На правах рукописи

Погибельский Дмитрий Александрович

МАТЕМАТИЧЕСКАЯ МОДЕЛЬ ВИРТУАЛЬНОЙ МАШИНЫ JAVA, АВТОМАТИЧЕСКИ АДАПТИРУЮЩЕЙСЯ К ОСОБЕННОСТЯМ ВЫПОЛНЯЕМОГО КОДА

Специальность 05.13.18 – Математическое моделирование, численные методы и комплексы программ.

АВТОРЕФЕРАТ

диссертации на соискание ученой степени

кандидата физико-математических наук

Москва–2007

Работа выполнена на кафедре прикладных информационных технологий Московского физико-технического института (государственного университета).

Научный руководитель: доктор физико-математических наук, член-корреспондент РАН, профессор Никитов Сергей Аполлонович

Официальные оппоненты: доктор физико-математических наук, профессор Крюковский Андрей Сергеевич

кандидат физико-математических наук Бельчик Александр Анатольевич

Ведущая организация: Институт проблем информатики РАН

Защита состоится 26 октября 2007 в 11^{30} часов на заседании диссертационного совета К212.156.02 в Московском физико-техническом институте (государственном университете) по адресу: 141700, Московская область, г. Долгопрудный, Институтский пер., д. 9, ауд. 903 КПМ.

С диссертацией можно ознакомиться в библиотеке Московского физико-технического института (государственного университета).

Автореферат разослан 25 сентября 2007.

Ученый секретарь
диссертационного совета К212.156.02
к. ф.-м. н. О. С. Федько

Общая характеристика работы

Актуальность работы

В середине 90-х годов появилась технология программирования Java. Это привело к бурному развитию и распространению концепции управляемого кода. На сегодняшний день реализации этой технологии, такие как Java и Microsoft.NET прочно закрепились на лидирующих позициях в вопросах разработки программного обеспечения. Главные их достоинства состоят в безопасности и переносимости кода, простоте и удобстве для разработчика, что позволяет быстро и качественно разрабатывать программное обеспечение (ПО) для разнообразного парка устройств, в том числе и мобильных. Это происходит за счет компиляции исходного текста программы в универсальный байт-код, одинаковый для всех платформ. К сожалению, производительность управляемого кода, как правило, ниже по сравнению с неуправляемым. Это вызвано наличием дополнительного слоя ПО, с помощью которого и выполняется управляемый код. Этот слой получил название контролирующего окружения или виртуальной машины. На сегодняшний день наиболее универсальным языком программирования, использующим технологию управляемого кода, является язык Java, потому сосредоточим внимание именно на нем.

Для выполнения Java-приложения на платформе должна быть реализована виртуальная машина – стековая машина для выполнения инструкций байт-кода, которая сама по себе является достаточно сложным и требовательным к ресурсам системы приложением. Широкое распространение технологии требует разработки такой виртуальной машины, которая могла бы функционировать на широком спектре различных платформ с минимальными переделками. Некоторые из этих платформ характери-

зуются крайне малым объемом доступной памяти, и это также должно быть учтено.

Одним из традиционных способов повышения быстродействия виртуальной машины является редукция языка, как это сделано в технологии Java Card, применяемой на микропроцессорных картах. Другим способом является предварительная компиляции кода и превращение виртуальной машины в компилятор, задачей которого является не интерпретация байт-кода, а генерация оптимального платформозависимого кода. Впервые такой подход был применен к языку Smaltalk, но нашел свое применение и в Java, и в .NET. Несмотря на существенное повышение производительности, эта технология абсолютно не переносима на другие платформы, и на сегодняшний день реализована лишь на ограниченном числе платформ, а именно на высокопроизводительных серверах, настольных компьютерах и лишь на некоторых топовых моделях мобильных устройств. Также проводились работы по созданию интерпретатора Java, использующего алгоритмы и типы данных, максимально оптимизированные под конкретную аппаратно-программную платформу. Несмотря на хорошие результаты, подход остается неприемлемым, поскольку решение не является переносимым и для каждой новой платформы необходим экспертный выбор оптимальных алгоритмов, и возможно, переделка некоторых частей системы.

Очевидно, что необходим способ оптимизировать процесс интерпретации байткода Java-приложения, абстрагируясь от конкретной реализации алгоритмов интерпретатора и, в то же время, максимально учитывая особенности самого исполняемого кода.

Цели и задачи диссертационной работы

Цель диссертационной работы состоит в том, чтобы разработать математическую модель интерпретатора Java, который бы адаптировался к особенностям выполняемого кода для повышения производительности и оптимизации использования ресурсов памяти. В рамках сформулированной цели необходимо:

1. Предложить критерий оптимальности выполнения Java-приложения;
2. Разработать способ выделения особенностей байткода кода Java;
3. Предложить конкретную методику оптимизации.

Научная новизна работы

Научная новизна работы заключается в разработке абстрактной модели представления метаданных Java-приложения, которая может быть использована для построения самоадаптирующихся интерпретаторов. Метаданные это структурированная полная информация о типах данных, используемых приложением.

На сегодняшний день вопросы построения трансляторов тщательно исследованы, однако, доминирующим направлением является изучение вопросов построения оптимального кода и вопросов организации языков программирования в целом. Вопросы оптимизации промежуточного представления кода и разработки интерпретаторов являются второстепенными и рассмотрены мало.

Основное достоинство модели заключается в абстрагировании от конкретных особенностей аппаратно-программной платформы, в то же время, она позволяет, вводить в рассмотрение формальные поправки для

учета таких параметров системы как объем памяти и производительность центрального процессора. Основываясь на предложенной модели и математическом аппарате теории графов и теории алгоритмов, сформулирована задача оптимизации процесса выполнения Java-приложения.

Применение математического аппарата генетических алгоритмов (ГА) позволило предложить эффективный метод численного решения поставленной задачи оптимизации.

На защиту выносятся:

1. Математическая модель, позволяющая учитывать произвольное число параметров платформы и выполняемого приложения, и сформулировать критерий оптимальности процесса выполнения Java-приложения.

2. Метод эффективного решения задачи структурной оптимизации графа метаданных Java-приложения, основанный на применении генетических алгоритмов.

3. Методика управления динамической памятью, демонстрирующая наилучшие асимптотические характеристики для большинства Java-приложений в случае дефицита памяти.

Апробация работы.

Материалы исследования докладывались и получили положительную оценку на научных конференциях и семинарах: II Московский научный форум, 6-я научно-практическая конференция «Московская наука, проблемы и перспективы» Москва, 13 - 17 июля 2005г.; V Международная конференция «Электроника и информатика 2005» Москва, 23 - 25 ноября 2005 г.; 13-я Всероссийская конференция «Микроэлектроника и ин-

форматика 2006», Москва, 19 - 21 апреля 2006 г.; «INTERMATIC-2006», Москва 5 - 9 декабря 2006 г., научные конференции МФТИ, Долгопрудный в 2005-2006 гг.; Семинарах кафедры прикладных информационных технологий МФТИ 2004-2007 гг.

Практическая ценность работы.

Результаты исследований по построению методики оптимального управления памятью [1] и модель самоадаптации виртуальной машины Java применялись при разработке интерпретатора Java для систем цифрового телевидения в компании «Samsung Electronics». Работа проводилась в рамках стажировки в лаборатории Infra Team IP STB Lab, г. Сувон, Южная Корея, в 2006 г.

Публикации.

По теме диссертации опубликовано восемь работ, в том числе две – из списка изданий, рекомендованных ВАК РФ.

Структура и объем диссертации.

Диссертация состоит из введения, трех глав, заключения, списка использованных источников, включающего 84 работы, и изложена на 100 страницах.

Основное содержание работы

Во введении обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, показана практическая значимость полученных результатов, представлены выносимые на защиту научные положения.

При выполнении байткода Java-приложения, интерпретатор создает в памяти сложную иерархическую структуру, содержащую метаданные приложения. Поскольку в технологии Java используется позднее связывание на основе символьных имен, то основной расход времени приходится на операции поиска нужных данных по текстовому ключу в структуре метаданных. Чтобы сократить время выполнения приложения необходимо, в первую очередь, оптимизировать структуру метаданных. В то же время, оптимизация должна сохранить все достоинства позднего связывания, поэтому известные технологии предварительного связывания неприемлемы.

Замечено, что программы, разработанные на одном языке программирования и для одной предметной области достаточно схожи между собой. Этому способствует принятая в настоящее время практика унификации подхода к разработке ПО, в первую очередь, применение шаблонов проектирования и повторного использования кода. Также необходимо принять во внимание тот факт, что при применении *javac* – самого распространенного компилятора Java, одинаковые синтаксические конструкции компилируются в одинаковые структуры байткода. Все это дает основания предполагать гомогенность атрибутов байт-кода по всему приложению. Для выделения этих атрибутов и разработки методики их учета требуется разработка математической модели.

В первой главе проводится рассмотрение и анализ структур метаданных, необходимых для выполнения приложения. Вводится модель структуры метаданных Java-приложения на основе взвешенного ориентированного графа. Вводятся величины, подлежащие оптимизации, выявляются параметры приложения, их определяющие.

Метаданные Java-приложения удобно моделировать взвешенным ориентированным графом $G = (V, E)$. Его вершинам V соответствуют эле-

менты метаданных, а ребра E представляют информационные зависимости между ними. Под информационной зависимостью b от a понимается тот факт, что для вычисления b необходимо вычислить a, например, чтобы при создании нового объекта определить размер выделяемого блока памяти (b), необходимо определить тип (a), создаваемого объекта.

Вершина R соответствует указателю на начало управляемой памяти, вершины $C(1)$, ..., $C(Mc)$, соответствуют данным загруженных классов. В каждом классе есть $Cp(1)$,...,$Cp(Mcp)$ – набор статических определений, F начало блока определения переменных, $F(1)$, ..., $F(Mf)$ – определения переменных, M – начало блока определения методов, $M(1)$, ..., $M(Mm)$ – определения методов. Через Mc, Mcp, Mf, Mm обозначены соответственно математические ожидания количества классов в приложении, длины списков определений, переменных и методов в каждом классе приложения. Функции распределения этих величин будем считать атрибутами байт-кода.

Другие вершины обозначают: s – размер переменных класса; ts – размер объекта класса, с учетом унаследованных переменных; n – имя класса; sn – имя родительского класса; sr – ссылка на одну из вершин C, соответствующую родительскому классу. Выполнение каждой команды байт-кода Java сводится к последовательности операций поиска путей между двумя вершинами на рассматриваемом графе. Такие операции будем называть элементарными, пусть они составляют множество Z. Например, поиск имени класса $C(1)$ соответствует поиску пути между $C(1)$ и n. Поиск одного из классов по имени соответствует последовательному просмотру всех вершин $C(1)$, ..., $C(Mc)$, и проведению для каждой из них поиска имени. На практике для ускорения таких операций поиска пользуются хэш-таблицами. На графе они промоделированы узлами CH для классов, CF и CM для полей и методов внутри класса, соответ-

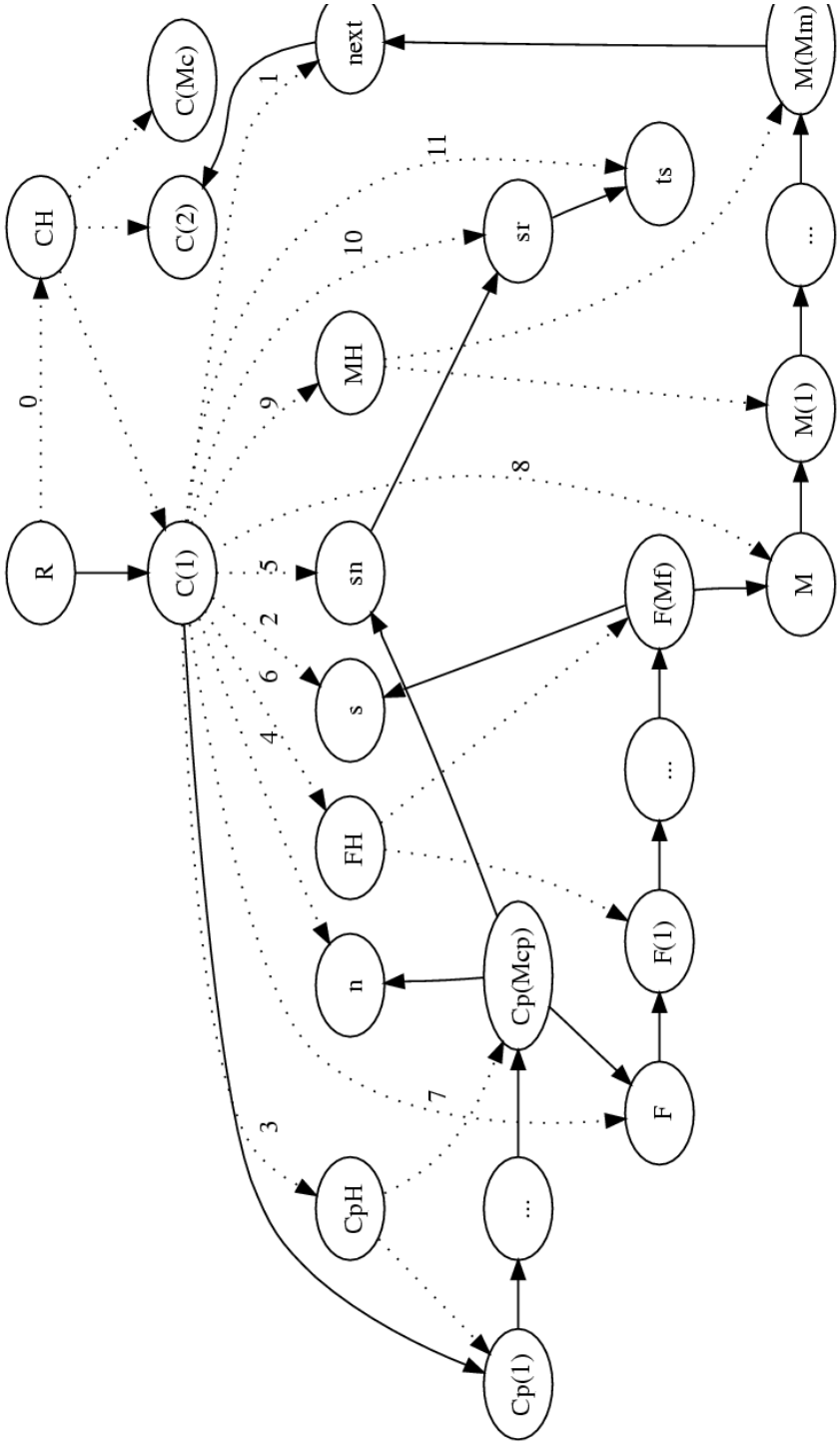

Рис. 1. Схема информационных зависимостей Java-приложения

ственно.

Обратим внимание на то, что множество E может состоять из ребер двух типов: ссылки получаемые «естественным путем», т.е. при загрузке связного списка данных с диска в память; и ссылки, добавляемые дополнительно для ускорения навигации по структуре метаданных. Первые составляют множество E_0 и на рисунке изображены сплошными линиями, а вторые составляют множество $\tilde{E}$. Они изображены пунктиром и помечены числами от 1 до 12. Очевидно, что $E_0 \cap \tilde{E} = \varnothing$ и любая конфигурация E может быть представлена как $E = E_0 \cup E'$, где $E' \subset \tilde{E}$. Добавление в граф ребер второго типа сопряжено с дополнительными расходами памяти. Например, эксперимент показывает, что разница необходимых объемов памяти для одного и того же приложения при $E' = \varnothing$ и $E' = \tilde{E}$ составляет до 40%.

Для простоты принято, что эффективность выполнения Java-приложения определяется только вычислительной сложностью производимых действий и объемом необходимой оперативной памяти. Каждому ребру графа G поставим в соответствие вес $\mathbf{w} = (w_c, w_m)^T$. Компонента w_c обозначает вычислительную сложность алгоритма перехода между вершинами, а w_m требуемый объем дополнительной памяти для организации такого ребра. Характерно, что $w_m = 0$ для ребер из E_0 и $w_c = 1$ для ребер из $\tilde{E}$. Сложность алгоритмов в каждом конкретном случае оценивается теоретически и уточняется в соответствие с их реализацией на языке программирования.

Для учета требований, обусловленных особенностями платформы вводится вектор штрафов $\mathbf{q} = (q_c, q_m)^T$, $0 < q_c,\ q_m$ и $q_c + q_m = 1$. Чем больше первая компонента, тем больше штраф за нагрузку на процессор, чем больше вторая - тем больше штраф за перерасход оперативной памяти.

В первом приближении набор команд байт-кода Java можно разде-

лить на $n_c = 4$ классов: «арифметические»; доступ к переменным объекта; доступ к методам объекта; создание нового объекта. Под «арифметическими» командами понимаются все операции, ограничивающиеся работой с локальными переменными, как арифметические, так и, например, условные операторы. Более глубокое расщепление команд на классы может привнести большую гибкость, но велик риск потерять наглядность.

Для каждого класса команд можно выделить строгую последовательность элементарных операций $\{z_k\}_{k=0}^{N_j}$. Например, чтобы создать объект, необходимо выделить для него место в памяти, для этого необходимо подсчитать необходимый размер, для это, в свою очередь, необходимо найти метаданные класса, к которому принадлежит объект и т. д. Эти операции необходимо производить именно в таком порядке и ни в каком другом. Каждой элементарной операции соответствует множество источников $R(z_k) \subset V$ и цель $d(z_k) \in V$. Тогда математическая сложность выполнения команды определенного типа может быть оценена как

$$\mathbf{l}_j(E) = \sum_{k=0}^{N_j} \hat{T}(E, z_k),$$

где $\hat{T}(E, z_k)$ – оператор, вычисляющий математическое ожидание сложности разрешения информационной зависимости для операции z_k. Она численно равна сумме весов ребер, входящих в кратчайший путь к вершине $d(z_k)$ из любой вершины $R(z_k)$. Эта величина носит вероятностный характер из-за того, что такой же характер носит сама структура графа G. Для «арифметических» операций положим $\mathbf{l}_1 \equiv (1, 0)^T$.

Введем вектор $\xi = (\xi_1, \ldots, \xi_{n_c})$, задающий частотное распределение для встречающихся в приложении команд байткода каждого класса,

$$\xi_i > 0,\ i = 1, \ldots, n_c \text{ и } \sum_{i=1}^{n_c} \xi_i = 1.$$

Все такие ξ образуют множество $\mathbf{\Xi}$. Вектор ξ также включим во множество атрибутов байт-кода. Можно составить функционал для расчета математического ожидания сложности выполнения Java-приложения

$$\mathbf{J}(\xi, \mathbf{q}, E_0 \cup o) = \sum_{j=1}^{n_c} \xi_j \, \mathbf{q}^T \, \mathbf{A} \, \mathbf{l}_j. \tag{1}$$

Матрица $\mathbf{A}$ служит для учета особенностей системы управления памятью на используемой платформе, если они не рассматриваются, то она тождественно равна единичной матрице. Если $\mathbf{q}$ и ξ рассматривать как параметры, то задача оптимального выполнения байт-кода Java сводится к задаче структурной оптимизации множества E за счет оптимального выбора ребер, формирующих множество E'. Пространством поиска является множество всех подмножеств множества $\tilde{E}$, обозначенное как O. Каждый его элемент задает допустимую конфигурацию множества ребер графа. Формально задача оптимизации может быть записана следующим образом:

$$o^* = \arg\max_{o \in O} \mathbf{J}(\xi, \mathbf{q}, E_0 \cup o) \tag{2}$$

Во второй главе демонстрируется способ учета в критерии оптимальности таких свойств моделируемой платформы, как объем памяти и производительность центрального процессора. Используя результаты профилирования типичных Java-приложений, в рассмотрение вводятся величины, характеризующие использование памяти. Используя их функции распределения, проводится анализ наиболее популярных методик управления динамической памятью. Разрабатывается собственная гибридная методика, являющаяся во многих случаях оптимальной.

Очевидно, что эффективность низкоуровневой работы с памятью критична для производительности виртуальной машины. В математической модели это нашло отражение посредством матрицы $\mathbf{A}$ в (1).

Реализация технологии Java требует организации автоматического управления памятью. Это означает, что должны решаться следующие две задачи:

1. Обеспечить выделение и освобождение блоков памяти;
2. Обеспечить способ отслеживать неиспользуемые более блоки и помечать их для освобождения и последующего повторного использования.

За последние годы разработано множество решений второй задачи, несмотря на это, все методики, удовлетворительно работающие в небольшом объеме памяти основаны на последовательном обходе графа структуры данных программы и маркировании всех достигнутых узлов. Затем маркер инвертируется и память, занятая помеченными узлами освобождается. Сложность такого алгоритма в любом случае линейна по числу узлов.

Разработано и подробно исследовано множество решений первой задачи. Однако, для повышения эффективности работы программной системы, разработчики каждый раз из различных методик динамического управления памятью делают выбор в пользу одной из них, руководствуясь при этом лишь экспертными оценками и качественными характеристиками использования динамической памяти прикладным программным обеспечением. Необходим формальный критерий для оценки накладных расходов каждой из методик, чтобы с его помощью можно было выяснить границы областей применения каждой из методик по отношению к какой-либо конкретной задаче. Привлекательной выглядит и возможность автоматизации выбора в системах автоматического проектирования.

Каждая методика характеризуется асимптотической сложностью выполнения следующих операций: выделения нового блока; освобождения блока; чтения или записи в блок. Также каждая методика характеризуется степенью фрагментации. Под степенью фрагментации понимается величина

$$\varphi = \frac{M_t - M_f}{\sum m_i} > 1,$$

где M_t – суммарный размер памяти, M_f – объем свободной памяти, m_i – размеры запрашиваемых у системы блоков. Чем ближе φ к 1, тем методика более эффективная.

В основу разработанной методики положена хорошо исследованная простейшая методика единого списка блоков. Отличие заключается в том, что доступная память делится на две части. В одной подряд выделяются блоки, а в другой ведется оглавление – список всех выделенных блоков, как свободных, так и занятых, по аналогии с методикой «битовой маски». В этом списке содержится информация о начальном адресе блока, его размере, соседях, а также может храниться любая служебная информация, регламентирующая работу с этим блоком. В ответ на запрос о выделении памяти прикладное программное обеспечение получает указатель на строку в этом списке. Оглавление отделено от списка для удобства работы прикладного программного обеспечения, если для уменьшения внешней фрагментации блок будет перемещен, то это никак не скажется на работе. Такой подход в литературе получил название косвенной адресации. Следуя принятой в литературе нотации для сложности алгоритмов, можно оценить сложность выполнения арифметических операций. Стоимость операции выделения нового блока в общем случае пропорциональна длине списка оглавления N_b. В случае если необходимо производить уплотнение памяти, то сложность этой операции также

пропорциональна количеству выделенных блоков $\Theta(N_b)$, таким образом, сложность выделения нового блока равна $\Theta(N_b)$. Операция освобождения блока сводится к добавлению пометки в списке и, возможно, проведению слияния блоков, если один или оба соседей на тот момент свободны. В любом случае, число требуемых операций фиксировано и может быть оценено как $r = \Theta(1)$. Поскольку выделенные блоки непрерывны и в прикладной программе содержится указатель на начало блока, то стоимость операций чтения или записи пропорциональна только длине данных, для одного машинного слова это $\Theta(1)$. Методика являет собой определенный компромисс между быстродействием и экономичностью памяти. Она разрабатывалась специально для применения во встраиваемых системах с жестким дефицитом памяти.

В контексте вопроса поиска в структуре метаданных, наиболее критичными представляются операции чтения/записи, а также, общая компактность создаваемых структур данных, то есть степень фрагментации. Тогда в простейшем случае, рассмотренном при построении критерия (1), матрица $\mathbf{A}$ может иметь вид:

$$\mathbf{A} = \begin{pmatrix} r & 0 \\ 0 & f \end{pmatrix}.$$

Матрица $\mathbf{A}$ имеет смысл некоторого уточняющего коэффициента, учитывающего особенности низкоуровневой работы с памятью.

В третьей главе предлагается решение задачи оптимизации структуры метаданных Java-приложения. Решение основано на применении математического аппарата генетических алгоритмов (ГА).

Применение классических методов оптимизации для таких задач как (2) чрезвычайно затруднено непредсказуемостью ландшафта целевой функции. Действительно, нельзя априори постулировать ни дифференциру-

емость, ни непрерывную зависимость от параметров, ни даже монотонность. В решении подобных задач хорошо себя зарекомендовали генетические алгоритмы (ГА). Их первые применения относятся к 70-м годам 20-го века, и к настоящему времени они доказали свою пригодность и к решению NP-трудных задач. Поскольку теоретические обоснования методики ГА выходят за рамки диссертационной работы, то при решении поставленной задачи оптимизации (2) применялись только простейшие методы, теоретические обоснования которых тщательно проверены.

Поскольку ГА осуществляет поиск в Хеммингговом пространстве бинарных строк соответствующей размерности, то прежде чем применить к решению задачи классический ГА, для каждого подмножества $\tilde{E}$ определяется преобразование представления следующим образом

$$e : O \to H_L \;\text{ пусть } \forall o \in O \; e(o) = (\chi_o(0), \ldots, \chi_o(L-1)),$$

где $L = 12$ число структурных элементов (в нашем случае ребер), подлежащих оптимизации, а $\chi_o(i)$ – характеристическая функция множества o. Она показывает, входит ли в множество o, ребро помеченное на рис. 1 числом i. В терминах ГА элементы H_L называются хромосомами, а элементы множества O особями. Очевидно, что такое преобразование представления обратимо и $\exists e^{-1} : H_L \to O$ так, что $\forall h \in H_L \; \exists e^{-1}(h) \in O$.Это позволяет эффективнее использовать аппарат ГА для поиска решения.

Общая схема применяемого генетического алгоритма выглядит следующим образом:

1. Случайным образом формируется начальная популяция I_0. Способ ее выбора влияет на скорость, но не на факт сходимости поиска. Текущим поколением объявляется начальная популяция: $I \leftarrow I_0$. Стрелка влево обозначает операцию присваивания. Наилучшая

особь определяется как:

$$o^* = \arg\max_{h \in I} f(e^{-1}(h)). \tag{3}$$

где $f : O \to R$, ограниченная функция, принимающая неотрицательные значения.

2. Анализируется условие остановки и схождения поиска.

3. Осуществляется переход к новому поколению n, пока $|I_n < N|$:

   а. Производится селекция $X = \hat{S}(I)$.

   б. Производится скрещивание $Y = \hat{B}(X)$.

   в. Производится мутация $\forall g \in Y \Rightarrow g \leftarrow \hat{M}(g)$.

   г. Потомство переносится в новую популяцию $I_n \leftarrow I_n \cup Y$.

4. Новое поколение объявляется текущим $I \leftarrow I_n$, $n \leftarrow n + 1$. Новая оптимальная особь по аналогии с (3) определяется как

$$o^* \leftarrow \max\left(\arg\max_{h \in I} f(e^{-1}(h)), o^*\right).$$

Для применения аппарата ГА к численному исследованию модели, автором был разработан программный комплекс на языке Java. На этапе селекции действует оператор $\hat{S} : I \to 2^I$, который выбирает из I некоторое его подмножество. В работе, в основном, использовался пропорциональный оператор селекции: за один шаг в множество X для скрещивания отбирается ровно два родителя и вероятность для каждой особи o быть выбранной пропорциональна $f(o)$:

$$P\{h \in X\} = \frac{f(e^{-1}(h))}{\sum_{g \in I} f(e^{-1}(g))}.$$

На этапе скрещивания действует простейший одноточечный оператор $\hat{B} : 2^{H_L} \to 2^{H_L}$. Он обладает несколько худшей способностью к перемешиванию, по сравнению с более сложными, однако его характеристики хорошо изучены. Его действие заключается в том, что случайным образом выбирается $k \in N : 1 \leq k \leq L-1$, обе родительские хромосомы разрезаются перед k-м битом и обмениваются фрагментами с k-го по L-й бит.

На этапе мутации действует оператор однородной мутации $\hat{B} : H_L \to H_L$. Он с небольшой вероятностью $p_m \in (0,1)$ инвертирует каждый бит хромосомы-аргумента. Так вероятность того, что хромосома $h = \{0000\}$ перейдет в хромосому $h' = \{0011\}$ составляет

$$P\{h \to h'\} = (1-p_m)\,(1-p_m)\,p_m\,p_m > 0.$$

Таким образом, каждая хромосома с ненулевой вероятностью может превратиться в любую другую. Этот факт важен для предотвращения преждевременной сходимости в популяциях с небольшой численностью. Этим же объясняется, почему выбор начального поколения не критичен для результата поиска.

Функция f построена на основе функционала (1) следующим образом:

$$f(o) = \frac{1}{\mathbf{J}(\xi, \mathbf{q}, E_0 \cup o)}.$$

Она удовлетворяет требованиям, предъявленным к f и принимает наибольшее значение на наилучшем графе G.

Численный эксперимент наглядно демонстрирует эффективность ГА и преимущества этого метода по сравнению с поиском оптимальной структуры графа полным перебором. В худшем случае, для решения задачи структурной оптимизации (2) требуется примерно на 50% меньше вычислений функционала (1), в лучшем – на 85%.

Есть основания считать, что надлежащий выбор ГА и адекватная детализация системы команд байт-кода позволят, оперативно отслеживая частотные характеристики команд, производить поиск оптимальной структуры метаданных прямо во время выполнения Java-приложения. Это привнесет большую гибкость и повысит эффективность самоадаптации виртуальной машины Java к выполняемому программному коду.

В то же время, полученные результаты исследования модели позволяют выдвинуть гипотезу, что если $o^*(\xi')$ – решение задачи оптимизации (2) при $\xi = \xi'$, для

$$\forall \xi', \xi'' : \|\xi' - \xi''\| < \delta \Rightarrow \|o^*(\xi') - o^*(\xi'')\| < \varepsilon(\delta),$$

где метрика в Ξ используется евклидова, а в H_L Хеммингова. Тогда достаточно будет предварительного вычисления $o^*(\xi)$ для ξ образующих δ-сеть над Ξ, и для текущего значения ξ не будет необходимости решать задачу оптимизации, а можно будет выбрать o^*, соответствующее ближайшему узлу сети. Впрочем, эта гипотеза нуждается в тщательной проверке, что и является главным направлением дальнейших исследований.

В заключении резюмируются результаты исследований, их практическая полезность и возможность применения, намечаются направления и задачи для дальнейшего исследования.

Основные результаты

1. Исследован процесс трансляции Java-приложения с точки зрения использования метаданных.

2. Разработана математическая модель процесса трансляции байт-кода Java. На ее основе сформулирован критерий оптимальности и по-

ставлена многопараметрическая задача оптимизации представления метаданных байт-кода Java во время трансляции.

3. Проведено теоретическое исследование эффективности сочетания явной и автоматической методик управления динамической памятью. На основе исследования разработана гибридная методика управления динамической памятью. Разработан соответствующий программный комплекс. Эффективность методики подтверждена экспериментально.

4. В рамках разработанной математической модели предложен эффективный способ оптимизации структуры метаданных Java-приложения, основанный на использовании аппарата генетических алгоритмов. Эффективность подтверждена экспериментально с помощью специально разработанного программного комплекса.

Список опубликованных работ

1. *Дьяков О. Н., Погибельский Д. А.* Организация управления динамической памятью во встраиваемых операционных системах, основанных на технологии Java // 6-я научно-практическая конференция «Московская наука, проблемы и перспективы»: Материалы конференции, – М.: МКНТ, 2005. – С. 694-702.

2. *Погибельский Д. А.* Оптимизация способа хранения метаданных в интерпретаторе Java.// Оборонный комплекс научно-техническому прогрессу. – 2007 – вып.3 – С. 61-65.

3. *Погибельский Д. А., Никитов С. А.* Применение генетических алгоритмов для оптимизации структуры метаданных Java-приложения.// Известия ВУЗов. Электроника. – 2007 – вып.5 – С. 63-68.

4. *Погибельский Д. А.* Гибридный механизм управления динамической памятью в системах, основанных на технологии Java.// Электроника и информатика – 2005. V Международная научно-техническая конференция: Материалы конференции. Часть 2. – М.: МИЭТ, 2005. – С. 43-44.

5. *Погибельский Д. А.* Механизм автоматического управления динамической памятью в системах, основанных на технологии Java.// Современные проблемы фундаментальных и прикладных наук. Часть VII. Прикладная математика и экономика: Труды XLVIII научной конференции. /Моск. физ. - техн. ин-т. – М. – Долгопрудный, 2005. – С. 157-158.

6. *Погибельский Д. А.* Математическая модель для исследования производительности подсистемы управления памятью с помощью математического аппарата цепей Маркова.// Микроэлектроника и информатика 2006. 13-я Всероссийская межвузовская научно-техническая конференция студентов и аспирантов: Тезисы докладов. – М.: МИЭТ, 2006. – С. 170.

7. *Погибельский Д. А.* Модель самоадаптирующегося контролирующего окружения для языка программирования Java.//Современные проблемы фундаментальных и прикладных наук. Часть VII. Управление и прикладная математика: Труды XLIX научной конференции. /Моск. физ. - техн. ин-т. – М. – Долгопрудный, 2006. – С. 158–159.

8. *Погибельский Д. А.* Математическая модель адаптивного контролирующего окружения для языка программирования высокого уровня.// «INTERMATIC-2006» Материалы Международной научно-технической конференции «Фундаментальные проблемы радиоэлектронного приборостроения», 5-9 декабря 2006 г., Москва. – М.: МИРЭА, 2006, часть 2. - С. 21.

В работах, выполненных в соавторстве, личный вклад автора состоит в следующем: [1] – построение и программная реализация алгоритмов управления динамической памятью; [3] – участие в разработке ключевых идей, построение и программная реализация алгоритма решения задачи оптимизации структуры метаданных.