Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Межпроцедурные оптимизации в среде PROMISE

Дипломная работа студента 59 группы
Шапоренкова Дмитрия Александровича

Научный руководитель:	.... (Булычев Д.Ю.)
Рецензент:	.... (Вояковская Н.Н.)
“Допустить к защите”, заведующий кафедрой, профессор,д.ф-м.н.	.... (Терехов А.Н.)

Санкт-Петербург
2002

Оглавление

Глава 1

Введение

С ростом возможностей современных процессоров значение оптимизирующих компиляторов не уменьшается, а наоборот, увеличивается. Это объясняется тем, что лишь компилятору под силу использовать весь потенциал архитектуры, применяя различные методы анализа и преобразования программы, позволяющие наилучшим образом "приспособить" ее для выполнения. Вместе с тем, низкоуровневые архитектурно-зависимые оптимизации не дадут хороший результат, если им не будут предшествовать оптимизирующие просмотры высокоуровневого представления программы (следует заметить, что под "высокоуровневым представлением" мы здесь понимаем внутреннее представление программы уровня более высокого, чем последовательность инструкций). Значение последних особенно велико в так называемых настраиваемых компиляторах, которые способны генерировать код для различных платформ.

1.1 Примеры межпроцедурных оптимизирующих преобразований

Оптимизирующие преобразования по области их действия можно разделить на следующие группы:

- **Локальные** преобразования, рассматривающие один фрагмент программы. В качестве фрагмента обычно выступает линейный участок. Примером подобного преобразования может послужить сокращение общих подвыражений в рамках одного линейного участка.

- **Глобальные** оптимизации, оперирующие на уровне процедуры. К числу таких оптимизаций относится, например, распространение констант и выполнение константных вычислений.

- **Межпроцедурные** оптимизации, действующие на всю программу и использующие информацию о структуре вызовов процедур в программе. Естественным представлением для таких оптимизаций является граф вызовов процедур.

Данная работа посвящена в основном межпроцедурным оптимизациям. Мотивацией для оптимизаций на межпроцедурном уровне является тот факт, что структурное программирование диктует необходимость выделения подзадач в отдельные процедуры. Такое разбиение программы улучшает читаемость и сопровождаемость, однако, как правило, не дается даром с точки зрения эффективности. В любой архитектуре с вызовом процедуры ассоциированы некоторые дополнительные действия (создание/свертка рамки стека, передача параметров и т.п.), замедляющие выполнение программы. Кроме того, для конвейерного процессора инструкция вызова сама по себе оказывает негативное влияние на производительность, так как с большой вероятностью вызывает сброс конвейера.

Помимо этих факторов, непосредственно влияющих на выполнение программы, следует заметить, что разбиение программы на процедуры снижает точность различных видов потокового анализа программы, производимого оптимизирующим компилятором. Если анализ работает лишь в рамках одной процедуры, то компилятор вынужден делать консервативные предположения о влиянии операторов вызова. Эти предположения могут значительно ухудшить точность результатов анализа, а в крайнем случае сделать эти результаты тривиальными. Например, рассмотрим следующий фрагмент кода:

```
...
int global;
void print (int);
int main ()
{
  ...
  int res[N];
  for (int i = 0; i < N; i++)
  {
     print (i);
     res[i] += 100 * global + 10 * i;
  }
  ...
}
```

В данном случае выражение "100 * global" инвариантно относительно цикла, и может быть вычислено перед циклом с сохранением его значения во временной переменной. Однако, компилятор не сможет выполнить такое преобразование, не будучи уверенным в том, что значение переменной global не меняется в результате вызова функции print.

В данной работе будут рассматриваться следующие межпроцедурные оптимизации:

- **Встраивание вызовов**. Это преобразование заключается в замене оператора вызова кодом вызываемой функции. В результате при выполнении не расходуется время на создание/свертку стековой рамки для вызываемой функции. Кроме того, встраивание вызова "проясняет" для компилятора ситуацию в точке вызова, т.е. делает возможным анализ вызываемого кода в контексте вызывающей функции, что открывает возможности для дальнейших оптимизаций.

  Вместе с тем, встраивание обладает и отрицательными побочными эффектами. Наиболее очевидным является увеличение размера кода, которое может привести к целому ряду негативных последствий для выполняемой программы. Например, после агрессивного встраивания размер кода может вырасти настолько, что (в системе с виртуальной памятью) число страниц с кодом превысит размер рабочего набора процесса, что приведет к необходимости подкачивать их с диска, и производительность программы резко упадет. Другим неприятным последствием может быть увеличение числа промахов в кэш инструкций в результате увеличения размера тела функции после встраивания вызываемых ею функций, что также негативно отразится на производительности.

Помимо увеличения размера кода, встраивание также увеличивает размер стековой рамки вызывающей функции, поскольку к ней присоединяются локальные переменные вызывающей функции. В результате может возникнуть проблема переполнения стека, как, например, в данной ситуации:

```
void f1 ()
{
  int arr[10000];
  ...
}
void f2 (int n)
{
  if (n == 0) f1 ();
  else
  {
    f2 (n-1);
    ... // какие-нибудь действия
  }
}
```

В данной ситуации большой массив arr на стеке для функции f1 захватывается лишь во время последней (т.е. самой верхней по стеку) рекурсивной активации функции f2. Предположим теперь, что вызов f1 () встроен в f2. В результате каждая активация f2 будет захватывать пространство на стеке для массива arr, что при больших значениях параметра n может привести к переполнению стека.

- **Межпроцедурное распространение констант**. Данное преобразование пытается найти функции, у которых какой-либо из параметров имеет одно и то же значение во всех точках вызова. В этом случае можно выполнить распространение констант внутри данной функции с учетом значения этого параметра. Кроме того, можно рассматривать и обратную задачу, а именно, идентификацию константных возвращаемых значений и распространение их в точки вызова.

- **Экономия вызовов**. Эта оптимизация устраняет лишние вызовы функций, т.е. вызовы, результат которых ничем не отличается от предыдущего вызова той же функции. Заметим, что это неформальное определение, разумеется, не может претендовать на строгость, поскольку не уточняет понятия “результат” и “предыдущий вызов”. Под “результатом” мы (опять-таки, неформально) понимаем возвращаемое значение функции, а также значения глобальных переменных, которые она изменяет (побочные эффекты). “Предыдущий вызов” может действительно предшествовать данному вызову в тексте программы (и тогда это преобразование является частным случаем экономии общих подвыражений), а может быть предыдущим лишь в смысле выполнения (и тогда это вынесение из цикла инвариантного выражения). Оба случая проиллюстрированы ниже:

```
int mystrlen (const char * s)
{
   for (const char * ptr = s; *ptr; ptr++) ;
   return ptr - s;
}
void f1 (char * s)
{
   int num1 = mystrlen (s) + 1;
   int num2 = 2 * mystrlen (s) - 1; // случай 1
   ...
   for (int i = 0; i < mystrlen (s); i++) // случай 2
     putc (s, stdout);
}
```

- **Удаление неиспользуемых функций**. Эта совсем несложная оптимизация удаляет из программы функции, которые никогда не вызываются и не используются другим образом (для языка C это означает, что они не адресуются с помощью операции взятия адреса). Такие функции могли быть в программе изначально (иногда такое можно наблюдать в больших программах) или появились позже в результате встраивания функции во все точки ее вызова. Следует заметить, что степень агрессивности этого преобразования зависит от того, какая часть программы доступна оптимизатору. Если оптимизатор уверен, что он “видит” перед собой всю программу, то можно удалять любые неиспользуемые функции. В противном случае, это преобразование применимо только к функциям с файловой областью видимости (т.е. объявленных с модификатором **static**), поскольку такие функции не могут использоваться компоновщиком при объединении объектных модулей.

Глава 2

Обзор существующих подходов и реализаций

Межпроцедурные оптимизации не снискали себе популярности среди разработчиков широко распространенных компиляторов C. Причина этого проста - межпроцедурный оптимизатор значительно усложняет структуру компилятора. Обычно компилятор C транслирует функции последовательно, в порядке их определения в исходном файле. Во время компиляции функции определения (точнее, внутренние представления) остальных функций из файла, как правило, недоступны для компилятора - они либо уже оттранслированы, либо еще не обрабатывались. Такая схема, применяемая в gcc, lcc и многих других популярных свободно распространяемых компиляторах, конечно, не дает возможности проводить межпроцедурный анализ и преобразования даже в рамках одного исходного файла. Помимо этого, надо отметить, что использование межпроцедурной, а тем более межмодульной, оптимизации может значительно увеличить продолжительность цикла "редактирование - компиляция - компоновка - запуск - отладка". Правда, при построении финальной версии программы это не имеет большого значения.

2.1 Компилятор gcc

Компилятор gcc, являющийся одним из лучших современных трансляторов, не производит оптимизации на межпроцедурном уровне. Из упоминавшихся преобразования компилятор имеет лишь ограниченную возможность автоматического встраивания функций. Для этого после компиляции функции, которая идентифицируется как "подходящая" для встраивания (это определяется исключительно из соображений размера кода функции), ее тело сохраняется в памяти, что делает его доступным для встраивания в последующие функции в этом файле. Недостатком является то, что встраиваемая функция обязательно должна быть определена перед точкой вызова, вместо которого она будет встроена. Например, в такой ситуации встраивание не сработает:

```
int f1 ();
void f2 ()
{
```

```
  int d = f1 (); // этот вызов не будет встроен
   ...
}
int f1 ()
{
  return 1;
}
```

Помимо этого, встраивание функции производится во все точки ее вызова, видимые в момент компиляции - специфика контекста вызова не принимается во внимание.

2.2 Компилятор C/C++ фирмы Intel

Компилятор C/C++ фирмы Intel является единственным (по крайней мере, для архитектуры **x86**) транслятором, поддерживающим достаточно полный набор межпроцедурных оптимизаций. В частности, компилятор выполняет автоматическое встраивание и межпроцедурное распространение констант, а также некоторые виды более низкоуровневых оптимизаций, например, переупорядочение функций в исполняемом файле с целью улучшения локальности кода (последнее делается на основе результатов профилирования инструментированной программы и последующего анализа компилятором сгенерированной трассы). Кроме того, поддерживается межмодульная оптимизация. Для этого исходные файлы компилируются в специальном режиме, в результате чего вместо файлов объектного кода создаются файлы, содержащие сериализованное внутреннее представление исходных модулей (т.н. mock files). Затем эти файлы обрабатываются компилятором, давая на выходе общий объектный файл или ассемблерный листинг.

Эксперименты, проведенные с этим транслятором, показали, что при отключенном встраивании вызовов эффект от межпроцедурных оптимизаций проявляется очень слабо. В частности, не удалось воспроизвести (даже на весьма простых примерах) ситуацию, где выполнялось бы межпроцедурное распространение констант или эффект от вызова функции учитывался бы при оптимизации вызывающей функции. По-видимому, основной упор при разработке компилятора был сделан на реализацию низкоуровневых оптимизаций (особенно межпроцедурного распределения регистров).

2.3 Компилятор vbcc (автор Volker Barthelmann)

Этот свободно распространяемый компилятор, совместимый со стандартом ISO C, является настраиваемым оптимизирующим компилятором, выполняющим большинство классических оптимизаций. Кроме того, компилятор поддерживает межпроцедурную оптимизацию, в частности, может выполнять автоматическое встраивание вызовов, межпроцедурный анализ потока данных, результаты которого учитываются в процессе внутрипроцедурных преобразований, а также межпроцедурное распределение регистров. Возможна межмодульная оптимизация, которая проводится одним из двух способов:

- Компилятор может работать в пакетном режиме, принимая на вход сразу несколько исходных файлов. В этом случае все исходные файлы компилируются вместе и порождается единственный результирующий объектный файл.

- В режиме раздельной компиляции файлы могут компилироваться не в объектный код, а в специальный формат, после чего файлы в таком формате читаются компилятором, который выполняет генерацию кода и оптимизацию на объединенном внутреннем представлении. По существу, этот подход совершенно аналогичен используемому в компиляторе фирмы Intel.

На небольших примерах этот компилятор порождает код, не уступающий по качеству коммерческим трансляторам и gcc. Тем не менее, как и в предыдущем примере, увидеть результаты эффект от межпроцедурных оптимизаций в сгенерированном ассемблерном коде не удалось.

2.4 Другие примеры

В статье [8] приводится описание оптимизирующего компоновщика. Компоновщик содержит оптимизатор и настраиваемый генератор кода, который, подобно используемому в `lcc`, основан на технологии BURS и отличается лишь тем, что генерирует непосредственно машинный код, а не ассемблерный текст (Автор увреждает, что генерация непосредственно машинного кода значительно увеличивает производительность системы из-за отсутствия ассемблирования как отдельного этапа трансляции). Компоновщик принимает на вход файлы с языко-независимым внутренним представлением и генерирует на выходе исполняемый файл. Автор подчеркивает, что компоновка именно промежуточного представления, а не объектного кода позволяет оптимизатору распознавать идиоматические последовательности, генерируемые компиляторами (например, косвенный вызов метода или обращение к полю). В настоящее время имеется два front-end"а к компоновщику - один для Modula-3, другой для C++. Среди выполняемых оптимизаций можно отметить распространение информации о типах и распространение констант. Первая предназначена для преодоления накладных расходов, связанных с наличием в Modula-3 так называемых "непрозрачных типов" (opaque types), т.е. объектов, в которых интерфейс отделен от реализации, и реализация недоступна для клиентов. Поскольку в момент компиляции кода клиента реализация непрозрачного типа невидима для компилятора, создание объектов такого типа и обращения к ним происходят через уровень индирекции. Компоновщик, имея в своем распоряжении полную систему типов, заменяет эти косвенные вызовы на непосредственные. Компоновщик может выполнять также встраивание вызовов. Процедуры, к которым применяются оптимизации, определяются исходя из результатов профилирования инструментированной программы.

В работе [9] описывается среда **FIAT** для решения различных задач анализа потока данных на межпроцедурном уровне. Внутренним представлением программы является аннотированный граф вызовов. Для каждого узла хранится информация *ProcSummary*, включающая идентификатор процедуры, операторы вызова, имеющиеся в этой процедуре, и данные, специфичные для конкретной задачи. В процессе решения задачи результаты для каждого узла графа вызова заносятся в эти аннотации. Среда предоставляет универсальный драйвер для решения задач межпроцедурного анализа потока данных, который параметризуется стандартным определением задачи. В это определение входят: функция *InitNode*, выполняющая локальный анализ и генерирующая начальное решение в рамках одной процедуры; функция *InitEdge*, генерирующая начальное решение для ребра графа вызовов; функция *Join*, объединяющая два решения; функция *Transfer*, определяющая преобразование решения при переносе информации через ребро графа; направление задачи (прямая/обратная). Кроме того, задача может использовать результаты решения других задач. Решение получается с помощью классического итеративного алгоритма анализа потока данных. При этом, если для решения данной задачи требуются результаты других задач, и они недоступны, то эти задачи решаются перед данной; таким образом, спецификации задачи неявно определяют порядок выполнения просмотров анализа (предполагается, конечно, отсутствие циклических зависимостей между задачами). Результаты решения задач кэшируются в аннотациях и выдаются по запросам до тех пор, пока сохраняют свою актуальность (т.е. пока структура графа вызовов остается неизменной). Проблема обновления решений при изменении графа остается открытой. Результаты решения задач анализа можно использовать при проведении преобразований, в частности, для определения выгоды того или иного преобразования. Наличие собственного, независимого от используемых front-end'а и back-end'а, представления программы позволяет интегрировать среду с любым компилятором (упоминаются реализации для компилятора SUIF [13], разработанного в Стэнфордском университете, и системы статического анализа производительности программ ParaScope [12]), но оставляет реализацию собственно преобразований практически полностью разработчику компилятора.

В качестве примера преобразования авторы приводят клонирование процедур, т.е. создание различных копий процедуры и специализация этих копий конкретными контекстами вызова. С этой целью среда используется для решения задачи межпроцедурного распространения констант. Клонированию подлежат те процедуры, у которых имеется параметр, принимающий несколько значений в различных контекстах вызова. Такая процедура клонируется на число копий, соответствующее числу значений параметра, после чего каждая копия специализируется с учетом (константного) значения параметра. Следует заметить, что при этом никак не принимается во внимание устройство потока управления в контекстах вызова (например, бесполезно клонировать процедуру, которая выполняется только один раз в одном контексте и один раз - в другом). В целом, для среды характерна практически исключительная ориентация на поток данных, что является ее существенным ограничением. В планы авторов входит дополнение внутреннего представления графом вызовов процедуры, что позволит строить решения для flow-sensitive задач анализа, а также учитывать поток управления при определении выгоды преобразований.

Глава 3

Окружение: выбор базового компилятора, интеграция в процесс трансляции, внутреннее представление программы

3.1 Компилятор lcc

В качестве базового транслятора для реализации оптимизатора был выбран компилятор lcc. Исходным языком для компилятора является ANSI C, который, несмотря на свой возраст, остается одним из основных языков системного программирования. Во многих ситуациях (например, разработка приложений для встроенных систем и систем реального времени) C и в настоящее время является практически единственной альтернативой ассемблеру.

Компилятор lcc, используемый в данной работе, является современным перенацеливаемым компилятором. Полное описание его приведено в книге [4]. lcc имеет машинно-независимый front-end, который в процессе синтаксического анализа входной программы строит внутреннее представление, передаваемое на вход back-end'у. Back-end (программа `pass2`) генерирует код для указанной целевой архитектуры. Для перенацеливания компилятора необходимо описать целевой процессор в виде спецификаций для генератора `lburg`. Такое описание включает в себя определение соответствия между деревьями выражений и инструкциями процессора в виде правил грамматики, при этом правила снабжаются стоимостями. Выбор инструкций при генерации кода происходит из соображений минимизации функции стоимости, представляющей суммарную стоимость всех правил, используемых в покрытии дерева программы, при построении решения используется метод динамического программирования (т.н. технология **BURS**). Кроме того, описание содержит ряд вспомогательных процедур, обеспечивающих генерацию кода для пролога/эпилога функции, генерацию экспортируемых и импортируемых символов с учетом синтаксиса целевого ассемблера, и некоторых других. Файл, содержащий это описание, обрабатывается генератором lburg, который создает исходный текст генератора кода. Сравнительная легкость перенацеливания на новую платформу (в частности, отсутствие необходимости писать вручную большое количетво кода) является несомненным достоинством компилятора lcc.

Следует, однако, заметить, что lcc не является оптимизирующим компилятором в полном смысле этого слова, т.е. не содержит отдельных оптимизирующих просмотров. В результате, как показывают результаты исследований ([10]), код, генерируемый lcc, обычно уступает по своему качеству коду, порождаемому такими трансляторами как `gcc` и некоторыми другими.

3.2 Среда PROMISE

Данная дипломная работа выполнена в рамках разработки среды PROMISE (PROgram MorphIng extsenSible Environment) для lcc. Целью является создание машинно-независимого оптимизатора, который, с одной стороны, сохранит легкость перенацеливания lcc, а с другой, позволит повысить качество порождаемого кода, приблизив его к лучшим оптимизирующим компиляторам. Выбор lcc в качестве базового транслятора дает следующие преимущества:

- lcc позволяет записывать внутреннее представление программы после front-end'а в сериализованной форме, которая может быть использована в качестве входных данных back-end'ом. Как показано ниже, это дает возможность относительно легко встроить оптимизатор в процесс трансляции и обойтись при этом без модификации кода самого компилятора. Кроме того, появляется возможность выбора языка реализации оптимизатора. Внутреннее представление lcc получено с помощью программы asdlGen, которая, принимая на вход абстрактное описание структур данных на языке ASDL (Abstract Syntax Definition Language), генерирует определения этих структур и операции с ними на различных языках программирования (C, C++, Java, SML и т.д.). Операции позволяют, в частности, сохранять структуры данных в бинарном виде, не зависящем от языка, и восстанавливать их. Рассматриваемый в данной работе оптимизатор реализован на языке C++, в то время как сам компилятор lcc написан полностью на C.

- Компилятор lcc не является экспериментальным транслятором (в отличие, например, от системы SUIF), мало пригодным для практического использования. Это подтверждается как широкой популярностью lcc, так и фактом существования версии lcc-win32, ориентированной на разработку приложений для ОС Windows. Это позволяет надеяться на то, что оптимизатор может найти практическое применение.

Рис. 3.1: Схема интеграции межпроцедурного оптимизатора в процесс трансляции

3.3 Интеграция оптимизатора в процесс трансляции

Приведем схему, изображающую устройство оптимизатора и интеграцию его в процесс трансляции программы с помощью lcc. Данная схема позволяет выполнять межпроцедурную оптимизацию как в рамках одного модуля (т.е. единицы трансляции) программы, так и на межмодульном уровне. Каждый файл исходной программы компилируется с помощью lcc. На выходе получается файл, содержащий сериализованное внутреннее представление модуля (дерево). Затем (в случае межмодульной оптимизации) файлы внутреннего представления компонуются специальной программой-компоновщиком, результатом работы которого является объединенное внутреннее представление нескольких модулей. При этом происходит разрешение тех внешних ссылок, которые удается разрешить в рамках данного набора деревьев программ. Скомпонованный модуль подается на вход оптимизатору, который загружает его и преобразует в свое собственное представление. Необходимость последнего объясняется тем, что собственное внутреннее представление компилятора lcc непригодно для модификации (например, там отсутствует возможность добавления/удаления элемента из произвольной позиции в списке), что и неудивительно, поскольку оно разрабатывалось исключительно с целью передачи дерева программы от front-end'а к back-end'у. Выполнив оптимизации на своем представлении, оптимизатор преобразует его обратно в представление lcc, которое транслируется back-end'ом, и дальнейший процесс компиляции не отличается от обычного с использованием lcc.

Этой схеме присущи определенные недостатки:

- Увеличивается время компиляции программы из-за большого размера обрабатываемых файлов: объединенного файла с внутренним представлением и ассемблерного листинга.
- Изменение одного исходного файла требует перекомпоновки промежуточного представления с последующей перекомпиляцией всей программы. Таким образом, нарушается принцип раздельной трансляции, игравший большую роль с самого начала развития языка C.

Вследствие этих обстоятельств, использование оптимизатора в рамках цикла “редактирование - компиляция - компоновка - запуск - отладка” для больших проектов затруднительно. Однако его вполне возможно применять на этапе финальной компиляции программы, когда требуется получить наиболее эффективный код, а время сборки не играет существенной роли. Ниже обсуждаются некоторые пути преодоления упомянутых недостатков оптимизатора.

3.4 Внутреннее представление программы, используемое в оптимизаторе

Мы уже упоминали, что реализация оптимизатора потребовала создания собственного внутреннего представления, которое можно было бы легко модифицировать в процессе преобразований. Поскольку в основном изменениям подвергается код функций, то внутреннее представление ограничено набором конструкций, необходимых для записи тела функции. В этот набор входят переменные, константы, оператор присваивания, оператор вызова, общее подвыражение [1], сравнение, а также унарная и бинарная операции. Таблица символов и типов не конвертируется, хотя может модифицироваться в процессе преобразований (путем добавления новых символов для создаваемых переменных).

Программа представляется в виде списка объектов верхнего уровня, т.е. функций и глобальных переменных. Каждая функция имеет двойное представление. С одной стороны, она представляется в виде графа потока управления *(CFG)*, узлы которого помечены операторами, а ребра бывают двух типов (простое ребро и переход) и отражают передачу управления в программе: простое ребро соответствует последовательному исполнению, а ребро перехода - переходу. С другой стороны, функция представляется в виде списка операторов, упорядоченного согласно исходному тексту программы (точнее, этот порядок есть порядок операторов в промежуточном представлении `lcc`, который в основном соответствует исходному тексту). Такое двойное представление несколько затрудняет модификацию кода при оптимизирующих преобразованиях, однако существенно облегчает его конвертацию обратно в дерево `lcc`. Граф потока управления необходим для выполнения почти всех оптимизирующих преобразований, но наличие только

[1] Необходимость в такой конструкции возникла потому, что front-end `lcc` выполняет выделение общих подвыражений в пределах линейного участка, представляя его в виде дэга

графового представления функции усложнило бы обратную конвертацию, поскольку привело бы к необходимости минимизации числа операторов перехода.

Помимо собственно кода, представление каждой функции включает в себя различную информацию, собираемую анализом потока данных. Примерами такой информации являются метки узлов графа потока управления, соответствующие операторам присваивания, и цепочки "использование-определения" и "определение - использования". Эта информация необходима для многих оптимизирующих преобразований. Сбор этой информации выполняется по принципу "вычисление по мере необходимости" (lazy evaluation). Каждая функция включает в себя признак актуальности этой информации (т.е. соответствия ее коду функции), которому изначально присваивается значение "ложь". Оптимизирующее преобразование, нуждающееся в этой информации, проверяет признак актуальности, и если его значение - "ложь", инициирует построение информации. При модификации кода функции в процессе преобразования, которое вызывает несоответствие этой информации телу функции, признаку актуальности вновь присваивается значение "ложь". Такая схема работы позволяет гарантировать корректность информации о потоке данных, избегая при этом лишних ее построений.

Замечание. Определенные виды сложных выражений, в частности, вложенные вызовы, преобразуются front-end'ом lcc в последовательность более простых. Это удобно для проведения большинства оптимизаций. Поэтому используемое нами представление является представлением среднего уровня (Middle-level intermediate representation, MIR), если следовать терминологии, принятой в [2].

Замечание. Всюду далее, если это не оговаривается специально, под термином "внутреннее представление" или "промежуточное представление" мы понимаем собственное представление оптимизатора.

Глава 4

Просмотры оптимизатора

В данном разделе рассматривается различные анализирующие и оптимизирующие просмотры, реализованные в оптимизаторе, их взаимное влияние и связь с классическими оптимизирующими преобразованиями, действующими в рамках одной процедуры.

4.1 Построение графа вызовов процедур

4.1.1 Построение графа вызовов при наличии внешних функций и указателей на функции. Асинхронные вызовы.

Как уже упоминалось, граф вызовов, т.е. граф, вершины которого соответствуют процедурам, а дуги - операторам вызова, является естественным представлением программы для межпроцедурного оптимизатора. Следует сразу заметить, что построение полного графа вызовов, соответствующего возможным выполнениям программы, невозможно на этапе компиляции. Причиной этому является наличие функций, код которых на данном этапе недоступен - таковыми, например, являются библиотечные функции или функции из других объектных модулей, присоединяемые к программе на этапе компоновки объектного кода. С одной стороны, это не является проблемой, поскольку нас не интересует как эти функции вызывают друг друга, так как никакие преобразования к ним все равно не применимы. С другой стороны, важно не потерять часть графа вызовов, поскольку возможна ситуация, когда пользовательская функция **f** передается по указателю в библиотечную - и такая ситуация должна быть отражена ребром в графе, поскольку означает, что функция **f** становится достижимой, т.е. может быть выполнена в процессе выполнения программы.

Другой проблемой является ситуация, когда из инструкции вызова не очевидно, какую именно процедуру она вызывает. Такое возможно, если в языке допускаются значения процедурных типов. В случае отсутствия в языке значений процедурных типов граф вызовов может быть построен простым сканированием кода каждой процедуры в поисках инструкций вызова. В нашем случае (язык C) ситуация гораздо более сложная, поскольку имеются указатели на функции, а наличие произвольных преобразований типов делает невозможным использование простого итеративного подхода, подобного описанному в [2]. В простейшей реализации необходимо консервативно предполагать, что вызов по указателю может привести к вызову любой функции (причем необязательно сигнатура функции должна совпадать с определением указателя - возможно преобразование типов между указателями на функции). Более точным методом является использование анализа синонимов. Правда, внутрипроцедурного анализа синонимов недостаточно для построения графа вызовов, как показывает следующий пример:

```
[file1.c]
extern void initp (void (**p) (void));
void fun (void)
{
...
}
int main ()
{
  void (*p) void;
  initp (&p);
  (*p) ();
...
}
[file2.c]
extern void fun (void);
void initp (void (**p) (void))
{
   *p = &fun;
}
```

Внутрипроцедурный анализ синонимов, ограниченный пределами процедуры **main**, не сможет выяснить, что указатель **p** в момент вызова содержит адрес процедуры **fun**, и придется консервативно предполагать, что он может содержать адрес любой процедуры. Однако, если код обоих модулей доступен оптимизатору, и проводится межпроцедурное распространение информации о синонимах, то значение указателя **p** может быть установлено. Но здесь возникает “порочный круг” - для межпроцедурного анализ синонимов требуется граф вызовов, который в этот момент еще не построен. Это означает, что построение графа вызовов должно проводиться одновременно с межпроцедурным анализом синонимов. Соответствующие алгоритмы рассматриваются, например, в [14]. В данной работе мы предполагаем, что указатель на функцию может содержать адрес любой процедуры.

Остановимся еще на одном моменте, который необходимо учитывать при построении графа вызовов. Речь идет об асинхронных событиях, иначе говоря, об обработке исключений. В С обработка исключений может быть реализована с помощью пары системных функций **setjmp/longjmp**, при этом setjmp устанавливает обработчик исключений, а longjmp вызывает обработчик. Это должно быть отражено в графе вызовов, поскольку означает достижимость во время выполнения функций, содержащих обработчик, из функций, вызывающих его. Для этого мы просто соединяем функции, содержащие вызовы longjmp, специально помеченными (с целью отличать их от обычного вызова) дугами с функциями, содержащими вызовы setjmp.

4.1.2 Аннотации узлов графа вызовов. Анализ графа.

Рассмотрим подробнее информацию, ассоциированную с узлами и ребрами графа вызовов. Мы храним в узле уникальный идентификатор функции, ссылку на код функции (если он доступен, т.е. функция не является внешней), а также аннотацию, содержащую специфичные для применяемых алгоритмов данные (например, с узлом ассоциируется множество идентификаторов переменных, которые встречаются в левых частях присваиваний в данной функции и, тем самым, могут быть изменены в результате ее выполнения). С ребром ассоциированы ссылка на оператор вызова (если последний имеется), тип ребра (обычный вызов, фиктивное ребро, используемое для представления вызовов через указатели, а также ребро, соответствующее асинхронному вызову), и, как и в случае узла, аннотация, специфическая для алгоритма.

После построения множества узлов и ребер графа вызовов выполняется анализ графа. В результате анализа строятся прямая и обратная нумерации и выделяются компоненты сильной связности графа (бикомпоненты). Бикомпоненты используются для определения рекурсивных функций: функция является рекурсивной тогда и только тогда, когда узел, представляющий ее в графе вызовов, содержит ребро-петлю (непосредственная рекурсия) или принадлежит бикомпоненте, в которой больше одного узла, т.е. входит в некоторый цикл графа вызовов (косвенная рекурсия). Анализ основан на алгоритмах, приведенных в [3]; подробности здесь опускаются.

4.2 Автоматическое встраивание вызовов

Встраивание вызовов процедур является одним из самых эффективных оптимизирующих преобразований. Устраняя накладные расходы на вызов процедур и открывая дорогу для других оптимизаций, встраивание способно значительно ускорить работу результирующей объектной программы. В то же время, неразумное встраивание может резко увеличить объем кода и привести к многочисленным негативным эффектам, которые упоминались выше.

4.2.1 Автоматическое встраивание и встраивание с помощью атрибутов, задаваемых пользователем

Сразу оговорим смысл слова “автоматическое”. Термин “автоматическое” означает, что решение о встраивании того или иного вызова принимается компилятором. Возможен и другой подход - подсказка компилятору со стороны пользователя в виде атрибута функции (пример - ключевое слово **inline**). Этот подход, однако, обладает следующим недостатком. Атрибут, представляющий собой подсказку, как правило (так обстоит дело в C,C++ и многих других языках), приписывается функции при ее определении и означает встраивание всех вызовов данной функции в данной единице трансляции. Таким образом, не делается различие между точками вызова, что ограничивает полезность такой оптимизации: например, встраивание вызовов, выполняющихся в цикле, очевидно, может дать гораздо больший эффект, чем встраивание вызовов, которые выполняются только один раз в процессе работы активации функции, содержащей данный вызов. Пользователь же обычно не имеет четкого представления о том, будет ли действительно полезным встраивание данной функции. Часто функции помечают как встраиваемые, лишь исходя из их размера. Если компилятор будет руководствоваться лишь указаниями пользователя, то он может встроить вызовы, которые этого "не заслуживают" и пропустить критические для производительности участки кода. [1]

Все это позволяет сделать вывод, что главным критерием при принятии решения о встраивании вызова для компилятора должны служить более веские доводы, чем предпочтения пользователя. Последние, однако, тоже могут приниматься во внимание, как будет показано ниже.

4.2.2 Использование динамической и статической информации при принятии решений о встраивании

При принятии решений о том, какие вызовы стоит встраивать, компилятор может руководствоваться двумя видами информации - статической и динамической. Под динамической информацией подразумевается использование результатов выполнения предварительно инструментированной путем добавления специального трассировочного кода программы (профилирование). Такой код выполняет подсчет количественных параметров, например, частоты вызовов функции. Эти данные компилятор может использовать при принятии решений о встраивании вызовов. Среди недостатков динамической информации необходимо отметить:

- Невозможно получить реалистичную картину типичного выполнения программы, если множество входных данных бесконечно. Результаты профилирования отражают выполнение программы на одном или на нескольких наборах входных данных. На других наборах эта картина может сильно отличаться.

- Для того, чтобы воспользоваться профилировочной информацией, необходимо предварительно скомпилировать программу с инструментированием, затем выполнить ее, записать результаты профилирования и только потом скомпилировать с оптимизацией, которая использует эти результаты. Такой подход, очевидно, неудобен из практических соображений. С этим, однако, можно было бы смириться (например, выполнять это сложную процедуру только при построении окончательной версии), но вполне может оказаться, что запустить программу на выполнение затруднительно или невозможно вообще (например, если целевая и инструментальная платформы не совпадают, как в случае кросс-компиляции).

На наш взгляд, динамическая информация является более полезной при компиляции "на лету" (*just-in-time*), поскольку в этом случае получаемая картина соответствует текущему выполнению программы, и для этого же выполнения проводится оптимизация. Достаточно полный обзор различных подходов к принятию решений о встраивании вызовов приведен в [5], [6]. Наша реализация использует исключительно статическую информацию.

[1] Противоположная точка зрения по этому вопросу высказывается автором языка C++ в книге [15]

4.2.3 Факторы, влияющие на принятие решений о встраивании

Сформулируем факторы, которые должны приниматься во внимание при проведении встраивания вызовов функций.

- Размер вызываемой функции. Небольшие [2] функции должны пользоваться преимуществом при встраивании. В пользу этого соображения служит как тот факт, что встраивание маленьких функций в меньшей степени увеличивает общий объем кода программы, так и то, что встраивание большой функции, вызов которой выполняется в цикле, может ухудшить локальность кода и приведет, например, к увеличению числа промахов в кэш инструкций процессора.

- Общий размер модуля. Необходимо также следить за общим размером кода. Прежде всего, объем памяти, в которой будет выполняться этот код, может быть ограничен (например, если речь идет о компиляции программы, предназначенной для встроенной системы). Если даже жестких ограничений на этот объем нет, то остаются соображения локальности кода, как и в предыдущем пункте, но на уровне процесса. Как уже упоминалось выше, разрастание размера кода может привести к увеличению рабочего набора процесса и активному обмену с диском для подгрузки страниц кода.

- Размеры локальных данных вызываемой и вызывающей функции. Этот аспект также затрагивался при обсуждении преобразования встраивания. Встраивание приводит к увеличению размера локальных данных вызываемой функции, поскольку к ней необходимо присоединить локальные данные вызываемой функции. Как показано ниже, частично этот эффект может быть преодолен за счет оптимизаций вызывающей функции после встраивания, однако это удается сделать не во всех случаях. Проблема заключается в том, что если до встраивания приращения стека при выполнении вызывающей функции могло и не произойти (если выполнение встраиваемого вызова зависит от некоторого условия), то после встраивания размер стековой рамки вызывающей функции увеличивается независимо от того, будет ли выполняться встроенный код. Особенно это применимо к ситуации, когда вызывающая функция является рекурсивной, как было показано выше.

- Является ли встраиваемая функция непосредственно рекурсивной, т.е. содержит ли вызов самой себя. (Это соответствует петле в графе вызовов). Вопрос, возникающий при встраивании непосредственно рекурсивных функций, состоит в том, стоит ли их встраивать и, если

[2] Необходимо сделать замечание об оценке размеров кода. Поскольку мы проводим оптимизацию на промежуточном представлении, до стадии генерации финального кода, то оценивать размеры кода можно лишь в терминах количества конструкций промежуточного представления. Такая оценка, конечно, является только грубым приближением к реальному числу машинных инструкций, получающихся в результате генерации кода. Однако в силу того, что наше внутреннее представление является достаточно низкоуровневым, можно рассчитывать на то, что размер реального кода будет отличаться на некоторый множитель, величина которого мало зависит от входной программы. Это подтверждается и экспериментами.

да, то сколько раз встраивать рекурсивный вызов. Если, например, известно, что глубина рекурсии для данной функции при вызове не превосходит небольшой константы, то можно соответствующее число раз “раскрутить” рекурсивный вызов. Однако, вообще говоря, для выяснения таких точек вызова требуется иметь информацию о возможных значениях выражений в условиях, контролирующих рекурсию. В простых случаях такая информация может быть извлечена из результатов распространения констант . Более совершенный способ, основанный на распространении информации о возможных значениях переменных в виде “множеств значений” (*value ranges*), и приписывании каждой ветви условного оператора вероятности ее выполнения, описывается в [11].

- Контекст вызова с точки зрения потока управления. Преимущество при встраивании должно отдаваться тем вызовам, встраивание которых обеспечивает наибольший эффект с точки зрения производительности. Хотя статически предсказать такой эффект невозможно, можно использовать разумные эвристические соображения. Одним из них является стуктура потока управления в точке вызова, точнее, выполняется ли вызов в цикле. В этом случае встраивание вызова даст больший эффект, чем если бы вызов выполнялся только один раз.

- Контекст вызова с точки зрения потока данных (аргументы вызова). Другим разумным соображением при определении приоритетов встраивания может послужить наличие у оператора вызова фактических параметров, для которых удается статически вычислить значения (например, выполнив предварительно распространение констант). В этом случае после встраивания можно будет провести оптимизации вызывающей функции, которые помогут устранить лишние переменные и вычисления во встроенном коде.

4.2.4 Формальная постановка задачи. Математическая модель.

С учетом упомянутых факторов поставим теперь задачу встраивания формально. Пусть $CG = (N, E, beg, end)$ - аннотированный граф вызовов программы, N - множество узлов графа, представляющих процедуры программы, E - множество ребер графа, представляющих операторы вызова,beg, end - отображения $E \to N$, сопоставляющие ребру графа его начало и конец [3]. Положим также $NAttr$ - множество аннотаций узлов, содержащих дополнительную информацию о процедурах, $EAttr$ - множество аннотаций ребер, содержащих дополнительную информацию об операторах вызова. Считаем, что заданы функции $in : N \to E$ по правилу $in(n) = \{(n', n) \in E\}$, $out : N \to E$ по правилу $out(n) = \{(n, n') \in E\}$, $NodeAttr : N \to NAttr$ и $EdgeAttr : E \to EAttr$.

Определение. Пусть $e = (n_1, n_2) \in E$ - ребро графа вызовов. *Встраиванием ребра e* назовем следующее преобразование графа:

$$Inline_e(CG) = CG', CG' = (N, E', beg', end'),$$

[3] Граф вызовов, очевидно, является мультиграфом.

где

$$E' = (E \setminus \{e\}) \cup E'', E'' = \{e' = (n_1, n_3) : (n_2, n_3) \in E\}$$

и функции beg', end' определены соответствующим образом.
Заметим, что при этом
$NodeAttr \to NodeAttr'$:

$$NodeAttr'(n) = \begin{cases} f_1(NodeAttr(n1), NodeAttr(n2)), & n = n_1 \\ NodeAttr(n), & \text{иначе} \end{cases}$$

а также $EdgeAttr \to EdgeAttr'$:

$$EdgeAttr'(e') = \begin{cases} f_2(EdgeAttr(e''), EdgeAttr(e)), & e' \in E'' \\ EdgeAttr(e'), & \text{иначе} \end{cases}$$

где $e'' = (n_2, n_3) \in E$, если $e' = (n_1, n_3)$, $f_1 : NAttr \times NAttr \to NAttr$ и $f_2 : EAttr \times EAttr \to EAttr$.

Поясним неформально смысл написанного выше. При встраивании происходит модификация графа вызовов: удаляется ребро, соответствующее встраиваемому вызову, а ребра, выходящие из узла, представляющего вызываемую процедуру, добавляются ко множеству выходных ребер узла, представляющего вызывающую процедуру. При этом меняются атрибуты данных ребер в зависимости от атрибутов встраиваемого ребра (поскольку соответствующие им вызовы находятся теперь в контексте другой процедуры и частично наследуют свойства встроенного оператора вызова; точный смысл этой фразы описан ниже), а также атрибуты вызывающей процедуры в зависимости от атрибутов вызываемой (например, увеличивается значение размера кода данной процедуры примерно на величину размера кода вызываемой функции). Изменения атрибутов выражаются с помощью функций f_1 и f_2.

Определение. Пусть $CG = (N, E, beg, end)$ - граф вызовов. *Планом встраивания* назовем последовательность $(e_0, ..., e_n)$, где для любого $i \in [0 : n]$ $e_i \in E_i$, и E_i определяется следующим образом: $CG_0 = CG, E_0 = E$, а при $i > 1$

$$CG_i = Inline_{e_{i-1}}(CG_{i-1}), CG_i = (N, E_i, beg_i, end_i).$$

Уточним теперь вид множеств $NAttr$ и $EAttr$, а также отображений f_1 и f_2. Множества $NAttr$ и $EAttr$ содержат информацию, отражающую факторы, влияющие на принятие решений о встраивании:

$$NAttr = \{(C, L)\},$$

где C и L - число конструкций промежуточного представления и размер локальных данных функции соответственно,

$$EAttr = \{(InLoop, CallArgs, RecursiveCounter)\},$$

где $InLoop$ - признак, показывающий выполняется ли соответствующий ребру оператор вызова в цикле в контексте содержащей этот оператор функции, $CallArgs$ - последовательность конструкций промежуточного представления, представляющих фактические параметры вызова,

$RecursiveCounter$ - счетчик, используемый для ограничения по глубине встраивания рекурсивных вызовов.

Далее, рассмотрим отображения f_1 и f_2.

Пусть $e = (Caller, Callee) \in E$, $Inline(e)$ - встраивание, соответствующее ребру e, $NodeAttr(Caller) = (C_{Caller}, L_{Caller})$, $NodeAttr(Callee) = (C_{Callee}, L_{Callee})$, $EdgeAttr(e) = (InLoop, CallArgs, RecCount)$.

f_1 выражает изменение аннотации вызывающей функции после встраивания. В первом приближении можно считать, что

$$f_1(NodeAttr(Caller), NodeAttr(Callee)) = (C_{Caller}+C_{Callee}, L_{Caller}+L_{Callee}).$$

Аналогично, f_2 выражает изменение аннотации ребер, добавляемых ко множеству ребер, выходящих из узла $Caller$. Эти ребра частично наследуют свойства встраиваемого ребра:

Если

$$e' = (Callee, n_3) \in E, EdgeAttr(e') = (InLoop', CallArgs', RecCount'),$$

то

$$f_2(EdgeAttr(e), EdgeAttr(e')) = (InLoop'', CallArgs, RecCount''),$$

где

$$InLoop'' = InLoop' \vee InLoop,$$

и

$$RecCount'' = \begin{cases} RecCount + 1, & \text{если } n_3 = Callee \\ RecCount', & \text{иначе} \end{cases}$$

Приведем некоторые пояснения. Отображения f_1 и f_2 построены таким образом, чтобы отражать изменения атрибутов при модификации графа вызовов. В частности, размер кода и локальных данных вызывающей функции увеличивается на соответствующие показатели вызываемой функции. Свойство "оператор вызова выполняется в цикле в контексте содержащей его функции" для операторов вызова, находящихся во встраиваемом коде, будет иметь место если они либо выполняются в цикле, который находится во встраиваемом коде (т.е. $InLoop' = True$), либо сам встраиваемый код выполняется в цикле, что соответствует случаю, когда выполняется в цикле встраиваемый оператор вызова ($InLoop = True$). Счетчик $RecursiveCounter$ используется для контроля за встраиванием рекурсивных вызовов, поэтому его значение увеличивается на единицу при добавлении к узлу вызывающей функции ребра, которое соответствовало рекурсивному вызову вызываемой функции.

Сформулируем теперь ограничения, влияющие на принятие решений о встраивании. Во-первых, мы считаем, что задано ограничение $CodeSizeGrowthFactor$ на общий размер кода (его удобно выражать в процентах от исходного размера). Во-вторых, мы должны принять во внимание эффекты, связанные с ростом размера локальных данных функции. Мы считаем, что это ограничение существенно только для рекурсивных функций.

Введем функции $Cost : E \to [0,\infty)$ и $Benefit : E \to [0,\infty)$, выражающие соответственно стоимость и выгоду встраивания ребра. Вид этих функций уточним позже.

Пусть $(e_0, ..., e_n)$ - план встраивания, $CG_n = (N, E_n, beg_n, end_n)$ - конечный граф вызовов, $CodeSize : NAttr \to [0,\infty)$, $LocalSize : NAttr \to [0,\infty)$ по правилу

$$CodeSize(n) = codesize, LocalSize(n) = locsize,$$

где $(codesize, locsize) = NodeAttr(n)$.

Определение. *Выгодой* плана встраивания назовем величину $\sum_{k=1}^{n} Benefit(e_k)$.

Теперь задача может быть сформулирована следующим образом: найти план встраивания с максимальной выгодой при ограничении

$$\sum_{n \in N} CodeSize(n) \le InitialCodeSize * (1 + CodeSizeGrowthFactor),$$

где $InitialCodeSize$ - начальный размер кода оптимизируемого модуля.

4.2.5 Функции стоимости и выгоды встраивания ребра. Алгоритм построения решения.

Пусть $e = (Caller, Callee) \in E$, причем $EdgeAttr(e) = (InLoop, CallArgs, RecCounter)$. Уточним определение функций выгоды и стоимости встраивания ребра. Определение функции стоимости должно отражать негативные последствия встраивания ребра. Мы считаем, что

$$Cost(e) = C_1 * CodeSize(Callee) + C_2 * LocalSize(Callee)$$

(здесь C_1, C_2 - масштабирующие коэффициенты; варьируя их, можно придавать больший вес тому или иному слагаемому) [4]. Для функции выгоды мы полагаем:

$$Benefit(e) = B_O + B_1 + B_2,$$

где B_0 - предопределенная константа, а B_1 и B_2 определяются так:

$$B_1 = \begin{cases} B_1', & \text{если InLoop = True} \\ 0, & \text{иначе} \end{cases}$$

$$B_2 = \begin{cases} B_2', & \text{если out (Callee)} = \emptyset \\ 0, & \text{иначе} \end{cases}$$

где B_1', B_2' - постоянные масштабирующие коэффициенты.

Слагаемое B_2 выражает тот факт, что приоритет при встраивании имеют листовые функции графа вызовов, т.е. узлы, у которых нет исходящих дуг. Это соображение подкрепляется тем, что такие функции, как правило, малы и выполняют какое-либо элементарное действие, а следовательно, являются хорошими кандидатами для встраивания.

[4]Заметим, что если нигде в программе не используется адрес функции Callee и in(Callee) $= 1$, то встраивание вызова заведомо не увеличивает объем кода, поскольку после встраивания такая функция может быть удалена. В этой ситуации мы полагаем $C_1 = 0$.

Приступим теперь к описанию алгоритма, позволяющего получать приемлемое решение для данной задачи. Предположим, что у нас имеются следующие примитивы:

- Предикат $IsInlineable : E \to \{False, True\}$ возвращающий $True$, если данное ребро графа вызовов может быть встроено. Этот предикат должен проверять, во-первых, доступны ли тела вызывающей и вызываемой процедур (т.е. не являются ли они внешними по отношению к программе); во-вторых, ребро должно соответствовать обычному вызову (а не передаче управления в результате возникновения исключения); в-третьих, имеется еще ряд специфических для языка условий, которые надо учитывать (для C, например, вызываемая функция должна иметь фиксированное число параметров). Здесь же необходимо проверять глубину рекурсии встраиваемого ребра:

```
function IsInlineable (e: CallGraphEdge): boolean
begin
  if not (IsNormalCall (e)) return False;
  if (IsExternalFunction (e.Caller)
      or IsExternalFunction (e.Callee))
    return False;
   ...
  if (e.RecCount > MaximumInlineRecursionDepth)
    return False;
  return True;
end
```

- Процедура $InlineEdge$, выполняющая встраивание ребра в графе вызовов. Эта процедура копирует тело вызываемой функции, объединяет его с телом вызывающей, создает необходимые временные переменные и присваивания для имитации передачи параметров и возвращаемого значения. В случае нашего внутреннего представления необходимо интегрировать граф потока управления вызываемой функции в граф потока управления вызывающей функции и добавить список операторов вызываемой функции в список операторов вызывающей. Кроме того, требуется заменить во встроенном коде операторы возврата на присваивания временной переменной, представляющей возвращаемое значение, и заменить оператор вызова на использование этого значения (если оператор вызова представляет собой конструкцию верхнего уровня, т.е. находится не в выражении, то его надо просто удалить). Детали этой процедуры опускаются за их громоздкостью и рутинностью.

- Процедура $UpdateCallGraph$ добавляет ребра, соответствующие ребрам, выходящим из узла вызываемой функции, к ребрам, выходящим из узла вызывающей функции. При этом аннотации ребер меняются так, как было описано выше:

```
function UpdateCallGraph (e: CallGraphEdge): set of
CallGraphEdge
  begin
    addedEdges: set of CallGraphEdge;
    for e' in out (e.Callee) do
    begin
      e'' = e';
      e''.InLoop = e'.InLoop or e.InLoop;
      if (e'.Callee = e.Callee) then
        e''.RecCount = e.RecCount + 1;
      out (e.Caller) = out (e.Caller) ∪ e'';
      addedEdges = addedEdges ∪ e'';
    end
    return addedEdges;
  end
```

Тогда алгоритм можно записать следующим образом:

Алгоритм. (*Автоматическое встраивание вызовов*).

```
procedure ProcedureIntegration (CG:CallGraph,
CodeSizeGrowthFactor: integer)
begin
  worklist: list of CallGraphEdge;
  totalCodeSize, initialCodeSize, maximumCodeSize: real;
  benefit, cost: map: CallGraphEdge → integer;

  (* Вычисляем начальный размер кода программы*)
  for n in CG.N do
  begin
    initialCodeSize = initialCodeSize + CodeSize (n);
  end
  maximumCodeSize = initialCodeSize *
      (1 + 0.01 * CodeSizeGrowthFactor);
  totalCodeSize = initialCodeSize;
  (* Инициализация рабочего списка *)
  for e in CG.E do
  begin
    if (IsInlineable (e))
    begin
      worklist = worklist ∪ e;
      (* Здесь также вычисляем benefit(e) и
         cost(e) по соответствующим формулам.*)
    end
  end
  (* Основной цикл *)
  while (not worklist.empty () and
    totalCodeSize < maximumCodeSize) do
```

```
       begin
         maxRatio := max (Benefit (e') / Cost (e'): e' ∈
worklist);
         e:= (e' ∈ worklist: Benefit (e') / Cost (e') =
maxRatio);
         worklist.remove (e);
         InlineEdge (e);
         addedEdges = UpdateCallGraph (e);
         (* Добавляем новые ребра к рабочему списку *)
         for e' in addedEdges do
         begin
            if (IsInlineable (e')) worklist = worklist ∪ e';
         end
         (* Необходимо также обновить benefit и
           cost для ребер, входящих в e.Caller *)
         for e' in in(e.Caller) do
         begin
         (* Перевычисляем benefit(e') и cost(e') с
           учетом изменений атрибутов e.Caller *)
         end
       end
     end
```

4.2.6 Некоторые замечания к алгоритму встраивания

Приведем некоторые комментарии к алгоритму встраивания вызовов. Прежде всего, заметим, что используемые функции оценки стоимости и выгоды встраивания ребра допускают широкую вариацию, касающуюся как выбора масштабирующих коэффициентов, так и введения дополнительных слагаемых, позволяющих учитывать другие факторы. Для функции выгоды одним из таких факторов могли бы стать, например, предпочтения пользователя, который может указать какие-либо функции как особенно желательные для встраивания. Однако, на наш взгляд, выбранные варианты формул представляют разумный компромисс между сложностью оценок и их подробностью.

Необходимо также заметить, что встраивание “портит” код вызывающей функции, добавляя в нее дополнительные переменные для параметров и возвращаемого значения. Очень часто код можно было бы значительно улучшить, например, в случае, когда фактический параметр является константой. С другой стороны, просто заменить все вхождения формального параметра на фактический во встроенном коде было бы некорректно, поскольку в этом коде может присутствовать операция взятия адреса формального параметра. Кроме того, такая замена не позволила бы выполнить возможные вычисления константных выражений. В качестве решения этой проблемы после работы алгоритма встраивания выполняется внутрипроцедурное распространение констант и копий (constant/copy propagation, CP) - оно позволяет “сгладить” результаты встраивания, удаляя ненужные переменные и присваивания и производя константные вычисления. Проиллюстрируем это на примере следующей программы:

```
     int f (int a, int b)
```

```
{
  return a + b;
}
int main()
{
  int c,d;
  ...
  d = f (c, 1);
  ...
}
```

В процессе встраивания функция **main** преобразуется так:

```
int main()                      int main()
{                               {
  int c, d;                       int c, d;
  int tmp0, tmp1, tmp2;           ...
  ...                             d = c + 1;
  tmp0 = 1;                       ...
  tmp1 = c;                     }
  tmp2 = tmp0 + tmp1;
  d = tmp2;
  ...
}
```

Слева показан код после встраивания вызова f (c,1), справа - после распространения констант и копирующих присваиваний. Очевидно, окончательный вариант не содержит никакого избыточного кода, связанного с поддержкой встраивания вызова.

4.3 Межпроцедурное распространение констант

Межпроцедурное распространение констант (interprocedural constant propagation, ICP) находит процедуры и формальные параметры в них, обладающие тем свойством, что формальному параметру соответствует одно и то же значение фактических параметров во всех точках вызова. Такая процедура может быть оптимизирована с учетом фиксированного значения формального параметра: внутрипроцедурное распространение констант с вычислением константных выражений и последующим устранением ненужного кода (dead code elimination, DCE) позволят получить “специализированную” данным значением версию процедуры. Необходимым является предположение о том, что доступны все точки вызова процедуры.

Задача межпроцедурного распространения констант является прямой задачей анализа потока данных на графе вызовов процедур, т.е. информация о константных значениях фактических параметров распространяется по дугам графа в направлении от вызывающей процедуры к вызываемой. Вклад вызывающей процедуры в вычисление значения фактического параметра вызываемой процедуры отражается с помощью так называемой “функции перехода”, которая может быть определена следующим образом:

Определение. Пусть $CG = (N, E, beg, end)$ - граф вызовов, $e = (p, q) \in E$ - ребро графа, соответствующее обычному вызову, $formals(p)$ - список формальных параметров функции p, $actuals(e)$ - список фактических параметров вызова, соответствующего ребру e, $formals(p)[i]$, $actuals(e)[i]$ - i-ый формальный и фактические параметры процедуры p (заметим, что фактический параметр является произвольным выражением в промежуточном представлении). *Функцией перехода через ребро* e называется отображение $J_e : formals(q) \to actuals(e)$ по правилу

$$J_e(formals(q)[i]) = actuals(e)[i].^5$$

Определение. *Функцией поддержки перехода* называется отображение $Jsupport_e : formals(q) \to \{$множество идентификаторов переменных$\}$ по правилу

$$Jsupport_e(formals(q)[i]) = \{id : id\text{−идентификатор переменной, входящей в } actuals(e)[i]\}.$$

Функция $Jsupport_e$, тем самым, определяет множество переменных, необходимых для вычисления фактического параметра. Поскольку значение выражения, являющегося результатом функции перехода, в процессе работы алгоритма должно вычисляться на основе имеющейся информации о значениях переменных, нужно ограничить допустимые значения функции перехода. Мы не собираемся во время работы межпроцедурного алгоритма выполнять анализ значений переменных внутри процедуры, поэтому приходится считать допустимыми только те значения $J_e(formals(q)[i])$, для которых

$$Jsupport_e(formals(q)[i]) \subseteq \{id : id\text{−идентификатор формального параметра } p\}.$$

Таким образом, функция перехода отражает лишь те значения фактических параметров, которые могут быть вычислены с помощью констант и формальных параметров вызывающей функции; в остальных случаях мы вынуждены предполагать,что фактический параметр не является константой и, следовательно, не может быть константой и формальный параметр вызываемой функции.

В работе [7] приведены результаты исследований эффективности алгоритма межпроцедурного распространения констант в зависимости от выбора класса допустимых значений функции перехода. В частности, рассматриваются следующие расширяющиеся классы допустимых значений:

- $actuals(e)[i]$ может быть только константой.
- $actuals(e)[i]$ может быть константой или формальным параметром вызывающей процедуры, если этот параметр не изменяется за время от начала процедуры до точки вызова вызываемой процедуры.

[5] Необходимо заметить, что в данном определении с целью упрощения изложения мы игнорируем возможность существования функций с переменным числом параметров и различных соглашений о вызовах. С практической точки зрения, мы не можем применять специализацию константным значением формального параметра к функции с переменным числом аргументов. Что же касается соглашений о вызовах, то для соглашения **cdecl**, принятого в языке С, необходимо в определении инвертировать список фактических параметров.

- $actuals(e)[i]$ может быть произвольным полиномом от формальных параметров вызывающей процедуры, если эти параметры не изменяются за время от начала процедуры до точки вызова вызываемой процедуры.

Исследования авторов работы [7] (проведенные на большом числе программ на FORTRAN) показывают, что в случае выбора второго класса допустимых значений функции перехода обнаруживается много больше межпроцедурных констант, чем в случае выбора первого. Третий же класс, несмотря на значительно более сложное вычисление фактического параметра, практически не дает преимуществ перед вторым. Руководствуясь этими соображениями, а также тем, что для реализации третьего варианта требуется выполнение константных вычислений, нами был реализован алгоритм, допускающий в качестве возможных значений фактических параметров константы и формальные параметры вызывающей процедуры.

Полурешетка, используемая в алгоритме, является стандартной полурешеткой высоты три, применяемой в задачах распространения констант. Три уровня включают в себя: минимальное значение $\perp$ (имеющее смысл "переменная не является константой"), уровень, на котором находятся все константы различных типов (целые, вещественные и т.д.), и максимальное значение $\top$, выражающее факт отстутствия информации о значении данной переменной. Операция *Meet* определяется так:

$$Meet(x,y) = \begin{cases} \perp, x = \perp \vee y = \perp \\ x, y = \top \\ x, x = y = const \\ \perp, & \text{иначе} \end{cases}$$

и является коммутативной: $Meet(x, y) = Meet(y, x)$.

В такой постановке задача может быть решена с помощью стандартного алгоритма анализа потока данных на графе вызовов (см, например, [1]). Заметим, что условие "формальный параметр процедуры не изменяется за время от начала процедуры до точки вызова" может быть проверено с помощью цепочек "использование-определения": условие справедливо, если ни одно из определений формального параметра в теле вызывающей процедуры не достигает точки вызова, где он используется в качестве фактического.

ICP находит формальные параметры, которые имеют одно и то же значение во всех точках вызова процедуры. Эта информация может быть использована для "специализации" процедуры следующим образом: в начало процедуры добавляется присваивание константного значения формальному параметру и выполняется внутрипроцедурное распространение констант с последующим удалением ненужного кода. Кроме того, константные формальные параметры удаляются из определения процедуры, а во всех точках вызова удаляются соответствующие им фактические параметры (это корректно, поскольку согласно определению функции перехода данные фактические параметры могут быть только выражениями вполне определенного вида и, в частности, не могут иметь побочных эффектов).

Проиллюстрируем действие межпроцедурного распространения констант на примере:

```
long sum;
int p (int n)
{
  return n*n - 3*n + 1;
}
void calc (int n)
{
  int i;
  for (i = 0; i < p(n); i++)

    sum += i & 1;

}
int main ()
{
  calc (15000);
  printf ("%d ",sum);
  return 0;
}
```

```
long sum;
int p ()
{
  return 224955001;
}
void calc ()
{
  int i;
  for (i = 0; i < p(15000); i++)

    sum += i & 1;

}
int main ()
{
  calc (15000);
  printf ("%d ",sum);
  return 0;
}
```

Заметим, что программу, конечно, можно упростить еще больше, учитывая, что вызов p () всегда возвращает одно и то же значение.

4.4 Межпроцедурный анализ потока данных

4.4.1 Вычисление множеств Mod и Ref

Межпроцедурный анализ потока данных (interprocedural data-flow analysis, IDFA) позволяет определить эффект вызова процедуры с точки зрения влияния на данные, представляющие контекст ее выполнения, который для языка C составляют глобальные переменные и локальные данные вызывающей функции, переданные в качестве параметров. В частности, межпроцедурный анализ позволяет установить, является ли функция "чистой" функцией (в смысле терминологии, используемой в функциональных языках программирования), т.е. не использует никаких данных, кроме параметров, и в результате вызова лишь возвращает значение, не обладая никакими побочными эффектами в виде модификации глобальных данных и вызовов внешних функций.

Наша реализация межпроцедурного анализа потока данных основана на подходе, описываемом в [2] и многих других источниках. Результатами анализа являются множества Mod и Ref, определяющие суммарный эффект вызова процедуры в терминах переменных, которые могут быть изменены в результате вызова (Mod), и переменных, значения которых могут быть использованы в результате вызова (Ref). В отличие от решения, предлагаемого в [2], мы связываем эту информацию не только с узлами, но и с ребрами графа вызовов, что увеличивает точность решения:

```
int inc (int p)
{
  return p++;
```

```
}
int main ()
{
  int a,b,i;
  i = inc(a); /* (1) */
  ...
  i = inc(b); /* (2) */
  ...
}
```

В данном примере Ref (inc) = {a,b}, Ref ((1)) = {a}, Ref ((2)) = {b}. Знание о том, что переменная b не используется в результате вызова (1) было бы утрачено, если бы множество Ref ассоциировались только с узлами графа вызовов.

Следует сразу заметить, что наша текущая реализация межпроцедурного анализа потока данных существенно ограничена с точки зрения подхода к анализу альтернативных имен (синонимов), который необходим для корректного вычисления множеств Mod и Ref. Именно, предполагается, что значение указателя может быть синонимом для любой переменной в программе; излишняя консервативность такого предположения снижает практическую полезность получаемых результатов. Для построения более точного решения необходимо использование межпроцедурного анализа синонимов. Возможные подходы к этой проблеме подробно рассматриваются, например, в работе [14].

Задача межпроцедурного анализа потока данных является примером обратной задачи на графе вызовов. Для данной задачи мы используем трехуровневую полурешетку, сходную с полурешеткой для задачи межпроцедурного распространения констант. Минимальное значение $\perp$ соответствует отсутствию содержательной информации о влиянии вызова данной процедуры и предположению о том, что процедура может изменять/использовать любую переменную; на промежуточном уровне находятся множества идентификаторов переменных; максимальное значение $\top$ обозначает, что влияние данной процедуры еще не известно. Операция на полурешетке вводится следующим образом:

$s1, s2$ - элементы полурешетки:

$$Join(s1, s2) = \begin{cases} \perp, s1 = \perp \vee s2 = \perp \\ s1, s2 = \top \\ s2, s1 = \top \\ s1 \cup s2, & \text{иначе} \end{cases}$$

Решение задачи состоит из двух этапов:

- Вычисление для каждого узла графа вызовов множеств LMod и LRef, представляющих собой множества переменных, которые могут быть изменены и использованы процедурой без учета содержащихся в ней вызовов процедур. Предполагается, что если в теле процедуры встречается присваивание, в левой части которого находится выражение вида *(expr) (где *expr* - допустимое адресное выражение), то LMod $= \perp$; если выражение вида *(expr) встречается в правой части операторов присваивания и других операторах, то LRef $= \perp$. Заметим, что вследствие применяемого нами загрубления решения, в множество LRef имеет смысл включать только информацию об используемых формальных параметрах и глобальных переменных, а в множестве LMod - только изменяемые глобальные переменные (поскольку единственным способом изменить значение локальной переменной вызывающей функции в C является передача ее адреса в вызываемую, а указатели мы в настоящей реализации не анализируем).

- Вычисление множеств Mod и Ref для узлов и дуг графа вызовов при помощи следующих уравнений $CG = (N, E, beg, end), n \in N, e = (p, q) \in E$ - ребро, соотвествующее обычному вызову:

  $Ref(n) = LRef(n) \bigcup_{e \in out(n)} Ref(e)$

  $Ref(e) = Map(Ref(q))$

  Для множества Mod первое уравнение выглядит так же, а второе упрощается вследствие рассмотрения только модификации глобальных переменных:

  $Mod(e) = Mod(q)$

  Функция Map определяется следующими правилами. Пусть s - элемент полурешетки, $actuals(e)$ - фактические параметры вызова, соответствующие ребру e. Определим функцию $Uses_e$, действующую на множестве формальных параметров функции q по правилу

  $$Uses_e(formals(q)[i]) = \begin{cases} \perp, & \text{если } actuals(e)[i] \\ & \text{содержит подвыражение вида } *(expr_1) \\ Jsupport_e(formals(q)[i]), & \text{иначе} \end{cases}$$

  а также вспомогательные функции

  $Formals_q(s) = \{v : v \in s \text{ и } v \text{ - идент. формального параметра } q\}$

  и $Num_q(v) = \{\text{номер } v \text{ в списке формальных параметров } q\}$

  (первая функция определена на множестве идентификаторов всех переменных, а вторая - на подмножестве идентификаторов переменных, являющихся формальными параметрами q). Положим также

  $MapFormals_e(s) = Join_{i:i=Num_q(v), v \in Formals_q(s)}(Uses_e(formals(q)[i]))$.

  Тогда функцию Map можно записать так:

  $$Map(s) = \begin{cases} \perp, s = \perp \\ Join(s \setminus Formals_q(s), MapFormals_e(s)), & \text{иначе} \end{cases}$$

  Вычисление множеств Mod и Ref по этим уравнениям выполняется с помощью итеративного алгоритма, в нашей реализации используется алгоритм, приведенный в [1]. В начале мы полагаем $\forall n \in N : Mod(n) = LMod(n), Ref(n) = LRef(n)$, а для внешних функций

$Mod(n) = Ref(n) = \perp$. Для улучшения сходимости алгоритма вершины графа вызовов на очередной итерации упорядочиваются в соответствии с нумерацией Rev_N, определяемой так: $Rev_N(n) = |N| - N(n)$, где N - обратная нумерация графа вызовов. Это означает, что вершина будет рассматриваться после ее потомков в графе вызовов и обеспечивает минимальное число итераций, а также сходимость за одну итерацию в случае отсутствия циклов в графе (т.е. рекурсии).

4.4.2 Использование результатов анализа для улучшения оптимизации общих подвыражений

Информация, представляемая множествами Mod и Ref, может быть использована для улучшения внутрипроцедурных оптимизаций. Мы рассмотрим два применения этой информации, реализованные в оптимизаторе в рамках просмотра внутрипроцедурного сокращения общих подвыражений (common subexpression elimination, CSE).

- Увеличение числа доступных выражений за счет менее консервативного рассмотрения эффекта вызовов функций. Задача определения доступных выражений является прямой задачей анализа на графе потока управления процедуры (см [1]). Вклад узла графа, представляющего оператор, состоит их множеств выражений Gen и Kill: первое представляет собой выражения, создаваемые этим оператором, второе - выражения, которые перестают быть доступными в результате выполнения этого оператора. В множество Kill входят выражения, содержащие переменные, значения которых могут изменяться данным оператором. Если оператор является вызовом функции, то в качестве множества Kill мы используем множество тех выражений, которые содержат переменные из множества Mod, соответствующего данному вызову.

- Экономия вызовов. Заметим, что сам вызов функции является выражением, которое также может рассматриваться как субъект для CSE. Зная множество Ref переменных, значения которых могут использоваться в результате данного вызова, мы определяем для произвольного узла графа, остается ли вызов доступным выражением после этого узла (если же для вызова множество Ref $= \perp$, то данный вызов не входит в число доступных выражений ни в одном узле графа потока управления процедуры). Кроме того, устранить повторный вызов функции можно только в том случае, если он не приводит к побочным эффектам (в виде вызовов внешних функций). В нашем случае учет побочных эффектов получается автоматически, поскольку для вызова с побочным эффектом Ref = Mod $= \perp$.

Использование межпроцедурного анализа для улучшения CSE показывает следующий пример:

```
int sum (int a, int b)
{
  return a + b;
}
```

Функция main до и после сокращения общих подвыражений:

```
int main ()                      int main ()
{                                {
int a, b, c, d;                  int a, b, c, d, tmp0;
  ...                              ...
  a = a + b + sum (c,d);           tmp0 = b + sum (c,d);
  c = d + b + sum (c,d);           a = a + tmp0;
  ...                              c = d + tmp0;
}                                  ...
                                 }
```

Знание о том, что для обоих вызовов функции sum в main Mod $= \emptyset$, Ref $= \{c,d\}$ помогает выделить общее подвыражение b + sum (c,d), сэкономив, в частности, один вызов sum.

4.5 Порядок применения оптимизирующих преобразований

Проблема порядка, в котором проводятся оптимизирующие преобразования, является одной из самых важных в практике оптимизации и имеет решающее значение для эффективности объектного кода. На нее влияют как взаимосвязи между самими преобразованиями, так и требования к качеству объектного кода. Рассмотрим некоторые из этих факторов:

- Без сомнения, самым эффективным межпроцедурным оптимизирующим преобразованием является автоматическое встраивание вызовов. В то же время, оно, как правило, является и наиболее "дорогим", поскольку может сильно увеличить размер объектного кода. Поэтому в определенных ситуациях встраивание неприменимо.
- В тех ситуациях, когда встраивание неприменимо, экономия вызовов может помочь очистить код от лишних вызовов функций. Заметим, что последовательное применение встраивания и сокращения общих подвыражений может дать худший результат, чем применение экономии вызовов. Это связано с тем, что возвращаемое встроенным вызовом значение записывается во временную переменную и последующий просмотр сокращения общих подвыражений может не "распознать" как одинаковые два выражения, содержащие встроенные вызовы.
- Между оптимизирующими преобразованиями наиболее очевидны следующие отношения:
  - Автоматическое встраивание требует последующего проведения распространения констант и копирующих присваиваний, а также удаления ненужного кода внутри вызывающей процедуры.
  - То же самое можно сказать и о межпроцедурном распространении констант: внутри процедуры с константным формальным параметром необходимо выполнить упомянутые в предыдущем пункте внутрипроцедурные преобразования.

– Внутрипроцедурное распространение констант может обнаружить, что фактические параметры вызова являются константами, расширяя тем самым возможности для межпроцедурного распространения констант. Кроме того, такие вызовы - хорошие кандидаты для встраивания.

– Экономия вызовов фактически является внутрипроцедурным преобразованием и проводится в рамках сокращения общих подвыражений, однако требует результатов межпроцедурного анализа потока данных.

– Встраивание вызовов может уменьшить число вызовов данной процедуры. В результате может оказаться, что в изменившемся графе вызовов к процедуре применимо межпроцедурное распространение констант.

С учетом сказанного выше, в оптимизаторе используется следующий порядок анализирующих просмотров и преобразований:

1. Внутрипроцедурное распространение констант, копий, удаление ненужного кода.

2. Построение графа вызовов

3. Межпроцедурный анализ потока данных

4. Экономия вызовов, т.е. сокращение общих подвыражений с учетом результатов межпроцедурного анализа.

5. Автоматическое встраивание вызовов.

6. Межпроцедурное распространение констант.

7. Внутрипроцедурное распространение констант, копий, удаление ненужного кода.

Заметим, что этот порядок, конечно, не является жестко определенным. Пользователю предоставляется возможность проводить оптимизации в любом порядке, чередуя внутрипроцедурные и межпроцедурные просмотры.

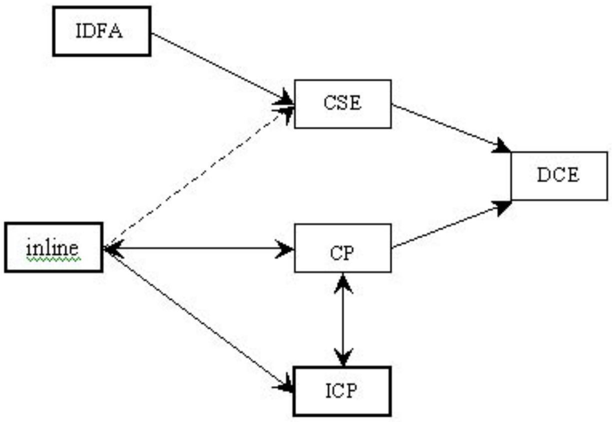

Рис. 4.1: Отношения между различными внтури- и межпроцедурными просмотрами. Пунктирные стрелки означают, что одно преобразование может препятствовать последующему выполнению другого.

Глава 5

Экспериментальные результаты

Для экспериментов использовались несколько тестов из набора SPECint 2000, а также программа ranking (реализация алгоритма Symbol Ranking Text Encoding, автор Д.Ломов). В приведенной ниже таблице даны результаты применения ICP и CSE с межпроцедурным анализом потока данных.

Тест	ICP	CSE	CSE + IDFA
ranking	2	0	1
gzip	2	1	1
bzip	3	30	31
parser	5	33	38

Цифры в соответствующих клетках означают количество межпроцедурных констант и общих подвыражений, найденных в результате анализа программы. Следует заметить, что внутрипроцедурные просмотры среды PROMISE (а именно, CSE и DCE), использовавшиеся для эксперимента, не содержат анализа указателей, и поэтому вынужденно делают консервативные предположения, значительно снижающие точность анализа (все тестовые программы очень интенсивно работают с указателями). Кроме того, ценность найденных констант и общих подвыражений в различных тестах не одинакова: если, например, в программе ranking обе найденные константы были параметрами процедур инициализации и обработки ошибок, то в parser одна из констант использовалась в качестве параметра в рекурсивной функции (выполняющей обход графа в глубину), к тому же рекурсивный вызов находился в цикле. В программе bzip процедура нижнего уровня, осуществляющая чтение в буфер из файла, вызывается в нескольких местах с одним и тем же фактическим параметром (размер буфера); аналогичная ситуация наблюдается в программе gzip с процедурой копирования блока памяти. В обоих случаях “специализация” процедур константным значением формального параметра устраняет несколько лишних проверок (в частности, на отрицательное и нулевое значение параметра) и позволяет надеяться на определенный выигрыш в производительности объектной программы.

Лишь в одном случае (ranking) удалось наблюдать действие экономии вызовов, причем был устранен лишний вызов функции, возвращающий текущую позицию в буфере, т.е. метода доступа в терминологии объектно-ориентированного программирования (этот вызов, конечно, можно было бы просто встроить). Зато в программе parser благодаря использованию межпроцедурного анализа потока данных удалось сэкономить несколько довольно сложных подвыражений, включающих несколько арифметических операций, в т.ч. умножение.

Приведем также результаты экспериментов со встраиванием вызовов. В процессе этих экспериментов задавались различные значения для ограничения на рост размера кода программы. После встраивания для всех функций выполнялось внутрипроцедурное распространение констант и удаление ненужного кода (впрочем, ограничения этих просмотров, упомянутые выше, сыграли свою негативную роль и здесь).

ranking:

Число встроенных вызовов	Размер исполняемого файла, Б	Время выполнения
44	24310	1 м 51.3 с
102	28932	1 м 49.7 с
214	33465	1 м 50 с

gzip:

Число встроенных вызовов	Размер исполняемого файла, Б	Время выполнения
146	92117	0 м 2.76 с
373	146957	0 м 2.73 с
582	220285	0 м 2.75 с

Приведем также результаты, полученные компиляцией обеих программ lcc без применения оптимизатора:

Тест	Размер исполняемого файла, Б	Время выполнения
ranking	23775	1 м 54 с
gzip	75535	0 м 2.80 с

Замечание. Обе программы запускались с одним и тем же входным файлом объемом около 420 Кб. Измерения проводились в ОС Linux на слабозагруженной системе (Intel Pentium III, 550 Mhz, ОЗУ 128 Мб), работающей в однопользовательском режиме, при помощи команды time. Время, данное в таблицах, представляет собой результат усреднения трех измерений.

Из этих таблиц можно сделать заключение относительно пользы агрессивного встраивания вызовов. В обоих случаях (ranking, gzip) прослеживается общая тенденция: встраивание дает выигрыш в производительности лишь до определенного предела, при слишком большом увеличении размера кода положительный эффект пропадает. Причины этого обсуждались выше.

Глава 6

Заключение

В рамках дипломной работы был реализован ряд межпроцедурных оптимизирующих просмотров для компилятора языка C, исследована их эффективность и отношения между ними. Автоматическое встраивание вызовов оказалось наиболее действенным оптимизирующим преобразованием. На одном из тестов наблюдались позитивные эффекты в результате сочетания межпроцедурного анализа потока данных с последующей экономией общих подвыражений. Эксперименты показывают, что межпроцедурное распространение констант на некоторых тестах также дает положительный эффект, поскольку найденные константные параметры могут использоваться в часто вызываемых функциях, и “специализация” таких функций позволяет получить выигрыш в производительности.

В качестве направлений дальнейшего развития системы можно назвать:

- Дополнение внутрипроцедурных оптимизаций (CP, CSE, DCE) использованием результатов анализа указателей. В текущей реализации внутрипроцедурные просмотры используют консервативные предположения о значениях указателей, что снижает их эффективность. Данный вопрос подробно рассматривается в работе [16].

- Реализацию межпроцедурного анализа указателей и модификацию алгоритма межпроцедурного анализа потока данных для с целью получения более точного решения, учитывающего возможные значения указателей. Это позволит намного повысить применимость межпроцедурного анализа к внутрипроцедурным оптимизациям, а частности, к CSE и экономии вызовов.

- Реализацию распространения констант по дугам графа вызовов в обратном направлении, т.е. “специализацию” точек вызова константными значениями, возвращаемыми функциями. Такое преобразование, в частности, приводило бы и к экономии вызовов, поскольку вызов, возвращающий константное значение, может быть устранен, если он не имеет побочных эффектов.

- Реализацию “чистки” циклов и использование межпроцедурного анализа потока данных для вынесения из циклов вызовов с параметрами, инвариантными относительно этого цикла, и не имеющих побочных эффектов (еще один вариант экономии вызовов).

- Исследование эффективности вариантов функции выгоды встраивания вызова в условиях жестких ограничений на рост объема кода. В настоящей реализации, значение функции выгоды увеличивается на константу, если вызов выполняется в цикле, при этом не учитывается уровень вложенности цикла. Более совершенная функция стоимости может включать составляющие, основанные на значении уровня вложенности. Для вычисления последнего может использоваться анализ участков повторяемости графа потока управления, описанный в [3].

Следует заметить, что текущая реализация оптимизатора обладает определенными недостатками и с точки зрения пользователя. Главным из них является необходимость полной перекомпиляции программы после внесения изменений в один из модулей. В качестве альтернативного подхода использованию компоновщика промежуточного представления можно было бы сохранять вместе с представлением программы информацию, необходимую для построения графа вызовов и проведения межпроцедурного анализа (например, результаты внутрипроцедурного анализа потока данных). При проведении преобразований программы перекомпилируются только модули, в которых были изменены какие-либо функции или глобальные данные, затем программа перекомпонуется традиционным компоновщиком объектных файлов. Такое решение значительно усложнило бы реализацию, однако позволило бы частично сочетать преимущества раздельной компиляции с возможностью межпроцедурной/межмодульной оптимизации программы.

Литература

[1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[3] В.Н.Касьянов. *Оптимизирующие преобразования программ.* Наука, 1988.

[4] C. Fraser, D. Hanson. *A Retargetable C Compiler: Design and Implementation*, Benjamin-Cummings, Redwood City, CA, 1995

[5] M. Arnold, S. Fink, V. Sarkar, P. Sweeney. *A comparative Study of Static and Profile-Based Heuristics for Inlining.* ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000.

[6] Wen-mei W. Hwu, Pohua P. Chang. *Inline Function Expansion for Compiling C Programs.* ACM, 1989.

[7] D. Grove, L. Torczon. *Interprocedural Constant Propagation: A Study of Jump Function Implementation.* ACM SIGPLAN Notices, 1993.

[8] M. Fernandez. *Simple and Effective Link-time Optimization of Modula-3 Programs.* ACM SIGPLAN Notices, 1995, Voi. 30, No. 6, pp 103-115.

[9] M. Hall, J. Mellor-Crummey, A. Carle, R. Rodriguez *FIAT: A Framework for Interprocedural Analysis and Transformation.* CRPC-TR95522-S, March, 1995.

[10] Д.Ю. Булычев, Е.Н. Вигдорчик, Д.С. Ломов, М.Н. Смирнов. *Настраиваемые средства генерации эффективного кода (предварительный отчет).* Январь, 2001.

[11] J. Patterson. *Accurate Static Branch Prediction by Value Range Propagation.* ACM SIGPLAN Notices, 1995, Voi. 30, No. 6, pp 67-78.

[12] K. Cooper, M. Hall et al. *The ParaScope Parallel Programming Environment.* Proc. of the IEEE, Vol. 81, No. 2, 1993, pp.244-263

[13] The Stanford SUIF Compiler Group.

http://suif.stanford.edu

[14] J. Lu *Interprocedural Pointer Analysis for C.* PhD. Thesis, Rice University, Houston, Texas, 1998.

[15] B. Stroustrup. *The C++ Programming Language (3rd Edition).* Addison-Wesley Longman, Reading Mass, USA, 1997.

[16] Симановский А.А. *Анализ указателей в среде PROMISE.* Дипломная работа, Санкт-Петербург, 2002.