Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Глобальный оптимизатор для .NET приложений

Дипломная работа студента 544 группы

Серебрянского Андрея Ильича

Научный руководитель: ..... Ломов Д. С.

Рецензент: .....к.ф.-м.н. Булычев Д. Ю.

«Допустить к защите»,

заведующий кафедрой,

д.ф.-м.н., профессор ..... Терехов А. Н.

Санкт-Петербург

2007

Аннотация

Создан программный комплекс для оптимизации многокомпонентных приложений на платформе .NET. В комплексе используется глобальный анализ и различные техники смешанных вычислений. Пользователям программного комплекса Spider предоставлено болшое количество настроек для управления процессом оптимизации, а также интерфейс для расширения возможностей комплекса в будущем.

Оглавление

1 Введение

С момента зарождения индустрии программного обеспечения большое внимание уделяется повторному использованию кода. В структурном программировании повторяющиеся фрагменты программы (либо фрагменты представляющие собой цельные вычислительные блоки) могут оформляться в виде подпрограмм (процедур и функций). Подпрограммы могут быть помещены в отдельные модули, которые могут быть использованы в нескольких программах. В объектно-ориентированном программировании в отдельные модули помещаются не только подпрограммы, но и структуры данных.

В последнее время повторное использование кода перешло на новый уровень. Приложения строятся из большого количества различных компонент. Компоненты создаются внутри компании для собственных продуктов (либо семейств продуктов [4]) либо приобретаются у сторонних компаний, которые профессионально занимаются созданием наборов компонент. Примерами сторонних компонент являются наборы элементов управления для программ в среде операционной системы Windows и интернет приложений, компоненты доступа к базам данных и пр. При этом компоненты могут быть написаны на различных языках программирования и даже расположены на различных машинах.

На сегодняшний день существует три основных компонентно-ориентированных платформы: COM, Java и .NET. COM (Component Object Model) – платформа, созданная Microsoft в 1993 году и используемая для меж-процессного взаимодействия и динамического создания объектов. COM дает разработчикам взможность создавать и использовать объекты написанные на различных языках программирования и даже расположенные на различных машинах. В последнее время данная технология считается устаревшей и ей на смену пришла технология COM+. Java – платформа разработанная фирмой Sun для программ написанных одноименном языке программирования. К преимуществам этой платформы можно отнести управляемое исполнение и независимость от низлежащей платформы. Основным недостатком является ориентированность только на один язык программирования, хотя в последнее время появились реализации таких яхыков как Ruby и Scala. Платформа .NET создана Microsoft в 2000 году как альтернатива платформе Java и является одной из передовых на данный момент. Именно эта платформа была выбрана автором в качестве основной в данной работе.

Стремление к возможности использования одной и той же компоненты в большом количестве различных приложений приводит к введению в компоненту различных архитектурных паттернов, абстракции алгоритмов и введению большого количества дополнительных параметров. Для конечного приложения это привносит следующие недостатки:

1. Конечное приложение, построенное из таких компонент, зачастую не использует большую часть их функциональности, и поэтому имеет больший размер чем необходимо. Здесь есть возможность убрать из конечного приложения неиспользуемую функциональность.

2. Такое приложение работает медленнее. В процессе работы компонентам приложения приходится проверять конкретные значения, выбирать конкретные имплементации абстрактных сущностей и алгоритмов. Перед поставкой приложения пользователю можно было бы убрать такие проверки на основе знаний о том, как компонента интегрируется в приложение.

С другой стороны, существует очень хорошо разработанная область смешанных вычислений ([7]), которая предлагает большое количество различных алгоритмов и техник для специализации программ на функциональных языках программирования. Несмотря на то, что эта область почти полностью исследована, на сегодняшний день практически ни одна из техник не применяется в настоящих больших приложениях (так называемых “real-world applications”). Самым большим препятствием стало то, что в то время как только такие языки как C/C++ считались промышленными, они были совершенно не приспособлены для смешанных вычислений из за сложной семантики указателей. С появлением таких платформ как Java и .NET смешанные вычисления стали возможны и для реальных приложений. Таким образом, возникает идея применить смешанные вычисления в больших реальных приложениях, но не в контексте одного метода как это делалось ранее а в глобальном контексте.

2 Постановка задачи

2.1 Постановка задачи

Целью данной работы является создание программного продукта, который автоматически оптимизирует конечное приложение, используя информацию о том как используются компоненты внутри приложения. При создании данного программного обеспечение особое внимание уделялось следующим аспектам:

1. *Максимальная гибкость*. Программа предоставляет максимальную гибкость при настройке параметров оптимизации, таких как, например, приоритетность методов, максимальный объем генерируемого кода, различные виды спецификации входных параметров и т.д.

2. *Расширяемость*. В программе предусмотрена возможность использования своей реализации алгоритмов, таким образом позволяя производить практически любые преобразования кода. Также есть возможность изменять существующие алгоритмы в некоторых пределах.

3. *Эффективность*. Система стремится получить максимальный прирост производительности в рамках заданных настроек. Для этого используется приоретизация методов и алгоритмы улучшения периода связывания (binding-time improvements)

4. *Безопасность*. Все преобразования производимые системой безопасны, т.е. они не меняют семантику оптимизируемой программы.

Оптимизатор использует глобальный анализ и техники смешанных вычислений. Интерфейсные методы и последовательность их оптимизации определяются в процессе предварительного анализа с помощью некоторой эвристики. В отличие от других работ, целью данной работы не было просто реализация смешанного вычислителя для языка CIL. Скорее, это попытка реализовать глобальный оптимизатор для .NET приложений, который использует техники смешанных вычислений для достижения своих целей. Глобальность оптимизации позволяет использовать некоторые преимущества:

1. Для работы оптимизатора нет необходимости явно указывать входные параметры для методов (хотя такая возможность тоже имеется), так как входные параметры определяются в процессе глобального анализа.

2. Глобальный анализ позволяет получить значения всех неизменяемых глобальных переменных и использовать их в процессе оптимизации.

Кроме того, автор применил несколько различных техник и ввел некоторые дополнительные алгоритмы, которые дают лучшую оптимизацию. Данная работа является экспериментальной, поэтому автор реализовал архитектуру, в которой можно использовать свои реализации алгоритмов или модифицировать реализации существующих. Таким образом, пользователи программы могут осуществлять практически любые преобразования CIL кода в контексте всего приложения, либо сравнить результаты работы различных алгоритмов.

В качестве компонентной платформы для создаваемой программы была выбрана платформа .NET. Данная платформа обладает несколькими преимуществами:

1. Это одна из наиболее перспективных и быстро развивающихся платформ на сегодняшний день

2. Основана на открытом стандарте

3. Предоставляет достаточно возможностей для анализа и модификации байткода

4. В данной платформе практически отсутствуют ограничения, которые сильно усложняли работу смешанных вычислителей в таких языках как C/C++. .NET не запрещает использование неуправляемых указателей, если это действительно необходимо при взаимодействии с другими платформами, но такие ситуации достаточно редки.

2. 2 Обзор существующих работ

За последние годы проведено большая работа в области смешанных вычислений для функциональных языков программирования и лишь небольшой объем работы был сделан в области объектно-ориентированных языков.

Первые попытки специализации объектно-ориентированных программ были предприняты в работе Partial Evaluation of an Object-Oriented Imperative Language ([9]). Авторами был разработан онлайн смешанный вычислитель для простого («игрушечного») объектно ориентированного языка.

Для языка Java было создано несколько смешанных вычислителей. В качестве примера может служить проект COMPOSE ([15]). В данном проекте внимание было уделено только неизменяемым (immutable) объектам. Другой проект, созданный Peter Bertelsen ([2]), работает на уровне байткода Java и ограничен только примитивным целочисленным типом и массивами. Объекты и интерфейсы не рассматриваются. Рассматриваются массивы целых, массивы массивов целых и т. д. Несмотря на такую ограниченность, уже в этой работе была продемонстрирована идея частично определенных объектов и массивов.

Самым известным из близких по тематике к данной работе является проект CILPE – смешанный вычислитель для .NET байткода ([8]). CILPE манипулирует неизменяемыми данными также, как смешанные вычислители для функциональных языков первого порядка и, кроме того, выделяет части изменяемых переменных (такие как поля объектов) как статические. В CILPE реализован offline смешанный вычислитель, основывающий свои действия на результатах предварительного поливариантного анализа периода связывания. В программе Spider, в отличие от CILPE, используется гибридная техника: сначала проводится анализ периода связывания, затем на его основе проводятся улучшения анализа периода связывания и затем производится онлайн специализация, которая может определить большее количество статических переменных. CILPE может работать со статическими объектами и простыми типами, с частично известными объектами, массивами с полностью или не полностью известными элементами и с динамическими объектами.

В программе Spider используется другой подход, нежели в CILPE. Специализация проводится не для каждого конкретного метода, а для всего приложения в целом. Этим достигаются следующие преимущества:

1. Отсутствует необходимость указывать входные данные

2. Глобальный анализ учитывает зависимости между методами и выбирает наиболее подходящие методы для оптимизации автоматически

3. Глобальный анализ определяет значения глобальных переменных и использует эти значения в процессе оптимизации

Кроме того, автор реализовал алгоритм улучшения периода связывания, отсутствующий в CILPE и использовал гибридный смешанный вычислитель, который является более эффективным. Spider также предоставляет большие возможности по конфигурированию, контролю за размером кода и расширению функциональности.

2.3 Обзор платформы .NET

.NET – это новая платформа, впервые объявленная Microsoft в июле 2000 года на выставке PDC, включающая в себя новый интерфейс к программному интерфейсу и сервисам семейства операционных систем Microsoft Windows, сервисы компонент COM+, платформу разработки веб приложений ASP и поддержку веб сервисов. Следует заметить что платформа Microsoft .NET это реализация открытого стандарта (ECMA-335 [3] и ISO/IEC 23271). Существуют также и другие реализации этого стандарта:

1. *Shared Source Common Language Infrastructure (SSCLI)* – реализация с открытым кодом
2. *.NET Compact Framework* – коммерческая реализация для портативных устройств
3. *Mono* – популярная реализация с открытым исходным кодом поддерживаемая компанией Novell
4. *Portable.NET* – еще одна реализация с открытым исходным кодом в рамках проекта dotGNU

Спецификацию ECMA-335 также принято называть Common Language Infrastructure (CLI). Спецификация CLI описывает среду, которая позволяет взаимодействовать программам и компонентам, написанным на разных высокоуровневых языках, быть использованным на различных платформах без переписывания. Кроме того, данная спецификация определяет следующие аспекты:

1. *Common Type System* – набор типов и операций над ними, общий для всех языков которые соответствуют спецификации. Описывает разделение типов на типы-значения и типы-ссылки. Определяет правила образования новых типов.
    a. Позволяет осуществлять межъязыковое взаимодействие и типовую безопасность
    b. Предоставляет объектно-ориентированный набор типов
    c. Определяет правила работы с типами
2. *Metadata* – языконезависимая информация о структуре программы и типах данных, позволяющая осуществлять межъязыковое взаимодействие.
3. *Common Language Specification* – общий набор правил, которым должны удовлетворять языки для того, чтобы взаимодействовать с другими языками. Содержит указания о том как нужно разрешать конфликты имен. Определяет какие символы могут быть использованы в именах а также пределы действия имен.
4. *Virtual Execution System* – загружает и исполняет CLI-совместимые программы. Комбинирует части написанные на различных языках с помощью метаданных.

В рамках данной работы наиболее интересны реализация VES в Microsoft .NET, так как средство работает с низкоуровневым байткодом, а также система организации компонент и их взаимодействия.

В качестве виртуальной машины служит Common Language Runtime ([1]), которая предоставляет среду выполнения для .NET приложений. Для того чтобы программа на высокоуровневом языке могла быть выполнена с помощью CLR, она компилируется в байткод на языке Common Intermediate Language. Во время исполнения байткод преобразуется в код низлежащей платформы с помощью JIT (Just-In-Time) компилятора и выполняется. Кроме того, виртуальная машины CLR предоставляет множество других сервисов:

- Управление памятью
- Управление потоками
- Обработка исключений
- Сборка мусора
- Обеспечение безопасности

Common Intermediate Language – это объектно-ориентированный ассемблерный язык, предполагающий выполнение на стековой машине ([6]). Таким образом, интерпретатор CIL кода можно представить в виде стековой машины, где каждая инструкция берет свои аргументы со стека и кладет результат своего выполнения в стек. В CIL встречается большое количество различных инструкций, но все их можно разделить на несколько типов: загрузка значений на стек, сохранение значений (например в локальные переменные или аргументы), вызов методов, арифметические, логические операции, операции сравнения, объектные операции (создание объектов, типовые операции), операции ветвления и пр.

Каждое .NET приложение состоит из одной или нескольких так называемых сборок (assemblies). Сборка – это скомпилированная в байткод библиотека, готовая для использования и установки. В реализации Microsoft .NET сборка – это PE (portable executable) файл ([5]), который может быть двух типов: исполняемый файл (exe) или библиотека (dll). Сборка состоит из одного или нескольких модулей (содержащих исполняемый код), ресурсных файлов и манифеста, который полностью описывает содержимое сборки.

Так как каждая сборка представляет собой отдельную компоненту и собирается тоже отдельно, компилятор не может учитывать зависимости между сборками и, соответственно, не может дать оптимальный код.

3 Программа Spider. Интерфейс пользователя.

Пользовательский интерфейс можно разделить на две части. Первая – проектная модель, способ указания приложения, которое подвергается оптимизации, задания настроек и входных параметров. Вторая часть – программный интерфейс, с помощью которого функциональность программы может быть расширена или изменена. Опишем здесь обе составляющие интерфейса.

3.1 Проектная модель

3.1.1 Общая структура

Единицей входных данных программы Spider является проект. Проект включает в себя спецификацию входящих в приложение компонент и сборок, явную спецификацию входных параметров и разнообразные настройки. Заметим, что на вход может подаваться спецификация нескольких проектов сразу. В этом случае оптимизация приложений будет производится одновременно в отдельных потоках. Набор проектов описывается при помощи XML файла, который и подается на вход программы во время выполнения. Опишем вкратце формат этого файла:

Каждый проект определяется элементом *<project></project>*. В аттрибутах проекта указываются:

- Имя проекта (*project_name*)
- Тип оптимизируемого приложения (*project_type*). На данный момент приложение Spider поддерживает 2 типа проектов:
    - Набор (*assembly_set*). Указывает, что приложение – это набор компонент и сборок (например библиотека), без явной точки входа (главного метода с которого начинается выполнение). В этом случае процесс оптимизации будет происходить начиная с наиболее приоритетных методов (алгоритм приоретизации методов будет описан отдельно).
    - Приложение (*application*). Указывает, что оптимизатор должен обращаться с набором сборок как с цельным приложением, у которого явно указанна точкой входа. В этом случае процесс оптимизации будет происходить только для входной точки.

Далее, с помощью элементов *<assembly></assembly>* пользователь должен указать сборки входящие в оптимизируемое приложение. Путь к сборке указывается в аттрибуте file. Кроме того, внутри этих элементов можно указать входные параметры для методов.

3.1.2 Спецификация входных параметров

Отдельной проблемой является предоставление возможности спецификации входных параметров. С одной стороны, объектная модель .NET позволяет создавать и использовать объекты любой сложности и оптимальной была бы возможность указывать любые такие объекты в качестве параметров. С другой стороны, необходимо иметь возможность указать значения параметров извне, уже после того как программа была скомпилирована. В связи с этим было принято решение воспользоваться возможностями сериализации которые предоставляет платформа .NET.

Сериализация – это процесс сохранения состояния объекта в некоторое хранилище. Во время этого процесса все видимые и скрытые поля объекта, так же как и полное имя

(включающее и имя сборки) класса объекта, преобразуются в набор байт который затем записывается в поток данных. Десереализация – это обратный процесс, при котором набор байт преобразуется в новый объект того же типа и с таким же состоянием. Бинарная сериализация не очень удобна, так как проектый файл имеет XML формат, поэтому было решено использовать XML сериализацию для указания значений параметров.

Пример сериализации объекта:

```
XmlSerializer serializer = new XmlSerializer(typeof(string));
serializer.Serialize(stream, "Test string");
```

Пример десериализации объекта:

```
XmlSerializer serializer = new XmlSerializer(typeof(string));
string str = (string)serializer.Deserialize(stream);
```

Вот так выглядит сериализованный в XML объект:

```
<?xml version="1.0" encoding="utf-16"?>
<TestClass xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <FirstProperty>10</FirstProperty>
      <SecondProperty>test string 1</SecondProperty>
      <ThirdProperty>
            <TestProperty1>100</TestProperty1>
            <TetsProperty2>test string 2</TetsProperty2>
      </ThirdProperty>
</TestClass>
```

К сожалению, XML-сериализация имеет свои ограничения, которые не позволяют указывать значения для любых объектов. На данный момент XML-сериализатор не работает для методов, индексаторов, скрытых полей и свойств которые можно только читать. Кроме того, для того чтобы сериализовать и десериализовать объект, тип объекта должен иметь конструктор без параметров.

Второй вариант позволяет указать не сериализованное представление объекта, а вызов конструктора объекта. Например, таким образом можно указать параметры конструктора двумерной матрицы заполненой нулями:

```
<constructor_params>
      <constructor_param index="0" type="System.Int32">
            <int>10</int>
      </constructor_param>
      <constructor_param index="1" type="System.Int32">
            <int>10</int>
      </constructor_param>
      <constructor_param index="2" type="System.Double">
            <double>0.0</double>
      </constructor_param>
</constructor_params>
```

Здесь каждый элемент *constructor_param* представляет параметр конструктора с индексом определяемым аттрибутом ‘index’ и типом определяемым аттрибутом *type*. Значение для параметров конструктора также может быть определено двумя способами – либо с помощью XML сериализованного представления, либо с помощью спецификации вызова конструктора.

Таким образом, внутри элемента *assembly* можно описать входные параметры для методов:

```
<assembly file="C:\WORK\Spider\SpiderTests\Numerics\MathNet.Numerics.dll">
```

```
      <type name="MathNet.Numerics.LinearAlgebra.CholeskyDecomposition">
            <method name="Solve">
                  <param index="0" type="constructor">
                        <constructor_params>
                              <constructor_param index="0"
                        type="System.Int32">
                                    <int>10</int>
                              </constructor_param>
                              <constructor_param index="1"
                        type="System.Int32">
                                    <int>10</int>
                              </constructor_param>
                              <constructor_param index="2"
                        type="System.Double">
                                    <double>0.0</double>
                              </constructor_param>
                        </constructor_params>
                  </param>
                  <param index="1">
                        <int>100</int>
                  </param>
            </method>
      </type>
</assembly>
```

3.1.3 Настройки проекта

Следующим по важности разделом является раздел ‘settings’, где определяются основные настройки оптимизации для данного проекта. Среди них стоит выделить:

- *loop_priority_multiplier*. Множитель для методов вызываемых в цикле. Используется при определении приоритета методов в очереди на оптимизацию. Обычно >> 1. По умолчанию его значение равно 100.

- *condition_priority_multiplier*. Множитель для методов вызываемых внутри ветки условного оператора. Также используется для определения приоритета метода. Обычно < 1 как и вероятность вызова такого метода. По умолчанию имеет значение 0.5.

- *size_limit*. Ограничение на размер кода. Некоторые оптимизации дублируют блоки кода и даже целые методы, раскручивают циклы и т.д. Поэтому нам необходимо установить ограничение на размер получающегося кода. Данный параметр может устанавливать размер как в абсолютных значениях (то есть в байтах), так и в процентах от начального размера кода. Значение по умолчанию предусматривает увеличение размера не более чем на 30%.

- *optimization_depth*. Глубина оптимизации. Если этот параметр задан, оптимизатор будет спускаться только на заданное количество уровней вглубь каждого оптимизируемого метода. Если же этот параметр не задан, оптимизация будет проходить на столько уровней вниз насколько это возможно.

- *split_blocks*. В зависимости от этого параметра оптимизатор будет решать, делать ли копии блока, если поток управления достигает его с различными значениями переменных (то есть блок имеет поливариантное разделение переменных). Если разделение блоков используется, то в конечном коде блоки будут лучше оптимизированы, но при этом возникнут дополнительные инструкции переходов. Таким образом, включение данной опции может как улучшить так и ухудшить производительность в

зависимости от приложения. Этот параметр оставляет нам возможности для экспериментирования. По умолчанию разделение блоков выключено.

- *cycle_unfold_limit*. Устанавливает ограничение на количество итераций цикла которые могут быть раскручены. Данный параметр может применяться для предотвращения зацикливания (несмотря на то что алгоримы раскрутки циклов в данной работе имеют встроенные средства для определения зацикливания) и для ограничения размера генерируемого кода. Если этот параметр не задан, оптимизатор раскручивает цикл до конца либо до момента определения зацикливания.

- *output_path*. Последний, но не менее важный параметр, который указывает путь к каталогу, где оптимизатор должен сохранить результат.

В следующей секции указываются идентификаторы алгоритмов которые будут использоваться оптимизатором для работы над проектом:

```
<strategies_configuration>
      <bta_strategy name="DEFAULT_BINDING_TIME_ANALYSIS_STRATEGY"/>
      <bti_strategy name="DEFAULT_BINDING_TIME_IMPROVEMENT_STRATEGY"/>
      <gva_strategy name="DEFAULT_GLOBAL_VARIABLE_MANAGER"/>
      <mc_strategy name="DEFAULT_METHOD_CACHE"/>
      <stack_machine name="DEFAULT_STACK_MACHINE"/>
      <main_strategy name="DEFAULT_SPECIALIZATION_STRATEGY"/>
</strategies_configuration>
```

Помимо проектов, внутри проектного файла указываются сборки в которых оптимизатор ищет реализации алгоритмов, которые могут быть использованы для работы над проектом:

```
<strategy_assemblies>
      <assembly
file="C:\WORK\Spider\Spider.Strategies\bin\Debug\Spider.Strategies.dll"/>
</strategy_assemblies>
```

3.2 Интерфейс расширения

В интерфейс расширения входят программные интерфейсы, с помощью которых можно реализовать нестандартные алгоритмы. Всего интерфейсов шесть и все они наследуются от базового интерфейса IStrategy.

Для того чтобы реализовать свою стратегию необходимо создать реализацию одного из интерфейсов и пометить ее аттрибутом Strategy, где указывается тип реализуемого интерфейса и строковый идентификатор, с помощью которого можно будет получить экземпляр этой реализации впоследствии.

Опишем вкратце назначение интерфейсов:

IBindingTimeAnalysisStrategy – стратегия для реализации алгоритма анализа периода связывания. Алгоритм помечает те инструкции которые могут быть выполнены статически, то есть оптимизированы.

IBindingTimeImprovementStrategy – стратегия улучшения периода связывания. Этот алгоритм в некоторых случаях может путем некоторых перестановок кода найти дополнительные инструкции которые могут быть выполнены статически.

IMethodCache – стратегия кэширования методов. Так как оптимизатор создает большое число дополнительных методов в процессе своей работы, данная стратегия позволяет повторно использовать тела оптимизированных методов.

IGlobalVariableManagementStrategy – стратегия работы с глобальными переменными. Отвечает за отслеживание значений таких переменных и определение переменных значение которых никогда не меняется.

IStackMachine – стратегия работы стековой машины. В ее обязанности входит абстрактное выполнение CIL кода и создание оптимизированного набора инструкций. Кроме того она используется в качестве вспомогательного инструмента для анализа живости определений, анализа зависимости инструкций и анализа возможности раскрутки циклов.

ISpecializationStrategy – стратегия оптимизации. Отвечает за управление предыдущими пятью алгоритмами с целью получения оптимизированного кода для всего приложения.

4 Обзор реализации программы Spider

В этом разделе рассматривается реализация программы, а точнее двух ее частей – общей управляющей части и реализации алгоритмов. Вначале рассмотрим синтаксис и семантику языка CIL (MSIL).

4.1 Обзор языка CIL

4.1.1 История

Первые версии IL ассемблера и дизассемблера (которые назывались Asm и Dasm) были разработаны в начале 1998 года Джонатаном Форбсом (Jonathan Forbes). Этот язык существенно отличался от того CIL который мы имеем на сегодняшний день. Единственное что объединяет их, это наличие точки перед управляющими ключевыми словами. Данная версия языка использовалась внутри команды разработчиков .NET для работы над Common Language Runtime. В 1999 году этот язык перешел в ведение Ларри Салливана (Larry Sullivan), главы группы CROEDT (Common Runtime Odds and Ends Development Team). После этого Сержу Лидину (Serge Lidin), Вэнсу Моррисону (Vance Morrison) и Джиму Миллеру (Jim Miller) было поручено провести полное переконструирование языка. К концу 1999 года был достигнут определенный прогресс: ILAsm и ILDAsm начали поддерживать ограниченный round-tripping, то есть стало возможно взять любой IL модуль, дизассемблировать его, добавить свой код и собрать обратно в модуль. После первой бета-версии в 2000м году IL Assembler и ILDASM стали полнофункциональными средствами разработки для .NET

4.1.2 Типы данных

В то время как .NET представляет достаточно богатую систему типов, а Common Language Specification определят подмножество типов которое можно использовать для межъязыкового взаимодействия, сама среда выполнения оперирует гораздо более простым набором типов. В этот набо входят:

- Подмножество числовых типов
- Ссылки на объекты
- Указатели

К числовым типам относятся целочисленные (размером от 1го байта до 8ми байт) и числа с плавающей точкой (размером 4, 8 байт а также специальный тип RPrecise). RPrecise имеет фиксированный размер для любой архитектуры, но не менее 64х бит и представляет числа с максимальной точностью которая может быть эффективно поддержана данной архитектурой. Следует заметить что стек виртальной машины может содержать только 4х или 8ми байтовые целые, поэтому для того чтобы поместить 1-2 байтовое число на стек оно дополняется нулем либо знаком (в зависимости от используемой инструкции), а при созранении значения в 1-2 байтовое хранилище может произойти переполнение.

Ссылки на объекты достаточно просты и прозрачны. Над ними запрещены любые арифметические операции, а из операций сравнения для них определены только равенство (и неравенство). Также для них не определены операции преобразования. Объектные ссылки можно создать с помощью CIL инструкций newobj (создает новый объект) и newarr (создает новый массив). Ссылки можно передавать в качестве аргументов при вызове методов, сохранять в локальные переменные, использовать как воздвращаемое значение, хранить в массивах и полях объектов.

Наиболее интересную и нетривиальную часть системы типов представляют указатели. Указатели бывают трех типов: неуправляемые, управляемые и переходные. Для указателей в рамках одного и того же объекта или массива определены следующие операции:

- Добавление целого числа к указателю.
- Вычитание целого числа из указателя
- Вычитание двух указателей одного типа

Ни одна из этих операций не допустима в верифрцируемом коде.

Неуправляемые указатели

Обычные указатели по семантике похожи на те, что используются в языках C/C++. Нет никаких ограничений на их использование, но они могут привести к тому что код станет не верифицируемым.

Управляемые указатели

Такие указатели могут указывать на поле объекта или типа-значения, на элемент массива или на адрес сразу за последним элементом массива. Они не могут быть нулевыми и всегда должны быть предоставлены сборщику мусора, даже если они не указывают на управляемую память (сборщик мусора не будет собирать указатели на память, которая не находится под его контролем).

Промежуточные указатели

Промежуточные указатели являются переходными между управляемыми и неуправляемыми указателями. Они используются внутри CLR и могут быть созданы с помощью некоторых CIL инструкций, но они не могут быть сохранены.

4.1.3 Виртуальная машина

Несмотря на то, что IL код, перед тем как быть исполненным, компилируется в код целевой архитектуры с помощью JIT компилятора, выполнение IL кода можно описать в терминах абстрактной виртуальной машины. Единицей исполнения в данном случае является метод. Во время исполнения метода IL коду доступны 3 категории локальной памяти и одна категория внешней памяти, которые представлены типизированными слотами (в отличие от указателей в неуправляемых архитектурах). Внешними, по отношению к методу, являются поля, внутренними – аргументы, локальные переменные и стек. Все инструкции либо записывают значение в стек, либо берут значение из стека, либо и то и другое одновременно (исключением из этого правила являются специальные префиксные интсрукции). Количество аргументов метода определяется в момент вызова метода (а не в момент его определения), так как в .NET количество аргументов может варьироваться с помощью директивы ‘vararg’. В языке более высокого уровня (C#) такой метод описывается помощью ключевого слова ‘params’, например так: void TestMethod(params p);

Каждая IL инструкция состоит из операционного кода (opcode) и нуля или одного параметра. Каждый операционный код имеет длину 1 или 2 байта. Существует также разделение на короткие и длинные инструкции. Те инструкции, параметр которых находится в пределах от -128 до 127 (или от 0 до 255 для беззнаковых параметров) называются короткими и имеют префикс s. Длинные инструкции подходят для любых

параметров, но использование их в случае когда можно обойтись короткой формой приводит к неоправданному увеличению IL кода. Рассмотрим вкратце типы инструкций.

Инструкции перехода

Включают в себя инструкции ветвления (условные и безусловные), инструкцию switch (которая выбирает из нескольких следующих инструкций), специальные инструкции выхода из SEH (Structural Exception Handling) секций и инструкции возврата

Арифметические инструкции

К ним относятся: инструкции манипуляции со стеком (дубликация значения на стеке и снятие значения со стека), инструкции загрузки констант, инструкции косвенной загрузки и хранения (которые работают с управляемыми или неуправляемыми указателями), арифметические операции (включая операции с проверкой переполнения), побитовые логические операции, операции сдвига, операции преобразования (включая операции с проверкой переполнения) и операции сравнения.

Операции с хранилищем

К ним относятся операции загрузки (на стек) и сохранения (из стека) работающие с локальными переменными, аргументами и полями

Вызовы методов

Включают вызовы виртуальных и невиртуальных методов, переход на код метода (jmp) и непрямые вызовы методов.

Работа с объектами

Инструкции по созданию, загрузке объектов, а также по приведению типов и проверки принадлежности к определенному типу.

Работа с массивами

Инструкции по созданию массивов, работе с элементами и длиной массива

4.2 Организация программы Spider

Всю программу можно условно разделить на несколько уровней, каждый из которых получает на вход результаты работы предыдущего уровня и передает результаты следующему уровню. Всего можно выделить четыре основных уровня:

1. Проектная модель и менеджер алгоритмов
2. Глобальный анализ
3. Оптимизатор (то есть реализация абстрактых алгоритмов)
4. Менеджер хранения

4.2.1 Проектная модель

Задачей проектной модели является перевод XML спецификации в объектное представление. Кроме того, она управляет процессами работы над проектами и следит за их завершением. Ключевыми в данном уровне являются элементы IProject - объектное представление проекта и его настроек, IProjectFactory – создает набор элементов IProject

по XML спецификации и IProjectManager – хранит набор текущих проектов и предоставляет методы для начала и завершения работы над ними.

Стоит заметить, что для полноты работы оптимизатора он должен знать достаточно информации о типах, определенных в приложении, над которым он работает и, более того, он должен уметь создавать экземпляры этих объектов. Для этого было решено воспользоваться механизмом .NET называемым рефлексия. Рефлексия – это механизм позволяющей работать с типовой информацией во время исполнения и динамически создавать объекты нужных типов. Для того чтобы механизм рефлексии работал, необходимо чтобы сборки, содержащие нужные объекты, были загружены в Application Domain (.NET абстракция которая представляет изолированную среду в которой выполняется приложение). Здесь возникает другая проблема: в домен приложения оптимизатора загружать сборки нельзя, так как CLR сразу заблокирует файлы сборок и оптимизатор не сможет записать результаты своей работы. Поэтому принято решение использовать метод теневого копирования. Каждый проект оптимизируется в отдельном домене приложения и каждая сборка перед тем как быть загруженной, копируется в кэш сборок и загружается оттуда.

Одной из приоритетных задач проектной модели является чтение кода приложения во внутреннее объектное представление, которым удобно манипулировать и которое можно обратно сохранить в скомпилированное приложение. На сегодняшний день существует несколько библиотек предоставляющих подобную функциональность, например, AbstractIL и Mono Cecil. AbstractIL – библиотека реализованная Microsoft Research, предоставляет богатые возможности по манипулированию CIL кодом. Одним из недостатков данной библиотеки является то, что она реализована на F# - функциональном языке программирования. Использование такой библиотеки из императивных языков программирования значительно затруднено и ее целесообразно использовать также из функциональных языков. Mono Cecil – библиотека с открытым исходным кодом реализованная в рамках проекта Mono. Несмотря на то что последняя доступная версия только 0.5, то есть библиотека все еще находится в разработке, ее уже можно использовать. В отличие от Abstract IL, она обладает простотой и удобством, а открытость исходного кода позволяет модифицировать ее для собственных нужд. Ее было решено использовать для чтения и модификации CIL кода.

4.2.2 Менеджер стратегий

Менеджер стратегий отвечает за извлечение, создание и предоставление стратегий. На запуске он сканирует все загруженные сборки и, используя механизм рефлексии, находит те типы, которые имеют аттрибут Strategy и реализуют интерфейс IStrategy. Затем менеджер создает экземпляры каждой стратегии и кэширует их. В момент когда менеджер проектов начинает работу над конкретным проектом, объекты необходимых стратегий извлекаются из кэша и им передается управление.

4.2.3 Глобальный анализ

Глобальный анализ работает в рамках одного проекта/приложения и собирает важную информацию, которая используется затем при оптимизации конечных методов. Рассмотрим задачи, выполняемые глобальным анализом и структуры данных которые он создает.

Разделение методов на блоки и построение графа потока управления

IL код метода представляется в виде линейного набора инструкций с аргументами, что само по себе не дает никакой информации о потоке управления метода и крайне неудобно для использования. В задачи глобального анализа входит построение блочной

структуры для каждого метода и графа потока управления (control-flow graph). Блок – это последовательный набор инструкций, который начинается либо с первой инструкции метода, либо с инструкции на которую переходит одна из инструкций перехода. Оканчивается блок либо инструкцией перехода, либо инструкцией возврата, либо инструкцией непосредственно предшествующей той, на которую переходит одна из инструкций перехода. Таким образом получается граф потока управления для метода. Каждый блок ссылается на один или 2 других блока (за исключением последнего блока который заканчивается возвратом) – тот блок на который перейдет управление, если инструкция перехода выполнится и тот блок который выполняется непосредственно после данного (если инструкция перехода не выполняется). На этом же этапе определяется набор циклов в методе и строится иерархия вложенных циклов. Цикл определяется по операции обратного перехода на одну из предыдущих инструкций.

Построение графа вызовов

Граф вызовов соединяет все методы приложения в один граф (в котором достаточно много компонент связности). Граф вызовов помогает определить зависимости между методами и определить приоритет каждого метода. Он ассоциирует множество вызовов с каждым методом:

$CG = \{(m, MC(m)) \mid m \in M\}$, где

$MC(m) = \{(i, m_i, pm) \mid i \in I(m), m_i \in M, pm \in \mathrm{N}\}$, где I(m) – множество инструкций, pm – приоритет вызова, о котором будет рассказано ниже.

Анализ приоритета методов

Так как у оптимизатора есть достаточно жесткое ограничение по объему генерируемого кода, в случае больших приложений это накладывает соответствующее ограничение на количество методов которые могут быть оптимизированы. Наибольшую выгоду, очевидно, может принести оптимизация тех методов, у которых показатель «используемости» (некоторый эвристический показатель, который приблизительно характеризует количество вызовов данного метода в процессе выполнения программы) наибольший. Следует заметить, что «используемость» показывает количество вызовов внутри программы и не учитывает внешнее использование.

Таким образом, после построения графа вызовов, глобальный анализ вычисляет «используемость» каждого метода и сортирует методы по этому критерию. Алгоритм вычисления «используемости» достаточно прост: она вычисляется как сумма приоритетов всех вызовов этого метода во всех стеках вызовов публичных методов. Изначально приоритет каждого вызова равен единице. Для вычисления приоритетности вызовов глобальный анализ проходит все публичные методы и для каждого метода проходит по стеку вызовов вглубь вычисляя приоритеты. Формула приоритета для метода x выглядит следующим образом:

$$p = \sum_{m \in M} \Big(\sum_{m_1 \in MC(m)} p(m_1) \cdot \Big(\sum_{m_2 \in MC(m_1)} p(m_2) \cdot \ldots \cdot \Big(\sum_{x \in MC(m_n)} p(x) \Big) \ldots \Big) \Big)$$

, где $MC(m_i)$ – множество вызовов в методе m_i, а

$$p(m_i) = \prod_{b \in B} k \cdot pm$$

, где k обозначает вложенность цикла (или условного предложения), а pm – это значение *loop_priority_multiplier*, если блок находится в цикле, либо *condition_priority_multiplier*, если вызов находится в ветке условного оператора, либо единица.

В итоге, оптимизатор начинает работу с наиболее «используемых» (в нашей эвристике) методов и последовательно переходит к менее «используемым». Но есть несколько исключений из этого правила: если оптимизируемый проект – приложение, работа ведется только над главным входным методом. Если в коде программы или с помощью проектного файла явно указаны значения для входных переменных некоторых методов, то эти методы помещаются в самое начало очереди и считаются наиболее приоритетными.

Анализ статических и глобальных переменных

Статические переменные – это те переменные, чьи значения не зависят от экземпляра класса. Такие переменные обычно инициализируются в статическом конструкторе, который вызывается при первом обращении к классу. Статические переменные в основном используются как хранилище для различных констант используемых в различных частях программы. Простой оптимизатор, работающий в контексте одного метода, не может ни получить текущего значения таких переменных, ни отследить изменения этого значения в других частях программы. Таким образом, обычно принято принимать статические и глобальные переменные за неизвестные. В данной работе автор предпринял попытку преодолеть это ограничение. Задачей анализа статических переменных является сбор необходимой информации о таких переменных и передача этой информации оптимизаторам работающим на уровне методов. Задачу можно разбить на две подзадачи:

1. Определение начального значения переменной. Начальное значение переменной задается внутри статического конструктора. Оптимизатор использует средства абстрактной интерпретации для выполнения конструктора и получает в этом процессе значения глобальных переменных.

2. Также необходимо убедиться что значение этой переменной не меняется в других частях программы. Если переменная имеет аттрибут ‘initonly’, она не может быть изменена и ее можно считать постоянной. Для остальных же переменных анализатор делает обход всех методов (учитывая область видимости переменных) и в каждом методе ищет операции хранения принимающие ссылку на одну из этих переменных. Если такие инструкции находятся, тогда переменная помечается как изменяющаяся и относительно нее оптимизации уже не проводятся. Заметим, что переменные могут быть не только простого (численные типы и строки), но и сложного типа. В случае со сложным типом анализатор также следит за компонентами (полями или элементами в случае массива) и помечает некоторые из них как изменяющиеся (о структуре сложных значений и ее поддержке оптимизатором будет рассказано позднее). Стоит заметить что константы тоже считаются постоянными но выгоды от этого немного, так как компилятор в большинстве случаев заменяет ссылку на константу на ее значение.

5 Процесс оптимизации метода

Рассмотрим теперь как происходит процесс оптимизации на уровне метода. Последовательность действий в данном процессе определяется используемой реализацией интерфейса *ISpecializationStrategy:*

```
public interface ISpecializationStrategy : IStrategy
{
      void Specialize(IDivision division, IMethod method);
}
```

Здесь уместно рассмотреть объектную систему абстракции значений, используемую оптимизатором.

5.1 Объектная модель

В процессе выполнения (а также оптимизации) методу доступны несколько типов памяти: локальные переменные, аргументы, поля объекта и глобальные переменные. Разделением (Division) называется разделение всех переменных на статические (значение таких переменных известно в момент оптимизации) и динамические (значение этих переменных неизвестно в момент оптимизации). Таким образом разделение содержит информацию о каждой переменной доступной методу (кроме глобальных переменных) и ее возможном значении. Управление глобальными переменными осуществляется отдельно с помощью реализации интерфейса IGlobalVariableManagementStrategy. Разделение в объектной модели представлено экземпляром интерфейса IDivision и состоит из отдельных компонент IDivisionComponent

```
public interface IDivisionComponent  :  ICloneable
{
      string ComponentName { get; }
      DivisionType DivisionType { get; set; }
      ComponentType ComponentType { get; }
      IValue Value { get; set; }
}
```

Каждая компонента уникально идентифицируется в контексте метода по имени. Имя компоненты составляется из полного имени держателя компоненты (класса либо метода), типа компоненты и ее индекса или имени. Например:

TestNamespace.TestClass@field:field1 – имя компоненты для поля объекта

TestRealMethod.TestClass::Power@param:1 – имя компоненты для аргумента

TestRealMethod.TestClass::Power@local:1 – имя компоненты для локальной переменной

Таким образом каждая компонента есть элемент отношения

$$R = N \times V \times \{S, D\}$$

, где N – множество имен компонент, V – множество значений, а {S, D} – константы соответствующие статическим и динамическим компонентам. Подмножество этого отношения и является разделением:

$Div \subset R$, такое что $\forall n \in N \; \exists! v \in V \wedge \exists! d \in \{S, D\} \; (n, v, d) \in Div$

Рассмотрим как оптимизатор абстрагирует реальные значения в процессе работы.

Базовым интерфейсом для всех значений является IValue:

```
public interface IValue : ICloneable
{
      Type ValueType { get; set; }
}
```

Единственное, что он определяет - это тип абстрагируемого значения. Так как все переменные в языке CIL имеют определенный тип, оптимизатор всегда может получить тип значения по типу переменной которая его содержит. Во время оптимизации, если встречается преобразование типа, соответствующим образом меняется и тип значения. Непосредственными наследниками IValue являются:

ISimpleValue – простые численные значения.

IDynamicValue – неизвестное (динамическое) значение. Только динамические компоненты могут иметь это значение

IArrayValue – массивы и вектора. Предоставляет доступ к отдельным элементам массива и его длине

```
public interface IArrayValue : IValue
{
      Type ElementType { get; }
      int Length { get; }
      IValue this[int index] { get; set; }
      bool HasElementAt(int index);
}
```

IComplexValue – значения, состоящие из нескольких подзначений. Например объект, у которого некоторые поля известны, а некоторые – нет. Доступ к подзначениям осуществляется по именам.

```
public interface IComplexValue : IValue
{
      IDictionary<string, IDivisionComponent> Fields { get; }
}
```

IObjectValue – представляет собой цельный объект.

IBoxedValue – представляет собой boxed значение. В .NET все типы делятся на типы-значения и типы-ссылки, как уже было рассказано ранее. Иногда возникает необходимость обращаться с типом-значением как с типом-ссылкой, например, для того, чтобы передать такое значение в метод ожидающий значение-ссылку. Преобразование типа-значения в тип-ссылку называется *упаковка* (boxing), а обратная операция называется *распаковка* (unboxing). Для поддержки таких значений и предназначен данный интерфейс.

Оказывается, что одного разделения недостаточно, так как в процессе выполнения метода поток управления может приходить в каждый блок кода с различными значениями переменных. Для решения этой проблемы существуют многовариантные разделения (polyvariant divisions, [11]), которые ассоциируют с каждым блоком набор разделений. В оптимизаторе используется интерфейс IPolyvariantDivision и объект реализующий данный интерфейс ассоциируется с каждым базовым блоком. Поливариатное разделение переменных ассоциирует с каждым блоком несколько разделений одновременно:

$$PDiv = \cup R_i$$

, где $R_i \subset N \times V \times \{S, D\}$ такое что $\forall n \in N \; \exists! v \in V \wedge \exists! d \in \{S, D\} \; (n, v, d) \in R_i$

5.2 Анализ периода связывания

Все смешанные вычислители делят на 3 типа: *неоперативные* (offline), *оперативные* (online) и *гибридные*, в зависимости от того, в какой момент вычислитель решает, какие действия предпринимать к конструкциям в коде. *Неоперативные* смешанные вычислители опираются на предварительный анализ (анализ периода связывания и, возможно, другие анализы) для того чтобы выбрать действия независимо от конкретных значений статических переменных. Результатом работы предварительного анализа является набор «помеченных» конструкций, для каждой из которых указано, какие действия следует предпринять. *Оперативные* вычислители, наоборот, определяют необходимые действия уже в процессе специализации относительно текущих значений переменных. *Гибридные* вычислители используют обе техники, то есть сначала производится предварительный анализ, а в процессе специализации результаты предварительного анализа уточняются. В данной работе реализован именно гибридный смешанный вычислитель.

В задачи анализа периода связывания входит определение статических переменных (с учетом того что разделение для аргументов метода уже известно) и разметка IL инструкций. На вход анализа периода связывания подается набор базовых блоков B и начальное разделение переменных Div. На выходе анализ дает набор помеченных блоков:

$$MB =< MI, PDiv >$$

, где MI – множество помеченных инструкций, PDiv – поливариантное разделение переменных ассоциированное с этим блоком. Таким образом, анализ периода связывания – это функция:

$$< B, Div > \rightarrow \{< MI, PDiv >\}$$

Она не вычисляет значения внутри PDiv, а только лишь создает разделение. Сами значения будут вычислены в процессе абстрактной интерпретации метода.

Реализация по умолчанию достаточно проста: это стековая машины в которой на стеке могут лежать лишь типы значений (статический или динамический). Операция с хотя бы одним динамическим аргументом считается динамической. Результаты работы динамической операции тоже динамические. Особое внимание уделяется операциям сохранения, и некоторые из переменных помечаются как статические.

5.3 Улучшения периода связывания

Даже полностью эквивалентные по семантике и работе программы могут быть специализированы по разному, с разной эффективностью, размером и использованием памяти. Поэтому, для получения наибольшего эффекта от специализации, программы должны быть написаны особым образом. Основное правило состоит в том, что статические и динамические переменные и операции нужно использовать, по возможности, отдельно. Простой пример:

Возьмем два выражения: $(x+1)+y$ и $(x+y)+1$. Если x – статическая, а y – динамическая, то смешанный вычислитель сможет вычислить x+1 в первом выражении, но не сможет ничего сделать со вторым выражением.

Улучшением периода связывания (binding-time improvement, [10]) называется такая трансформация исходной программы, которая не меняет семантику, но приводит к лучшим результатам работы смешанного вычислителя.

Существует несколько стратегий улучшения периода связывания. Например можно использовать continuation passing style (CPS) преобразование ([13]). Оно линеаризует исполнение программы и может совместить те части программы, которые в изначальном варианте были разделены. Недостатком такого подхода является то, что CPS делает программы более высокоуровневыми, что в свою очередь может мешать работе смешанного вычислителя. В случае оптимизатора рассматриваемого в данной работе CPS - не подходящий способ, так как единицей оптимизации является метод и введение дополнительных методов не только не улучшит оптимизируемость программы, но и добавит работы вычислителю.

Улучшения периода связывания представлены как реализации интерфейса *IBindingTimeImprovementStrategy:*

```
public interface IBindingTimeImprovementStrategy : IStrategy
{
      void Improve(MarkedMethod method, IDivision division);
}
```

В данной работе автор реализовал свой алгоритм улучшения периода связывания. Этот алгоритм не использует никаких дополнительных конструкций, а производит преобразования арифметических выражений, по возможности группируя статические инструкции вместе. Алгоритм работает следующим образом: для каждого блока строится набор деревьев, по дереву на каждое арифметическое выражение. Далее для каждого дерева алгоритм пытается передвинуть статические подвыражения так что они группируются в одной части дерева. Отметим, что подвыражения передвигаются только в том случае, если они находятся под аддитивной инструкцией (то есть сложением или вычитанием), либо если выражение можно выделить как общий множитель. В случае вычитания знаки меняются соответствующим образом.

5.4 Оптимизация метода

Процесс оптимизации метода происходит в несколько этапов:

1. Абстрактная интерпретация тела метода с использованием виртуальнй стековой машины
    a. Выполнение кода внутри блоков и получение списка остаточных инструкций
    b. Обработка циклов. Определение раскручиваемости и их раскручивание.
    c. Определение выполнимости конструкторов и других методов
    d. Постановка неустранимых вызовов методов в очередь на специализацию
2. Анализ достигающих определений (reaching definitions) и удаление остаточных присваиваний.
3. Обработка остаточных вызовов. Поиск методов в кэше
4. Сжатие переходов
5. Компиляция

5.4.1 Абстрактная интерпретация

Абстрактный интерпретатор выполняет тело метода, вычисляя разделение переменных PDiv для каждого блока и интерпретируя блок. Выражение для разделения переменных каждого блока можно записать так:

$$PDiv_b = \bigcap_{bb \in MB, Jump(bb)=b \vee Cont(bb)=b} PDiv_{bb}$$

, где Jump(bb) – блок, на который передается управление при выполнении инструкции перехода в bb, Cont(bb) – блок, на который передается управление при невыполнении или отсутствии инструкции перехода.

Для интерпретации IL кода используется заданная реализация стековой машины, а точнее интерфейса IStackMachine. Стековая машина реализованная автором поддерживает практически весь набор IL инструкций за исключением некоторых очень редко используемых. Стек машины содержит значения IValue и, кроме того, поддерживает граф зависимости инструкций, о котором будет рассказано позднее. При выполнении каждой инструкции машина берет аргументы со стека (если они нужны) и кладет результат выполнения инструкции на стек. Если хотя бы один из аргументов инструкции динамический (то есть IDynamicValue), машина вычисляет тип результата инструкции и кладет IDynamicValue соответствующего типа на стек (то есть производит динамическое выполнение).

Особое внимание стоит уделить вопросу того, как стековая машина обрабатывает конструкторы и методы.

5.4.2 Выполнение конструкторов

Операция создания объекта имеет следующий вид: *newobj <ссылка на конструктор>*, где ссылка на конструктор – это MethodReference (в рамках Mono Cecil). В первую очередь, необходимо убедиться, что конструктор может быть выполнен во время оптимизации, то есть конструктор должен быть полностью статически выполним (вместе с внутренними вызовами методов) и он не должен иметь побочных эффектов. Здесь под побочными эффектами понимается изменение любых глобальных значений (то есть не относящихся к полям данного класса, агрументам и локальным переменным конструктора). Для того чтобы убедиться, что конструктор действительно выполним, создается разделение для конструктора (в которое входят аргументы, локальные переменные и поля создаваемого объекта), а затем происходит абстрактная нитерпретация тела конструктора с отслеживанием глобальных переменных или динамических операций. Как только встречается динамическая операция, либо модификация глобальной переменной, конструктор считается невыполнимым. Если же конструктор выполним, то с помощью механизма рефлексии создается объект, оборачивается в IObjectValue и кладется на стек.

5.4.3 Выполнение методов

Выполнение методов во многом схоже с выполнением конструктора. Отличия возникают, когда вызывается метод объекта (а не статический метод). В этом случае по спецификации виртуальной машины кроме аргументов на стеке должен также лежать объект, метод которого вызывается. Таким образом для выполнимости метода необходимо дополнительное уловие: на стеке должен лежать объект IObjectValue. Все остальное происходит аналогично выполнению конструктора.

Если метод оказывается невыполним, то возможны несколько вариантов действий. Если данный метод с данным набором агрументов уже оптимизировался, можно его

заменить на вызов кэшированного метода. Иначе, оптимизатор просто добавляет метод в очередь на оптимизацию.

5.4.4 Удаление остаточных инструкций

В процессе абстрактной интерпретации статические инструкции, которые возвращают простые значения, заменяются стековой машиной на инструкции загрузки констант (ldc.i4, ldc.r4, ldc.r8 и т.д.)

Этого недостаточно, так как необходимо также убрать все дополнительные инструкции, которые привели к формированию этого значения на стеке. В этих целях стековая машина поддерживает граф зависимости инструкций. Граф содержит зависимости инструкция-значение-инструкция, которые формируются следующим образом: если инструкция использует значения в качестве аргументов, то она зависит от этих значений, если инструкция кладет значение на стек, то данное значение зависит от инструкции:

$DG =< ID, VD >$, где

$ID = \{(i, v) \mid i \in MI,\ v \in V,\ \textit{инструкция } i \textit{ использует значение } v\}$

$VD = \{(v, i) \mid v \in V,\ i \in MI,\ \textit{значение } v - \textit{результат выполнения } i\}$

Таким образом, оптимизатор определяет набор зависимых инструкций путем обхода цепочки зависимостей. Все такие инструкции также можно удалить за исключением тех, что имеют побочный эффект. Такими, например, являются инструкции хранения, вызовы методов, которые меняют не только свои локальные переменные и локальные переменные.

5.4.5 Обработка циклов

Обработка циклов происходит особым образом. Следует заметить, что оптимизатор, реализованный автором, поддерживает работу только с цельными циклами (то есть с теми циклами, у которых есть единая точка выхода – после инструкции обратного перехода и нет переходов из тела цикла в другие части программы). Все современные компиляторы для платформы .NET производят цельные циклы, поэтому ситуации когда данный недостаток может играть роль очень редки.

Первое, что определяет оптимизатор – это возможность раскручивания цикла. Если управляющий переход цикла может быть выполнен статически после абстрактного выполнения тела цикла, то цикл можно раскрутить. Более того, цикл можно удалить если управляющий переход изначально не выполним. Стоит заметить, что если цикл содержит вложенные циклы, которые не могут быть раскручены, переменные модифицируемые во внутренних циклах становятся динамическими, что может повлиять на раскручиваемость внешнего цикла

Оптимизатор выполняет тело цикла, вставляет остаточные инструкции в некоторый контейнер, копирует тело цикла, обновляет инструкции переходов и повторяет эти действия пока выполним управляющий переход. После этого проверяется размер контейнера и, если он не превышает допустимых размеров, цикл в методе заменяется на содержимое контейнера и поправляются соответствующие инструкции перехода. Используется два метода, с помощью которых оптимизатор определяет зацикливание: во первых в настройках проекта можно указать максимальное количество раскручиваемых итераций, во вторых оптимизатор умеет отслеживать значения переменных, от которых зависит условие цикла. С помощью графа зависимости инструкций, оптимизатор находит

переменные управляющие циклом. Если эти переменные перестают меняться в процессе раскрутки, оптимизатор прекращает процесс и сообщает пользователю о том, что в данном месте возможно зацикливание.

5.4.6 Удаление неиспользуемых присваиваний

На данной стадии производится анализ достигающих определений (reaching definitions), в котором определяются присваивания, которые не используются в процессе выполнения остаточного метода. Такие присваивания удаляются вместе с зависимыми инструкциями.

5.4.7 Сжатие переходов

После оптимизации в коде метода остается большое количество излишних инструкций перехода. На данной стадии излишние переходы удаляются путем копирования блоков и код программы становится более линейным.

5.4.8 Компиляция

На выходе оптимизатора получается набор абстрактных блоков, а не цельное тело метода. В задачи компилятора входит получение полного набора инструкций по набору блоков и встраивание этого набора в тело исходного метода.

5.4.9 Завершаемость

Большое внимание в данной работе автор уделил завершаемости оптимизации. Первым барьером на пути зацикливания является ограничение на размер генерируемого кода. Если процесс оптимизации метода войдет в бесконечный цикл, в большинстве случаев границы размеров будут достигнуты очень быстро. Данный подход обладает большим недостатком: определить место, в котором произошло зацикливание, невозможно. На самом деле, зацикливание может произойти в трех местах: при раскрутке циклов, при оптимизации рекурсивных методов и при сжатии переходов. Во всех трех случаях автором предприняты меры для определения зацикливаемости.

При раскрутке циклов, оптимизатор отслеживает значения переменных влияющих на условие цикла и, если ни одна из переменных не изменилась за одну итерацию, сообщает о возможном зацикливании. В данном случае также помогает ограничение на количество раскручиваемых итераций.

При оптимизации рекурсивных вызовов помогает кэш методов. Если параметры рекурсивного вызова не меняются, оптимизированный метод уже есть в кэше и необходимо лишь заменить инструкцию вызова. В данном случае также помогает ограничение на глубину оптимизации.

При сжатии переходов, оптимизатор отслеживает все копируемые блоки и следит за тем, чтобы сжатие переходов не производилось для одного и того же блока несколько раз.

6 Пример

В качестве примера можно рассмотреть программу вычисляющую степень заданного числа. В этом разделе приводятся различные варианты реализации Power и результаты работы специализатора. Самый простой вариант вычисления степени такой (на языке C#):

```
public int Power(int n, int x)
{
       int y = 1;
       for (int i = 0; i < n; i++)
            y *= y;
      return y;
 }
```

В языке CIL данная программа выглядит следующим образом:

```
.method public hidebysig instance int32 Power(int32 n, int32 x) cil managed
{
  .maxstack 3
  .locals init (
    [0] int32 num,
    [1] int32 num2,
    [2] int32 num3,
    [3] bool flag)
  L_0000: nop
  L_0001: ldc.i4.1
  L_0002: stloc.0
  L_0003: ldc.i4.0
  L_0004: stloc.1
  L_0005: br.s L_000f
  L_0007: ldloc.0
  L_0008: ldloc.0
  L_0009: mul
  L_000a: stloc.0
  L_000b: ldloc.1
  L_000c: ldc.i4.1
  L_000d: add
  L_000e: stloc.1
  L_000f: ldloc.1
  L_0010: ldarg.1
  L_0011: clt
  L_0013: stloc.3
  L_0014: ldloc.3
  L_0015: brtrue.s L_0007
  L_0017: ldloc.0
  L_0018: stloc.2
  L_0019: br.s L_001b
  L_001b: ldloc.2
  L_001c: ret
}
```

Программу вычисления степени можно усложнить обернув параметры в объект-контейнер:

```
public long Power(PowerParamsContainer container)
{
      long result = 1;
      for (int i = 0; i < container.N; i++)
            result *= container.X;
      return result;
}
```

и использовать глобальную переменную при вызове этого метода:

```
Math math = new Math();
PowerParamsContainer powerParams = new PowerParamsContainer();
powerParams.X = x;
powerParams.N = GlobalSettings.N;
long result = math.Power2(powerParams);
```

В обоих случаях оптимизированные программы получаются практически одинаковыми, отличаясь только в способе получения значения x (в первом случае значение берется как значение аргумента, а во втором как значение поля аргумента). Вот результат работы оптимизатора для программы возводящей x в степень 3:

```
.method public hidebysig instance int64
Power2_9ad883fa5a9540b5a5b52f049d2ae9d5(class TestPower.PowerParamsContainer
container) cil managed
{
  .maxstack 3
  .locals init (
    [0] int64 num,
    [1] int32 num2,
    [2] int64 num3,
    [3] bool flag)
  L_0000: nop
  L_0001: ldc.i8 1
  L_000a: ldarg.1
  L_000b: ldfld int32 TestPower.PowerParamsContainer::X
  L_0010: conv.i8
  L_0011: mul
  L_0012: stloc.0
  L_0013: ldloc.0
  L_0014: ldarg.1
  L_0015: ldfld int32 TestPower.PowerParamsContainer::X
  L_001a: conv.i8
  L_001b: mul
  L_001c: stloc.0
  L_001d: ldloc.0
  L_001e: ldarg.1
  L_001f: ldfld int32 TestPower.PowerParamsContainer::X
  L_0024: conv.i8
  L_0025: mul
  L_0026: stloc.0
  L_0027: ldloc.0
  L_0028: stloc.2
  L_0029: ldloc.2
  L_002a: ret
}
```

Что эквивалентно

```
public long Power2_9ad883fa5a9540b5a5b52f049d2ae9d5(PowerParamsContainer
container)
{
  long num = container.X;
  num *= container.X;
  return (num * container.X);
}
```

на языке C#.

7 Тестирование

Тестирование программы производилось на приложениях состоящих из нескольких компонент и имеющих в целом следующую структуру: основная сборка приложения с управляющим кодом и дополнительные сборки с различными математическими методами.

Результаты тестирования приведены в таблице 1:

Название теста	Исходная программа	Оптимизированная программа	Ratio
power	323974614 ticks	272032608 ticks	1.2
	16384 bytes	5632 bytes	
matrix_multiplication	3875373035 ticks	2407462475 ticks	1.61
	16384 bytes	30208 bytes	
cos	52023591 ticks	28433314 ticks	1.83
	16384 bytes	4608 bytes	
ray_tracing	408398290 ticks	282160410 ticks	1.45
	16384 bytes	30720 bytes	
dot	221543368 ticks	202639389 ticks	1.1
	16384 bytes	4608 bytes	
chebyshev	53595808 ticks	49220640 ticks	1.1
	16384 bytes	6144 bytes	
fft	108432642 ticks	112338717 ticks	0.96
	16384 bytes	38400 bytes	

Комментарии к таблице

Тестирование производилось на компьютере с центральным процессором Intel Pentium D с тактовой частотой 3.2 GHz и 2 Gb оперативной памяти. Для каждого теста в первой строке указано время работы программы, а во второй строке – размер программы. Время работы измерялось специальных единицах времени (ticks). 1 tick = 100 ns.

power – возведение в степень. Используется разделение переменных, при котором степень равна 15.

matrix_multiplication – умножение матриц размером 30x30. Используется разделение переменных, в котором одна из матриц известна в момент оптимизации для каждого вызова.

cos – вычисление косинуса угла через разложение в ряд Тейлора. Используется разделение переменных, в котором количество членов ряда известно в момент оптимизации и равно 10.

ray_tracing – программа трассировки лучей, которая по заданной двумерной сцене (состоящей из нескольких объектов) и набору лучей определяет пересечения лучей с объектами. Использовалось разделение переменных, в котором сцена известна в момент оптимизации и состоит из 100 объектов типа «окружность».

dot – программа вычисляющая произведение (dot product) двух векторов размером 30x30. Используется разделение переменных, в котором один из векторов известен в момент оптимизации.

chebyshev – вычисление коэффициэнтов Чебышева при рассчете аппроксимации некоторой функции. Используется разделение переменных, в котором количество точек и концы отрезка известны в момент оптимизации. Для данного теста количество точек равно 10 и используется отрезок $[1.0, 10.0]$.

fft – Быстрое дискретное преобразование Фурье (вещественное). Используется разделение переменных, в котором количество точек известно в момент оптимизации и равно 128.

Стоит обратить внимание на то, что тесты *dot, chebyshev* и *fft* показывают очень плохие результаты. Это происходит в силу нескольких причин. В данных тестах активно используется арифметика над числами с плавающей точкой, а также вычисления синусов и косинусов. Несмотря на то, что остаточная программа содержит гораздо меньшее количество выполняемых инструкций и все возможные значения с плавающей точкой вычислены и подставлены в код, JIT компилятор создает гораздо более эффективный код для исходной программы, нежели для остаточной и это приводит к таким результатам. Предметом дальнейшего исследования может являться изучение особенностей JIT компиляции и создание более эффективного алгоритма оптимизации с учетом этих особенностей.

8 Заключение

Результатом дипломной работы явилось создание глобального оптимизатора для приложений на платформе .NET Spider.

Spider поддерживает практически весь набор инструкций и среду выполнения .NET 2.0. Программа показала себя пригодной для оптимизации реальных приложений. Программа обладает большим количеством настроек для управления процессом оптимизации и удобна в использовании.

Тестирование программы показало, что реализация достаточно эффективна и в большинстве случаев можно получить более быстрые программы и компоненты, чем исходные.

Программа Spider показала себя применимой в качестве исследовательского инструмента, с помощью которого можно применять различные преобразования к программам на CIL и сравнивать эффективность алгоритмов.

В дальнейшем в программу Spider можно ввести:

1. *Поддержка коллекций.* В современных .NET приложениях зачастую используются коллекции (наборы элементов) такие как списки, хэш-таблицы, словари и т.п. Если ввести для них такую же поддержку как для массивов (то есть известны некоторые элементы или длина), можно достичь лучших результатов.

2. *Де-виртуализация*. Методы с абстрактными аргументами обычно используются с аргументами конкретных типов. Возможно создание нескольких вариантов метода для каждого возможного типа аргументов, снизив, таким образом, расходы на работу с абстрактными типами.

3. *Встраивание динамических полей*. Динамические поля можно заменить на локальные переменные во время специализации.

4. *Учет особенностей JIT компилятора.* Алгоритм можно модифицировать так, чтобы он порождал код легко оптимизируемый JIT компилятором. Это поможет добиться прироста производительности в большем количестве случаев.

Литература

[1] *.NET Framework Common Language Runtime Architecture. Version 1.9 Final, 1999 Microsoft Corporation*

[2] **Peter Bertelsen.** *'Binding-time analysis for a JVM core language, April 1999'.* *ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/bta-core-jvm.ps.gz*

[3] *ECMA International. Common Language Infrastructure (CLI), Standard ECMA-335, 2nd edition (December 2002).* *http://www.ecma-international.org/publications/standards/ecma-335.htm*

[4] **Krzysztof Czarnecki, Ulrich Eisenecker. '***Generative Programming: methods, tools and applications' Addison-Wesley, 2000; ISBN 0-210-30977-7*

[5] *Microsoft Portable Executable and Common Object File Format Specification. Version 8.0, May 16, 2006 Microsoft Corporation.* *http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx*

[6] *MSIL Instruction Set Specification. Version 1.9 Final, 1999 Microsoft Corporation*

[7] **Neil D.Jones, Carsten K. Gomard, Peter Sestoft**. *'Partial Evaluation and Automatic Program Generation' Prentice Hall International, 1993; ISBN 0-13-020249-5*

[8] **Andrei M. Chepovsky, Andrei V. Klimov, Arkady V. Klimov, Yuri A. Klimov, Andrei S. Mishchenko, Sergei A. Romanenko, and Sergei Yu. Skorobogatov. '***Partial Evaluation for Common Intermediate Language' Lecture Notes in Computer Science, 171-177, Volume 2890/2003, Springer Berlin/Heidelberg*

[9] **Morten Marquard, Bjarne Steensgaard.** *'Partial Evaluation of an Object-Oriented Imperative Language' Master's Thesis, Department of Computer Science, University of Copenhagen, Denmark, April 1992.* *ftp://ftp.research.microsoft.com/users/rusa/thesis.ps.Z*

[10] **C. Consel, O. Danvy.** *'Partial evaluation of pattern matching in strings' Information Processing Letters, 30:79-86, January 1989*

[11] **M. A. Bulyonkov. '***Polyvariant mixed computation for analyzer programs' Acta Informatica, 21:473-484, 1984*

[12] **Ulrik P. Schultz: Partial Evaluation for Class-Based Object-Oriented Languages. In O. Danvy, A. Filinski (eds.).** *'Program as Data Objects' Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001. LNCS 2053, Springer-Verlag 2001, pages 173–197.*

[13] **O. Danvy, A. Filinski.** *'Representing control: A study of the cps transformation', Mathematical Structures in Computer Science, 2(4):361-391, 1992*

[14] **J. Neighbours.** *'Software construction using components' Ph. D. Thesis, (Technical Report TR-160), Department Information and Computer Science, University of California, Irvine, 1980*

[15] **Ulrik P. Schultz, Julia L. Lawall, Charles Consel, Gilles Muller.** *'Towards Automatic Specialization of Java Programs'. 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, Portugal, June 1999. LNCS 1628, Springer-Verlag 1999, pages 367–390.*