ISSN 2306-1561
Automation and Control in Technical Systems (ACTS)
2015, No 3, pp. 16-30.
DOI: 10.12731/2306-1561-2015-3-2

Technology Global Optimization of User Computer Program Code

Murat Hasanbekovich Tomaev

Russian Federation, Ph. D., Associate Professor, Department of «Automated Data Processing».

North Caucasian Institute of Mining and Metallurgy (State Technological University), NCIMM (STU), 362021, Russian Federation, North Ossetia-Alania, Vladikavkaz, Nikolaeva Str., 44. Tel.: +7 (8672) 40-71-01. http://www.skgmi-gtu.ru

tmxwork@mail.ru

Abstract. The paper describes the main challenges of development and implementation of specialized software tools that perform code optimization by automatic and automatic modes. Reveals the basic principles of the method of optimal decomposition, proposed methods for solving optimization problems.

Keywords: system, software, program code, optimization, data processing, decomposition.

ISSN 2306-1561
Автоматизация и управление в технических системах (АУТС)
2015. – № 3. – С. 16-30.
DOI: 10.12731/2306-1561-2015-3-2

УДК 681.343.001

Технологии глобальной оптимизации пользовательских программных кодов

Томаев Мурат Хасанбекович
Российская Федерация, кандидат технических наук, доцент кафедры «Автоматизированной обработки информации».
ФГБОУ ВПО «Северо-Кавказский горно-металлургический институт (государственный технологический университет)» (СКГМИ (ГТУ)), 362021, Российская Федерация, РСО-Алания, г. Владикавказ, ул. Николаева, д. 44, Тел.: +7 (8672) 40-71-01, http://www.skgmi-gtu.ru
tmxwork@mail.ru

Аннотация. В работе описываются основные проблемы и задачи разработки и внедрения специализированных программных средств, выполняющих оптимизацию программного кода пользователя в автоматическом и автоматизированном режимах. Раскрываются основные принципы метода оптимальной декомпозиции, предлагаются методы решения оптимизационных задач.

Ключевые слова: система, программное обеспечение, программный код, оптимизация, обработка информации, декомпозиция.

1. Введение

Одним из способов повышения качества программных продуктов является внедрение автоматизированных средств оптимизации исходных программных кодов. С развитием аппаратных средств вычислительных систем (ВС) и соответствующего программного обеспечения (ПО) эволюционируют и технологии оптимизации программных продуктов. Хорошо прослеживается зависимость между способами оценки качества ПО и степенью опережающего развития отдельных компонентов ВС: процессоров, модулей оперативной памяти, графических подсистем и других [1…6].

Начальные этапы развития элементной базы характеризовались ценностью всех видов ресурсов ВС. Пользовательская программа должна была занимать минимальные объемы процессорного времени, а также оперативной и внешней памяти. Актуальными в этих условиях являются многокритериальные оптимизационные модели – к примеру, задача минимизации стоимости используемых программой ресурсов каждого типа.

Быстро наметившийся разрыв в стоимости и производительности памяти различного уровня: внешней (дисковые и твердотельные накопители, сохраняющие информацию после отключения питания), оперативной памяти и кэш-памяти на кристаллах процессора, привел к развитию моделей и методов кэширования в многоуровневой памяти ЭВМ.

Достигнутый на определенном этапе качественный скачок в технологии изготовления полупроводниковых систем и последующее снижение удельной стоимости ресурсов всех видов (процессоров, модулей памяти, коммуникационных систем) приводит к очередной смене приоритетов: Если для системного ПО критерии качества не изменились, то для программных продуктов прикладного назначения традиционный критерий качества – «оптимальное расходование ресурсов», – уходит на второй план. Новыми ориентирами становятся субъективные критерии, определяемые рыночной конъюнктурой: высокая степень автоматизации прикладных (научных, производственных и т.д.) процессов, достигаемая с помощью ПО; простота сопровождения (низкие требования к квалификации пользователя); высокая надежность программного продукта, по возможности, не зависящая от «человеческого фактора». Основным направлением развития ОС на данном этапе стало обеспечение максимальной изоляции уровня прикладных приложений от технических характеристик конечного устройства. Кроме того, получившая популярность идея многоплатформенности (на уровне ОС) прикладных приложений привела к стремительному развитию виртуальных машин (ВМ) [7, 8], которые поставляются либо в виде отдельных продуктов («Oracle Java VM», «MS .NET Framework») либо в виде компонентов расширения существующих приложений (VBA в MSOffice, PHP, python, Ruby и другие в Web-серверах) либо являются неотъемлемой частью других приложений (Javascript в браузерах).

2. Классификация оптимизационных подходов

Машинно-зависимые методы оптимизации в полной мере остаются доступны лишь системным программистам, частично могут использоваться прикладными разработчиками на языках низкого уровня (при этом низкоуровневый код – к примеру, ассемблерные вставки в «C»/«C++»,– не должен конфликтовать с функциями ядра ОС) [9…11] и недоступны разработчикам на языках высокого уровня. В связи с этим, растет роль машинно-независимых методов оптимизации. Классифицируя их по области применения оптимизационного подхода, можно выделить 2 группы моделей: локально-оптимизирующие и глобально-оптимизирующие. Первая группа описывает методы улучшения качества отдельных участков кода, без учета их влияния на другие компоненты приложения. Вторая – представляет собой набор методов, в которых для соответствия критериям оптимальности требуется анализ и, при необходимости, модификация всего алгоритма программы. Благодаря простоте реализации локально-оптимальные методы получили широкое распространение среди производителей так называемых интегрированных сред разработки. И, напротив, глобально-оптимальные подходы, позволяющие достичь значительно большего качественного прироста

разрабатываемого ПО, тем не менее, получают слабое распространение по двум причинам:

Методы решения подобных задач имеют, в общем случае, переборный характер и требуют значительных ресурсов процессорного времени;

Высокая сложность программных средств поддержки глобально-оптимизирующих преобразований: Эффективная реализация оптимизационных подходов часто требует включения различных алгоритмов для частных случаев задач, кроме того с ростом размерности задач, получить решения за разумное время возможно только с включением комбинаторных процедур с поиском на частичных планах, включая различного рода рандомизированные подходы.

3. Метод оптимальной декомпозиции

В результате выполнения научных работ в рамках данной тематики на кафедре Автоматизированной обработки информации Северо-Кавказского горно-металлургического института(СКГМИ) был разработан математический аппарат оптимизации ПО, которые стали основой для дальнейших исследований и позволили получить тестируемую в настоящий момент программную платформу, объединяющую различные оптимизационные методы в рамках одного инструментального средства. Полученные на настоящий момент результаты основаны на моделях двух классов:

1) Задачи оптимальной декомпозиции программного кода.
2) Модели выбора оптимальной стратегии размещения пользовательских данных в многоуровневой памяти ЭВМ.

Использование терминологии системного анализа в названии первого класса моделей (в частности, «декомпозиция») обусловлено смыслом применяемого оптимизационного подхода, который заключается в улучшении одного из критериев качества программы (производительности, занимаемого объема оперативной памяти) за счет изменения состава подсистем. Для различных условий функционирования программы предлагается два подвида оптимизационных моделей:

1) Модели оптимальной декомпозиции программных систем, функционирующих в условиях недостатка оперативной памяти.
2) Модели «экстремального программирования».

Задачи первой группы формулируются для систем, размер которых не позволяет разместить весь код в оперативной памяти(ОП) при запуске приложения. В этом случае выполняется декомпозиция системы на подгружаемые модули (каждый из которых, отработав, освобождает ресурсы и загружает следующий). Так как разница в скорости работы оперативной и внешней памяти достигает нескольких порядков, то минимизация числа подсистем может дать серьезный прирост в производительности. Набор подсистем очень часто выбирается из субъективных соображений: простота сопровождения группы функций, объединенных в рамках одного модуля; специфика организации процесса разработки – для упрощения контроля версий каждый разработчик (группа разработчиков) «ведет» свой модуль и т.д. Разработанные модели

призваны помочь разработчику принять решение о выборе того или иного варианта декомпозиции на основе объективных оценок.

Оптимизационный процесс можно описать в виде следующих шагов:

1) Идентификация типа пользовательского алгоритма.
2) Выбор критерия оптимальности и ограничений.
3) Решение задачи выбора оптимальной декомпозиции.

Используемый оптимизационный подход использует следующую классификацию пользовательских алгоритмов:

1) Конечные, время поиска решения которых конечно при любых входных данных.
2) Зацикленные – в алгоритме которых отсутствуют варианты завершения.
3) Склонные к зацикливанию – данный тип алгоритма, в зависимости от входных данных, может либо находить решение за конечное время либо зацикливаться.

Для конечного алгоритма критерий оптимальности легко формулируется если программа имеет линейную структуру (т.е. когда все операторы вызываются строго последовательно) – достаточно разбить её на минимальное число подгружаемых подпрограмм, при условии, что размер каждой из них не превысит доступный приложению размер оперативной памяти (ОП) ЭВМ.

В любой момент состояние программы можно характеризовать текущим значением вектора переменных. На рисунке 1 изображен граф, изображающий исходный алгоритм, содержащий четыре оператора. Здесь вершины соответствуют состояниям данных программы, а дугам – операторы, переводящие программу из одного состояния в другое.

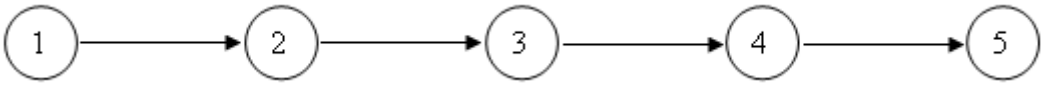

Рисунок 1 – Линейный алгоритм

На рисунке 2 – граф, содержащий все допустимые варианты подпрограмм, размер которых находится в пределах ограничения на объем ОП.

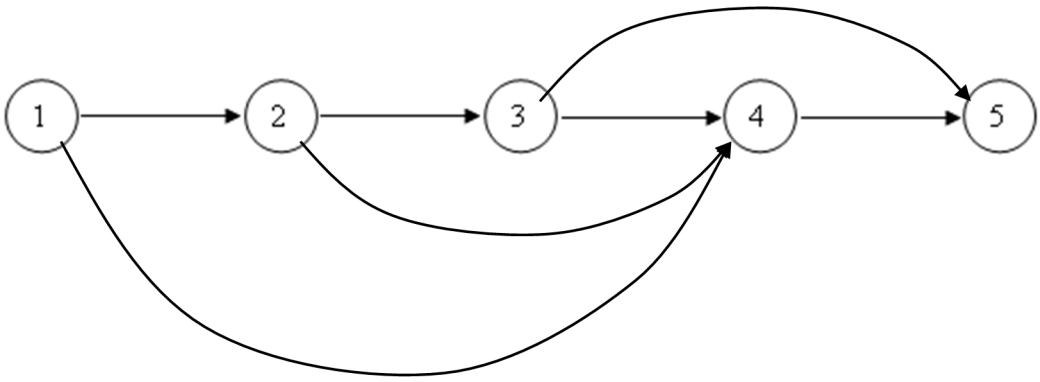

Рисунок 2– Граф вариантов подпрограмм

Каждая дуга графа на рисунке 2 представляет собой один из вариантов объединения операторов – к примеру, дуга (1, 2) означает вариант подпрограммы, состоящий всего лишь из первого оператора, а дуга (2, 4) – из второго и третьего операторов и так далее.

Если для данного примера допустить, что времена загрузки с внешнего устройства хранения в оперативную память для всех подпрограмм идентичны (соответственно равны веса дуг), то оптимальной декомпозиции будет соответствовать решение задачи о кратчайшем пути из начального состояния программы (вершины 1) в конечное (вершина 5), т.е. путь (1,4), (4,5). Таким образом, для данного примера оптимальной декомпозицией является разбиение исходной версии программы на 2 модуля: первый включает операторы 1,2,3 (дуги (1,2), (2,3), (3,4)), а вторая подпрограмма содержит четвертый оператор (дуга (4,5)).

На практике линейные алгоритмы встречаются редко, как правило, структура является ветвящейся и в этом случае критерий оптимальности не столько однозначен. К примеру, пусть в алгоритме существуют 2 различных варианта пути L_1 и L_2 из начального состояния программы в одно из конечных, соответствующих одному из вариантов вычисления решения, и известно, что вероятность реализации операторного переходов по пути L_1 равна 0,1 а по L_2 соответственно 0,9. Если множества операторов, составляющих оба пути являются пересекающимися, то, в общем случае, может существовать вариант разбиения программы, при котором суммарное число подпрограмм будет минимальным, но при этом разбиение подмножества операторов на пути L_2 не будет минимально возможным. Так как информацию о вероятностях операторных переходов невозможно получить в автоматическом режиме, в предлагаемой технологии все переходы приняты равновероятными, а для выделения подмножества наиболее «значимых» путей используется суммарное время выполнения операторов, составляющих этот путь. Существует, очевидно, две стратегии оптимизации: программист-«пессимист» упорядочит все операторные пути в порядке обратном времени их выполнения, затем, найдя оптимальную декомпозицию для первого, самого длительного по времени, пути (соответствующего верхней границе времени поиска решения), продолжит декомпозицию остальных путей, соблюдая непротиворечивость решений. «Оптимист» выберет обратную стратегию, когда в первую очередь осуществляется декомпозиция операторов, приводящих к решению за кратчайшее время. Помимо описанной выше стратегии оптимизации, формулировки задач учитывают режим, в соответствии с которым осуществляется загрузка модулей-подпрограмм. При последовательной загрузке (режим «Chain») отработавший модуль передает управление следующей подпрограмме и освобождает выделенные для своей работы ресурсы. На практике более распространен другой вариант, когда в памяти, после запуска программы, подгружается и постоянно находится головной модуль, управляющий работой остальных подпрограмм, при этом загрузка остальных модулей осуществляется в порядке, аналогичном режиму «Chain». Так как многокритериальная оптимизация может обладать высокой вычислительной сложностью, то одним из

способов адаптации данного подхода к реальным условиям является одного, главного, критерия, в соответствии с которым будет выполняться декомпозиция. В работе предлагаются два таких критерия: минимизация верхней границы времени поиска решения и минимизация нижней границы времени поиска решения (соответственно стратегии «Пессимист» и «Оптимист»). К примеру, модель оптимизации конечного ветвящегося алгоритма для стратегии «Пессимист» формулируется следующим образом:

$$\begin{cases} F_1 = \max\limits_{x_q \subset X^T} \max\limits_{f} \sum\limits_{j \in \{L^f(0,q)\}} \tau(j) sign \sum\limits_{i} z(i,j) \to \min; \\ \forall i, \sum\limits_{j} z(i,j) = 1; \\ \max\limits_{j} V(j) sign \sum\limits_{i} z(i,j) \le V; \\ z(i,j) = 1,0; i = 1,2,..,n; j = 1,2,... \end{cases} \quad (1)$$

где

V – объем свободной оперативной памяти;

$z(i,j)$ – булева переменная, равная единице, если i-й оператор алгоритма пользователя реализуется j-й программной единицей, и нулю в противном случае;

$G(X,U)$ – взвешенный ориентированный граф, вершины которого отвечают состояниям вектора переменных, а дуги – операторам алгоритма пользователя, переводящим вектор переменных из одного состояния в другое;

$\{L^j(p,q)\}$ – j-ое подмножество программных единиц, реализующих все операторы, отвечающие дугам пути $L(p,q)$;

$X^T \subset X$ – множество терминальных вершин графа $G(X,U)$;

$x_q \in X^T$ – вершина, соответствующая q-му варианту завершения программы.

Модель (1) представляет собой формальную постановку антагонистической позиционной игры двух лиц с полной информацией: минимизирующий игрок объединяет операторы в программные единицы, а максимизирующий выбирает условия их завершения – соответственно для её решения можно применить методы теории игр. Другой подход, двухэтапный, заключается в локализации подмножества вариантов, соответствующих стратегии оптимизации и последующей его декомпозиции на минимальное число подпрограмм. Операторы, не вошедшие в состав подпрограмм, составляющих оптимальную декомпозицию, могут быть распределены по модулям в произвольном порядке (с учет лишь ограничения на доступный объем ОП). Переходу к стратегии «Оптимист» соответствует замена целой функции на (2):

$$F_2 = \min\limits_{x_q \subset X^T} \min\limits_{f} \sum\limits_{j \in \{L^f(0,q)\}} \tau(j) sign \sum\limits_{i} z(i,j) \to \min; \quad (2)$$

Для зацикленных программных алгоритмов термин «время поиска решения» не имеет смысла, так как программа не имеет условий завершения. Одним из объективных критериев оптимальности декомпозиции для этого типа пользовательских программ является минимизация верхней («Пессимист») либо нижней(«Оптимист») границы однократного зацикливания. Соответствующая целевая функция примет вид (3):

$$F_3 = \max_{a_k \in A(G)} \sum_{j \in H(a_k)} \tau(j) sign \sum_i z(i, j) \to \min; \qquad (3)$$

где

$A(G)$ – множество контуров на графе $G(X,U)$.

$a_k \in A(G)$ – k-й контур на графе $G(X,U)$.

$H(a_k)$ – подмножество программных единиц, реализующих операторы, отвечающие дугам, принадлежащим контуру $a_k \in A(G)$;

Для решения задачи оптимальной декомпозиции зацикленных алгоритмов, как и в случае с ветвящимися алгоритмами, можно воспользоваться игровым подходом. При этом меняется лишь алгоритм построения дерева игры – последний ярус составляют вершины, соответствующие всем вариантам зацикливания программы.

Для частных случаев, используются более эффективные алгоритмы. К примеру, поиск оптимальной декомпозиции ветвящихся алгоритмов, минимизирующей нижнюю границу времени поиска решения (стратегия «Оптимист») можно свести к задаче поиска кратчайшего пути на графе, полученного из исходного алгоритма добавлением дуг, соответствующим вариантам подпрограмм, а также фиктивной вершины, являющейся «стоком» для терминальных вершин.

Модели оптимизации декомпозиции актуальны как для сверхбольших систем с большим объемом кода (к примеру, сложные АСУ технологических процессов и производств), так и компактных систем, изначально ориентированных на работу в условиях недостатка оперативной памяти (операционные системы для бытовых приборов, компактных средств коммуникации и подобных устройств).

Вторая группа оптимизационных моделей – задачи «экстремального программирования»,– актуальна для программ, функционирующих в условиях избытка оперативной памяти, к этой категории можно отнести как системные утилиты и службы, так и прикладные приложения, для которых производительность является критически важным фактором. Один из таких подходов – увеличение производительности кода за счет сокращения суммарного процессорного времени, затрачиваемого на операции выделения и освобождения стековых переменных. Для объектов, расположенных в статической (глобальной) памяти, временем создания можно пренебречь, так как они создаются однократно в момент загрузки процесса в оперативную память. Стековые (локальные) переменные автоматически создаются каждый раз, когда начинается выполнение блока операторов, в пределах которого они объявлены, и уничтожаются, когда блок завершается. К примеру, если локальная переменная объявлена в пределах блока цикла «for», суммарное время, затрачиваемое

на операции создания и удаления этой переменной будет пропорционально числу циклов. Таким образом, имеется очевидный потенциал роста производительности в результате замен стековых локальных (класс «auto») переменных на статические локальные («static») либо глобальные переменные. Очевидно, существуют два ограничения на использование подхода: нельзя применять к рекурсивным функциям; метод не поддерживает многопоточность. Учитывая, что по завершении работы функции стековая память, выделенная для её работы, освобождается, легко подсчитать объем дополнительной оперативной памяти, который будет занимать программа после оптимизации (4):

$$V_{advanced} = \sum_{i=1}^{n} v_i - \max_i(v_i). \tag{4}$$

где

v_i – размер стека, необходимый для размещения локальных стековых переменных i-й функции.

Если принять линейный характер зависимости времени выделения стека от размера выделяемой стековой области, то суммарное время работы всех функций уменьшится на следующую величину (5):

$$dT = k\sum_{i=1}^{n} v_i \tag{5}$$

где

k – коэффициент пропорциональности, отражающий зависимость времени выделения стека от его размера;

dT – выигрыш во времени.

В том случае, если доступной для оптимизации памяти недостаточно для модификации всех функций, то выбор тех из них, к которым будет применяться замена класса памяти, можно описать следующей моделью:

$$\begin{cases} F_4 = \max_{x_q \subset X^T} \max_f \sum_{j \in \{L^f(0,q)\}} z_j N_j k v_j \to \min; \\ \sum_{i=1}^{n} z_i v_i - \max_i(z_i v_i) \le V. \\ z_i = 1,0; i = 1,2,..,n; j = 1,2,... \end{cases} \tag{6}$$

где

N_j – верхняя граница вызовов j-й функции;

V – верхняя граница доступной для оптимизации дополнительной памяти;

z_i – булева переменная, равная 1, если для i-й функции выполнена замена стековых переменных статическими.

Другой моделью экстремального программирования является метод макрозамен, популярный метод повышения производительности кода, который заключается в том, что код функции вставляется в место вызова. Выигрыш в производительности составляют в этом случае накладные расходы машинного времени, затрачиваемого на вызов функции, выделение и освобождение локального стека и передачу параметров.

$$\begin{cases} F_4 = \max\limits_{x_q \subset X^T} \max\limits_{f} \sum\limits_{j \in \{L^f(0,q)\}} (z_j \sum\limits_{i=1}^{n} b_{ij} t_i^{call}) \to \min; \\ \sum\limits_{j=1}^{m} (z_j (\sum\limits_{i=1}^{n} b_{ij} v_i^{code})) \le V; \\ \forall j : \sum\limits_{i=1}^{n} b_{ij} = 1; \\ z_i = 1,0; i = 1,2,..,n; j = 1,2,...,m \end{cases} \quad (7)$$

где

m – количество вызовов функций в программе;

n – количество функций;

b_{ij} – вспомогательная булева переменная (входные данные), равная 1, если j-й вызов передает управление i-й функции;

t_i^{call} – время, затрачиваемое программой на передачу управления (и возврат из нее) i-й функции.

v_i^{code} – размер кода i-й функции;

V – верхняя граница доступного для оптимизации дополнительного объема оперативной памятиж

z_j – булева переменная, равная 1, если для j-й вызова выполняется макрозамена, и 0 – в противном случае.

В моделях (6) и (7) , по аналогии с задачами оптимальной декомпозиции, решение выбирается на подмножестве функций, составляющих верхнюю границу времени поиска решения. Для нахождения оптимального плана $\vec{z}$ можно воспользоваться сочетанием методов теории графов – в частности, поиск длиннейшего пути для выделения верхней границы времени поиска решения($\max\limits_{f} L^f(0,q)$), – и методами дискретного программирования для вычисления оптимальных значений z_j. Причем

индексы j, рассматриваемые на втором шаге локализованы в пределах найденного на первом шаге пути (т.е $j \in \{L^f(0,q)$).

Для успешной реализации программных средств оптимизации исходных кодов необходимо решить ряд прикладных задач в том числе:

1) Разработка прикладных методов оценки объективных характеристик пользовательского программного алгоритма и подготовка исходных данных для оптимизатора.
2) Выбор технологии интеграции разрабатываемых решений в существующие среды разработки.
3) Обеспечение максимальной автоматизации процесса оптимизации – пользователь должен отвлекаться на ввод только тех параметров, которые невозможно вычислить автоматически на основе анализа кода.

Одной из наиболее трудоемких задач первого типа является оценка объема оперативной памяти, занимаемой пользовательским кодом. Общий объем памяти, занимаемый программой, представляет собой сумму четырех компонентов:

1) размер машинных инструкций;
2) размер статических данных (глобальные переменные и массивы, а также локальные статические переменные);
3) стековые переменные (локальные не статические);
4) верхняя граница выделяемой динамической памяти.

Вычисление занимаемого подпрограммой объема ОП зависит от типа языка.

Для интерпретируемых языков (виртуальные машины) размер программы приблизительно можно вычислить как сумму объемов данных (переменных, массивов) и текста программы. Размер байт-кода и другие ресурсы ОП, выделяемые виртуальной машиной точно оценить невозможно.

Для кода, написанного на компилируемых языках, размер занимаемой оперативной памяти можно оценить с высокой степенью точности, воспользовавшись тем фактом, что размер объектного кода, полученного при компиляции исходной версии программных кодов, является точным объемом статической памяти, занимаемого инструкциями и статическими переменными.

Для вычисления размера данных (переменных и массивов) можно реализовать рекурсивным разбором синтаксических структур типа class (пользовательские классы), struct (структуры) и union(объединение). На каждом шаге разбора суммируются размеры полей стандартных типов, в том случае, если поле имеет пользовательский тип, размер которого пока неизвестен, то выполняется его рекурсивный разбор. Пользуясь данным методом можно получить размеры стековых областей для каждой функции. При этом, из рассмотрения следует исключать локальные объявления переменных и массивов с ключевым словом static и volatile. Спецификатор volatile в «Си»-подобных языках назначается для переменных(или массивов) значение которых может непредсказуемо измениться в результате действия внешних программ, поэтому произвольно изменять класс памяти подобных данных нельзя.

4. Результаты работы

На текущем этапе, получены ряд важных научных и прикладных результатов: математические модели оптимизации программных кодов и методы их решения; программные средства оптимизации – в частности «САПР оптимальных программных комплексов», представляющий собой демонстратор технологий оптимизации. Программа выполнена в виде самостоятельного приложения, написанного на языке C++ для платформы Windows и включает реализации методов оптимальной декомпозиции исходных программных продуктов для различных типов алгоритмов (конечных и циклящихся), а также алгоритмы вычисления оптимальных размеров кэши-блоков для используемых информационных файлов [1].

Интерфейс программы выполнен в соответствии со стандартом MDI и поддерживает 3 типа документов:

1) Окно редактирования исходного кода (рисунок 1) с возможностью переключения к просмотру алгоритма в виде графа (рисунок 2). Анализ исходного кода, построение алгоритма и соответствующего графа выполняются автоматически.

2) Окно ввода параметров оптимизации, в котором пользователь вводит объем доступной для оптимизации оперативной памяти (рисунок 3).

3) Окно вывода результатов оптимизации, содержащее оптимальное разбиение исходного кода на подпрограммы (рисунок 4). Данное окно поддерживает просмотр каждой подпрограммы в текстовом виде либо в виде графа.

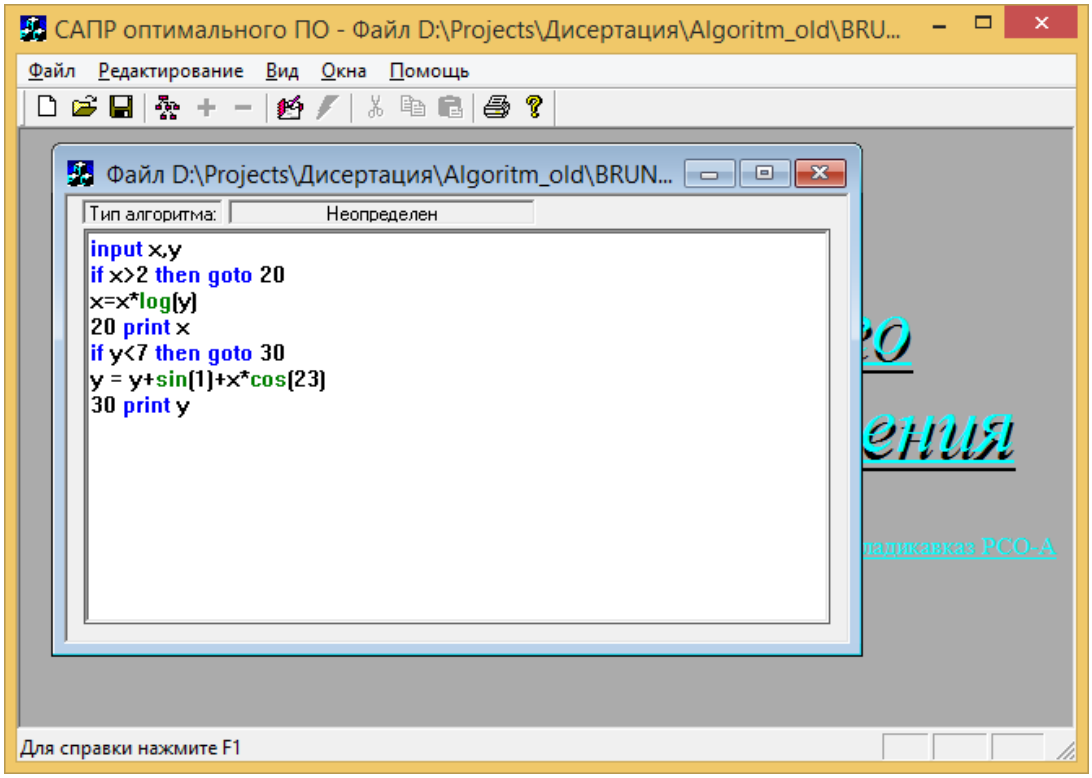

Рисунок 1 – Окно первого модуля с исходным кодом на языке BASIC

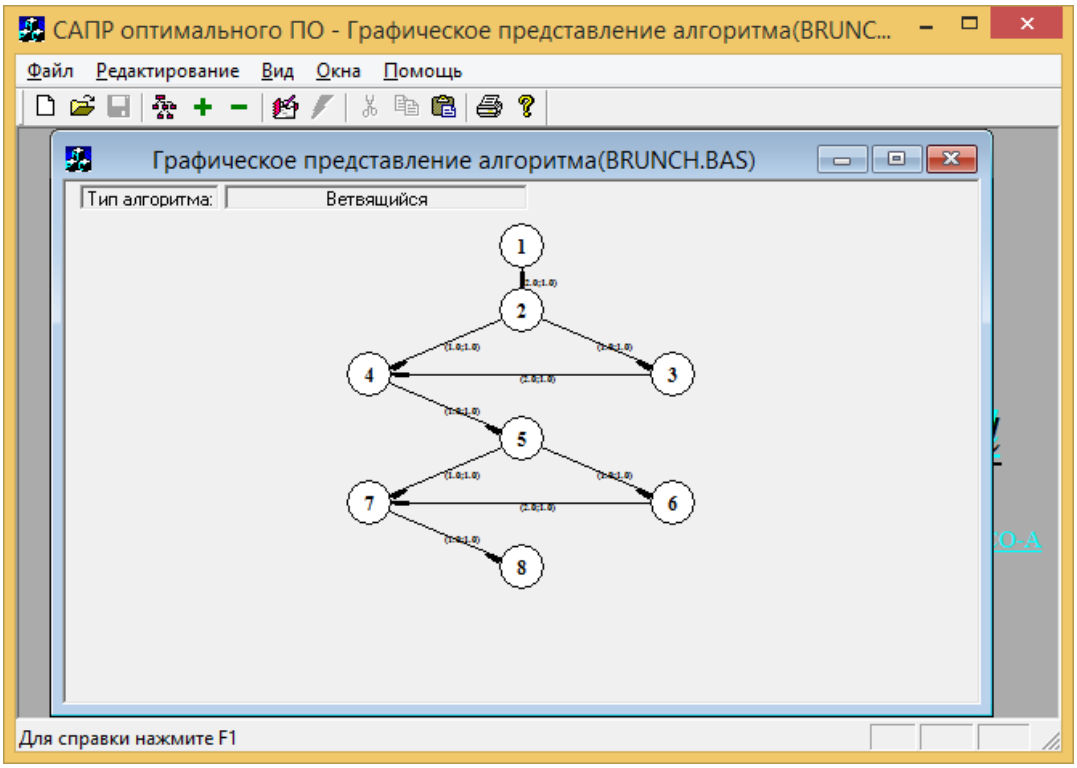

Рисунок 2 – Представление алгоритма пользователя в виде графа

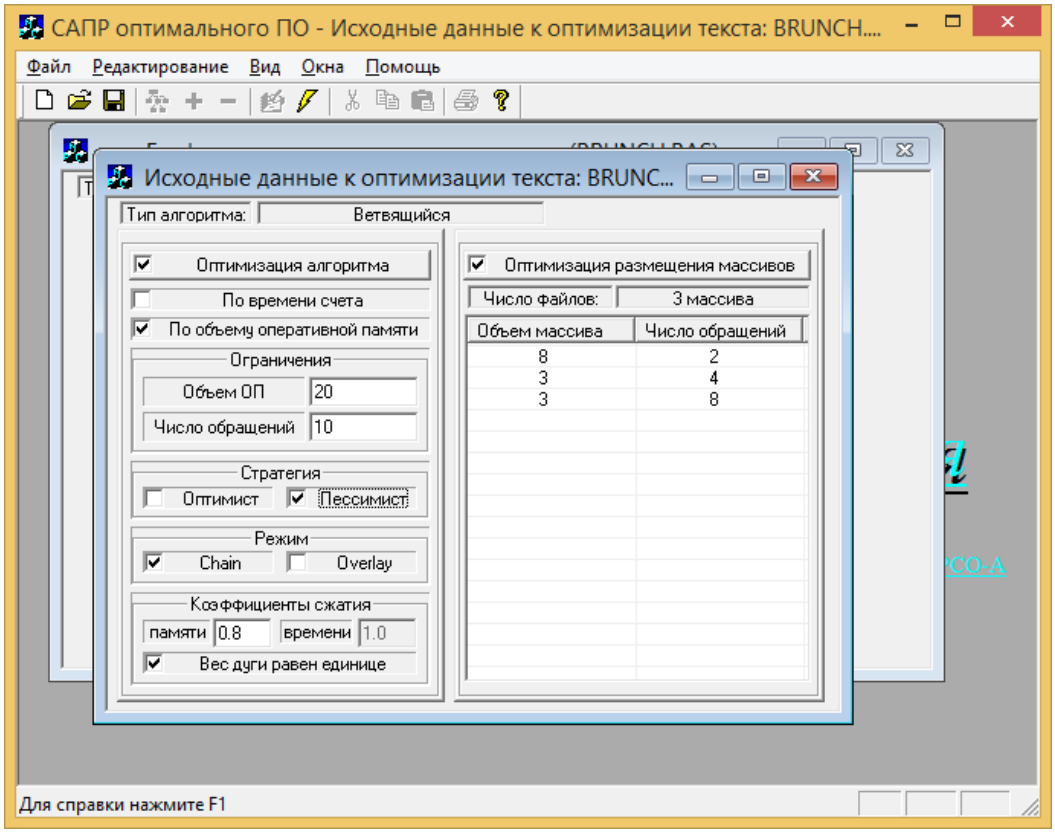

Рисунок 3 – Панель ввода параметров оптимизации

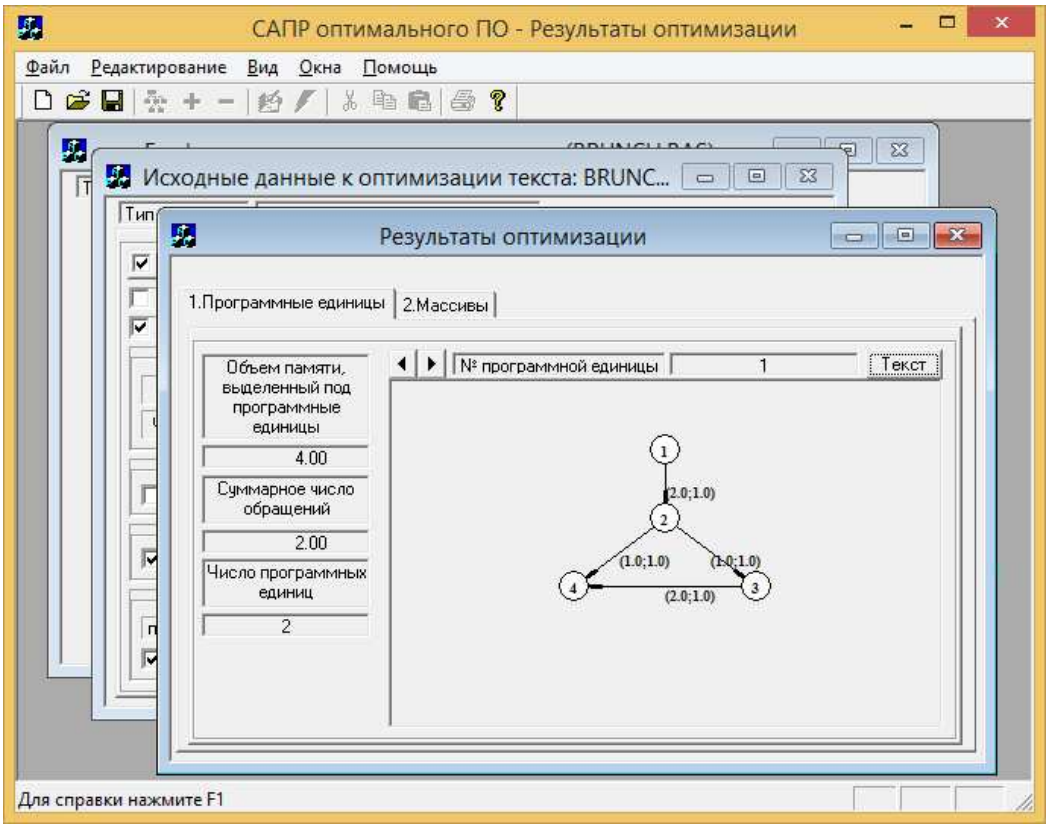

Рисунок 4 – Оптимальная декомпозиция программного кода – представление каждого программного модуля в виде подграфа

5. Заключение

Разработанный программный продукт используется в учебном процессе, а методы и модели, полученные при его разработке используются в оптимизационном продукте следующего поколения, работы над которым в настоящее время продолжаются – он будет представлять собой оптимизационную надстройку для одной из наиболее популярных на платформе Windows сред разработки MS Visual Studio, созданную по технологии «MS Add-In» и позволит автоматизировать вычисление оптимальных планов оптимизационных задач, предлагая разработчику наилучшую компоновку приложения, улучшающую его производительность. В заключение следует отметить, что создание и внедрение программных средств поддержки технологии оптимизации программных продуктов позволит не только улучшить качество разрабатываемого ПО – появление удобных инструментальных средств также будет способствовать более широкому распространению идей и принципов оптимизации.

Список информационных источников

[1] Гроппен В.О. Эффективные стратегии использования кэш-памяти // Автоматика и телемеханика № 1, 1993 г., с. 173-179.

[2] Остроух А.В. Ввод и обработка цифровой информации: учебник для нач. проф. образования / А.В. Остроух. – М.: Издательский центр «Академия», 2012. – 288 с. – ISBN 978-5-7695-9457-1.
[3] Остроух А.В. Основы информационных технологий: учебник для сред. проф. образования / А.В. Остроух. – М.: Издательский центр «Академия», 2014. – 208 с. – ISBN 978-5-4468-0588-4.
[4] Остроух А.В. Мультимедиа-технологии / А.В. Остроух, А.М. Васьковский, А.Б. Николаев. – Saarbrucken, Germany: Palmarium Academic Publishing, 2012. – 228 p. – ISBN 978-3-659-98030-5.
[5] Демидов Д.Г. Программные и аппаратные средства систем мультимедиа. Часть 1. Аппаратные средства: учебное пособие / Д.Г. Демидов, А.М. Васьковский, А.Б. Николаев, А.В. Остроух, П.И. Лукащук, В.А. Виноградов. – М.: МГУП имени Ивана Федорова, 2014. – 78 с.
[6] Демидов Д.Г. Программные и аппаратные средства систем мультимедиа. Часть 2. Программные средства: учебное пособие / Д.Г. Демидов, А.М. Васьковский, А.Б. Николаев, А.В. Остроух, П.И. Лукащук, В.А. Виноградов. – М.: МГУП имени Ивана Федорова, 2014. – 70 с.
[7] Пастухов Д.А., Юрчик П.Ф., Остроух А.В. Сравнительный анализ гипервизоров // Международный журнал экспериментального образования. – 2015. – № 3-3. – С. 346-350.
[8] Пастухов Д.А., Юрчик П.Ф. Сравнительный анализ гипервизоров // Автоматизация и управление в технических системах. – 2014. – № 4 (12). – С. 129-140. DOI: 10.12731/2306-1561-2014-4-13.
[9] Сальный А.Г., Остроух А.В. Исследование производительности файловых систем ядра Linux // Автоматизация и управление в технических системах. – 2014. – №4 (12). – С. 158-167. DOI: 10.12731/2306-1561-2014-4-16.
[10] Сальный А.Г., Остроух А.В. Исследование эффективности структур хранения данных ядра Linux // Международный журнал экспериментального образования. – 2015. – № 3 – С. 366-367.
[11] Терентьев Д.И., Николаев А.Б., Остроух А.В. Исследование дисковых массивов RAID по параметрам надежности и быстродействия // Международный журнал экспериментального образования. – 2015. – № 3-3. – С. 423-427.