соколов С. Ю., Шалагин С. В.** Полиномиальное представление конечноавтоматных случайных последовательностей над полем Галуа // Вестник Казан. гос. техн. ун-та им. А. Н. Туполева. Вып. 2. Казань: Изд-во КГТУ им. А. Н. Туполева, 2003. С. 24—28.
7. **Захаров В. М., Эминов Б. Ф.** Анализ нелинейных моделей преобразователей случайных последовательностей над полем $GF(2^n)$ на основе стохастических матриц // Вестник Казан. гос. техн. ун-та им. А. Н. Туполева. Вып. 3. Казань: Изд-во КГТУ им. А. Н. Туполева, 2006. С. 41—46.
8. **Габбасов Н. З.** Задача об имплицирующем векторе и ее приложения // Вероятностные методы и кибернетика. Вып. 23. Казань: Изд-во Казан. ун-та, 1987. С. 36—50.
9. **Кузнецов С. Е., Нурмеев Н. Н., Салимов Ф. И.** Задача о минимальном имплицирующем векторе // Математические вопросы кибернетики: Вып. 3. Сб. статей / Под ред. С. В. Яблонского. М.: Наука, 1991. С. 199—216.
10. **Кемени Дж., Снелл Дж.** Конечные цепи Маркова. М.: Наука, 1970. 272 с.

УДК 004.3.06(075)

А. Д. Десятов, А. А. Сирота, д-р техн. наук, проф.,
Воронежский государственный университет

Имитационное моделирование систем с адаптивной структурой на основе технологий автоматизированного создания моделей в среде MatLab + Simulink + Stateflow

Описывается применение интегрированной инструментальной среды MatLab + Simulink + Stateflow для моделирования систем с адаптивной структурой. Особое внимание уделяется вопросам автоматизированного создания многоэлементных, многоуровневых моделей систем с изменяемой структурой на основе базовых компонентов. В качестве примера такой системы рассматривается служба поддержки на предприятии в сфере оказания информационных услуг.

Введение

В настоящее время система MatLab, помимо развитого языка для проведения математических вычислений, предоставляет широкий набор средств для решения прикладных задач в различных областях науки и техники. Наличие подсистемы Simulink для визуального программирования, имитации и анализа динамических систем

делает MatLab одним из самых мощных средств имитационного моделирования. Включаемый в Simulink пакет Stateflow позволяет описывать событийно-управляемые системы в виде SF-диаграмм состояний, опирающихся на формализм гибридных автоматов (карт состояний Харела [1—3]). Одновременно присутствует возможность интеграции среды с компонентами, написанными с помощью современных универсальных языков программирования (Java, C++ и т. д.), что позволяет исследовать стратегии поведения систем с использованием общепринятых шаблонов объектно-ориентированного программирования. Таким образом, имеет смысл говорить об интегрированной среде, содержащей большое число готовых компонентов для исследования систем различного назначения.

В то же время возможности среды в области моделирования раскрыты еще не полностью. В практике создания имитационных моделей событийно-управляемых систем часто возникает задача исследования многоэлементных систем с адаптивно изменяющейся в процессе симуляции структурой. Для решения этой задачи необходимо разработать две технологии:

- генерации в автоматизированном режиме SF-модели произвольной структуры, реализующей использование заданного набора функционально законченных элементов системы (модели);
- изменения структуры модели под воздействием соответствующих управляющих команд непосредственно в процессе симуляции.

В настоящей статье описывается реализация подобного механизма на примере моделирования службы поддержки предприятия в сфере оказания информационных услуг.

Функциональная модель системы

Служба поддержки (СП) на предприятии (*Service Desk*) — это некоторая диспетчерская служба, которая в полной мере ответственна перед клиентом за предоставление согласованных с ним сервисов. Аналогичная структура может также именоваться "Центром приема сообщений" (*Call Centre*), "Центром технической поддержки" (*Technical Support Center*), "Центром диспетчерской помощи клиентам" (*Help Desk*) и т. п. [4]. Различие в наименовании несет с собой, как правило, некоторые функциональные особенности, не принципиальные в рамках данного рассмотрения. Функционирование подобной системы в виде диаграммы деятельности представлено на рис. 1.

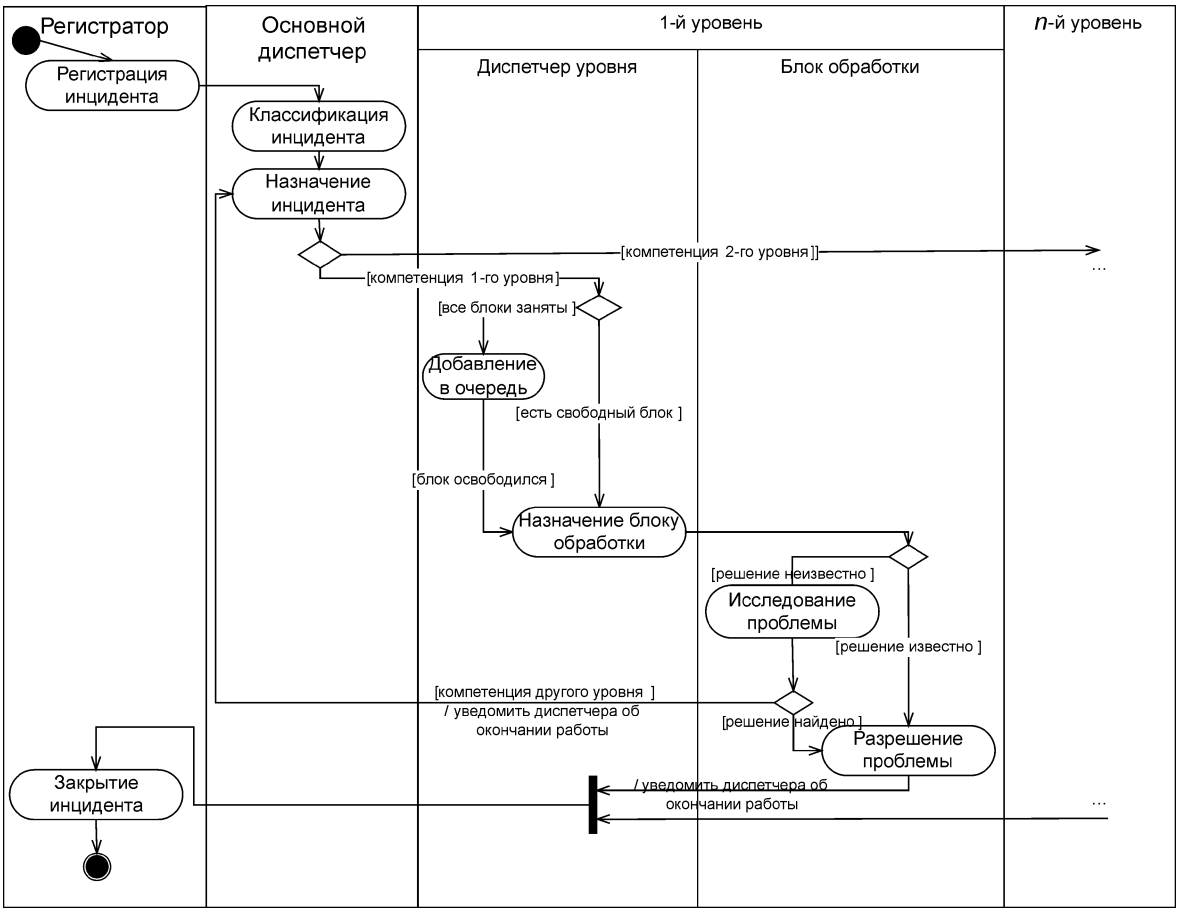


Рис. 1. Схема работы многоуровневой службы поддержки

Процесс обработки заявки начинается с обнаружения ошибки в системе или регистрации уведомления о неисправности от клиентов сервиса (блок "Регистрация инцидента"). СП отвечает за процесс разрешения всех зарегистрированных инцидентов (заявок) и является его владельцем. Устранение проблемы или поиск временного решения должны быть проведены как можно быстрее, чтобы потери пользователей сервиса были минимальными. В зависимости от сложности информационных сервисов на предприятии могут существовать группы, занимающие различные уровни поддержки в процессе обслуживания клиентов. Диагностирование и исследование представляет итеративный процесс, который начинается в одной группе специалистов и сопровождается устранением рассмотренных причин из списка возможных (первый уровень поддержки) или передается другой группе с более высокой квалификацией. Проблемы, которые не могут быть решены СП самостоятельно, могут назначаться специалистам сторонних организаций. После устранения причины инцидента и восстановления функционирования сервиса инцидент закрывают [4].

Следует отметить, что на рис. 1 приведена схема, в которой на первом уровне показан только один блок обработки. В реальных системах подобных блоков может быть множество, причем их конфигурация может меняться как за счет использования дополнительных блоков обработки на каждом уровне поддержки, так и за счет увеличения числа уровней. В связи с этим организация гибкой структуры СП является решающим фактором, определяющим эффективность ее функционирования. В зависимости от характеристики входного потока сервис-заявок может возникнуть необходимость изменения структуры, например, добавление на определенный уровень дополнительных блоков обработки. Для исследования различных стратегий организации СП с изменяющейся структурой целесообразно использовать методы и средства имитационного моделирования. Отдельный интерес представляет анализ поведения системы на участках перехода от одной структуры к другой.

Технология автоматизированной генерации модели с использованием Stateflow API

Рассмотрим традиционный способ визуального конструирования SF-моделей [1—3]. Для построения модели заданной структуры пользователь на основе технологии "drag-and-drop" добавляет используемые компоненты в окно графического редактора SF-модели (изначально открытой из Simulink), настраивает их параметры, устанавливает взаимосвязи между ними, а также с Simulink и MatLab и запускает модель на исполнение. Если необходимо изменить структуру модели, то пользователь останавливает процесс и повторяет эти действия в рамках новой модели. Описанный подход требует весьма существенных затрат, если требуется создавать модель для системы, состоящей из большого числа элементов. Учитывая, что невозможно предусмотреть и создать все возможные варианты построения СП, задача сводится к генерации модели с использованием ограниченного набора базовых компонентов, в которых определяются основные функциональные элементы системы и устанавливаются отношения между ними.

Для решения задачи генерации многоэлементной модели на основе базовых компонентов предлагается использовать инструмент Stateflow API (*Stateflow Application Programming Interface*), позволяющий создавать и изменять Stateflow-диаграммы посредством команд пакета MatLab. Stateflow API предоставляет возможность работы с основными графическими и неграфическими объектами диаграмм: State, Transition, Junction и т. д. Каждый из этих объектов характеризуется определенными методами и свойствами, путем изменения которых происходит редактирование объектов диаграммы. Таким образом, становится возможной автоматизация процесса создания достаточно сложных моделей систем в Stateflow.

Общая схема предлагаемой технологии создания модели представлена на рис. 2. В ней сначала создается некоторая базовая модель в виде набора SF-диаграмм, которая служит "заготовкой" для последующего создания более сложной модели. Основным же здесь является написание специальной процедуры генерации (script-файл на языке MatLab), оперирующей базовыми объектами диаграмм с помощью Stateflow API. Процедура генерации получает на вход базовую модель, содержащую основные функциональные блоки, и выполняется до запуска модели. В результате ее выполнения формируется многоуровневая, многоэлементная Stateflow-модель, соответствующая структуре имитируемой системы, на которой впо-

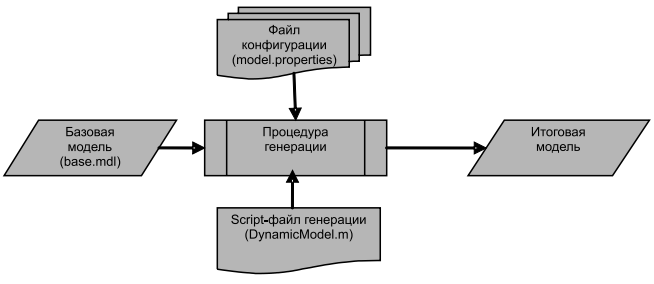


Рис. 2. Схема процесса создания многоуровневой модели

следствии проводятся испытания. Вынесение всей необходимой информации относительно конфигурации создаваемой модели (число уровней поддержки, число блоков обработки на уровне) в отдельный файл (model.properties) позволяет локализовать данные, необходимые для автоматизированного создания модели.

В основе процедуры генерации лежит метод, позволяющий создавать копии существующих объектов диаграммы. В этом плане Stateflow API предоставляет эффективный способ — копирование сгруппированных состояний (*Copying by Grouping*). При копировании сгруппированного состояния копируются и все дочерние состояния и переходы.

Рассмотрим реализацию этой технологии на примере построения модели СП. Первоначально при отработке технологии были созданы базовые SF-компоненты, отвечающие исходной модели функционирования СП, представленной на рис. 1. Поскольку СП является, по существу, системой массового обслуживания (СМО), данные блоки реализуют ее активные, активно-пассивные и пассивные элементы с управляемыми параметрами в среде MatLab + + Simulink + Stateflow, принципы создания которых подробно рассмотрены в [3].

На рис. 3 представлена S-модель, являющаяся носителем предлагаемой SF-модели (блок Chart). При построении S-модели, в которой используются блоки Stateflow, содержащие в себе переходы, инициируемые истечением некоторых интервалов времени (что характерно для СМО), требуется обеспечить синхронизацию внутреннего системного времени Stateflow и Simulink. Для этого в качестве входного параметра в SF-диаграмму подается системное время (*systime*) из Simulink, и это время используется для составления условий переходов. Реализация с внешним источником предоставляет дополнительные возможности [3] применения неравномерной шкалы системного времени. Входной поток заявок также формируется в Simulink и подается на вход SF-модели в виде последовательности событий-заявок. Поток формируется на выходе блока Fcn, реализующего логическое сравнение входного сигнала, имеющего равномерное распределение в диапазоне значений [0, 1], с величиной $Pi = Li \cdot Ts$, зависящей от интенсивности потока заявок (Li) и от интервала дискретизации (Ts). Каждое событие, связанное с поступлением входной заявки, определяется в SF-модели под именем newRequest (*Input from Simulink event*). Для активизации SF-диаграммы на каждом шаге изменения времени S-модели в блок-диаграмме рис. 3 вводится генератор событий (CLK Generator). Процесс функционирования SF-модели характеризуется визуальным отображением активных состояний и переходов, что позволяет наглядно определить динамику работы системы. Процессы и события на выходе SF-модели визуализируются в блоках Scope S-модели.

Продемонстрируем построение SF-модели на примере копирования уровня обслуживания базовой модели рис. 1 (компонент ServiceLevel, соответствующий уровню с одним блоком обработки). Уровень обслуживания представлен на рис. 4 и имеет два параллельно функционирующих основных блока: блок Dispatcher ("Диспетчер уровня"

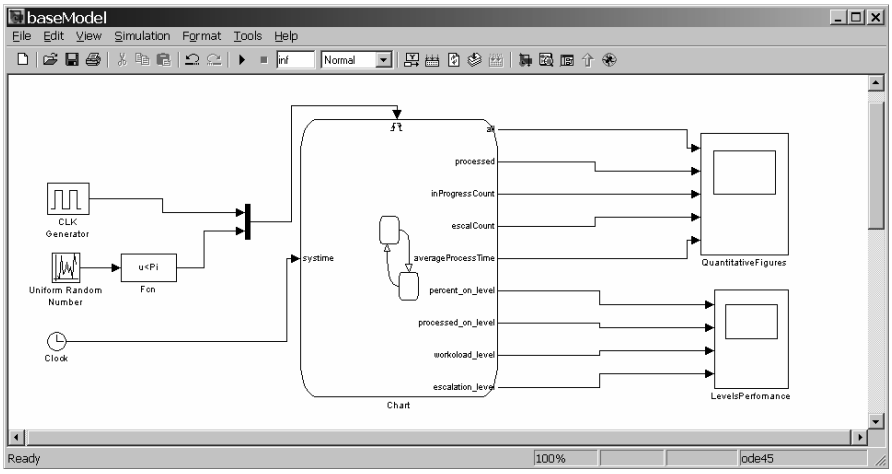


Рис. 3. Схема генерации событий в S-модели

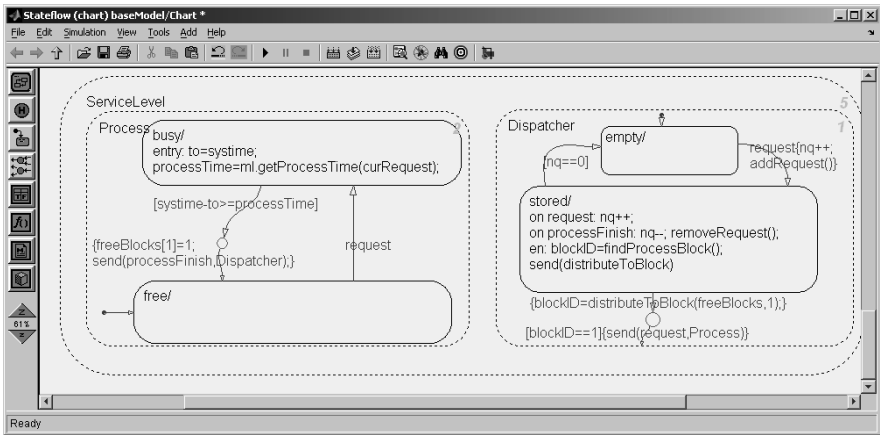


Рис. 4. Схема работы уровня обслуживания

на рис. 1), где осуществляется хранение поступающих заявок и их распределение между блоками обработки на уровне, и блок Process ("Блок обработки" на рис. 1), непосредственно реализующий обработку. Каждый из них, в свою очередь, содержит взаимоисключающие состояния более низкого уровня, переход между которыми осуществляется при наступлении определенных событий в системе. Так, при получении события request (статус Local) происходит переход блока Process в состояние busy, а Dispatcher оказывается в состоянии empty после завершения обработки в блоке Process (получение события processFinish (статус Local)) при условии отсутствия заявок в очереди.

Имея базовую модель, можно перейти к реализации предлагаемого механизма построения многоуровневых моделей с использованием средств Stateflow API. Ниже приведен пример копирования уровня обслуживания ServiceLevel — создание дополнительно уровня на основе существующего:

```
% получаем значение свойства IsGrouped для состояния ServiceLevel
prevGrouping = ServiceLevel.IsGrouped
% если объект не был сгруппирован, сгруппируем его
if (prevGrouping == 0}
    ServiceLevel.IsGrouped = 1
end
% получаем указатель на буфер (копирование происходит через буфер)
cb = afclipboard
% копируем сгруппированный объект в буфер
cb.copy(ServiceLevel)
% вставляем объект из буфера на диаграмму
cb.pasteTo(chart)
% устанавливаем значение свойства IsGrouped в предыдущее состояние
ServiceLevel.IsGrouped = prevGrouping
```

Тот же самый подход копирования сгруппированных состояний применяется и для тиражирования необходимого числа других блоков обработки на каждом уровне поддержки.

Помимо копирования состояний диаграммы Stateflow для получения работоспособной функциональной модели необходимо установить соответствующие переходы между состояниями. Для получения требуемого перехода создается объект Transition диаграммы Stateflow. У созданного объекта устанавливается исходное и конечное состояния. Для задания условий перехода, а также действий, совершаемых при переходе, используется атрибут LabelString:

```
% создаем объект перехода на текущей диаграмме
transitionHandle = Stateflow.Transition(chart);
% устанавливаем исходное состояние для перехода
dispatcherHandle = chart.find('-isa', 'Stateflow.State', 'and', 'Name', 'Dispatcher');
transitionHandle.Source = dispatcherHandle;
% устанавливаем конечное состояние для перехода
sLevel = chart.find('-isa', 'Stateflow.State', '-and', 'Name', levelName);
transitionHandle.Destination = sLevel;
% задаем условие перехода
transitionHandle.LabelString = strcat ('[currentLevel == ', levelName, ']');
```

В приведенном примере метод chart.find() используется для поиска требуемых объектов на Stateflow диаграмме.

Механизм изменения структуры модели в процессе симуляции

Для решения поставленной задачи помимо автоматизированного конструирования моделей посредством Stateflow API необходимо обеспечить возможность перестройки структуры модели в процессе симуляции. Наличие данного механизма позволяет исследовать различные аспекты поведения моделируемой системы при добавлении/удалении элементов, а именно: изменение показателей эффективности при изменении структуры; динамику перехода системы в стабильный режим работы; загрузку блоков обработки при пиковой нагрузке.

Строго говоря, MatLab на уровне языка в явном виде не поддерживает изменение структуры модели в режиме симуляции. Однако данное ограничение можно обойти путем использования управляющей программы (УП), написанной в MatLab, назначением которой является управление режимом симуляции. Эти возможности предоставляет Simulink API, имеющий необходимый набор методов, позволяющих динамически запускать или останавливать модель.

Например, запуск модели осуществляется из УП следующим образом:

set_param('<*mdl_name*>', '*SimulationCommand*', '*start*'),

где '<*mdl_name*>' — имя модели;

'*SimulationCommand*' — имя изменяемого параметра;

'*start*' — новое значение параметра.

Соответственно, для останова модели выполняется команда

set_param('<*mdl_name*>', '*SimulationCommand*', '*stop*').

Нерешенной пока остается проблема сохранения данных — результатов работы модели при останове симуляции, а также их восстановление при запуске после изменения структуры. Для этих целей вполне естественно использовать MatLab Workspace — хранилище, содержащее множество переменных, созданных в процессе работы про-

граммы. Объекты Stateflow-диаграммы при этом необходимо настроить таким образом, чтобы инициализация значений переменных, а также сохранение их значений при окончании симуляции происходили с использованием переменных, определенных в Workspace. Так, для корректного восстановления состояния системы после изменения структуры необходимо сохранять число заявок в очереди на каждом уровне обслуживания. Для реализации хранения этих данных в Workspace необходимо осуществить настройку параметров переменной, соответствующей числу заявок в очереди, в Model Explorer на закладке Value Attributes. При сохранении данных в Workspace необходимо выполнить установку параметра 'Save final value to base workspace', а для последующей инициализации данных при запуске модели нужно установить значение атрибута 'Initialize from' в значение 'Workspace'. Отметим, что доступ к базовому хранилищу Workspace имеют как объекты SF-диаграммы, так и содержащая ее S-модель, что позволяет сохранить текущее состояние всех объектов интегрированной модели.

Использование Workspace обеспечивает сохранение результатов работы модели во внешней памяти и не ограничивает время жизни переменных рамками одной сессии MatLab. Таким образом, в среде принципиально возможно сохранить состояние перед формированием управляющего воздействия с последующим восстановлением исходного состояния в будущем, что облегчает отладку модели и снимает необходимость ожидания момента перехода системы в стабильный режим работы после запуска.

Общая схема, реализующая данный подход, представлена на диаграмме последовательности рис. 5. В ходе изменения структуры удаляются и/или добавляются блоки обработки и, следовательно, может изменяться размерность пространства состояний. При этом очевидно, что процесс перехода должен базироваться на определенном наборе формальных правил, регламентирующих действия, выполняемые при изменении структуры системы:

- все заявки, находящиеся в ожидании, сохраняются, но прекращается их прием на обслуживание;
- останов модели для перестройки структуры осуществляется в тот момент, когда заканчивается обслуживание заявок на всех уровнях;
- заявки, находящиеся в очередях, распределяются на каждом уровне равномерно между старыми и новыми элементами обслуживания.

Рассмотрим процесс перестройки модели на следующем примере. Пусть к определенному моменту времени была сформирована модель трехуровневой службы поддержки, схема которой приведена на рис. 6 (исходная модель представлена элементами, изображенными сплошной линией). Регистратор и основной диспетчер на рис. 6 не

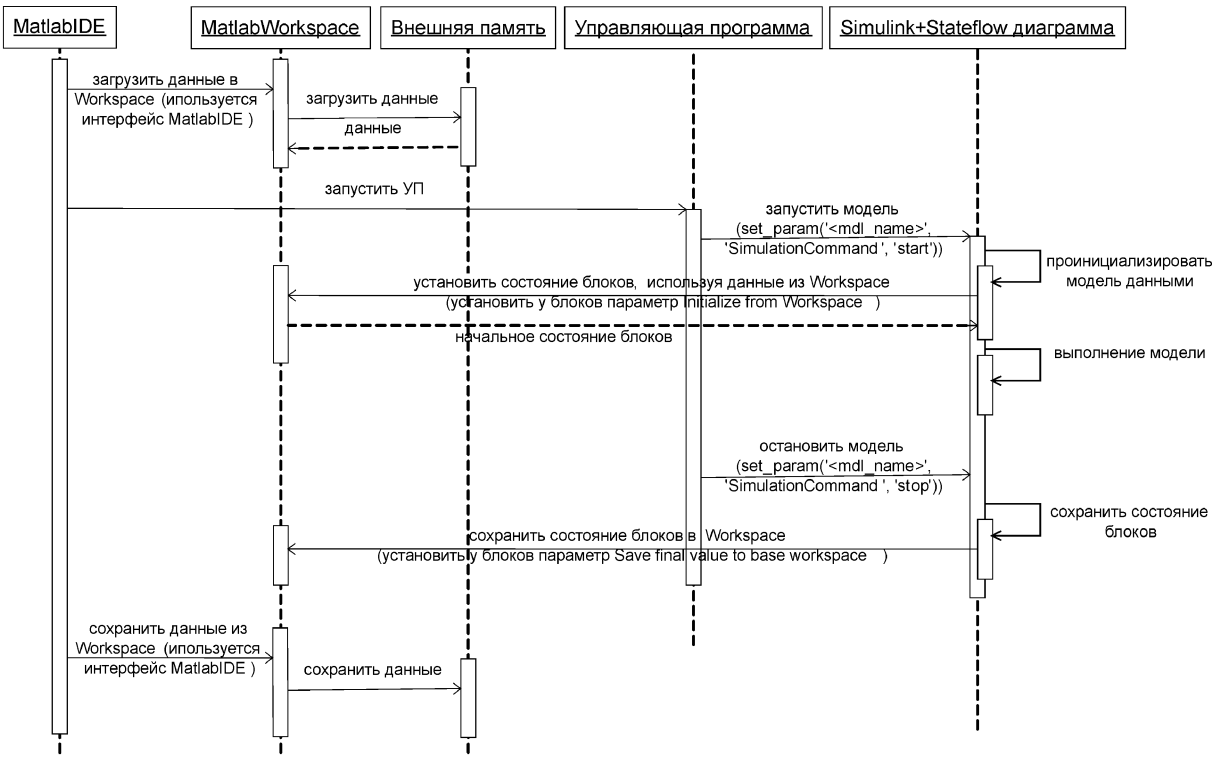


Рис. 5. Сохранение состояния модели во внешней памяти

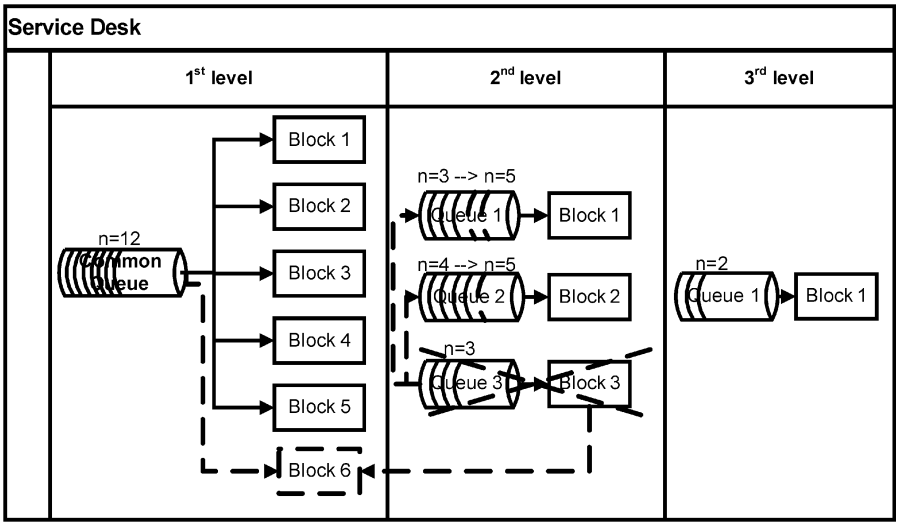


Рис. 6. Схема службы поддержки до и после изменения структуры

изображены, поскольку рассматривается оптимизация работы уровней обслуживания. Заявки в системе распределяются на уровнях следующим образом:

- на первом уровне решаются достаточно простые задачи и все блоки обработки в состоянии обслужить приходящие заявки (или же передать на следующий уровень поддержки в случае недостатка квалификации). Поэтому здесь не требуется формировать отдельную очередь для каждого блока, и действует общая очередь всего уровня, заявки из которой равномерно распределяются между блоками обработки (1st level на рис. 6);
- на втором и третьем уровнях уже есть привязка заявок к конкретному блоку обработки и, следовательно, здесь очереди формируются для каждого отдельного элемента (2nd level и 3rd level на рис. 6). В то же время соответствие заявок блоку обработки не является жестким, т. е. допускается переназначение заявки в рамках уровня поддержки, что позволяет организовать гибкий алгоритм распределения очереди.

В примере, отображаемом на рис. 6, наибольший объем заявок в очереди накоплен на первом уровне (n = 12), в то время как загрузка на втором и третьем уровнях достаточно слабая. Таким образом, первый уровень является "узким местом" с точки зрения производительности системы. Количественное исследование эффективности работы системы дает тот же результат: среднее время обработки заявок превышает установленный порог при сравнительно низком коэффициенте использования блоков обработки на втором уровне.

Перестроим структуру СП в соответствии с возникшей критической ситуацией. Для нормализации работы необходимо в терминах предметной области переместить одного человека (группу специалистов) со второго уровня поддержки на первый. Изменение структуры СП представлено на рис. 6 штриховыми линиями. Конкретно в процессе перестройки были проведены следующие действия: перемещение блока обработки со второго на первый уровень; равномерное распределение заявок, находящихся в очереди перемещаемого блока, между другими элементами второго уровня поддержки. При этом добавление дополнительного элемента (Block 6) не повлекло за собой других изменений на первом уровне, поскольку механизм распределения заявок при наличии общей очереди не связан с конкретным элементом и не зависит от числа элементов.

Динамическое изменение структуры с использованием описанного механизма позволило повысить эффективность системы. Это видно из графика на рис. 7, где представлено усредненное значение времени обслуживания заявок, полученное на основе 100 прогонов модели, в рамках каждого из которых исследуется работа СП в течение 10 ч. Интенсивность потока заявок около 40 заявок в час (данные соответствуют практике функционирования реальной службы поддержки). Вертикальной штриховой линией на графике отмечен момент перестроения структуры.

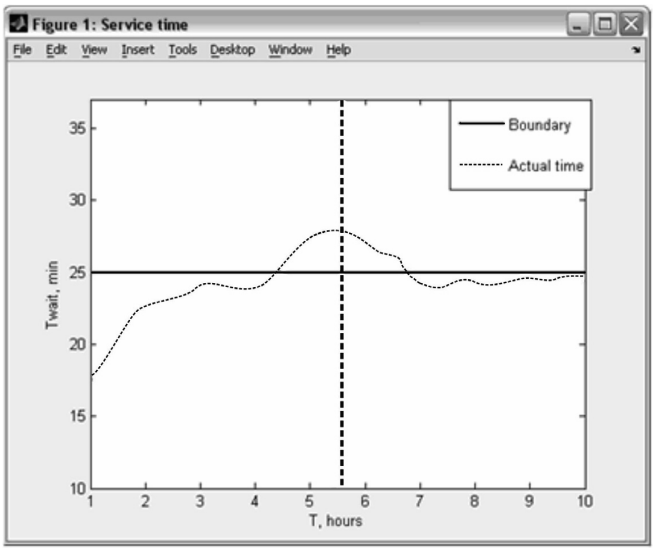


Рис. 7. Среднее время обслуживания заявок

Таким образом, в статье предложен механизм автоматизированного построения моделей многоэлементных систем в среде MatLab + Simulink + + Stateflow, обеспечивающий исследование поведения систем с переменной структурой в динамике функционирования. Предлагаемая технология апробирована на примере модели сервисной службы поддержки предприятия в сфере оказания информационных услуг.

Список литературы

1. **Бенькович Е. С., Колесов Ю. Б., Сениченков Ю. Б.** Практическое моделирование динамических систем. СПб.: БХВ — Петербург, 2002. 464 с.
2. **Гультяев А. А.** Визуальное моделирование в среде MatLab: учебный курс. СПб.: Питер, 2000. 432 с.
3. **Сирота А. А.** Компьютерное моделирование и оценка эффективности сложных систем. М.: Техносфера, 2006. 280 с.
4. **Алехин З.** Service Desk — цели, возможности, реализации // Открытые системы. 2001. № 5—6. С. 43—48.

БЕЗОПАСНОСТЬ ИНФОРМАЦИИ

УДК 004.491

С. М. Авдошин, канд. техн. наук, проф.,
А. С. Зайцев,
Государственный университет —
Высшая школа экономики, г. Москва

Технология извлечения функциональной нагрузки вредоносного кода

Рассмотрена технология автоматизированного выявления действий вредоносных программ, заключающаяся в пошаговом извлечении поведенческих сигнатур и организации их в каталоги поведения. Эта технология многократно облегчает работу эксперта, анализирующего функциональную нагрузку вредоносного кода.

Введение

Сложность вредоносного кода постоянно растет. В свою очередь, необходимость своевременного и однозначного понимания потенциальных угроз, связанных с распространением вредоносного программного обеспечения, растет еще быстрее. Сегодня анализ действий любой программы, в том числе вредоносной, связан с колоссальными затратами времени и усилий квалифицированных, а следовательно, дорогостоящих экспертов. Даже в идеализированном случае абсолютно точного толкования специалистом всех возможных действий программы в любых ситуациях, время, затраченное на анализ, может быть неудовлетворительно большим и издержки субъекта, нанимающего эксперта, за период анализа функционала (а с точки зрения заказчика — период простоя системы, пораженной вредоносным программным обеспечением) могут быть недопустимо велики.

Как же сделать данный анализ более надежным и быстрым?

Существующие технологии анализа, такие как автоматизированное тестирование [1] и проверка на моделях [2], позволяют для некоторого набора ситуаций получить полезную информацию. Однако для разработки эффективных средств противодействия, специалисту необходимо полное и всеобъемлющее понимание поведения анализируемого программного кода.

Технология извлечения функциональной нагрузки обеспечивает максимально возможную глубину анализа, раскрывая поведение программного кода посредством применения математического подхода.

Основы технологии извлечения функциональной нагрузки

Функционально-теоретическая модель программного обеспечения рассматривает программы, как правила для математических функций или отношений. Целью автоматического вычисления поведения программ является извлечение картины полного функционального поведения программ, т. е. идентификация и подробная детализация процессов преобразования входных данных в выходные во всех возможных ситуациях с последующим четким представлением вычлененного набора поведений в удобной для анализа форме.

Основа технологии извлечения функциональной нагрузки лежит в понимании того факта, что в то время как большие программы содержат фактически бесконечное число возможных путей выполнения, они все же построены из конечного числа вложенных и упорядоченных структур

управления, из которых и складывается конечный сценарий поведения программы.

Эти структуры задают математические функции или отношения, т. е. отображения входного набора данных на выходной. Заданные отображения могут быть автоматически извлечены посредством пошагового процесса обхода конечной иерархии управляющих структур.

На каждом шаге локальные детали кода и данных исключаются из рассмотрения, в то время как эффекты их взаимодействия сохраняются и приводятся в извлеченном сценарии поведения. Так как не существует общего подхода к вычленению циклов, для этой цели применяют рекурсивные выражения и паттерны циклов.

При более подробном рассмотрении функционально-теоретическая основа подразумевает генерацию уравнений, показывающих результирующее влияние управляющих структур на данные и обеспечивающих отправную точку для реализации механизма извлечения поведения. Эти уравнения представлены в терминах функциональной композиции, *case*-анализа и для итерационных структур — рекурсивного выражения, основанного на эквивалентном представлении этих структур, как композиции ветвления и итерации. Ниже приводятся уравнения [3], в которых P — управляющая структура; g и h — операции над данными; q — предикат; f — функция программы; o — оператор композиции, | — оператор "ИЛИ", функция программы также заключается в квадратные скобки.

Функция программы — следование (P: g; h): f = [P] = [g; h] = [h] o [g] (рис. 1).

Данное уравнение показывает, что функция следования может быть вычислена путем обычной функциональной композиции своих составных частей.

Функция программы — ветвление (P: if q then g else h fi): f = [P] = [if q then g else h fi] = = ([q] = true → [g] | [q] = false → [h]) (рис. 2).

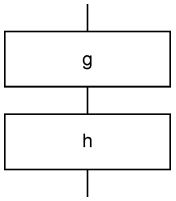


Рис. 1. Функция f — следование

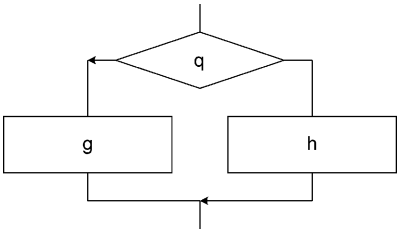


Рис. 2. Функция f — ветвление

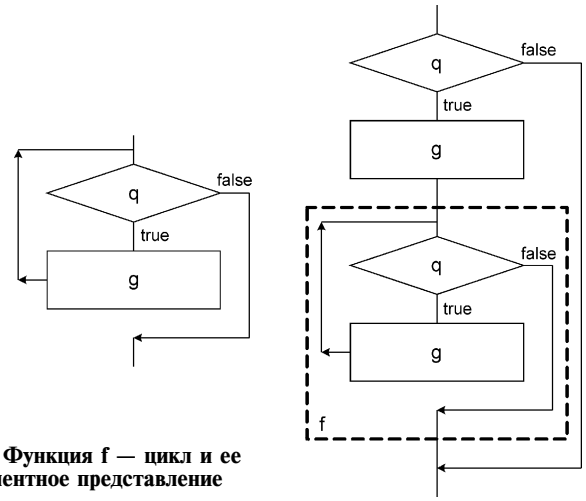


Рис. 3. Функция f — цикл и ее эквивалентное представление

Это уравнение дает возможность сведения к одиночной конструкции функции программы, заданной как case-анализ true- и false-ветвей.

Функция программы — цикл (P: while q do g od):

f = [P] = [while q do g od] = [if q then g; while q do g od fi] = [if q then g; f fi], при этом f должна удовлетворять условию: f = ([q] = true → [g] o [f] | [q] = = false → I) (рис. 3).

Эти уравнения определяют алгебру функций, которые могут быть применены в восходящем порядке к иерархии управляющих структур программы при пошаговом процессе извлечения функционала. Этот процесс распространяет и сохраняет результирующее воздействие управляющих структур путем последовательного абстрагирования, оставляя позади сложность локальных вычислений, не требуемых для выражения поведения на более высоком уровне.

Пошаговое извлечение функциональной нагрузки

Рассмотрим процесс пошагового извлечения сценария поведения программы [4] (рис. 4).

Вопрос: Что делает программа, представленная на рис. 4?

Программа берет на вход и выдает на выходе очередь целых чисел, названную Q. В ней также определены локальные очереди, названные odds и evens, и целочисленная переменная x. Операция ‖ есть конкатенация, <> — "не равно". Процесс извлечения поведения расписан по шагам последовательно слева направо. Управляющие структуры программы формируют иерархическую структуру с конечным числом "листьев". В начале последовательного процесса извлечения низший уровень — "лист" if-then-else и управляющая структура while-do — абстрагируются в поведенческую сигнатуру, представленную как условное присваивание. На следующем шаге три структуры while-do, которые теперь являются "листом", могут быть та-

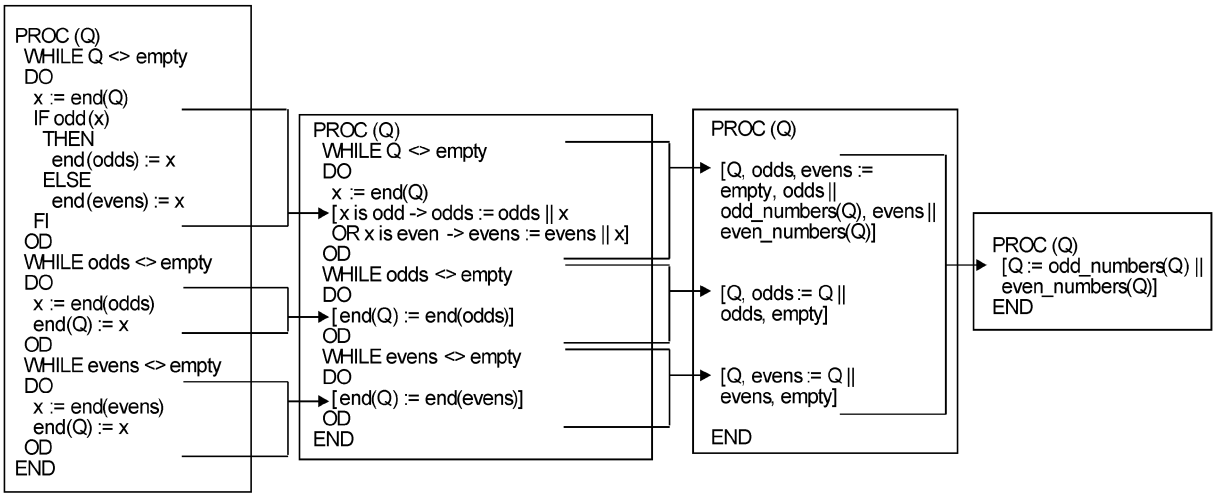


Рис. 4. Пример пошагового извлечения сигнатур поведения

ким же образом абстрагированы в условные правила и присваивания. Наконец, три поведенческих сигнатуры могут быть объединены в одно присваивание, являющееся конечной сигнатурой поведения программы в целом.

В результате видно, что программа создает новую очередь Q, содержащую ее первоначальные нечетные числа, за которыми следуют первоначальные четные числа. Во время этого процесса промежуточные управляющие структуры и данные исключаются путем включения их функциональной нагрузки в более высокоуровневые абстракции. Основа процесса вычисления поведения — составление функциональной композиции путем подстановки значений, что по определению исключает промежуточные выражения на последующих уровнях абстракции. Как отмечено выше, программы могут иметь огромное число путей исполнения, но они составлены из конечного числа управляющих структур, таким образом, процесс вычисления поведения гарантированно является конечным. Кроме того, на каждом шаге текущий сценарий поведения сохраняется, чтобы предоставить аналитику функциональную документацию для каждого уровня абстракции.

Этот пример иллюстрирует пошаговый процесс извлечения поведенческих сигнатур, который является инвариантным относительно объема — одна и та же математика, одни и те же операции используются на всех уровнях извлечения, независимо от размера программы.

Автоматизация

Приведенные выше методы формализации и последовательной абстракции управляющих структур позволяют сделать вывод о возможности автоматизировать процесс извлечения функциональной нагрузки, представленный на рис. 5, тем самым многократно облегчая работу человека, анализирующего поведение кода [4].

Функциональная семантика определена для структур управления и данных целевого языка и правил их комбинации, а также для форм выражений поведения, с помощью которых будет представлено извлеченное поведение.

Сам "экстрактор" использует правила абстракции и упрощения для пошагового извлечения функциональной нагрузки управляющих структур входной программы. Вычисленные наборы поведения передаются графическому интерфейсу в целях создания их представления, удобного для конечного пользователя.

Таким образом, есть возможность скрыть от человека-аналитика многочисленные промежуточные вычисления и прочие процессы, не имеющие непосредственного отношения к конечной модели поведения программы. Специалисту по анализу нет необходимости понимать математическую основу происходящих в программе про-

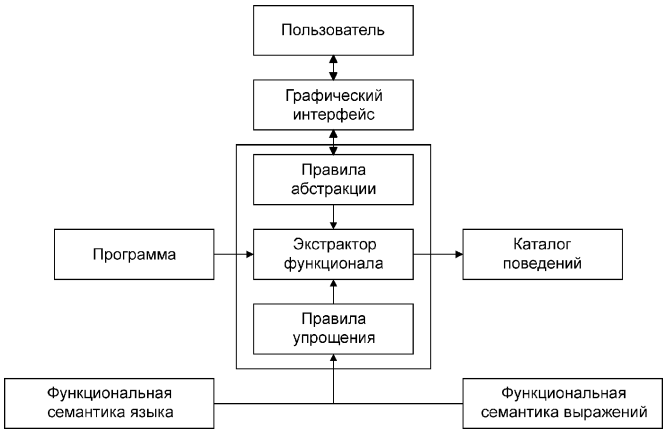


Рис. 5. Принципиальная схема работы "экстрактора функционала"

цессов и в то же время он может быть уверен, что абстракция, полученная после применения программы автоматического извлечения функционала, основана на формально подтвержденных математических методах.

Извлечение функциональной нагрузки на примере вредоносного кода

Сценарии поведения больших программ удобно организовывать в так называемые каталоги поведения [5]. Каталог поведения — это репозиторий сценариев поведения программ. Он реализован в графически структурированной форме, представляющей поведенческие выражения посредством набора параллельных присваиваний и соответствующих правил ветвления, которые, в свою очередь, индексируются с помощью своих предикатов. Все выражения поведения представлены в едином синтаксисе и семантике на всех уровнях абстракции. Таким образом, каталог представляет собой цельную, иерархическую структуру, которую можно обходить, осуществлять по ней поиск и анализировать в соответствии с тем, как это необходимо пользователю для каждой исследуемой программы. В качестве примера возьмем небольшую финансовую Java-программу [5], текст которой представлен на рис. 6.

Допустим, что такой мы ее получили от разработчика. Каталог поведения для данной программы показан на рис. 7. Он демонстрирует достаточно нетривиальное поведение, идентификация которого путем чтения программного кода потребовала бы значительных усилий.

Каталог поведения представляет собой древовидную структуру, которая обходится слева направо. Раздел Summary определяет ключевые свойства программы, а именно то, что AccountRecord остается неизменной, а AdjustRecord изменяется по одному из четырех возможных сценариев. В разделе Definitions представлены промежуточные выражения, введенные "экстрактором" для упрощения и систематизации выражений поведения. Четыре возможных варианта поведения справа приведены раздельно, каждый в своем блоке, ход их выполнения определяется посредством анализа результатов вычисления предикатов, указанных вверху каждого из четырех блоков.

Хотя операции функционального отображения приведены последовательно, это сделано лишь для удобства чтения, на самом же деле они параллельны. Поведенческие выражения являются процедурно-независимыми.

Беглый осмотр четырех блоков справа выявляет, что программа имеет дело с балансами счетов, которые могут быть отрицательными, и предоставляет займы в размере 100 единиц, наращивая их по мере необходимости и в рамках допустимой банком нормы. Разбор первого случая показывает: при условии, что acctRec.balance не отрицателен, программа копирует AccountRecord в AdjustRecord и устанавливает in_default в false. Третий случай отвечает за ситуацию, когда можно добавить 100 единиц к учетной записи, чтобы обратить баланс в положительное значение, не превышая максимального кредита, и в in_default также помещается false. Программист или финансовый аналитик может быстро понять, что в случаях 1 и 3 необходимые банковские функции выполняются корректно. Но в случаях 2 и 4, где in_default устанавливается в true, очевидно подозрительное поведение программы. В случае 2, когда acctRec.balance отрицателен и добавление 100 единиц превышает acctRec.loan_max, с поля баланса снимаются пени и добавляются к AdjustRecord.spec.balance. То же самое происходит в случае 4, который выполняется, когда добавление 100 единиц, которое может быть осуществлено без превышения максимума, недостаточно для того, чтобы сделать баланс положительным. Поведенческая сигнатура данного вредоносного кода есть неизбежный результат расчета поведения программы. Неважно, как тщательно вредоносный код скрыт в программном коде, он будет укрупняться и собираться в определенные варианты про-

```
public class AccountRecord {
  public int acct_num;
  public double balance;
  public int loan_out;
  public int loan_max;
} // end of AccountRecord

public class AdjustRecord extends AccountRecord {
  public boolean in_default;
  public static AdjustRecord spec;
} // end of AdjustRecord

public static AdjustRecord classify_account (AccountRecord acctRec) {
  AdjustRecord adjustRec = new AdjustRecord();
  adjustRec.acct_num = acctRec.acct_num;
  adjustRec.balance = acctRec.balance;
  adjustRec.loan_out = acctRec.loan_out;
  adjustRec.loan_max = acctRec.loan_max;
  while ((adjustRec.balance < 0.00) &&    ((adjustRec.loan_out + 100) <= adjustRec.loan_max)) {
    adjustRec.loan_out += 100;
    adjustRec.balance += 100.00;
  }
  adjustRec.in_default = (adjustRec.balance < 0.00);
  if (adjustRec.balance < 0.00) {
    adjustRec.balance -= 0.01;
    AdjustRecord.spec.balance += 0.01;
  }
  return adjustRec;
}
```

Рис. 6. Пример Java-программы

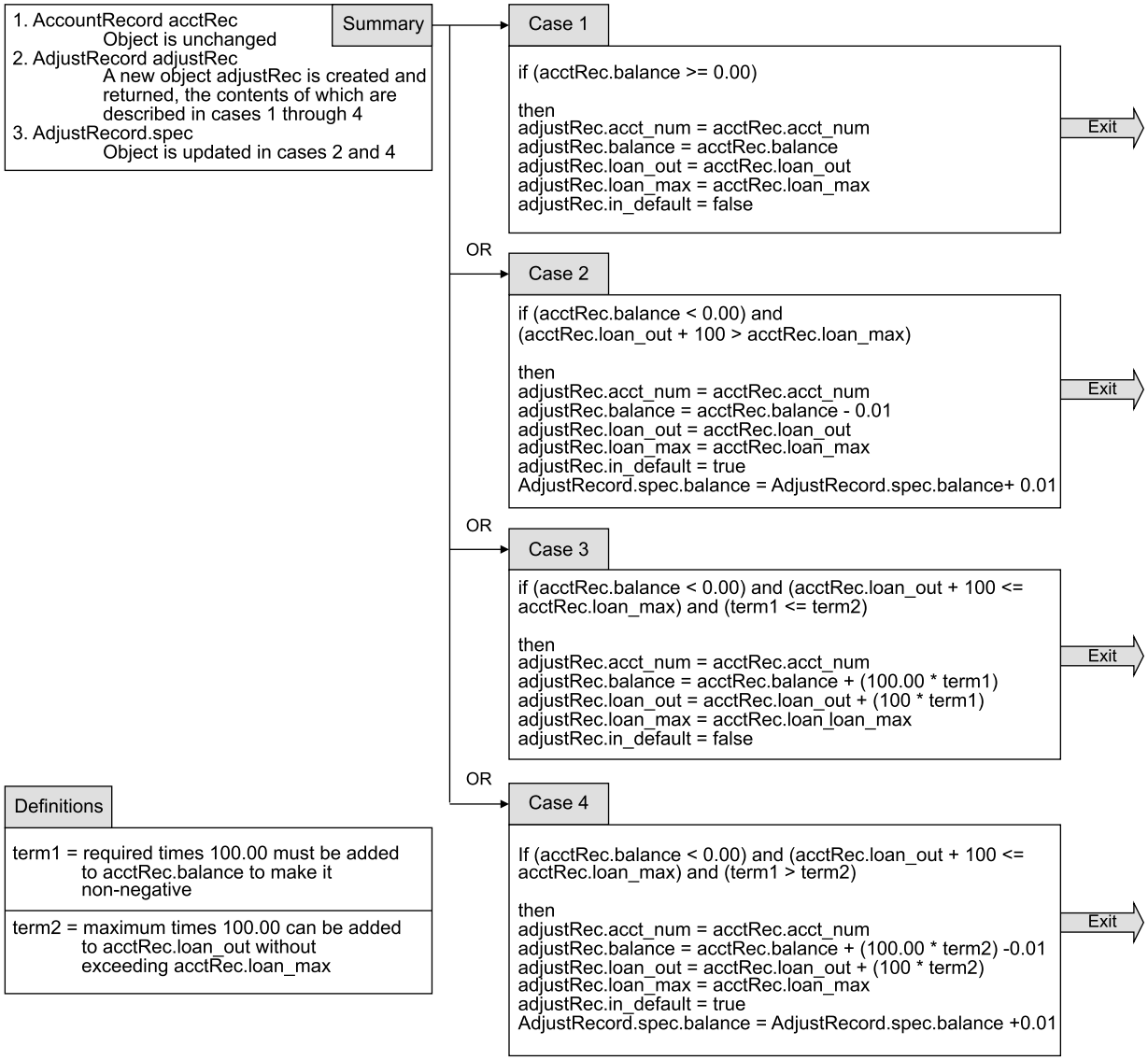


Рис. 7. Каталог поведений для финансовой Java-программы

граммного поведения. Также важно заметить, что процесс извлечения функциональной нагрузки не зависит от именования данных. Вредоносный код не может быть замаскирован путем изменения названий, так как процесс извлечения функционала неизбежно приводит к вычленению поведенческой сигнатуры, которая не зависит от именования.

Заключение

Первым шагом в анализе вредоносного кода является преобразование запутанной структуры враждебной программы в хорошо структурированную, удобную для чтения форму. Такое преобразование помогает аналитикам понимать логику программы и служит подходящей основой для расчета поведения кода. Результатом является функционально-эквивалентная версия, которую можно сопоставить с оригинальной программой и которая может использоваться для понимания и сравнения. Начальная версия реализации технологии извлечения функционала вредоносного кода, предоставляющая такую возможность, была разработана в CERT (Computer Emergency Response Team — www.cert.org) в 2005 году.

Для исследования потенциала данной технологии был разработан прототип, рассчитывающий поведение программы, написанной на небольшом подмножестве языка Java, и представляющий результаты работы в виде каталогов поведения. Был проведен эксперимент в целях сравнения традиционных методов анализа и анализа с применением новой технологии извлечения функциональной нагрузки. Опытные программисты были поделены на две группы: контрольную,

использующую традиционный подход, и экспериментальную, применяющую разработанный прототип. Каждой группе были заданы вопросы относительно понимания и верификации Java-программ. Результаты [6] показали, что экспериментальная группа справилась с ответами на вопросы в 4 раза лучше контрольной, при этом затратив на это лишь четверть отведенного времени.

На основании оценки успеха, достигнутого на этапе тестирования ограниченного прототипа, можно с уверенностью сказать, что технология извлечения функциональной нагрузки в будущем может существенно повлиять на развитие программной инженерии в целом.

Список литературы

1. **Zallar K.** Practical Experience in Automated Testing. http://www.methodsandtools.com/archive/archive.php?id = 33.

2. **Palshikar G. K.** An introduction to model checking. http://www.embedded.com/showArticle.jhtml?articleID = 17603352.

3. **Pleszkoch M., Linger R.** Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior // Proc. of the 37th Hawaii International Conference on System Sciences (HICSS-37). Hawaii, January 5—8, 2004. Los Alamitos. CA: IEEE Computer Society Press, 2004.

4. **Collins R., Walton G., Hevner A., Linger R.** The CERT Function Extraction Experiment: Quantifying FX Impact on Software Comprehension and Verification (CMU/SEI-2005-TN-047). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tn047.html.

5. **Hevner A., Linger R., Collins R., Pleszkoch M., Prowell S., Walton G.** The Impact of Function Extraction Technology on Next-Generation Software Engineering (CMU/SEI-2005-TR-015). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tr015.html.

6. **Function** Extraction for Malicious Code (FX/MC). Automated Calculation of Program Behavior for Security Analysis. http://www.cert.org/sse/fxmc.html.

УДК 004.056

А. В. Душкин, канд. техн. наук, доц.,
Воронежский государственный
технический университет

Распознавание и оценка угроз несанкционированного воздействия на защищенные информационно-телекоммуникационные системы

Показано моделирование процесса распознавания и оценивания угроз несанкционированного воздействия в виде компьютерной разведки и программно-технического воздействия на защищенные информационно-телекоммуникационные системы на основе декомпозиционного подхода.

В последнее время устойчиво наблюдается рост числа атак на информационные ресурсы и инфраструктуры информационно-телекоммуникационных систем (ИТКС). Во многих случаях это происходит в виде несанкционированного воздействия на информацию, средства ее передачи и обработки. Под несанкционированным воздействием (НСВ) будем понимать воздействие на защищаемую информацию с нарушением установленных прав и/или правил доступа, приводящее к утечке, искажению, подделке, уничтожению, блокированию доступа к информации, а также к утрате, уничтожению или сбою функционирования носителя информации [1].

Цель настоящей работы — показать декомпозиционный подход к созданию модели процесса оценивания угроз НСВ на ИТКС на основе их распознавания и оценки степени опасности для возможности принятия адекватного решения на применение средств защиты информации. В рамках исследования будем рассматривать *преднамеренные угрозы*, которые могут быть различных видов: от небрежного анализа, использующего легко доступные средства управления, до изощренных вторжений с использованием специальных сведений о системе. Учитывая преднамеренный характер, реализация угроз НСВ на ИТКС может осуществляться посредством компьютерной разведки (КР) и/или программно-технического воздействия (ПТВ).

Компьютерной разведка — специально подготовленная, согласованная по месту, времени и формам деятельность, направленная на извлечение, систематизацию и специальную обработку открытой информации из информационно-вычислительных сетей, телекоммуникационных систем, а также информации об особенностях их построения и функционирования [2].

Программно-техническое воздействие — совокупность согласованных по цели, месту, времени и формам действий, направленных на модификацию информационной технологии выполнения установленных функций автоматизированной системы, атрибутов ее компонентов и снижение эффективности ее функционирования [3].

Для моделирования угроз НСВ на ИТКС необходимо разобраться с подходом. Классическим

стало определение трех основных угроз: нарушение конфиденциальности, целостности и доступности.

Если детально разобраться, то оказывается, что в чистом виде для информации наиболее актуальны угрозы нарушения конфиденциальности и целостности. Угроза нарушения доступности связана с процессом обработки информации и определяется реализующими его средствами. Таким образом, нарушение доступности напрямую связано с нарушением технологического процесса доступа к информации для проведения санкционированных операций по ознакомлению, документированию, модификации и уничтожению. Другими словами, если нет доступа к информации, невозможно нарушить ее конфиденциальность и целостность. В то же время при НСВ нарушается конфиденциальность и целостность всего технологического процесса обработки информации или его составляющих и появляется возможность нарушения конфиденциальности и целостности самой информации. Таким образом, при определении угроз НСВ на ИТКС целесообразно выделить два основных компонента — *нарушение конфиденциальности* и *целостности* [3, 4].

Под ***нарушением конфиденциальности*** будем понимать утечку информации, а под ***нарушением целостности*** — воздействия на компоненты и/или структуру ИТКС. При этом будем различать ***служебную информацию,*** к которой относятся сведения об используемом техническом и программном обеспечении, протоколах управления и взаимодействия средств вычислительной техники (СВТ), используемых средствах и методах защиты, и ***пользовательскую,*** к которой относится все остальное. ***Служебная конфиденциальность*** определяется состоянием ИТКС, при котором обеспечивается конфиденциальность служебной информации, а ***пользовательская,*** соответственно, — пользовательской информации.

Компонентная целостность — это состояние ИТКС с однозначно определенными и неизменными атрибутами ее компонентов. ***Функциональная целостность*** — состояние ИТКС, в котором она способна реализовать информационную технологию выполнения установленных функций с заданной эффективностью при соблюдении неизменности технологического процесса.

Процедура осуществления НСВ характеризуется большим числом вариантов. В связи с этим целесообразно рассмотреть **типовую модель реализации угроз НСВ на защищенные ИТКС,** последовательно уточняя взаимосвязь основных угроз и способов их реализации.

Учитывая типовую стратегию НСВ на ИТКС, всю процедуру разобьем на четыре этапа [5]:

1) исследование механизмов доступа в ИТКС;

2) исследование механизмов защиты информации в ИТКС;

3) исследование информационных процессов;

4) несанкционированное манипулирование информацией в ИТКС.

Типовая модель реализации угроз НСВ на ИТКС представлена на рис. 1. Но в каждом част-

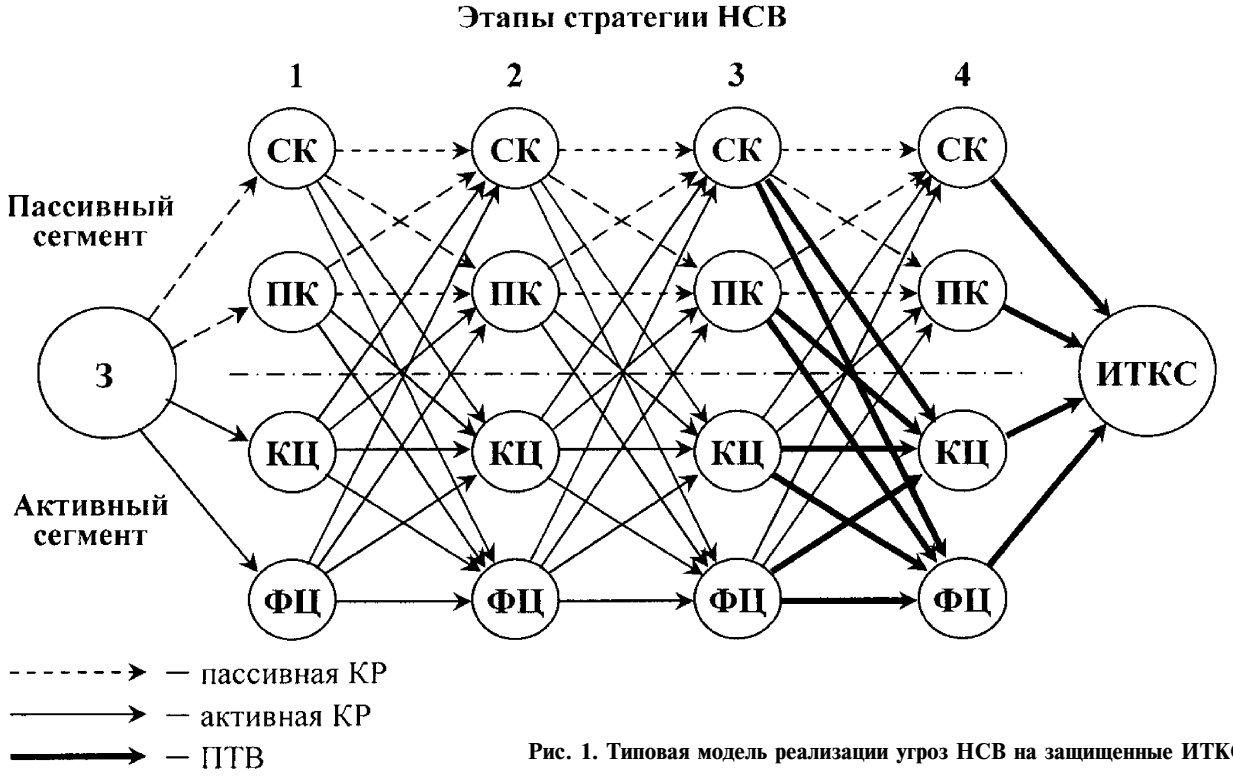


Рис. 1. Типовая модель реализации угроз НСВ на защищенные ИТКС

ном случае модель может видоизменяться, хотя основной подход к ее построению остается тот же.

На каждом этапе решается частная задача НСВ, решением которой является реализация одной из угроз безопасности ИТКС — нарушение служебной (СК) или пользовательской (ПК) конфиденциальности, а также компонентной (КЦ) или функциональной (ФЦ) целостности. Причем целесообразно отдельно выделить два сегмента — пассивный и активный, что соответствует сохранности или нарушению целостности. Это обусловлено характером ведения компьютерной разведки (КР). При ***пассивной КР*** процедура добывания разведданных не нарушает целостности ИТКС и ее элементов, а при ***активной КР*** происходит ее нарушение. Кроме того, активная КР, как правило, осуществляется с использованием ПТВ. При этом с помощью ПТВ нарушается целостность ИТКС, а далее, с помощью КР, — конфиденциальности в виде утечки информации из ИТКС.

Для описания наиболее полной, детальной классификации, которая учитывает основные существующие угрозы ИТКС и при этом каждая из угроз подпадает только под один классификационный признак, разработана **декомпозиционная модель угроз НСВ на защищенные ИТКС на основе способов несанкционированного доступа** (НСД), изображенная на рис. 2. Декомпозиция угроз реализуется методом последовательного деления, осуществляемого на основе отдельно сгруппированных способов НСД. Код модуля отображается в виде **a.b.c.** Нумерация сторон получившихся фигур осуществляется от левого верхнего угла слева-направо, сверху-вниз.

Способ НСД определяется на основании типа доступа, точки доступа, источника, активности, доступности и опосредованности:

- по *типу доступа* различают ***локальный*** (без использования линий связи СВТ) и ***удаленный*** (с использованием линий связи СВТ) НСД;
- по *точке доступа* — НСД ***к СВТ*** и ***линии связи***;
- по *источнику* — ***внешний*** (вне пределов контролируемой зоны) и ***внутренний*** (в пределах контролируемой зоны) источник НСД;
- по *активности* — ***активный*** (с нарушением целостности) и ***пассивный*** (без нарушения целостности) НСД;
- по *доступности* — ***есть*** доступ к элементам ИТКС или ***нет***;
- по *опосредованности* — ***непосредственный*** (без элементов посредничества в процедуре НСД) и ***опосредованный*** (с элементами посредничества) НСД.

Такой подход к моделированию позволяет получить классификацию угроз НСВ на защищенные ИТКС с непересекающимися признаками и необходимым уровнем детализации, что, в свою очередь, позволяет снизить их неопределенность и разработать соответствующие алгоритмы распознавания угроз.

Учитывая изложенное, целесообразно рассмотреть подход к созданию **модели процесса оценивания угроз НСВ на защищенные ИТКС** на основе распознавания угроз и оценки степени их опасности для возможности принятия адекватного решения на применение средств защиты информации. Он заключается в следующем (рис. 3).

Распознавание угрозы НСВ на защищенные ИТКС осуществляется на основе данных об ис-

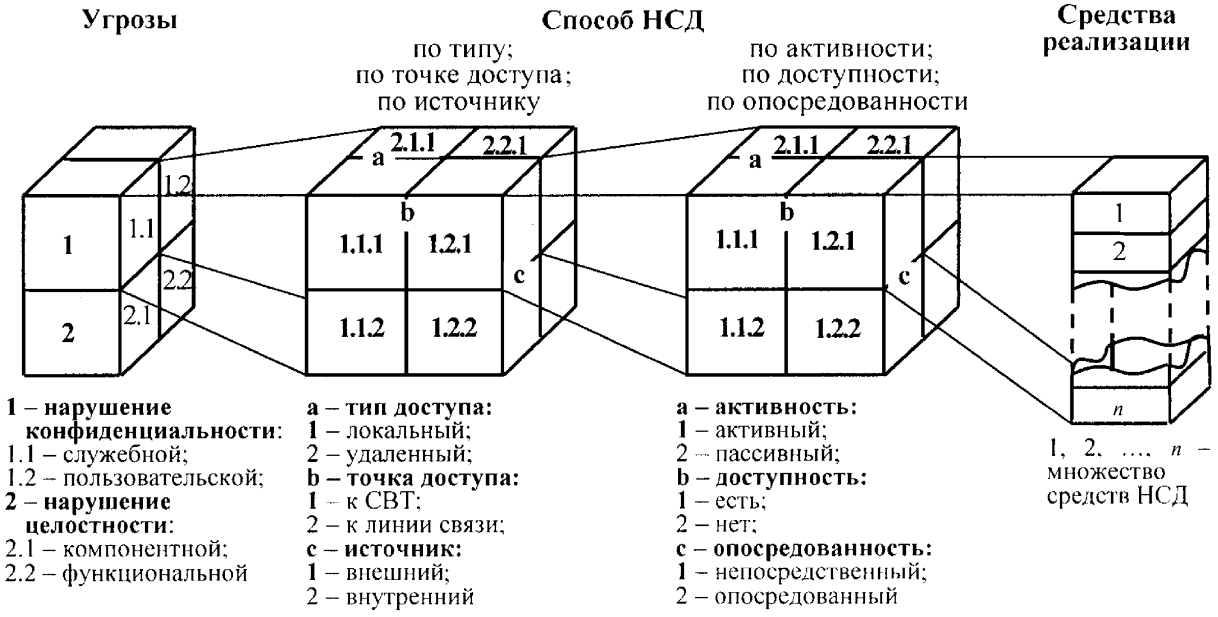


Рис. 2. Декомпозиционная модель угроз НСВ на защищенные ИТКС на основе способов НСД

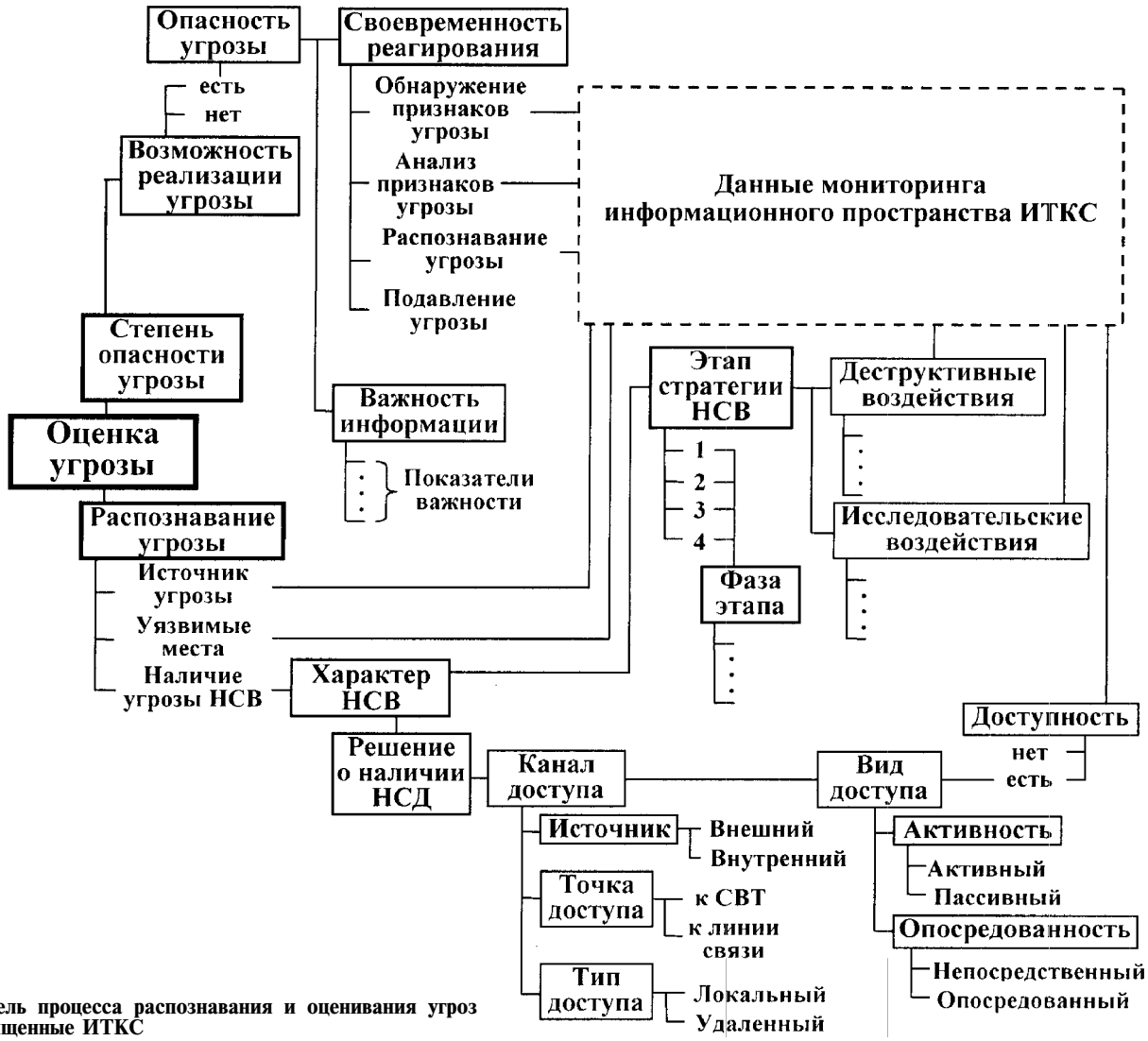


Рис. 3. Модель процесса распознавания и оценивания угроз НСВ на защищенные ИТКС

точнике угрозы, уязвимых местах системы и наличии угрозы. При этом информация о вероятных источниках угрозы и уязвимых местах системы считается априорно известной. Она выявляется на основе предварительных данных мониторинга информационного пространства ИТКС по результатам аудита объектов информатизации.

Вывод о наличии угрозы формируется с учетом характера НСВ. Для этого необходимо, во-первых, определить доступность элементов ИТКС для осуществления НСВ и в случае положительного решения выявить вид и канал доступа, параметры которых находятся в соответствии с декомпозиционной моделью угроз НСВ на защищенные ИТКС (см. рис. 2). Во-вторых, на основе данных мониторинга информационного пространства ИТКС выявить исследовательские и деструктивные свойства НСВ, а с их помощью установить этап стратегии НСВ и, по возможности, фазу этапа.

Степень опасности угрозы во многом зависит от возможности ее реализации. В свою очередь, опасность угрозы определяется важностью подвергающейся воздействию информации и своевременностью реагирования средств защиты.

Важность информации устанавливается на основе интегрирования показателей важности, имеющих разные весовые коэффициенты для каждой конкретной ситуации.

Своевременность реагирования средств защиты ИТКС определяется через показатель эффективности противодействия таким угрозам [6]. Учитывая, что наибольшее внимание на эффективность мероприятий по защите элементов ИТКС оказывает соотношение между временными характеристиками угроз и средств противодействия угрозам, в качестве основы для конструирования показателя эффективности целесообразно брать время $\tau_{пд}$ противодействия угрозе НСВ на ИТКС, т. е. время с момента возникновения угрозы до момента ее подавления. Противодейст-

вие угрозе считается своевременным, если время $\tau_{пд}$ не превышает времени $\tau_{сущ}$ ее существования, т. е. при выполнении неравенства

$$\tau_{пд} \leqslant \tau_{сущ}.$$

Формально процесс противодействия угрозам безопасности элементам ИТКС представляется последовательностью состояний (процедур):

1) обнаружение признаков угрозы;
2) анализ признаков угрозы;
3) распознавание угрозы;
4) подавление угрозы.

Время $\tau_{пд}$ в общем случае можно представить в виде выражения

$$\tau_{пд} = \tau_{пд_1} \circ \tau_{пд_2} \circ \tau_{пд_3} \circ \tau_{пд_4},$$

где $\tau_{пд_1} \div \tau_{пд_4}$ — временные характеристики процедур противодействия угрозам безопасности; $\circ$ — знак композиции случайных величин.

Подводя итог изложенному в работе, можно констатировать, что представленный подход моделирования дает возможность правильного распознавания угроз НСВ и оценки степени их опасности для защищенных ИТКС, а это, в свою очередь, приводит к принятию адекватного решения на применение средств защиты информации.

Список литературы

1. **ГОСТ 51583—00.** Защита информации. Порядок создания автоматизированных систем в защищенном исполнении. Общие положения. М.: Изд-во стандартов, 2000.
2. **Меньшаков Ю. К.** Защита объектов и информации от технических средств разведки: Учебн. пос. М.: РГГУ, 2002. 399 с.
3. **Валуйских С. А., Рог А. И.** О подходе к классификации угроз безопасности автоматизированных систем // Информация и безопасность. 2003. № 2. С. 76—79.
4. **Душкин А. В.** Декомпозиционное моделирование угроз безопасности защищенным информационно-телекоммуникационным системам на основе способов несанкционированного доступа. Деп. ЦСИФ МО РФ, № 14739 от 19.07.2006. Сб. реф. деп. рук. Серия Б. Вып. 76. М.: ЦВНИ МО РФ, 2006. 21 с.
5. **Скрыль С. В., Киселев В. В.** Аналитическая разведка в оценке угроз информационной безопасности // Системы безопасности. 2003. № 6 (48). С. 96—97.
6. **Душкин А. В., Киселев В. В., Скрыль С. В., Ярош В. А.** Распознавание и оценка угроз несанкционированного доступа к элементам информационно-телекоммуникационных систем // Наука — производству. 2006. № 5/2 (92). С. 11—13.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

УДК 519.683.2

С. П. Копысов, канд. физ.-мат. наук,
А. К. Новиков, канд. физ.-мат. наук,
А. Б. Пономарев,
В. Н. Рычков, канд. физ.-мат. наук,
Ю. А. Сагдеева,
Институт прикладной механики УрО РАН

Программная среда построения расчетных моделей метода конечных элементов для параллельных распределенных вычислений

Предлагается среда подготовки расчетных данных (конечно-элементная сетка с заданными характеристиками материала и граничными условиями), тесно связанная с описанием геометрии области. Рассматривается реализация алгоритмов, позволяющих строить, перестраивать и разделять сетки для параллельных вычислений методом конечных элементов.

Введение

При решении задач математического моделирования одной из основных проблем является приведение геометрической модели к расчетной. По описанию тела необходимо построить расчетную сетку (двумерную — 2D, трехмерную — 3D), затем поставить расчетные условия (граничные условия, свойства среды или материала и т. д.) и разделить сетку для параллельных распределенных вычислений. Все эти этапы требуется выполнять интерактивно и визуализировать результаты.

Современные программные решения (ICEM CFD, Pro/Engineer, SDRC I-DEAS, SolidWorks и др.) предоставляют комплексы специализированных программных средств для пре- и постпроцессинга в задачах механики жидкостей и газов, деформируемого твердого тела и др. Сеточные генераторы имеют в них прямые интерфейсы к CAD (*Computer Aided Design*)-системам, которые не просто позволяют построить любую расчетную модель, но и взаимодействуют с расчетными модулями.

Важные проблемы параллельных и распределенных вычислений связаны с необходимостью подготовки больших расчетных сеток и обработки

массивов полученной информации до и после вычислений на многопроцессорных системах.

Расчетная конечно-элементная сетка (в трехмерном случае) представляет собой взаимосвязанную совокупность сеточных объектов — конечных элементов, их граней, ребер, узлов — и связанных с ними расчетных условий и данных. Односвязное объединение конечных элементов Ω_e будем называть сеточной областью $\Omega^h = \bigcup_e \Omega_e^h$, ее замыкание обозначим $\overline{\Omega}^h = \Omega^h \cup \Gamma^h$, где Γ^h — множество граничных граней. Расчетная сетка называется неструктурированной, если она не имеет топологически подобных конечных элементов и простых формул для определения инцидентности (нумерации или адресации) смежных элементов, ребер и узлов. Характерное для нее сложное информационное описание объясняется тем, что взаимосвязи между сеточными объектами определяются только полным перечислением всех необходимых адресов и ссылок.

Для параллельных вычислений в методе конечных элементов (МКЭ) сеточная расчетная область Ω^h разделяется на сеточные подобласти: $\Omega^h = \bigcup_p \Omega_p^h$, где p — число процессоров, участвующих в вычислениях. Сеточные объекты, образующие подсетки, параллельно рассылаются, перестраиваются и перераспределяются в процессе решения задачи и балансировки вычислительной нагрузки процессоров.

В задачах с большими распределенными неструктурированными расчетными сетками резко возрастает вероятность появления ошибки в исходных данных. Кроме того, все больше задач моделирования требуют прямого обращения к геометрическим ядрам и функциям из пользовательского параллельного распределенного приложения [1—3]. Поэтому актуальным является построение CAE (*Computer Aided Engineering*)-системы, включающей в себя кроме обычных функций ряд специфических: подготовку и работу с распределенными данными на подсетках; функции проверки правильности введенных сеточных данных; оптимальную передачу и визуализацию распределенных сеток с возможностью фильтрации ненужной в данный момент информации и др.

Структура среды построения расчетных моделей

Среда построения расчетных моделей состоит из пяти логических составляющих: геометрии, сетки, расчетной сетки, разделенной сетки и распределенной сетки (строки таблицы). Каждая составляющая содержит компоненты ввода/вывода, объектно-ориентированной модели и визуализации (столбцы таблицы).

Объектно-ориентированная модель разбивается на структуры данных и алгоритмы. Деление на уровни отражает принцип декомпозиции сложных задач на более простые компоненты. Логические составляющие допускают наращивание функциональности. В таблице перечислены некоторые свойства компонентов, реализованные на соответствующем логическом уровне. Составляющие геометрии и сетки определяют базовый уровень, каждая следующая логическая составляющая зависит от предыдущих уровней.

При построении программной среды стояла задача получить по возможности платформонезависимое программное обеспечение с использованием следующих технологий: объектно-ориентированное проектирование (UML — *Unified Modeling Language* [4]), объектно-ориентированное программирование (C++, CORBA — *Common Request Broker Architecture* [5]).

Модель геометрии

Создание геометрической модели расчетной области является первым этапом при построении расчетной сетки (первая строка таблицы). Как правило, геометрические модели строятся в CAD-системах и экспортируются в файл для дальнейшей работы с ним [6].

Для работы с геометрией на этапе генерации сетки и при выполнении операций над ней (уточнение, огрубление и др.) была использована открытая система геометрического моделирования Open CASCADE [7]. Open CASCADE предоставляет разработчику доступ к исходным кодам структур данных 3D-геометрии (от объемных примитивов до сложного поверхностного моделирования) и включает алгоритмы моделирования (булевы операции, удаление невидимых линий, сглаживание и снятие фасок).

Использование классов топологических форм объектов (твердое тело, геометрическая грань, ребро, вершина) позволяет реализовать операции, требующие уточнение сетки по геометрическому описанию области.

Модель сетки

Выбор структуры для представления сетки оказывает существенное влияние на трудоемкость алгоритмов, а также на скорость конкретной реализации сеточной модели. Выбрана структура данных, которая задает связанность объектов вида узлы — ребра — грани — ячейки. Данная структура содержит все возможные типы объектов.

Объектно-ориентированная модель расчетной сетки. Основу базовой объектно-ориентирован-

Структура программной среды FEStudio

Логические составляющие среды	Ввод/вывод	Объектно-ориентированная модель		Визуализация
		Структуры данных	Алгоритмы	
Геометрия	Импорт/экспорт (IGES/STEP/STL); автоматизированная процедура проверки и восстановления геометрии	Твердые тела; поверхности; каркасные модели; воксели	Поверхностная триангуляция	Структура визуального приложения на основе Open CASCADE
Сетка	Формат файла; импорт/экспорт (CGNS); автоматизированная процедура проверки и восстановления сетки	2D/3D сетки; ячейки произвольной формы; уникальные индексы объектов; размещение объектов в памяти; свойства сеточных объектов; связь с геометрическими данными	Построение сеток: · поверхностных; · тетраэдальных; · шестигранных; · смешанных. Оптимизация: · сглаживание; · огрубление. Адаптивное перестроение сеток по шаблону	Расширение Open CASCADE
Расчетные данные	Расширение формата файла для расчетных данных; импорт/экспорт (CGNS); связь с геометрическими сеточными данными	Расчетные данные: · граничные условия; · материальные характеристики; расчетная геометрия расчетная сетка	Расстановка расчетных данных; перенос граничных условий с геометрии на сетку	Визуализация исходных данных; визуализация результатов
Разделенная сетка	Ввод/вывод разделенной сетки	Иерархия сеток: · подсетка есть сетка; · граф подсеток; · заместители сеточных объектов	Методы статического разделения: · разделение взвешенного узлового (элементарного) графа; · разделение на основе геометрической информации	Визуализация разделенной сетки
Распределенная сетка	Параллельный ввод/вывод	Взаимодействие с объектами инфраструктуры CORBA; SPMD-объекты; взвешенный граф МВС	Параллельное построение и перестроение сеток; методы динамического перераспределения; мониторинг загрузки	Визуализация удаленных данных; визуализация больших объемов данных

ной модели сетки (вторая строка таблицы) составляют два класса (рис. 1): `Object` и `Mesh` (в диаграмме имена классов начинаются с префикса `Mesh_ModelingData`, префиксами обозначается принадлежность класса той или иной логической подсистеме). Класс `Object` является базовым для всех сеточных объектов. `Object` содержит функции по восстановлению древовидной структуры сетки (связности между объектами разных уровней), по установке и получению свойств объектов, по вычислению центра масс (`Vector CentreOfMass ()`). Большинство из этих функций реализуются только в классах-наследниках, описанных ниже.

Классы `Node`, `Edge`, `Polygon`, `Cell` представляют объекты соответствующих уровней (размерностей): узел, ребро, грань, ячейка (рис. 1). Между соседними уровнями существует двунаправленная связь — ребро содержит ссылки на узлы, которыми оно образовано, а узел — на соответствующие ребра. Связи "вниз" (от ребра к узлу и т. д.) создаются и заполняются при инициализации объекта. Связи "вверх" (от узла к ребру и т. д.) используются при определении соседних элементов в различных операциях над сетками. Связи "вверх" представляются динамическими массивами (`Sequencese : : Vector`), которые заполняются по мере создания объектов. Координаты узла выделяются динамически, за счет этого реализуются как 3D, 2D, так и смешанные сетки.

Класс `Property` присваивает сеточному объекту произвольные свойства, например, характеристики материала, граничные условия. Для хранения свойств в объекте выделяется динамический список. В классе `Property` определены функции сравнения двух свойств на совпадение по типу (также сортировка по типу для упорядоченного хранения в объекте) и сравнения однотипных свойств по значению параметров. На основе данного класса реализуются расчетные данные (рис. 2).

Класс `Mesh` служит контейнером для всех сеточных объектов и свойств, обеспечивает единственность объектов. Новые объекты и свойства

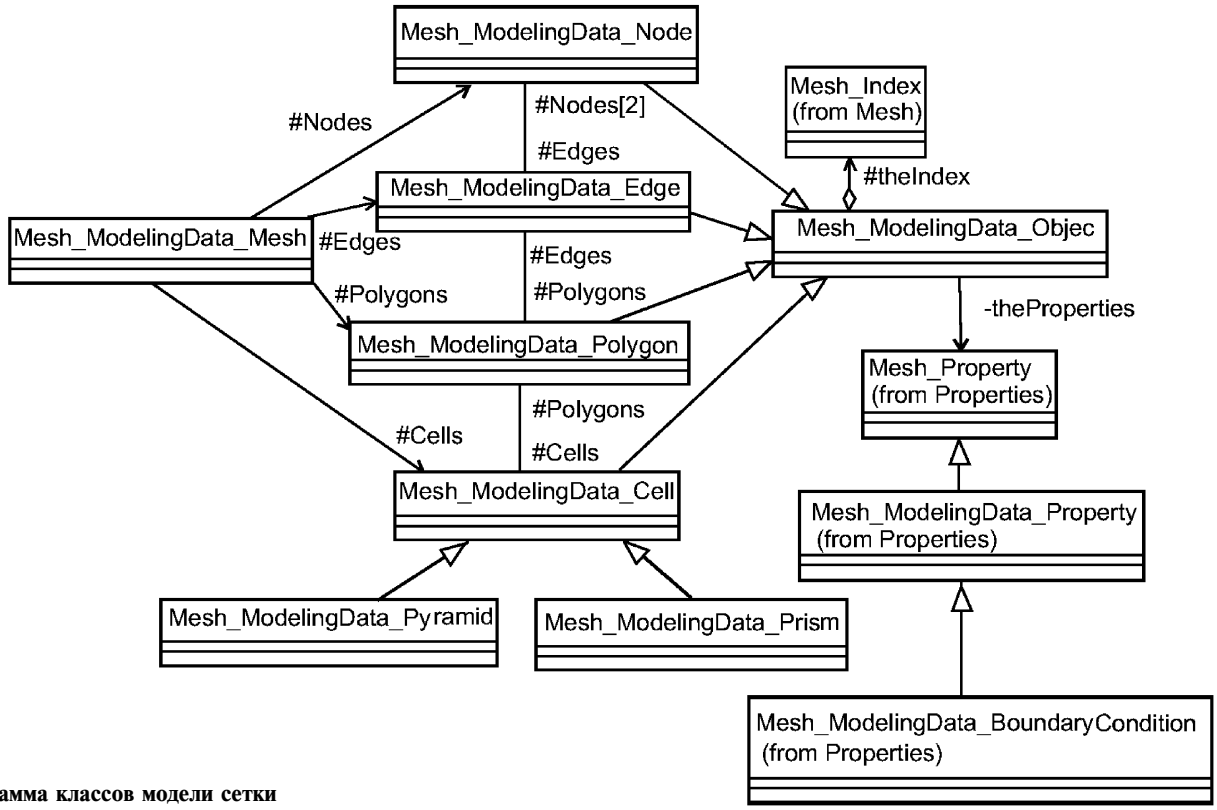


Рис. 1. Диаграмма классов модели сетки

создаются вызовом соответствующих функций сетки, которые проверяют наличие такого объекта в списке, и, если объект уже создан, функция возвращает уже имеющийся, в противном случае новый объект вносится в список. Единственность объектов гарантируют их уникальные индексы.

Функции по созданию, добавлению, удалению и поиску сеточных объектов позволяют реализовывать различные алгоритмы построения сетки. В интерфейс класса сетки добавлены функции для нахождения проекции точек на ребра или поверхности геометрической модели.

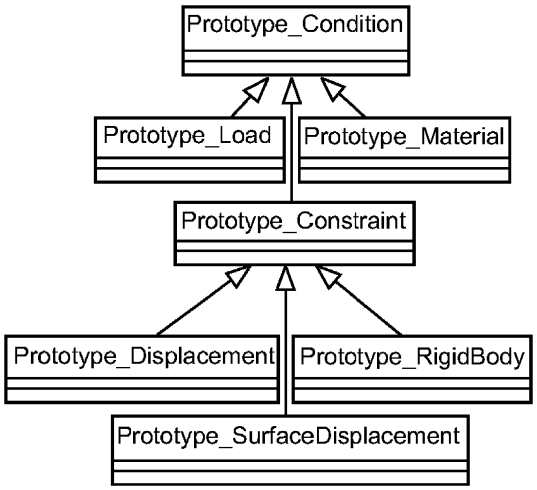


Рис. 2. Реализация расчетных данных

Сеточные генераторы. Построение неструктурированной трехмерной сетки в данной программной среде реализовано в три этапа:

- анализ CAD-геометрии и построение реберной сетки, т. е. аппроксимация топологических криволинейных ребер (каркасной модели) отрезками;
- построение поверхностной сетки методом сжатия текущей границы (развивающегося фронта) с адаптацией к геометрии;
- генерация объемной сетки — алгоритм Делоне с ограничениями [8].

В системе предусмотрено подключение внешних библиотек для построения сеток, и такая возможность была реализована на примере генератора сеток GRUMMP [9].

Перестроение сеток. В числе процедур, реализованных для перестроения сеток, можно выделить такие, как преобразование тетраэдальной сетки в шестигранную (разделение тетраэдра на четыре гексаэдра), регулярное деление тетраэдров, адаптивное перестроение сетки (например, деление по большой стороне). В этих процедурах на вход подаются тетраэдры, которые разбиваются по определенному принципу, а образующиеся неконформные элементы подаются на вход процедуры согласования.

При разбиении тетраэдра появляются новые узлы. Как правило, новые узлы ставятся в середины исходных ребер, треугольников и элементов,

при этом описание исходной геометрии ухудшается, поскольку новые точки уже не принадлежат границе области. В системе положение новых узлов уточняется с использованием классов Open CASCADE, при этом сеточное описание геометрии значительно улучшается.

Модель разделенной сетки

Модель разделенной сетки (четвертая строка таблицы) базируется на модели сетки (см. рис. 1), введенной выше. Классы разделенной модели наследуют соответствующие им классы базовой модели. На рис. 3 представлена модель разделенной сетки. Особенностью разделенной модели является представление сетки и подсетки одним и тем же классом. Класс `Mesh` ссылается сам на себя. Таким образом, подсетки могут рассматриваться как самостоятельные сетки, что позволяет применять к подсеткам те же алгоритмы и использовать их в тех же расчетах, что и разделенные сетки без изменения программного кода. Примером является алгоритм адаптивного перестроения сетки.

Эффективность параллельного решения задач зависит от качества разделения неструктурированных сеток. Алгоритмы разделения сетки условно делятся на две группы: геометрические и графовые [10]. В системе применяются оба класса алгоритмов.

Процесс разделения сетки содержит следующие шаги:

- *Подготовка объектов*. Определяются граничные и внутренние объекты, которые должны быть созданы в подсетках, формируются списки объектов для пересылки в подсетки.
- *Рассылка объектов*. Выполняется последовательная рассылка объектов. Последующие процессы осуществляются на подсетках параллельно. В подсетках создаются клиенты на граничные объекты, затем локальные узлы согласно передаваемому списку координат, вслед за узлами — ребра, затем грани и ячейки.

Модель распределенной сетки

Распределение сеточных объектов, составляющих подсетку, требует расширения модели разделенной сетки и создания новой подсистемы, описывающей распределенную сетку (см. таблицу), т. е. сетку, элементы которой находятся в различных адресных пространствах, в том числе и на различных вычислительных узлах многопроцессорной вычислительной системы. Распределенная сетка должна обеспечивать коммуникации между сеткой и подсетками, между подсетками и между объектами, находящимися в разных подсетках.

В распределенной модели введено понятие остова сетки, обеспечивающего единую точку входа и синхронизации для всех алгоритмов, применяемых к сетке. Остов управляет подсетками и организует коммуникации между ними. При разделе-

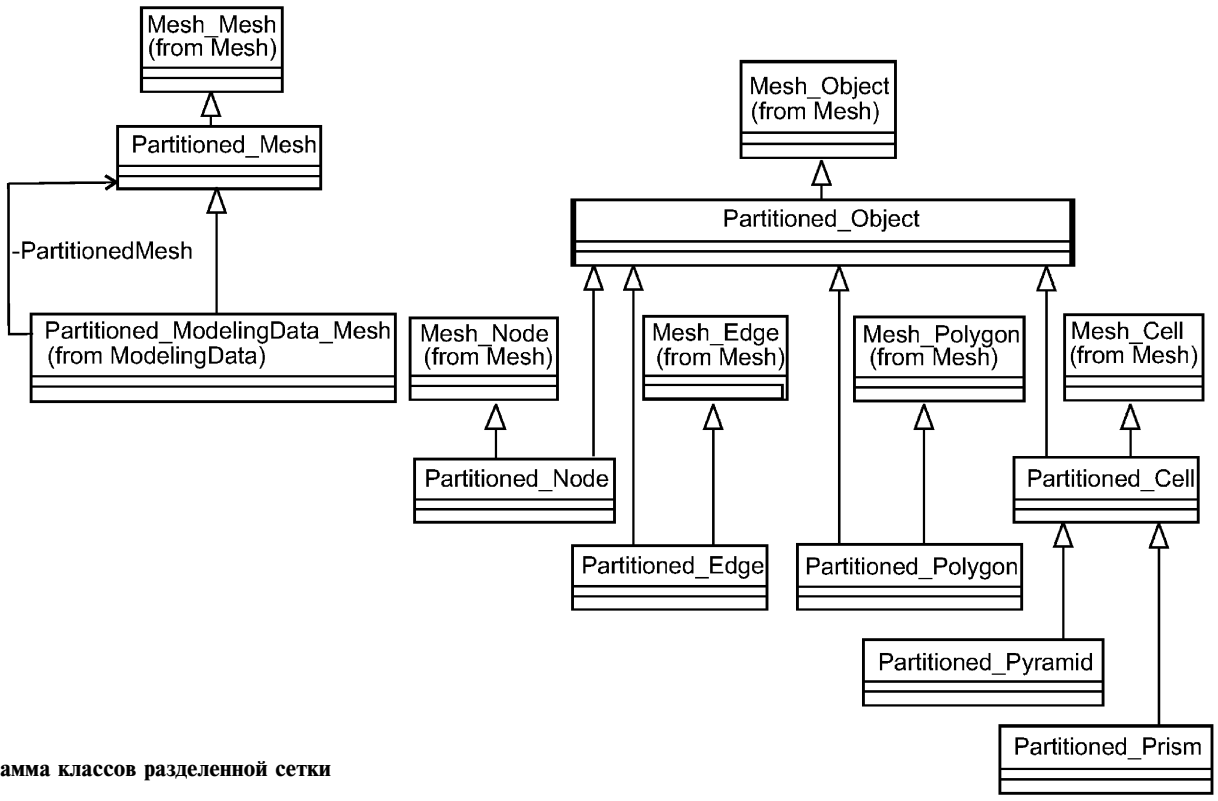


Рис. 3. Диаграмма классов разделенной сетки

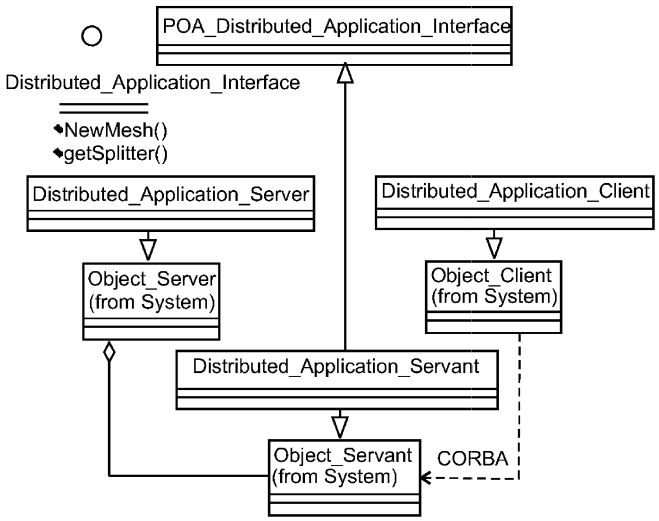


Рис. 4. Набор классов для взаимодействия между клиентом и удаленными серверами

нии сетки возникают граничные объекты, т. е. объекты, принадлежащие двум или более подсеткам. В рассматриваемой модели распределенной сетки граничные объекты хранятся в остове, а в подсетках создаются заместители, которые отличаются от обычных сеточных объектов только внутренней реализацией. Заместители обеспечивают доступ к реальному объекту посредством коммуникаций, которые реализованы с применением технологии CORBA [11].

Такой подход обеспечивает прозрачность граничных объектов для используемых алгоритмов. Для управления распределенной сеткой построен набор сервисных классов, которые обеспечивают доступ к остову сетки и доступ остова к подсеткам, расположенным на других узлах вычислительной системы. На рис. 4 показан CORBA-интерфейс (`Distributed_Application_Interface`) и его абстрактная реализация, содержащая вспомогательные методы, но не включающая реализаций, описанных в интерфейсах — все они создаются разработчиком.

Взаимодействие с удаленными серверами осуществляется вызовом методов клиентов, связанных с этими серверами. Связь клиента с сервантом обеспечивается CORBA, сервант же имеет прямой доступ к серверу, так как находится в том же адресном пространстве. Основой распределенной модели являются классы: "клиент", "сервант" и "сервер".

Класс "клиент" не содержит в себе никаких данных и потому наследуется от абстрактного класса `Mesh`. Класс "сервер" же, наоборот, представляет собой сетку, поэтому должен содержать все ее данные, в силу чего он унаследован от класса `ModelingData_Mesh`, который реализует абстрактный класс `Mesh`. Разделение "клиента" и "сервера" на уровне классов не означает четкого разграничения их функций. Остов и любая из подсеток могут быть как "сервером", так и "клиентом". Отдельное рассмотрение классов "клиент" и "сервер" необходимо для того, чтобы на уровне программного кода в любой момент времени можно было определить, какую роль играет сетка. Реализация методов удаленного доступа к сетке содержится в соответствующем классе серванта. Работа с сеткой, находящейся на удаленном сервере, аналогична работе с обычной сеткой, все преобразования аргументов и вызовы методов выполняют клиент и сервант сетки, связь с удаленным сервером происходит через CORBA.

Интерфейс удаленного взаимодействия остова и подсетки является важной, но не единственной составляющей распределенной модели. Необходимо также обеспечить взаимодействие объектов сетки между собой и сеткой. Классы, обеспечивающие удаленный доступ, изображены на рис. 5.

Взаимодействие с удаленным объектом является двухсторонним. Поскольку в модели клиенты не являются полноценными объектами, а только

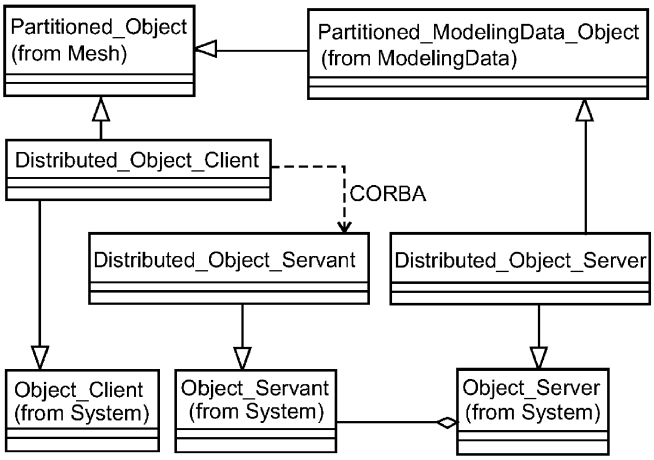


Рис. 5. Классы, обеспечивающие удаленный доступ к объекту

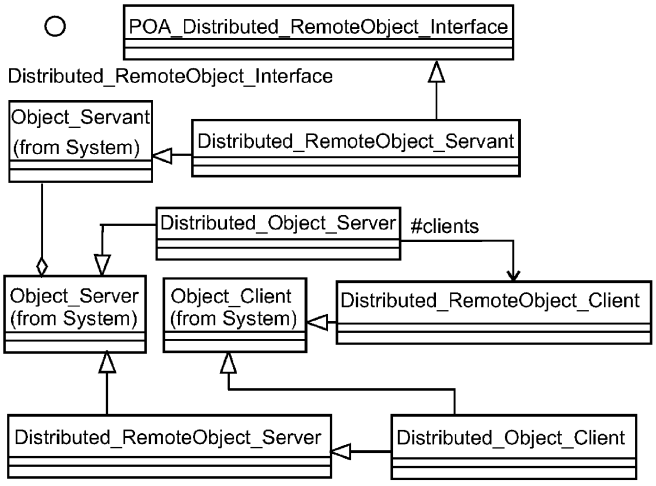


Рис. 6. Связь понятий распределенного и удаленного объектов

лишь их заместителями, взаимодействие с ними организовано по другому интерфейсу. Как видно из диаграммы (рис. 6), удаленным объектом считается клиент для объекта, находящегося в остове (в случае граничного объекта) или в другой подсетке. В свою очередь, сам клиент является сервером (в терминах данной модели сервером удаленного объекта `Distributed_RemoteObject_Server`). Введение понятия удаленного объекта и совмещение ролей клиента и сервера у заместителя объекта обеспечило двустороннее взаимодействие, которое является необходимым для правильного функционирования программной среды.

Сеточные объекты в распределенной модели внешне ничем не отличаются от локальных аналогов, например из разделенной сетки, а все необходимые преобразования и коммуникации скрыты во внутренней реализации, что позволяет избежать модификаций используемых алгоритмов.

Динамическое перераспределение сеточных объектов применяется при балансировке нагрузки между вычислительными узлами. При перераспределении сеточных объектов помимо загруженности вычислительных узлов немаловажную роль играют затраты на выполнение непосредственно пересылки данных сеточных объектов [11].

Выделим три стратегии перемещения объектов:

- перемещение объекта со всеми свойствами (базовая в модели);
- перемещение сеточного объекта с созданием некоторых (или всех) его свойств заново. В функции перемещения (у класса) указывается удаление свойства с последующей инициализацией его на другой стороне;
- удаленный доступ через заместителя без реального перемещения объекта. При попытке доступа к объекту из другой подсетки автоматически создается его заместитель.

Визуализация

Основанное на данной среде визуальное окружение интегрировано с расчетными модулями метода декомпозиции области и дает простой интерфейс для задания граничных условий и редактирования расчетных сеток [12, 13]. С использованием модулей Open CASCADE реализован визуальный редактор расчетных моделей FEStudio [14], содержащий окна отображения геометрии, древовидной структуры документа и окно установки свойств объектов. В библиотеке визуализации сетки реализованы интерфейсы для импорта/экспорта ряда сеточных форматов (GRUMMP, CGNS [15] и др.).

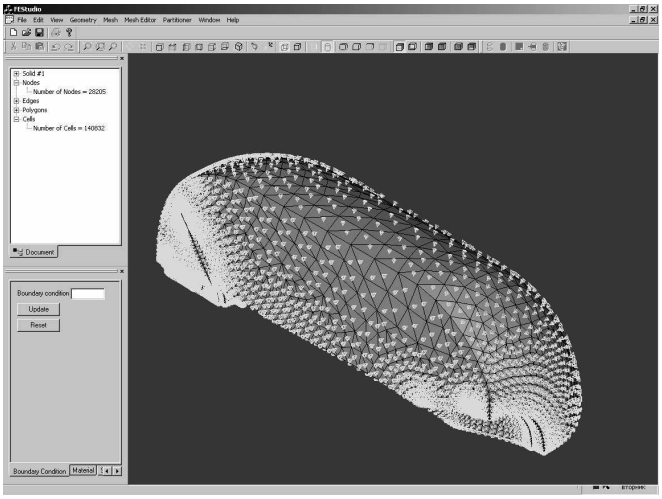


Рис. 7. Задание граничных условий Дирихле

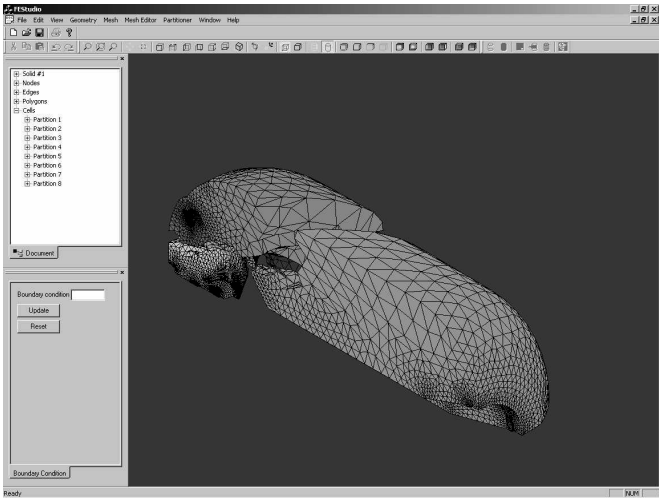


Рис. 8. Разделенная тетраэдальная сетка

Библиотека содержит функции по сбору статистических данных, таких как длины ребер, площади полигонов, объемы ячеек, плоские и двугранные углы, радиусы вписанных и описанных окружностей и сфер, которые выводятся в виде гистограмм.

Заданные граничные условия изображаются схематически в окне отображения геометрии (рис. 7). Для улучшения восприятия расчетные данные масштабируются относительно габаритных размеров расчетной области.

Примеры построения расчетных моделей

Предлагаемая программная среда используется:

- при создании расчетной сетки и модели;
- в ходе вычислений при параллельной адаптации сетки к решению и геометрии области;

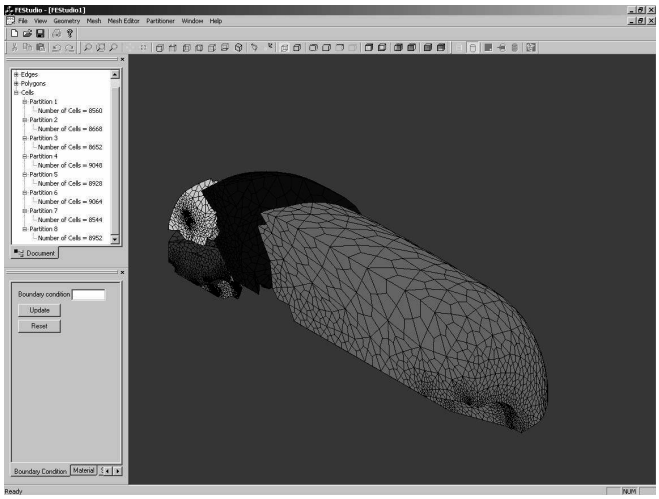


Рис. 9. Разделенная сетка шестигранных элементов

- для визуализации полученных сеток и результатов.

Применение программной среды для параллельного построения больших неструктурированных сеток позволило объединить построение и разделение сетки, уменьшая ввод/вывод и накладные расходы на пересылку данных. На рис 8, 9 в рабочем окне FEStudio представлены разделенные сетки с разнесением подсеток из тетраэдров (рис. 8) и шестигранников (рис. 9).

Заключение

Основная идея данной работы — построение и использовании открытой платформонезависимой программной среды подготовки данных для параллельных распределенных вычислений в МКЭ. Построена модель неструктурированной сетки и на ее основе — модели разделенной и распределенной сеток. Все вычисления на сетке связаны с геометрической моделью, а расчетные данные — как с геометрией, так и сеткой. Впервые реализованы алгоритмы параллельного построения и оптимизации сетки, разделения ее для параллельных вычислений и адаптации сетки к геометрии и решению в процессе высокопроизводительных вычислений методом конечных элементов в рамках единой программной среды для многопроцессорных систем с распределенной памятью.

Построенная среда и визуальное окружение используется в практике вычислений в Институте прикладной механики УрО РАН для решения сопряженных задач механики деформированного твердого тела и газовой динамики.

Работа выполнена при финансовой поддержке РФФИ, гранты № 06-07-89015, 07-01-96069-р_урал_а.

Список литературы

1. **Tautges T. J.** CGM: A Geometry Interface for Mesh Generation, Analysis and Other Applications // Eng. With Computers. 2001. V. 17. P. 299—314.

2. **Четверушкин Б. Н., Гасилов В. А., Поляков С. В., Якобовский М. В.** и др. Пакет прикладных программ GIMM для решения задач гидродинамики на многопроцессорных вычислительных системах // Математическое моделирование. 2005. Т. 17. № 6. С. 58—74.

3. **Ильин В. П.** Геометрическое и функциональное моделирование в задачах математической физики // Вычислительные технологии. 2001. Т. 6. Ч. 2. С. 315—321.

4. **Ларман К.** Применение UML и шаблонов проектирования. М.: Изд. дом "Вильямс", 2001.

5. **Цимбал А.** Технология CORBA для профессионалов. СПб: Изд. Дом "Питер", 2001.

6. **Kopyssov S. P., Rychkov V. N., Ponomaryov A. B.** The integration of CAD-systems and generators of unstructured 3D mesh // Proceedings of the workshop "Grid generation: theory and applications", June 24—28, 2002, Dorodnicyn Computing Centre RAS, Moscow, Russia. 2002. P. 218—229.

7. **Open** CASCADE, simulation integrator: [http://www.opencascade.com].

8. **Копысов С. П., Новиков А. К., Пономарев А. Б.** Методы параллельного построения неструктурированных сеток // Численная геометрия, построение расчетных сеток и высокопроизвод. вычисления / Под ред. Ю. Г. Евтушенко, М. К. Керимова, В. А. Гаранжи. М.: ВЦ РАН, 2006. С. 125—130.

9. **Olivier-Gooch C.** GRUMMP Version 0.2. User's Guide // Department of Mechanical Engineering University of British Columbia. 2001.

10. **Копысов С. П., Новиков А. К.** Параллельные алгоритмы адаптивного перестроения и разделения неструктурированных сеток // Математическое моделирование. 2002. Т. 14. № 9. С. 91—96.

11. **Копысов С. П., Пономарев А. Б., Рычков В. Н., Зубцовский С. Н.** Расчетные неструктурированные сетки для распределенных вычислений // Сибирская школа-семинар по параллельным вычислениям. Томск: Изд-во Том. ун-та. 2005. С. 19—25.

12. **Копысов С. П., Красноперов И. В., Рычков В. Н.** Объектно-ориентированный метод декомпозиции области // Вычислительные методы и программирование. 2003. Т. 4. С. 176—193.

13. **Копысов С. П., Красноперов И. В., Рычков В. Н.** Реализация объектно-ориентированной модели метода декомпозиции области на основе параллельных распределенных компонентов CORBA // Вычислительные методы и программирование. 2003. Т. 4. С. 194—206.

14. **Копысов С. П., Пономарев А. Б., Рычков В. Н.** Открытое визуальное окружение для взаимодействия с геометрическими ядрами, генерации/перестроения/разделения сеток и построения расчетных моделей // Прикладная геометрия, построение расчетных сеток и высокопроизвод. вычисления / Ред. Ю. Г. Евтушенко, М. К. Керимов, В. А. Гаранжа. М.: ВЦ РАН. 2004. Т. 2. С. 154—164.

15. **Rumsey C. L., Poirier D. M. A., Bush R. H., Towne C. E.** CFD General Notation System. A User's Guide to CGNS. Overview and Entry-Level Document. Mid-Level Library: [http://www.cgns.org].

УДК 004.451

П. В. Емельянов, К. С. Коротаев,
Московский физико-технический институт

Математическая модель обеспечения гарантий выделения ресурсов операционной системы

Описывается подход к организации работы менеджера ресурсов операционной системы, ориентированный на гарантированное выделение заданного объема ресурсов. В основу положен принцип ограничения потребления имеющихся ресурсов в целях создания свободного резерва требуемого объема. Приводится математическое обоснование подхода.

Введение

Ядро любой операционной системы предоставляет ее пользователям различные ресурсы [1]. При этом наряду с задачей выделения ресурсов часто возникает задача их распределения — различные пользователи должны быть ограничены в потреблении того или иного ресурса.

В первом приближении ресурсов всего два — это процессорное время и объем памяти. Однако очень часто возникает необходимость вводить большее количество ресурсов, например, открытые файлы и сетевые соединения, объекты ядра операционной системы и сегменты программ, размеры сетевых буферов отправки и получения данных и многие другие. И потребление каждого из этих ресурсов требуется контролировать.

В простейшем виде задача ограничения выглядит следующим образом. Пусть u_i — объем определенного ресурса, потребляемый пользователем i. С течением времени эта величина может меняться произвольным образом, но операционная система следит за тем, чтобы всегда выполнялось неравенство $u_i \leqslant l_i$, где l_i — это предел потребления ресурса данной группой, задаваемый, как правило, администратором. В случае, если выделение очередной порции ресурса по запросу от пользователя приведет к нарушению этого неравенства, в выделении ресурса отказывается. Но поскольку данное неравенство может быть выполнено всегда в случае $u_i = 0$, то наряду с вопросом ограничения использования ресурса возникает вопрос о том, можно ли задать такой объем ресурсов g_i, который будет предоставлен пользователю при любом стечении обстоятельств[1], т. е. требуется вычислить g_i:

$$g_i \leqslant u_i. \qquad (1)$$

В данной статье вводится определение гарантии предоставления ресурсов, даются пути решения задачи предоставления гарантий и приводится обоснование одного из путей и анализ полученного результата. В заключение показываются достоинства и недостатки построенной модели.

Определение гарантии и способы ее обеспечения

Гарантией называется объем ресурсов, который будет предоставлен пользователю по запросу независимо от объема ресурсов, потребляемого другими пользователями системы.

В соответствии с введенным определением существует два "базисных" способа обеспечения гарантии для отдельного пользователя (все остальные способы получаются в результате комбинации этих двух).

Первый способ очевиден — резервирование ресурса в момент появления нового пользователя. Тогда при наличии необходимого "запаса" для него вопрос предоставления требуемого объема ресурсов решается естественным образом. Однако такому способу присущи следующие недостатки. Во-первых, не все ресурсы можно резервировать, например, процессорное время или память на NUMA-системах — в таких системах она выделяется процессу в зависимости от того, на каком процессоре он в данный момент выполняется [2]. Здесь резервирование нереализуемо, так как невозможно предугадать, на каком процессоре будет выполняться процесс. Резервирование же для всех процессоров или миграция резервов с одного процессора на другой создают дополнительные неудобства в работе. Во-вторых, неясно, что делать в случае разделения ресурсов между разными пользователями. Например, если брать ресурс из резервов одного из них, то в случае, если этот пользователь перестает использовать выделенный ресурс, придется его выделять заново из резерва другого, что может оказаться невозможным. Тем не менее, этот подход является самоочевидным и простым в реализации и применяется практически во всех современных системах контроля ресурсов, в которых существует механизм гарантий [3—5].

Остановимся подробнее на втором способе. Что делать, если нельзя отложить ресурсы "про запас"? В этом случае выход один — работа системы должна быть организована таким образом, чтобы

[1] Разумеется, за исключением случаев выхода системы или ее компонентов из строя.

максимальное суммарное потребление ресурсов остальными пользователями системы оставляло свободным необходимое рассматриваемому пользователю количество ресурсов, т. е. каждый пользователь системы должен быть ограничен в потреблении ресурсов определенным образом. В следующем разделе показывается, как вычисляются требуемые пределы.

Математическая модель

Для построения математической модели дополнительно к g_i введем следующие обозначения:

- R — общий объем имеющихся в распоряжении ресурсов;
- n — число пользователей в системе. Интерес представляет случай $n > 1$;
- l_j — предел потребления ресурса, который необходимо установить пользователю j, чтобы обеспечить необходимую гарантию для g_i пользователя i, $j = \overline{1, n}$, $j \neq i$.

Для обеспечения гарантии g_i необходимо выполнение следующего условия:

$$\sum_{1 \le j \le n, j \ne i} l_j = R - g_i. \tag{1}$$

Очевидно, что условие (1) будет выполняться автоматически. Решений у этого уравнения бесконечно много, однако его можно решать для всех пользователей одновременно, обеспечивая, таким образом, гарантии в рамках всей системы. Следовательно, чтобы найти необходимые пределы, требуется решить следующую систему линейных уравнений:

$$\begin{cases} \sum\limits_{i \ne 1} l_i = R - g_1; \\ \vdots \\ \sum\limits_{i \ne n} l_i = R - g_n. \end{cases} \tag{2}$$

В этом случае может существовать лишь одно решение. Заметим, что для каждого пользователя вычисленный предел должен быть корректен, т. е. в соответствии с условием (1) не меньше, чем требуемая гарантия:

$$l_i \geqslant g_i, \ \forall i. \tag{3}$$

Также естественно предположить, что требуемые гарантии также корректны, т. е. суммарный объем предоставляемых гарантий не превосходит объем имеющихся в наличии ресурсов, что выражается следующим соотношением:

$$\sum_{i=1}^{n} g_i \leqslant R. \tag{4}$$

Перепишем уравнение (2) в матричной форме

$$A \cdot l = r, \tag{5}$$

где

$$A = \begin{vmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{vmatrix} - E; \qquad l = \begin{vmatrix} l_1 \\ \vdots \\ l_n \end{vmatrix}, \ r = \begin{vmatrix} R - g_1 \\ \vdots \\ R - g_n \end{vmatrix}. \tag{6}$$

Для того чтобы решить уравнение (5), достаточно найти матрицу, обратную к матрице A. Для этого вычислим ее квадрат. Используя выражение (6), легко получить следующее равенство:

$$A \cdot A = (F - E)(F - E) = (n - 2)F + E,$$

где

$$F = \begin{vmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{vmatrix}, \tag{7}$$

откуда

$$A \cdot A - (n - 2)(A + E) = E.$$

Перепишем полученное равенство в виде

$$A \cdot \frac{A - (n-2)E}{n-1} = E,$$

откуда, в соответствии с определением обратной матрицы, находим

$$A^{-1} = \frac{A - (n-2)E}{n-1}.$$

Следовательно, решение системы (2)

$$l = A^{-1} \cdot r = \frac{A - (n-2)E}{n-1} \cdot r,$$

или, учитывая (6),

$$l = \frac{F \cdot r - (n-1)r}{n-1}.$$

Вычислим вектор l покомпонентно. Учитывая (7), получаем

$$F \cdot r = \Sigma_i (R - g_i) \begin{vmatrix} 1 \\ \vdots \\ 1 \end{vmatrix},$$

и, следовательно, для компонент вектора l имеет место следующее выражение:

$$l_i = \frac{\Sigma_i (R - g_i) - (n-1)(R - g_i)}{n-1}.$$

После упрощения получаем

$$l_i = \frac{1}{n-1}(nR - \Sigma_i g_i) - R + g_i.$$

Для анализа полученного результата удобно переписать его в виде

$$l_i = \tilde{R} + g_i, \tag{8}$$

где

$$\tilde{R} = \frac{R - \Sigma_i g_i}{n - 1}. \qquad (9)$$

Из уравнения (8) видно, что условие корректности пределов (3) выполняется в том и только в том случае, когда $\tilde{R} \geqslant 0$. Вместе с тем из (9) следует, что полученное условие $\tilde{R} \geqslant 0$ равносильно условию $R - \Sigma_i g_i \geqslant 0$, которое является условием корректности требуемых гарантий (4). То есть корректность необходимых гарантий в смысле (4) является необходимым и достаточным условием корректности вычисленных пределов (3).

Случай нарушения пределов

В некоторых случаях, однако, операционная система допускает кратковременное нарушение пользователем установленных пределов [6]. Естественно ожидать, что в течение этого периода времени остальные пользователи системы могут страдать от возможной нехватки ресурсов. Описываемая модель позволяет показать, как именно.

Перепишем (2) в следующем виде:

$$\begin{cases} g_1 = R - \sum\limits_{i \neq 1} l_i; \\ \vdots \\ g_n = R - \sum\limits_{i \neq n} l_i. \end{cases} \qquad (10)$$

Нарушение пользователем j своего предела эквивалентно установлению ему нового предела $\tilde{l_j} = l_j + \Delta l_j$. Подставим это выражение в (10) и вычислим, какие гарантии $\tilde{g_i}$ будут предоставляться всем остальным пользователям:

$$\begin{cases} \tilde{g}_1 = R - \sum\limits_{i \neq 1} l_i - \Delta l_j; \\ \vdots \\ \tilde{g}_n = R - \sum\limits_{i \neq n} l_i - \Delta l_i. \end{cases}$$

Нетрудно видеть, что

$$\tilde{g_i} = g_i - \Delta l_j.$$

Итак, в случае, если операционная система искусственно позволяет одному пользователю нарушить установленный предел, гарантии всех остальных пользователей уменьшаются на одну и ту же величину — на величину нарушения — независимо от числа пользователей в системе.

Заключение

Описанная в статье модель позволяет решить задачу предоставления гарантий выделения ресурсов с помощью механизма ограничения пользователей. Данная модель обладает следующими достоинствами.

Во-первых, модель показывает, что задачи предоставления гарантий и задача ограничения потребителей в объеме ресурса являются эквивалентными — зная пределы потребления, можно вычислить, какие гарантии предоставляются пользователям, и наоборот, исходя из желаемых событий можно вычислить необходимые для этого пределы.

Во-вторых, реализация данной модели может быть выполнена на существующих системах по контролю ресурсов [4, 5], поскольку все они обеспечивают необходимый механизм организации потребления ресурсов. Заметим также, что расширение функционально сосуществующих механизмов не приведет к потерям в производительности, поскольку необходимые дополнительные вычисления требуются лишь в момент создания нового пользователя.

В-третьих, применение модели возможно для любого типа ресурсов. Например, легко видеть, что задача обеспечения гарантии на память в NUMA-системах, не имеющая, как было показано, эффективного решения с использованием механизма резервирования, изменяется с применением предложенной модели. Предоставление гарантий на процессорное время может быть также решено при объединении данной системы с моделью ограничения групп процессов в процессорном времени, например, описанной в работе [7].

В качестве недостатков модели можно отметить следующее.

Сложность процедуры создания нового пользователя становится как минимум $O(n)$. Несмотря на то, что в большинстве систем контроля ресурсов такая операция является крайне редкой, применение модели в системах, где пользователи создаются и исчезают очень часто, может существенно ограничить эффективность работы.

Список литературы

1. **Tanenbaum A. S., Woodhuli A. S.** Operating Systems: Design and Implementation (Second Edition). Prentice Hall, 1997.

2. **Kleen A.** A NUMA API for Linux. Novell inc, 2006. http://www.novell.com/collateral/4621437/4621437.pdf.

3. **Braham P., Dragovic B., Frascr K.** Xen and the Art of Virtualization University of Cambridge, 2003. http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xenosp.pdf.

4. **CKRM** Linux Open Source project. Class Based Kernel resource management. http://ckrm.sourceforge.net/.

5. **Емельянов П. В., Савочкин А. В.** Расширение функциональности существующей модели контроля памяти в ядре Linux // Процессы и методы обработки информации. Долгопрудный: Изд-во МФТИ. 2005. С. 77—80.

6. **Boswell W.** Inside Windows Server 2003. Addison-Wesley Professional, 2003.

7. **Коротаев К. С., Емельянов П. В.** Многоуровневый планировщик процессорного времени для групп процессов, обеспечивающий гарантии в обслуживании // Информационный технологии. 2006. № 6. С. 58—63.

Поздравляем юбиляра!

Заслуженному деятелю науки РФ, заведующему кафедрой "Прикладная математика" МГТУ им. Н. Э. Баумана, члену редколлегии журнала "Информационные технологии", доктору технических наук, профессору

Владимиру Степановичу ЗАРУБИНУ,

известному ученому, крупному специалисту в области прикладной математики, механики и современных вычислительных технологий исполняется **75** лет.

Владимир Степанович является основателем одной из ведущих отечественных научных школ в области прикладной математики и механики, создателем нового научного направления — вычислительной термомеханики деформируемого твердого тела. Им опубликовано большое число монографий и учебных пособий. Под его руководством подготовлена большая плеяда научных и научно-педагогических кадров в области теплопрочностного расчета конструкций.

Возглавляемая В. С. Зарубиным кафедра занимает ведущее место в системе подготовки студентов по математике на ведущих факультетах МГТУ им. Н. Э. Баумана, а также подготовки инженеров-математиков по специальности "Прикладная математика".

Под его руководством выпущена широко известная серия учебных пособий "Математика в техническом университете", охватывающих основные разделы математической подготовки студентов технических университетов.

В. С. Зарубин является членом редколлегий ряда академических и научно-технических журналов. Большой вклад он внес в становление и развитие журнала "Информационные технологии".

Талантливый ученый, блестящий педагог, высококвалифицированный специалист, чуткий, отзывчивый и обаятельный человек, Владимир Степанович пользуется глубоким уважением учеников, коллег, товарищей.

Сердечно поздравляем юбиляра, желаем ему крепкого здоровья, благополучия и новых творческих успехов в научной и педагогической деятельности.

Редколлегия и редакция журнала

CONTENTS

Адрес редакции:

107076, Москва, Стромынский пер., 4/1

Телефон редакции журнала **(495) 269-5510**

E-mail: it@novtex.ru

Дизайнер *Т.Н. Погорелова*. Художник *В.Н. Погорелов*.
Технический редактор *О. А. Ефремова*. Корректор *О. А. Шаповалова*

Сдано в набор 28.12.2007. Подписано в печать 15.02.2008. Формат 60×88 1/8. Бумага офсетная. Печать офсетная.
Усл. печ. л. 10,78. Уч.-изд. л. 12,57. Заказ 157. Цена договорная.

Журнал зарегистрирован в Министерстве Российской Федерации по делам печати, телерадиовещания и средств массовых коммуникаций.
Свидетельство о регистрации ПИ № 77-15565 от 02 июня 2003 г.

Отпечатано в ООО "Подольская Периодика"
142110, Московская обл., г. Подольск, ул. Кирова, 15