Министерство науки и высшего образования Российской Федерации

Новосибирский государственный технический университет

Кафедра ТПИ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Расчетно-графическое задание

Методы оптимизации кода. Оптимизация выражений.

Факультет: ПМИ

Преподаватели:
Еланцева И. Л.,
Петров Р. В.

Группа: ПМ-81

Студенты: Ефремов А. А.,
Ртищева К. С.

Бригада: 1

Вариант: 3

Новосибирск
2021

1. Оптимизация кода

Оптимизация кода - различные методы преобразования кода ради улучшения его характеристик и повышения эффективности. Среди целей оптимизации можно указать уменьшения объема кода, объема используемой программой оперативной памяти, ускорение работы программы, уменьшение количества операций ввода-вывода.

Главное из требований, которые обычно предъявляются к методу оптимизации - оптимизированная программа должна иметь тот же результат и побочные эффекты на том же наборе входных данных, что и неоптимизированная программа. Впрочем, это требование может и не играть особой роли, если выигрыш за счет использования оптимизации может быть сочтен более важным, чем последствия от изменения поведения программы.

2. Методы оптимизации кода

Peephole-оптимизация

Локальные peephole-оптимизации (англ. *peephole — «глазок»*) рассматривают несколько соседних инструкций, чтобы увидеть, можно ли с ними произвести какую-либо трансформацию с точки зрения цели оптимизации. В частности, они могут быть заменены одной инструкцией или более короткой последовательностью инструкций.

Например, удвоение числа может быть более эффективно выполнено с использованием левого сдвига или путём сложения числа с таким же.

Локальная оптимизация

В локальной оптимизации рассматривается только информация одного базового блока за один шаг. Так как в базовых блоках нет переходов потока управления, эти оптимизации требуют незначительного анализа (экономя время и снижая требования к памяти), но это также означает, что не сохраняется информация для следующего шага.

Внутрипроцедурная оптимизация

Внутрипроцедурные оптимизации — глобальные оптимизации, выполняемые целиком в рамках единицы трансляции (например, функции или процедуры). При такой оптимизации задействовано гораздо больше информации, чем в локальной, что позволяет достигать более значительных эффектов, но при этом часто требуются ресурсозатратные вычисления. При наличии в оптимизируемой программной единице глобальных переменных оптимизация такого вида может быть затруднена.

Оптимизация циклов

Существует большое количество оптимизаций, применяемых к циклам. При большом количестве повторений цикла такие оптимизации чрезвычайно эффективны, так как небольшим преобразованием влияют на значительную часть выполнения программы. Поскольку циклы — весомая часть времени выполнения многих программ, оптимизации циклов существуют практически во всех компиляторах и являются самыми важными.

Например, выявив инварианты цикла, иногда можно вынести часть операций из цикла, чтобы не выполнять избыточные повторные вычисления.

Межпроцедурная оптимизация

Такие виды оптимизаций анализируют сразу весь исходный код программы. Большее количество информации, извлекаемой данными методами, означает что оптимизации могут быть более эффективным по сравнению с другими методами. Такие оптимизации могут использовать довольно сложные методы, например, вызов функции замещается копией тела функции (встраивание или inline).

3. Оптимизация выражений

Большинство задач программирования нуждаются в применении математических и логических выражений. Сложные выражения обычно дороги, но есть способы их удешевления.

Алгебраические тождества

Алгебраические тождества не редко позволяют заменить «дорогие» операции на более «дешевые». Так, следующие выражения логически эквивалентны:

not a and not b

not (a or b)

Выбор второго выражение вместо первого, экономит одну операцию.

Избавление от одной операции not, не приведет к заметным результатам, но тем не менее этот принцип значительно полезен. Джон Бентли отмечает, что в одной программе проверялось условие sqrt(x) < sqrt(y). Так как sqrt(x) меньше sqrt(y), только когда x меньше, чем y, исходную проверку можно заменить на x < y. С учетом дороговизны метода sqrt(), можно сказать, что достигнута существенная экономии.

Снижение стоимости операции

Как уже было сказано, снижение стоимости операций подразумевает замену дорогой операции более дешевой. Вот некоторые возможные варианты:

замена умножения сложением;

замена возведения в степень умножением;

замена тригонометрических функций их эквивалентами;

замена типа long long на long или int (следите при этом за аспектами производительности, связанными с применением целых чисел естественной и неестественной длины);

замена чисел с плавающей запятой числами с фиксированной точкой или целые числа;

замена чисел с плавающей запятой с удвоенной точностью числами с одинарной точностью;

замена умножения и деления целых чисел на два операциями сдвига.

Инициализация во время компиляции

Если при вызове метода, передается ему в качестве единственного аргумента именованная константа или непосредственное значение, лучше заранее вычислить нужное значение, присвоить его константе и избежать вызова метода.

Недостатки системных методов

Системные методы очень дорогие и часто обеспечивают избыточную точность. Зачастую такая предельная точность не нужна, не стоит тратить на нее время. Еще один вариант оптимизации основан на том факте, что деление на 2 аналогично операции сдвига вправо. Двоичный логарифм числа равен количеству операций деления на 2, которое можно выполнить над этим числом до получения нулевого значения.

Пример метода, определяющего примерное значение двоичного логарифма с использованием оператора сдвига вправо:

```
unsigned int Log2( unsigned int x ) {
    unsigned int i = 0;
    while ( ( x = ( x >> 1 ) ) != 0 )  i++;
    return i;
}
```

Использование констант корректного типа

Используйте именованные константы и литералы, имеющие тот же тип, что и переменные, которым они присваиваются. Если константа и соответствующая ей переменная имеют разные типы, перед присвоением константы переменной компилятор должен будет выполнить преобразование типа.

Устранение часто используемых подвыражений

Вместо повторяющихся несколько раз выражений, следует присвоить его значение константе и использовать ее там, где ранее вычислялось само выражение.

4. Применяемая реализация

В данной работе было реализовано снижение стоимости операции, а именно, замена деления умножением, сокращение количества операций, например

До оптимизации	После оптимизации
int i = (a + b) * c + (a + b) * d	int i = (a + b) * (c + d)
int i = c / (a + b) + d / (a + b)	int i = (c + d) / (a + b)
int i = b / a / c	int i = b / (a * c)
int i = b / (a / c)	int i = b * c / a

Оптимизация выражений была встроена в синтаксический анализатор во время построения обратной польской записи выражений.

5. Тестовые примеры

№	Входные данные	Выходные данные до оптимизации выражений “postfix.txt”	Выходные данные после оптимизации выражений “postfixSimple.txt”	Назначение

1	int main()
	{
	int a = 1, b = 2, c = 3;
	int d = (a + b) * b - (a + b) * c;
	int f = (a * b) * b + c * (a * b) ;
	int e = b * (a / b) - c * (a / b) ;
	int g = b * (a - b) + (a - b) * c;
	return 0;
	}
		a 1 =
		b 2 =
		c 3 =
		d a b + b * a b + c * - =
		f a b * b * c a b * * + =
		e b a b / * c a b / * - =
		g b a b - * a b - c * + =
			a 1 =
			b 2 =
			c 3 =
			d a b + b c - * =
			f a b * b c + * =
			e a b / b c - * =
			g a b - b c + * =
				int main()
				{
				int a = 1, b = 2, c = 3;
				int d = (a + b) * (b – c);
				int f = (a * b) * (b + c);
				int e = (a / b) * (b – c);
				int g = (a - b) * (b + c);
				return 0;
				}

2	int main()
	{
	int a = 1, b = 2, c = 3;
	int d = b / (a + b) + c / (a + b) ;
	int f = b / (a - b) - c / (a - b) ;
	int e = b / (a * b) + c / (a * b) ;
	return 0;
	}
		a 1 =
		b 2 =
		c 3 =
		d b a b + / c a b + / + =
		f b a b - / c a b - / - =
		e b a b * / c a b * / + =
			a 1 =
			b 2 =
			c 3 =
			d b c + a b + / =
			f b c - a b - / =
			e b c + a b * / =
				int main()
				{
				int a = 1, b = 2, c = 3;
				int d = (b + c) / (a + b);
				int f = (b – c) / (a – b);
				int e = (b + c) / (a * b);
				return 0;
				}

3	int main()
	{
	int a = 1, b = 2, c = 3;
	int d = b / (a / c) ;
	int f = b / a / c ;
	int e = b / a / c + d / a / c ;
	int i = b / (a / c) - d / (a / f) ;
	return 0;
	}
		a 1 =
		b 2 =
		c 3 =
		d b a c / / =
		f b a / c / =
		e b a / c / d a / c / + =
		i b a c / / d a f / / - =
			a 1 =
			b 2 =
			c 3 =
			d b c * a / =
			f b a c * / =
			e b d + a c * / =
			i b c * d f * - a / =
				int main()
				{
				int a = 1, b = 2, c = 3;
				int d = b * c / a;
				int f = b / (a * c);
				int e = (b + d) / (a * c);
				int i = (b * c – d * f) /a;
				return 0;
				}

6. Текст программы

```
struct tree
{
	string elem;
	tree* left, * right;
	tree(string _elem = "@", tree* _left = NULL, tree* _right = NULL)
		: elem(_elem), left(_left), right(_right) {}
};

tree* buildTree(deque<string>& postfix)
{
	string c;
	tree* t;

	c = postfix.back();
	postfix.pop_back();

	if (c == "*" || c == "/" || c == "+" || c == "-" || c == "=")
	{
		t = new tree(c);
		t->right = buildTree(postfix);
		t->left = buildTree(postfix);
		return t;
	}
	else
	{
		t = new tree(c);
		return t;
	}
}

bool equalTrees(tree* t1, tree* t2)
{
	if (t1 == NULL && t2 == NULL)
		return true;
	if (t1->elem == t2->elem)
	{
		if(equalTrees(t1->left, t2->left) && equalTrees(t1->right, t2->right) || equalTrees(t1->left, t2->right) &&
			equalTrees(t1->right, t2->left) && t1->elem == "+")
			return true;
		else return false;
```

```
	}
	else return false;
}

void standIn(tree** t)
{
	tree* d = *t, * f1, * f2, * temp;
	bool isProcessed;
	stack<tree*> s;
	stack<bool> was;
	stack<int> count;

	do
	{
		while (d)
		{
			s.push(d);
			was.push(false);
			d = d->left;
		}
		if (!s.empty())
		{
			do
			{
				d = s.top();
				s.pop();
				isProcessed = was.top();
				was.pop();
				if (isProcessed)
				{


					if (d->elem == "/" && d->left->elem == "/")
					{
						d->right = new tree("*", d->left->right, d->right);
						d->left = d->left->left;
					}

					if (d->elem == "/" && d->right->elem == "/")
					{
						d->left = new tree("*", d->left, d->right->right);
						d->right = d->right->left;
```

```
}

if (d->elem == "+" || d->elem == "-")
{
	if (d->left->elem == "*" && d->right->elem == "*")
	{
		f1 = d->left;
		f2 = d->right;
		if (equalTrees(f1->left, f2->left))
		{
			d->left = f1->left;
			d->right = new tree(d->elem, f1->right, f2->right);
			d->elem = "*";
		}
		else
			if (equalTrees(f1->left, f2->right))
			{
				d->left = f1->left;
				d->right = new tree(d->elem, f1->right, f2->left);
				d->elem = "*";
			}
			else
				if (equalTrees(f1->right, f2->left))
				{
					d->left = f1->right;
					d->right = new tree(d->elem, f1->left, f2->right);
					d->elem = "*";
				}
				else
					if (equalTrees(f1->right, f2->right))
					{
						d->left = f1->right;
						d->right = new tree(d->elem, f1->left, f2->left);
						d->elem = "*";
					}
	}

	if (d->left->elem == "/" && d->right->elem == "/")
	{
		f1 = d->left;
		f2 = d->right;
		if (equalTrees(f1->right, f2->right))
```

```
                                        {
                                                d->right = f1->right;
                                                d->left = new tree(d->elem, f1->left, f2->left);
                                                d->elem = "/";
                                        }
                                }
                        }
                }

                } while (isProcessed && !s.empty());
                if (!isProcessed)
                {
                        s.push(d);
                        was.push(true);
                        d = d->right;
                }
        }
    } while (!s.empty());
}

void PostOrder(tree* t, deque<string>& postfix)
{
    stack<tree*> s;
    stack<bool> was;
    bool isProcessed;

    do
    {
        while (t)
        {
                s.push(t);
                was.push(false);
                t = t->left;
        }
        if (!s.empty())
        {
                do
                {
                        t = s.top();
                        s.pop();
                        isProcessed = was.top();
                        was.pop();
```

```
                        if (isProcessed)
                                postfix.push_back(t->elem);
                } while (isProcessed && !s.empty());
                if (!isProcessed)
                {
                        s.push(t);
                        was.push(true);
                        t = t->right;
                }
        }
    } while (!s.empty());
};

void simplification(deque<string>& postfix)
{
    tree* t = buildTree(postfix);
    standIn(&t);
    PostOrder(t, postfix);

}
```