Оптимизации по размеру с преобразованием реализации кода

Ю. С. РАССОХИН
Московский государственный университет им. М. В. Ломоносова
e-mail: yuri.rassokhin@gmail.com

УДК 519.671

Ключевые слова: оптимизации по размеру, сигнатура функции, оценка эффективности оптимизации по размеру.

Аннотация

В статье рассматривается новый подход к построению алгоритмов оптимизации кода по размеру, допускающий глубокие преобразования кода оптимизатором. Основное внимание уделяется преобразованию сигнатур функций. Приводятся соответствующие алгоритмы оптимизации, оценки эффективности алгоритмов, примеры применения, а также адекватные и нерекомендуемые сценарии использования.

Abstract

Yu. S. Rassokhin, Code optimizations for size based on architecture transformations, Fundamentalnaya i prikladnaya matematika, vol. 14 (2008), no. 4, pp. 193—212.

The article reveals a new approach for code optimizations for size, which is based on allowing deep transformations of the source code when performing optimizations. Transformations of function signatures are described as a basis. The article contains algorithm descriptions including efficiency evaluations, examples, and appropriate use cases.

Введение

Развитие информационных технологий в современном мире приобрело замечательное свойство, которое можно назвать *двойственностью технологий*.

С одной стороны, развивающиеся технологии производства вычислительной техники приводят к совершенствованию техники: центральные и специализированные процессоры становятся всё более производительными, носители памяти всё более объёмными и т. д. К программному обеспечению для новых аппаратных компонент предъявляются новые требования. В качестве примеров можно привести требование генерирования компилятором максимально эффективного кода для таких процессорных архитектур, как

- архитектура с широким словом, или EPIC;
- симметричные мультипроцессорные системы, или SMP;
- многоядерные архитектуры процессора, такие как Cell.

Фундаментальная и прикладная математика, 2008, том 14, № 4, с. 193—212.

Каждая из приведённых архитектур диктует специфические требования к генерируемому компилятором коду. Некоторые из них появились сравнительно недавно, другие, существовавшие в прошлом, напротив, сегодня потеряли актуальность. Так, типичным примером устаревшего требования в области «больших» вычислительных систем может служить требование генерирования наиболее компактного кода.

С другой стороны, появляются новые и развиваются существующие области применения информационных технологий, отличных от «больших» вычислительных систем и, зачастую, также от встроенных систем. Более того, развитие этих «малых» систем в последние годы всё более и более расширяется.

«Малые» системы включают в себя любые вычислительные устройства, для которых критично ограничение размера кода и менее критична эффективность кода. Это могут быть как *встроенные системы*, например *беспроводные сенсорные сети* (БСС), так и иные вычислительные устройства, к встроенным системам не относящиеся, например операционные системы на носителях EEPROM или системы BIOS, содержащие дополнительное программное обеспечение.

Примитивизм аппаратной части «малых» систем обусловлен такими факторами, как

- бизнес-модель системы, ограничивающая стоимость используемой аппаратуры;
- требования надёжности, тяготеющие к использованию «примитивной» аппаратуры;
- ограничения, обусловленные сценарием применения системы. Так, узлы БСС имеют жёсткие ограничения на физические размеры в силу специфики их применения согласно концепции smart dust.

В итоге на любом этапе развития информационных технологий в части «малых» систем используются аппаратные компоненты с техническими характеристиками, формально принадлежащими далёкому прошлому.

В качестве примера актуальности оптимизаций по размеру опишем проблематику бурно развивающейся сферы беспроводных сенсорных сетей.

На сегодняшний день узел БСС, исходя из маркетинговых факторов, располагает памятью EEPROM и RAM, в сумме не превышающей 128 килобайт. В то же время концепция БСС требует разработки достаточно сложного программного обеспечения для работы на узлах сети. Подразумевая наличие на узле операционной системы реального времени как само собой разумеющееся, перечислим необходимые сервисы узла БСС:

- сетевой стек, удовлетворяющий тому или иному стандарту;
- поддержка шифрованной передачи в сетевом стеке;
- алгоритмы оптимальной маршрутизации;
- динамическое перестроение маршрутизации при изменении топологии сети;
- алгоритмы экономного использования источника питания;
- гарантированное отсутствие потери данных;

- надёжная передача с уведомлением;
- динамический мониторинг топологии БСС;
- кластеризация сети на сегменты введением отношения «родитель—потомок» между узлами;
- распределённая обработка данных и поддержка распределённой базы данных на узлах;
- связь с внешними по отношению к БСС устройствами через интерфейсы USB или Ethernet;
- динамическая загрузка исполняемого кода в узлы БСС «по воздуху».

Набор перечисленных сервисов с очевидностью демонстрирует актуальность в области БСС таких традиционных требований к разработке программного обеспечения, как требование генерирования компилятором максимально компактного кода.

Таким образом, разбивая направления развития информационных технологий на «большие» и «малые» системы, мы видим, что традиционные требования к программному обеспечению, такие как компактность кода, ни в коей мере не исчезают со временем, но перемещаются из «больших» систем во всё новые подобласти «малых» систем.

Принятые соглашения

Известно, что задача оптимизации кода состоит из взаимно исключающих подзадач: оптимизации по размеру и оптимизации по эффективности. Принимая это во внимание, всюду в данной работе мы руководствуемся следующим правилом:

> к эффективности генерируемого компилятором кода не предъявляется никаких требований; другими словами, с целью уменьшения размера кода допускается генерирование сколь угодно «неэффективного» кода.

Заметим, что данное правило отнюдь не является порочным или неадекватно абстрактным, но находит отражение в практике. Например, сетевой стек беспроводной сенсорной сети на основе протокола ZigBee, подобный описанному выше, должен предоставить возможность обработки всего лишь нескольких единиц обращений к произвольному узлу БСС за одну секунду.

Однако задача построения оптимизаций кода по размеру, в свою очередь, распадается на две подзадачи, исключающие друг друга. Именно, оптимизация кода по размеру может стремиться либо использовать стек наиболее экономно, либо использовать стек без каких-либо ограничений. Заметим, что выбор одной из двух стратегий использования стека, как правило, играет жизненно важную роль при проектировании программного обеспечения для «малых» систем, в частности для БСС. Дело в том, что стек в «малых» системах ограничен

по размеру не менее чем RAM или EEPROM. В то же время переполнение стека ведёт к фатальным ошибкам, которые трудно устранить ввиду их принципиальной природы. Кроме того, зачастую на «малых» системах отсутствует аппаратное обнаружение переполнения стека, вследствие чего затруднительно обнаружить само возникновение переполнения стека.

Принимая во внимание как важность непереполнения стека, так и тот факт, что оптимизации по размеру зачастую противоречат политике экономии стека, в настоящей работе мы формулируем дополнительные соглашения об использовании стека:

1) в работе не накладывается никаких ограничений на использование стека в коде, генерируемом оптимизатором;
2) в то же время если конкретный алгоритм оптимизации кода генерирует код, для которого размер стека является узким местом, то проводится дополнительный анализ использования стека получаемым кодом.

Программно-аппаратная платформа

Следующая программно-аппаратная платформа используется и подразумевается всюду ниже в настоящей работе при тестировании, анализе и оценке описываемых алгоритмов.

- Фрагменты кода приводятся на стандарте ISO 9899:1999 языка программирования C, при необходимости дополненном расширениями GNU GCC.
- Используемые расширения GNU GCC соответствуют версии компилятора gcc 4.1.0.
- Данные о скомпилированном двоичном коде приводятся для сборки под целевую аппаратную платформу i386.
- Искусственное придуманные фрагменты кода используются для наглядного представления об описываемом алгоритме оптимизации.
- Для демонстрации практической эффективности алгоритма используются реальные фрагменты кода из исходных кодов компилятора gcc 4.1.0, сборка от 25 мая 2006 г.

Отметим, что описываемые алгоритмы, являясь высокоуровневыми, не зависят от целевой аппаратной платформы. В связи с этим использование платформы i386 не ограничивает общности изложения.

Оценки эффективности

Сформулируем способ построения оценок эффективности предлагаемых в настоящей работе алгоритмов оптимизации. Поскольку предлагаемые оптимизации являются высокоуровневыми, то не рассматривается никакое фиксированное низкоуровневое представление кода.

Все рассматриваемые примеры кода написаны на языке C. В связи с этим численная оценка, показывающая изменение размера оптимизируемого кода, приводится в количестве конструкций языка C, различающихся между исходным кодом и кодом, полученным в результате оптимизации.

Всюду далее в настоящей работе при построении оценок эффективности приняты следующие обозначения для операций либо, в соответствующем контексте, для стоимостей соответствующих операций:

- Push — сохранение регистра в стек;
- Pop — восстановление регистра из стека;
- Cmp — сравнение переменной или формального параметра с некоторой константой;
- Call — вызов функции без учёта формирования и восстановления стекового фрейма;
- Land — логическое AND над произвольными выражениями без учёта вычисления операндов;
- Mov — копирование значения переменной или формального параметра в регистр;
- Shift — сдвиг значения переменной или формального параметра на количество разрядов, заданное некоторой константой.

Перечисленные обозначения будем называть *базовыми операциями*.

Наконец, всюду далее будем считать, что все экземпляры базовых операций имеют одинаковую стоимость. Такое допущение верно при отсутствии иных оптимизаций, кроме той, для которой строится оценка эффективности.

Пример. Пусть оптимизация удаляет из кода $n-1$ вызов функций. Тем самым происходит уменьшение кода на $n-1$ операцию Push, Pop и Call. Поскольку все экземпляры операций сохранения фрейма и операций восстановления фрейма имеют одинаковые стоимости, код уменьшается на величину

$$(\text{Load} + \text{Push} + \text{Call})(n-1),$$

значение которой в байтах определяется выбором архитектуры целевого процессора, наличием прочих оптимизаций и т. д.

Как видно из приведённого примера, описанное построение оценки эффективности позволяет определить эффективность алгоритмов оптимизации без привязки к конкретной программно-аппаратной платформе.

Всюду в настоящей работе примем следующее определение.

Определение 1. *Оценкой эффективности* алгоритма оптимизации для кода F называется количество N языковых конструкций C, различающихся между исходным кодом F и кодом, полученным в результате применения рассматриваемого алгоритма, причём N выражено через базовые операции.

Предлагаемая парадигма

Приводимые ниже построения оперируют базовым понятием «реализация кода». Данное понятие призвано обозначить разницу между чисто семантическим разделением программы на логически завершённые части и реализацией частей программы путём задания определённых интерфейсов между логическими частями программы с использованием тех или иных конструкций заданного языка программирования.

Пример. Программа сортировки заданного массива логически разделяется на следующие логические части:

- инициализация: выделение памяти под массив данных;
- ввод данных: чтение данных из файла;
- сортировка: запуск основного алгоритма для прочтенного массива данных;
- вывод данных: вывод отсортированного массива в файл;
- завершение работы: вывод заключительных сообщений, удаление временных динамических объектов;
- генератор сообщений: вывод сообщения в стандартном виде, содержащего заданный текст.

Заметим, что описание логического строения программы, приведённое выше, само по себе не требует задания интерфейсов между частями, которые могут выглядеть следующим образом:

- инициализация: `void initialization(void);`
- ввод данных: `double *input(FILE *);`
- сортировка: `double *sort(double *);`
- вывод данных: `void output_results(double*);`
- завершение работы: `void finalize(void);`
- генератор сообщений: `void message(char *, ...);`

На понятийном уровне первое описание структуры программы является чисто семантическим, а второе фиксирует конкретный способ выражения семантики каждой части. Так, определяется однозначная реализация ввода данных в виде одной функции, которая получает на входе файл, из которого выполняется чтение данных.

Понятие реализации кода формализует разницу между приведёнными двумя способами описания структуры программы. Соответствующее определение формулируется следующим образом.

Определение 2. *Реализацией кода* называется набор фиксированных значений всех свойств программы из числа перечисленных ниже:

1) набор функций, кроме, быть может, некоторого множества функций из кода программы;
2) набор переменных, кроме, быть может, некоторого множества переменных из кода программы;

3) сигнатуры функций;
4) типы значений, возвращаемых функциями;
5) значения, возвращаемые функциями;
6) типы переменных.

Теперь можно сказать, что первое описание в приведённом примере не фиксирует никакую реализацию кода, тогда как второе описание задаёт определённую реализацию кода. При этом заметим, что для любой программы в принципе возможен поиск альтернативных реализаций кода, возможно более оптимальных с точки зрения компактности кода.

Тем самым намечен подход, являющийся основой для предлагаемых в настоящей работе оптимизаций:

> рассмотрение реализации кода в качестве инварианта для оптимизатора сужает возможности для оптимизации кода. Напротив, разрешение оптимизатору произвольно изменять реализацию входного кода даёт возможность отыскать более оптимальную реализацию кода.

Пример. Приведём некоторые «классические» оптимизации, сохраняющие реализацию кода:

1) подстановка тела функции в вызов;
2) удаление неиспользуемого кода;
3) вычисление констант;
4) оптимизации циклов.

Все способы модификаций реализации кода разделяются на две большие группы.

Определение 3. Преобразование кода называется *трансформацией*, если это преобразование сохраняет семантику кода.

Определение 4. Преобразование кода называется *адаптацией*, если это преобразование изменяет семантику кода с целью получения дополнительных возможностей оптимизации этого кода.

Пример. Следующая адаптация кода является распространённым приёмом при рефакторинге программного проекта.

Пользовательские функции зачастую возвращают в качестве неуспешного результата числовое значение, как правило являющееся уникальным. Уникальность такого «кода ошибки» позволяет упростить отладку программы. Однако на практике уникальностью неуспешного результата часто можно пожертвовать ради уменьшения размера кода. Происходящая при этом замена уникальных кодов ошибочных результатов на единое «ошибочное» значение есть не что иное, как адаптация кода.

Дав представление о видах преобразований реализации кода, всюду далее в настоящей работе будем рассматривать только трансформации кода.

Наконец, сделаем несколько замечаний относительно общих критериев целесообразности применения трансформаций кода.

Трансформация, по сути являясь изменением интерфейсов между частями программы, может усложнять раздельную компиляцию Так, изменение сигнатуры функции, не квалифицированной как `static`, фактически исключает возможность раздельной компиляции. Один из выходов состоит в автоматической генерации компилятором заголовочного файла, содержащего объявления всех функций с изменёнными сигнатурами.

Таким образом, область применения трансформаций и адаптаций — разработка встроенных и прочих «малых» систем, в которых некоторое усложнение процесса разработки является приемлемой ценой за получение ещё более компактного кода.

Алгоритмы оптимизаций

Примеры

Прежде чем приступить к описанию оптимизаций по размеру, проиллюстрируем идеи предлагаемых оптимизаций, описав несколько фрагментов кода, которые часто встречаются на практике.

1. Ширина целочисленного типа, используемого в качестве логического, заведомо больше, чем это действительно нужно для хранения двух логических значений true и false.
2. Параметры, играющие роль логических флагов, представляются целочисленным типом, хотя требуемая семантика состоит только в различении двух разных значений. Важно отметить, что величины самих значений, принимаемых параметром, несущественны, поскольку параметр используется только как логический флаг.
3. Согласно C99, использование типа `enum` эквивалентно использованию `int`, хотя требуемая от `enum` ширина значительно меньше, чем `sizeof(int)`, уже на архитектуре с шириной типа `int`, равной 8 битам.
4. Программа содержит набор функций $f_1, f_2, \ldots, f_n$, которые возвращают *разные* целочисленные значения в случае ошибки. Возвращаемые значения выбраны уникальными с целью упрощения отладки, хотя с точки зрения семантики программы существенная информация состоит только в самом факте возникновения ошибки, независимо от того, какая функция f_i вернула ошибочное значение.
5. Функция f имееет формальные параметры $p_1, p_2, \ldots, p_n$, каждый из которых принимает во всех вызовах f одно из двух значений, означающих true и false. Если все p_i имеют тип `int`, то суммарный диапазон значений по всем параметрам p_i равен $(256 \cdot \texttt{sizeof(int)})^n$, что много больше требуемого диапазона значений, состоящего из не более чем 2^n векторов логических значений true и false. Типичным классом программ,

содержащих функции с большим числом параметров, используемых в качестве логических флагов, являются трансляторы и компиляторы. Для программ этого класса характерны обширные структурированные типы данных с обилием логических флагов, обозначающих наличие или отсутствие специфических свойств объекта. Так, структура, описывающая объявление объекта, может содержать логические поля для обозначения таких свойств, как асинхронный доступ к объекту, необходимость побитовой упаковки и др.

Далее приводится подробное описание оптимизаций по размеру, улучшающих фрагменты кода, имеющие специфику, описанную в примерах выше.

Внутрипроцедурное слияние статических флагов

Описание алгоритма

Определение 5. Параметр p функции или переменная p называется *флагом*, если удовлетворяет следующим условиям:

1) p имеет скалярный тип;
2) p входит хотя бы в одно условие хотя бы одного из операторов `if... else`, `while`, `do... while`, `for`, `switch`;
3) количество значений, которые p может принимать в программе (всюду далее *диапазон флага*), известно статически.

В качестве типичных кандидатов на роль флагов можно назвать формальные параметры и переменные типа перечисления `enum` и логического типа `_Bool`.

Заметим, что согласно определению 5 диапазон флага формируется всеми присваиваниями значений параметру p в программе независимо от того, действительно ли они выполнятся на этапе выполнения. При сочетании с другими оптимизациями слияние флагов следует выполнять после dead code elimination, чтобы полученные диапазоны флагов были минимальны.

Пользуясь определением флага, сформулируем алгоритм оптимизации под названием «слияние статических флагов».

Исходные данные

Программа P, содержащая такую функцию f, что

1) некоторые параметры f являются флагами (без ограничения общности будем считать, что флагами являются первые n параметров $p_1, \ldots, p_n$, $n \in \mathbb{N}$, в противном случае переименуем формальные параметры f);
2) все флаги принимают константные значения в вызовах f. При этом любой флаг может принимать *разные* константные значения в разных вызовах.

Алгоритм

1. Найти диапазоны D_i всех флагов p_i.

2. Найти число $N_{p_1,\ldots,p_n}$ векторных значений, которые могут принимать флаги $p_1,\ldots,p_n$ в программе P, по формуле
$$N_{p_1,\ldots,p_n} = \prod_{i=1}^{n} D_i.$$
3. Найти минимальную ширину $K_{p_1,\ldots,p_n}$ целочисленного типа, который может хранить $N_{p_1,\ldots,p_n}$ значений, по формуле
$$K_{p_1,\ldots,p_n} = \min\{k \in \mathbb{N} \mid 2^k \geqslant N_{p_1,\ldots,p_n}\}.$$
4. Если $K_{p_1,\ldots,p_n} = N_{p_1,\ldots,p_n}$, завершить работу без применения оптимизации.
5. Если $K_{p_1,\ldots,p_n}$ превышает максимальную ширину целочисленного типа, аппаратно поддерживаемую целевой платформой, то применить оптимизацию для набора флагов $p_1,\ldots,p_{n-1}$.
6. Построить взаимно-однозначное отображение
$$F_f\colon A_1 \times \ldots \times A_n \mapsto \mathbb{N},$$
отображающее наборы значений флагов $p_1,\ldots,p_n$ на значения нового флага, где A_i — множество значений, которые флаг p_i может принимать в P.
7. Применить отображение F_f на всех первых n фактических параметрах во всех вызовах функции f в P.
8. Заменить формальные параметры $p_1,\ldots,p_n$ функции f единственным формальным параметром $P_{p_1,\ldots,p_n}$.
9. Заменить условия, в которые входят флаги $p_1,\ldots,p_n$, в соответствии с полученным отображением F_f.

Заметим, что код функции, полученный после применения приведённого выше алгоритма, всё ещё может содержать параметры-флаги. Следовательно, слияние статических флагов следует применять, вообще говоря, до тех пор, пока сигнатура оптимизируемой функции не перестанет изменяться.

Пример. Исходный код:

```
#include <stdio.h>
#include <string.h>

/* Режим работы функции f1 */
typedef enum { C1, C2, C3 } mode;

/* Функция обрабатывает входную строку str
в зависимости от режима работы, заданного флагами a и b */

char * f1(mode a, mode b, char* str) {
    if ( (a==C1) && (b==C2) ) {
        return (str+1);
    }
```

```
    if ( (a==C2) && (b==C2) ) {
        return str;
    }
    if ( (a==C3) && (b==C3) ) {
        return (str+strlen(str)-1);
    }
    return NULL;
}
```

Результат слияния флагов:

```
#include <stdio.h>
#include <string.h>

/* Функция обрабатывает входную строку str
в зависимости от режима работы, заданного флагом flag,
полученным в результате слияния флагов */

char * f1(char flag, char* str) {
    if (! flag ) {
        return (str+1);
    }
    if (flag==1) {
        return str;
    }
    if (flag==2) {
        return (str+strlen(str)-1);
    }
    return NULL;
}
```

Размеры сегмента с кодом программы до и после слияния флагов:

- 4c и 32 с ключом -Os, уменьшение кода на 26 байт, или 66 %;
- 5b и 3b с ключами -O2 и -O3, уменьшение кода на 32 байта, или 65 %.

Как видно по приведённому примеру, построенная оптимизация может делать код неудобочитаемым для человека. Следовательно, она более адекватна для реализации в оптимизаторе, но не для ручного применения при рефакторинге или разработке программного проекта.

Оценка эффективности

Утверждение 1. *Пусть задана функция f в программе P, и пусть n параметров функции f являются флагами, $n \in \mathbb{N}$, причём слияние статических флагов ставит в соответствие этим флагам ровно один результирующий флаг. Тогда*

для статического слияния параметров-флагов в коде функции f имеет место следующая оценка $E_{f,P}$:

$$E_{f,P} \in [A, A + d],$$

где

$$A = (n-1)\big(D_{p_1,\ldots,p_n} \cdot \big(\mathrm{Cmp}(p, C_j) + \mathrm{Land}()\big) + C_f(\mathrm{Push} + \mathrm{Pop})\big),$$
$$d = C_f(n-1)\mathrm{Mov}()$$

и приняты обозначения $D_{p_1,\ldots,p_n}$ — количество операций сравнения набора исходных флагов с набором констант в программе P, p — параметр-флаг, полученный после слияния флагов, C_j — константы.

Доказательство. Пусть $p_1, \ldots, p_n$ — параметры-флаги функции f.

Пусть в программе P имеется $D(p_1, \ldots, p_n)$ выражений вида

$$O_j = (p_1 == C_{ji})\ \&\&\ \ldots\ \&\&\ (p_n == C_{jn}),$$

каждое из которых сравнивает все p_i с константами C_{ji}.

Без ограничения общности можно считать, что алгоритм оптимизирует все n флагов, не сужая множества рассматриваемых флагов. Действительно, в противном случае набор $p_1, \ldots, p_q$ флагов, уменьшенный алгоритмом оптимизации, выберем в качестве исходного множества флагов.

В результате слияния статических флагов каждая операция O_j преобразуется в операцию $\mathrm{Cmp}(P, C_j)$, где p — единственный параметр-флаг и C_j — константа.

Таким образом, операция O_j уменьшена на величину

$$(n-1) \cdot D_{p_1,\ldots,p_n} \cdot (\mathrm{Cmp}(p, C_j) + \mathrm{Land}()).$$

Далее, поскольку слияние статических флагов изменяет как условия, включающие параметры-флаги, так и сигнатуру функции f, также примем во внимание изменение кода, формирующего стековый фрейм в вызовах оптимизированной функции f.

Поскольку рассматривается стековая машина, то формирование стекового фрейма заключается в сохранении регистров в стек перед вызовом f и восстановлении регистров из стека по завершении вызова f. Следовательно, при уменьшении количества формальных параметров-флагов в функции f происходит уменьшение кода, формирующего стековый фрейм, на величину

$$(\mathrm{Push} + \mathrm{Pop})(n-1)C_f,$$

где C_f — количество операторов вызова функции f в исходном коде программы P.

Кроме того, если какой-либо из исходных флагов не загружен в регистр перед сохранением в стек, то происходит копирование значения флага в регистр.

Таким образом, окончательная оценка $E_{f,P}$ эффективности слияния статических флагов $f_1, \ldots, f_n$ функции f в программе P ограничена снизу величиной

$$(n-1)\big(D_{p_1,\ldots,p_n} \cdot \big(\mathrm{Cmp}(p, C_j) + \mathrm{Land}()\big) + C_f(\mathrm{Push} + \mathrm{Pop})\big)$$

и сверху величиной

$$(n-1)\Big(D_{p_1,\ldots,p_n}\cdot\big(\mathrm{Cmp}(p,C_j)+\mathrm{Land}()\big)+C_f\big(\mathrm{Push}+\mathrm{Pop}+\mathrm{Mov}()\big)\Big),$$

что и требовалось доказать. □

Из утверждения 1 вытекают следующие свойства слияния статических флагов.

1. Величина уменьшения кода при внутрипроцедурном слиянии статических флагов зависит от параметров n, C_f и $D_{f_1,\ldots,fn}$, которые специфичны для оптимизируемой программы P.
2. Оптимизация является глобальной.
3. Полученная оценка демонстрирует величину $E_{f,P}$ уменьшения всего кода рассматриваемой программы P и величину E_f уменьшения функции f, рассматриваемую вне программы P, причём величина E_f — точная величина уменьшения размера кода функции f. В самом деле, для получения оценки уменьшения кода только функции f достаточно положить $C_f = 0$.
4. Компоненты оценки, кроме n, C_f и $D_{f_1,\ldots,fn}$, являются постоянными коэффициентами, зафиксированными выбором трёх составляющих: используемого компилятора, целевой аппаратной платформы, набора выполняемых оптимизаций, не считая рассматриваемой.
5. Величина уменьшения линейна по каждому из своих параметров.

Кроме того, из полученной оценки вытекает следующее утверждение.

Утверждение 2. *Слияние статических флагов не увеличивает размер кода.*

Доказательство. В самом деле, поскольку число n первоначальных флагов в функции f, подвергающейся слиянию статических флагов, не меньше единицы по построению алгоритма, то величина полученной оценки неотрицательна, что и требовалось доказать. □

Пример. Пусть оптимизация производит слияние трёх статических флагов p_1, p_2, p_3 — параметров заданной функции f. Пусть имеются три операции сравнения флагов в теле функции f, т. е. $D_{p_1,p_2,p_3} = 3$. Пусть также $\mathrm{Cmp} = 4$, $\mathrm{Land} = 3$, $\mathrm{Mov} = 3$. Наконец, пусть стоимости Push и Pop равны 1 и 3 байта соответственно и функция вызывается три раза, т. е. $C_f = 3$. В таком случае оценка уменьшения всего кода программы P будет $66 \leqslant E_{f,P} \leqslant 84$ байт. Положив $C_F = 0$, получим, что код самой функции f уменьшен на $E_f = 42$ байта.

Флаги в переменных

Данная оптимизация применяется к коду, содержащему достаточно большое количество переменных, играющих роль флагов. Как правило, на практике эта ситуация реализуется выделением в исходном коде проекта специального

модуля, содержащего все переменные, значения которых диктуют режимы работы тех или иных алгоритмов и которые потенциально предназначены для использования в любом модуле проекта. В проектах на языке C такой модуль традиционно называется flags.h и/или flags.c.

Идея алгоритма заключается в слиянии двух или более флагов $p_1, \ldots, p_n$ в результирующий флаг $P_{p_1,\ldots,p_n}$ так, что декартово произведение множеств значений флагов p_i однозначно отображается на множество значений флага $P_{p_1,\ldots,p_n}$.

Исходные данные

Программа P, содержащая набор переменных $p_1, \ldots, p_n$, которые являются флагами.

Алгоритм

1. Найти диапазоны D_i всех флагов p_i.
2. Найти число $N_{p_1,\ldots,p_n}$ наборов значений, принимаемых флагами $p_1, \ldots, p_n$, по формуле
$$N_{p_1,\ldots,p_n} = \prod_{i=1}^{n} D_i.$$
3. Найти минимальную ширину $K_{p_1,\ldots,p_n}$ аппаратного целочисленного типа, который может хранить $N_{p_1,\ldots,p_n}$ значений, по формуле
$$K_{p_1,\ldots,p_n} = \min\{k \in \mathbb{N} \mid 2^k \geqslant N_{p_1,\ldots,p_n}\}.$$
4. Если $K_{p_1,\ldots,p_n} = N_{p_1,\ldots,p_n}$, завершить работу без применения оптимизации.
5. Если $K_{p_1,\ldots,p_n}$ превышает максимальную ширину целочисленного типа, аппаратно поддерживаемую целевой платформой, применить оптимизацию для набора флагов $p_1, \ldots, p_{n-1}$.
6. Построить взаимно-однозначное отображение
$$F_f \colon A_1 \times \ldots \times A_n \mapsto \mathbb{N},$$
отображающее наборы значений флагов $p_1, \ldots, p_n$ на значения нового флага, где A_i — множество значений, которые может принимать флаг p_i в P.
7. Применить отображение F_f на всех обращениях к p_i.
8. Заменить условия, в которые входят флаги $p_1, \ldots, p_n$, в соответствии с полученным отображением F_f.

Внутрипроцедурное слияние динамических флагов

Описанные выше оптимизации, построенные на определении флага, содержат существенное ограничение: флагам присваиваются только константные значения. В настоящем разделе обобщим оптимизации флагов на случай произвольного изменения значений флагов на этапе выполнения программы.

В общем случае использование в программе некоторого набора флагов $p_1, \ldots, p_n$ можно описать следующим образом.

- Изменения флагов $p_1, \ldots, p_n$ изменяют значения некоторого флага $P_{p_1,\ldots,p_n} = P(p_1, \ldots, p_n)$.
- Условные операторы и выражения зависят только от значений флага $P_{p_1,\ldots,p_n}$.
- Без ограничения общности можно считать, что для заданного набора $p_1, \ldots, p_n$ в программе нет условий, зависящих от собственного непустого подмножества множества флагов $p_1, \ldots, p_n$. В самом деле, допустив, что в программе есть условие, зависящее от $p_2, \ldots, p_n$, достаточно рассмотреть в качестве исходного множества флагов множество $p_2, \ldots, p_n$ и флаг $P_{p_2,\ldots,p_n} = P(p_2, \ldots, p_n)$.

Таким образом, задача оптимизации по размеру флагов в общем случае сводится к построению такой функции $P_{p_1,\ldots,p_n}$, вычисление которой на этапе выполнения программы выполняется наиболее экономно с точки зрения размера кода.

При построении функции $P_{p_1,\ldots,p_n}$ используются следующие возможности, полученные в результате отказа от фиксированной реализации кода:

1) возможность хранения флага $P_{p_1,\ldots,p_n}$
   - в глобальной переменной;
   - в локальной переменной, значение которой передаётся между функциями;
   - в локальной переменной только в той функции, в которой нужно получить значение флага $P_{p_1,\ldots,p_n}$;
2) аналогичные возможности хранения флагов $p_1, \ldots, p_n$. Заметим, что для разных p_i могут использоваться разные способы хранения;
3) необязательность взаимной однозначности соответствия значений флага $P_{p_1,\ldots,p_n}$ и наборов значений флагов $p_1, \ldots, p_n$. Например, если флаги $p_1, \ldots, p_n$ имеют тип `int`, но во всех условиях различаются только два значения каждого флага (моделирующие истину и ложь), то в таком случае все значения флага, означающие истину (например, все ненулевые значения) могут отображаться на одно и то же значение.

Одна из естественных реализаций функции $P_{p_1,\ldots,p_n}$ «упаковывает» исходные флаги в итоговый флаг с помощью побитового сдвига. Приведём пример построения такой функции $P_{p_1,\ldots,p_n}$.

Исходный код, подлежащий оптимизации:

```
typedef enum _state { one = 0, two = 1, three = 2 } state;

/* Все параметры являются флагами,
значения которых известны лишь на этапе выполнения */
int f(state p_1, state p_2, state p_3) {
    if ( (p_1==one) && (p_2==three) && (p_3==two) ) {
        return 0;
    }
```

```
    if ( (p_1==two) && (p_2==one) && (p_3==two) ) {
        return 1;
    }
}
```

Согласно определению типа `state`, каждый из флагов p_1, p_2, p_3 принимает только три различных значения. Строго говоря, согласно семантике типов языка C каждый флаг p_i может принимать любые целочисленные значения. Однако для соблюдения эквивалентности семантики в оптимизированном коде достаточно учитывать только три различные значения флагов, так как только три разных значения флагов используются в логических условиях, определяющих семантику программы.

Поскольку для хранения трёх значений от 0 до 2 достаточно двух бит, то функция $P_{p_1,\ldots,p_n}$ принимает вид

$$P_{p_1,p_2,p_3} = P(p_1,p_2,p_3) = (p_1 \texttt{ << 4) \& (}p_2 \texttt{ << 2) \& (}p_3\texttt{)}.$$

Код, полученный после оптимизации, выглядит следующим образом:

```
typedef enum _state { one = 0, two = 1, three = 2 } state;

int f(state p_1, state p_2, state p_3) {
    short p = ( (p_1 << 4) & (p_2 << 2) & p_3 );
    if ( p == ((2 << 2) & 1)) {
        return 0;
    }
    if ( p == ((2 << 4) & (1 << 2) & 2)) {
        return 0;
    }
}
```

В построенном коде константные условия условных выражений вычисляются на этапе компиляции. Таким образом, вычисление условий двух условных выражений заменено одним вычислением локальной переменной p, являющейся итоговым флагом.

Размер сегмента с кодом программы:

- 1b и 27 байт при оптимизации -O2, код уменьшен на 17 байт, или 39 %.

Отметим особенности алгоритма, продемонстрированные в примере.

1. В зависимости от целевой платформы оптимизация может не приводить к уменьшению кода, если исходных флагов p_i не более двух.
2. Оптимизация применима, только если выполнены следующие ограничения:
   - количество значений, принимаемых флагами $p_1,\ldots,p_n$, известно на этапе компиляции. Заметим, что диапазон значений, принимаемых

флагами, может не играть роли: если диапазон значений больше количества значений, принимаемых флагами, компилятор может изменить значения, принимаемые флагами и используемые в условиях, так, чтобы эти диапазоны совпадали.

- количество значений, принимаемых флагами, кодируется количеством бит, как минимум втрое меньшим одного из целочисленных типов, поддерживаемых аппаратно. Это условие необходимо для эффективной «упаковки» исходных флагов $p_1, \ldots, p_n$, $n \geqslant 3$, в итоговый флаг $P_{p_1,\ldots,p_n}$.

3. Вычисление значения $P_{p_1,\ldots,p_n}$ в той функции f, в которой оно используется, почти всегда оптимально с точки зрения размера кода для любого количества и любого расположения вызовов функции f в графе вызовов программы. Исключением может являться ситуация, когда f — статическая функция, не привязана к прерыванию, имеет ровно один вызов, но по каким-либо причинам оптимизатор не выполняет для неё подстановку `inline`. В таком случае более выгодно расположить вычисление фукнции $P_{p_1,\ldots,p_n}$ *перед* вызовом функции f.

Оценка эффективности

Нетрудно заметить, что изменение кода при выполнении слияния динамических флагов происходит так же, как при слиянии статических флагов, но в оптимизированную функцию добавляется код для вычисления значения результирующего флага $P_{p_1,\ldots,p_n}$, которое не может быть вычислено статически.

Таким образом, нижняя и верхняя оценки эффективности внутрипроцедурного слияния динамических флагов равны соответствующим оценкам внутрипроцедурного слияния статических флагов, уменьшенным на стоимость вычисления $P_{p_1,\ldots,p_n}$.

Вообще говоря, вычисление $P_{p_1,\ldots,p_n}$ может выполняться по разным формулам, однако побитовая упаковка представляется наиболее простой и эффективной. Размер кода, необходимого для побитовой упаковки, равен n размерам операции $\mathrm{Shift}(A, C)$ побитового сдвига заданного значения A на заданную константу C:

$$P_{p_1,\ldots,p_n} = n \cdot \mathrm{Shift}(A, C).$$

Использование побитовой упаковки максимально эффективно на целевых архитектурах с поддержкой машинных команд с встроенным побитовым сдвигом операнда.

Слияние флагов с условиями общего вида

Все приведённые оптимизации слиянием флагов легко обобщаются на условия в условных операторах, содержащие произвольные логические выражения.

Для реализации условий общего вида достаточно построить отображение

$$F_f\colon (l_1, h_1) \times \ldots \times (l_n, h_n) \mapsto \mathbb{N}$$

без требования взаимной однозначности.

Пример. Условие `if (!(a==2))` для флага a в функции f, принимающего значения из диапазона от 0 до 255, описывает множество натуральных чисел от 0 до 255, за исключением 2. Все значения из этого множества заменяются на значение, например, 0 и значение 2 заменяется на другое значение, например 1.

Практический пример применения оптимизаций

Рассмотрим функцию `build_array_declarator` компилятора gcc, которая создаёт декларатор переменной типа массив.

```
/* Построение декларатора массива.
EXPR — выражение внутри [] или NULL_TREE.
QUALS — квалификаторы типа, указанные внутри [] (заданные для
применения к типу—указателю, к которому преобразуется параметр массива).
STATIC_P — логический флаг:
ИСТИНА, если внутри [] указано "static", иначе ЛОЖЬ.
VLA_UNSPEC_P — логический флаг:
ИСТИНА, если массив является [*]—массивом, т. е. массивом
переменной длины (VLA), который тем не менее является полным типом
(в данный момент не поддерживается компилятором gcc),
в противном случае ЛОЖЬ.
Наконец, предполагается, что поле встроенного декларатора
будет заполнено функцией set_array_declarator_inner. */

struct c_declarator * build_array_declarator
    (tree expr, struct c_declspecs *quals, bool static_p, bool vla_unspec_p) {
    struct c_declarator *declarator =
        XOBNEW (&parser_obstack, struct c_declarator);
    declarator->kind = cdk_array;
    declarator->declarator = 0;
    declarator->u.array.dimen = expr;
    if (quals) {
        declarator->u.array.attrs = quals->attrs;
        declarator->u.array.quals = quals_from_declspecs (quals);
    }
    else {
        declarator->u.array.attrs = NULL_TREE;
        declarator->u.array.quals = 0;
    }
```

```
    declarator->u.array.static_p = static_p;
    declarator->u.array.vla_unspec_p = vla_unspec_p;
    if (pedantic && !flag_isoc99) {
        if (static_p || quals != NULL)
            pedwarn ("ISO C90 does not support %<static%> or type"
                " qualifiers in parameter array declarators");
        if (vla_unspec_p)
            pedwarn ("ISO C90 does not support %<[*]%> array declarators");
    }
    if (vla_unspec_p)
        warning (0, "GCC does not yet properly implement"
            " %<[*]%> array declarators");
    return declarator;
}
```

Оптимизированный код имеет следующий вид:

```
struct c_declarator * build_array_declarator
        (tree expr, struct c_declspecs *quals, int flag) {

/* Оба исходных флага отображены на объединенный флаг flag,
который имеет более широкий тип. */

    struct c_declarator *declarator =
        XOBNEW (&parser_obstack, struct c_declarator);
    declarator->kind = cdk_array;
    declarator->declarator = 0;
    declarator->u.array.dimen = expr;
    if (quals) {
        declarator->u.array.attrs = quals->attrs;
        declarator->u.array.quals = quals_from_declspecs (quals);
    }
    else {
        declarator->u.array.attrs = NULL_TREE;
        declarator->u.array.quals = 0;
    }

/* Все пары значений исходных флагов отображаются на значения флага flag,
поэтому вся семантическая информация представляется значениями
последнего. */

    declarator->u.array.static_p = flag;

/* Замечание: на самом деле в данном контексте необходимо
только одно из полей static_p и vla_unspec_p.
```

Таким образом, имеется возможность дополнительной оптимизации структуры array. Однако в настоящей статье мы не рассматриваем эту оптимизацию и потому оставляем оптимизированный код всё же несколько избыточным. */

```
    declarator->u.array.vla_unspec_p = flag;
    if (pedantic && !flag_isoc99) {

/* Пары (static_p==TRUE,ANY) отображаются в 1. */

        if ( (flag==1) || quals != NULL)
            pedwarn ("ISO C90 does not support %<static%> or type"
                "  qualifiers  in  parameter array declarators");

/* Пары (static_p==ANY,vla_unspec_p!=0) отображаются в 0. */

        if (!flag)
            pedwarn ("ISO C90 does not support %<[*]%> array declarators");
    }
    if (!flag)
        warning (0, "GCC does not yet properly implement"
                " %<[*]%> array declarators");
    return declarator;
}
```

Литература

[1] Касперски К. Техника оптимизации программ. Эффективное использование памяти. — СПб.: БХВ-Петербург, 2003.

[2] Фаулер М. Рефакторинг: улучшение существующего кода. — СПб.: Символ-Плюс, 2005.

[3] Muchnik S. Advanced Compiler Design and Implementation. — Academic Press, 1997.

[4] Muchnik S. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. — Academic Press, 2002.