Nome: Bruno Corrêa Zimmermann
Cartão: 00313985
Login: bczimmermann

Introdução

Este relatório visa explicar o desenvolvimento da etapa 7 do trabalho da disciplina de compiladores. Serão apresentadas aqui, quatro otimizações e seus resultados, assim como uma descrição da recuperação de erros da análise sintática.

Otimizações

A respeito de otimizações, foi escolhida a classe de otimizações *peephole*, exceto pela última otimização. As otimizações foram: remoção de `mov`*s* duplicados, redução de variáveis temporárias em uma função, troca das instruções `addq $1` e `subq $1` por `inc` e `dec`, respectivamente, e a troca de multiplicações com operandos na base 2 e divisões com divisores na base 2 por shift-lefts e shift-rights aritméticos. Para ligar todas as otimizações, basta passar a *flag* `-O` para o compilador. Individuais otimizações podem ser ligadas pelas suas respectivas *flags*.

Movs Duplicados

Nessa otimização, o número de `movs` é reduzido o máximo possível, evitando redundância. O objetivo dessa otimização é principalmente reduzir o tamanho do código gerado, mas também há o objetivo de melhorar o tempo de execução. A otimização pode ser ligada passando ao compilador a *flag* `-fdedup-movs`.

Exemplo 1

Antes:

```
movq -8(%rbp), %rax
addq %rax, %rsi
```

Depois:

```
addq -8(%rbp), %rsi
```

Exemplo 2

Antes:

```
addq -8(%rbp), %rcx
movq %rcx, %rdx
```

Depois:

```
addq -8(%rbp), %rdx
```

Incrementos e Decrementos

Nessa otimização, instruções de adição e subtração com operando imediato um (1) são substittuídas por instruções de incremento e decremento. O objetivo dessa otimização também é reduzir tamanho do código gerado, mas melhorar tempo de execução com instruções potencialmente mais rápidas. A otimização pode ser ligada passando ao compilador a *flag* `-finc-decs`.

Exemplo 1

Antes:

```
addq $1, %rsi
```

Depois:

```
incq %rsi
```

Exemplo 2

Antes:

```
subq $1, %rsi
```

Depois:

```
decq %rsi
```

Multiplicação e Divisão por Potências de Dois

Nessa otimização, multiplicações e divisões por potênciais de dois são convertidas em deslocamentos de bits. Essa otimização foi feita tanto a nível de TACs quanto de código assembly. Para tal, foram introduzidas duas novas operações: `TAC_SHMUL` (SHift-MULtiply) e `TAC_SHDIV` (SHift-DIVide), e então na geração de assembly essas duas TACs são implementadas de acordo com a plataforma. A otimização pode ser ligada passando ao compilador a *flag* `-fpower-of-two`. A otimização também aceita operandos negativos.

Exemplo 1

Antes:

```
TAC(TAC_MUL, Y, X, 8)
```

Depois:

```
TAC(TAC_SHMUL, Y, X, 3)
```

Exemplo 2

Antes:

```
TAC(TAC_DIV, Y, X, 16)
```

Depois:

```
TAC(TAC_SHDIV, Y, X, 4)
```

Reuso de Temporários

Nessa otimização, as variáveis temporárias, locais de cada função, são reutilizadas ao invés de criar-se uma nova. Essa otimização leva em consideração o conceito de blocos básicos, valores e valores locais de um bloco. Sem a otimização, para cada valor, cria-se uma variável. Com a otimização, quando valores locais de um bloco chegam à sua última leitura, eles passam a ser reusados pelo próximo valor local dentro da mesma função.

O objetivo dessa otimização é principalmente reduzir o uso de bytes da pilha, economizando memória e possibilitando níveis mais profundos de recursão, Por exemplo, na função `processar_bloco` da amostra `sample-md5.txt` (anexo I), reservava 448 bytes do stack frame para temporários. Após, a otimização, passou a se reservar 128 bytes. Secundariamente, poderia-se fazer melhor uso de memória cache, o que melhoraria tempo de execução. A otimização pode ser ligada passando ao compilador a *flag* `-freuse-tmps`.

Exemplo 1

Antes:

```
TAC(TAC_ADD, TMP0, x, 1)

TAC(TAC_DIV, TMP1, y, 2)

TAC(TAC_ADD, TMP2, TMP0, 3)

TAC(TAC_PRINT, NULL, TMP1, NULL)

TAC(TAC_PRINT, NULL, TMP2, NULL)
```

Depois:

```
TAC(TAC_ADD, TMP0, x, 1)
```

```
TAC(TAC_DIV, TMP1, y, 2)

TAC(TAC_ADD, TMP0, TMP0, 3)

TAC(TAC_PRINT, NULL, TMP1, NULL)

TAC(TAC_PRINT, NULL, TMP0, NULL)
```

Benchmarks

Para medir-se a performance, foi escrito um programa que calcula a hash MD5 da entrada (anexo I ao final do documento ou `sample-md5.txt` no projeto). O programa lida com entrada e saída do usuário com palavras de 32 bits sem sinal, apesar da implementação da linguagem fornecer inteiros de 64 bits com sinal. Para analisar a performance, foram feitas quatro medições:

1. Performance de tempo em função do tamanho da entrada da função MD5;
2. Performance de tempo em função de otimizações individuais;
3. Tamanho do executável em função de otimizações individuais;
4. Tamanho da seção de código (.text) em função de otimizações individuais.

Tabela 1: performance de tempo de execução em função do tamanho da entrada, para um programa não-otimizado versus um programa compilado com todas otimizações ligadas.

Tamanho	**Não-Otimizado**	**Otimizado Completamente (`-O`)**
65535	0,703s	0,508s
1048575	11,191s	7,837s
16777215	181,516s	125,054s

Pela tabela 1, pode-se perceber que as otimizações ligadas melhoraram o tempo de execução aproximadamente, em média, 1,42 vezes. Para medir o tempo, foi usado o comando `time`, e redirecionou-se para a entrada do programa um arquivo com dados aleatórios.

Tabela 2: performance de tempo de execução em função de otimizações individuais.

Otimização	Tempo
Não-otimizado	11,191s
`-fdedup-movs`	11,299s
`-finc-decs`	11,251s
`-fpower-of-two`	7,946s
`-freuse-tmps`	11,330s
otimizado completamente (`-O`)	7,837s

Pela tabela 2, pode-se perceber que a única otimização que contribuiu significativamente para a mellhora do tempo de execução foi a otimização de multiplicação e divisão por potências de dois. A medida foi feita da mesma forma da tabela 1, exceto que dessa vez apenas com o arquivo com entrada de tamanho 1048575. A otimização que emprega as instruções `inc` e `dec` foi combinada com a remoção de `mov`s duplicados, porque sem essa otimização auxiliar, a otimização de `-finc-decs` não detecta o padrão para ser aplicada.

Tabela 3: tamanho de executável em função de otimizações individuais.

Otimização	Tamanho do Executável
Não-otimizado	24112 bytes
`-fdedup-movs`	24112 bytess
`-finc-decs`	24112 bytes
`-fpower-of-two`	24112 bytes
`-freuse-tmps`	24112 bytes
otimizado completamente (`-O`)	20016 bytes

Pela tabela 3, observa-se que o executável não teve tamanho reduzido em otimizações individuais, mas ainda assim, combinando todas, houve redução. Como será visto em seguida, apesar disso, a seção de código diminui de tamanho com

qualquer otimização. Essa contradição pode ser explicada com a inserção de *padding* para “arredondar” o tamanho do binário. A medida do tamanho do executável foi feita com o comando `stat`, após a remoção dos símbolos, com o comando `strip`.

Tabela 4: tamanho da seção de código (.text) em função de otimizações individuais.

Otimização	**Tamanho da Seção `.text`**
Não-otimizado	10680 bytes
`-fdedup-movs`	9629 bytess
`-fdedup-movs -finc-decs`	9608 bytes
`-fpower-of-two`	10637 bytes
`-freuse-tmps`	10386 bytes
otimizado completamente (`-O`)	9271 bytes

Por fim, pela tabela 4, é evidente que qualquer otimização reduz o tamanho do código gerado, ainda que o executável final tenha tamanho arredondado. A otimização individual que melhor reduziu o tamanho da seção `.text` foi a combinação de remoção de `movs` duplicados com o emprego das instruções `inc` e `dec`, somente atrás da combinação de todas otimizações. O tamanho da seção foi medido com o comando `size`.

Podemos concluir que as otimizações emplacadas foram todas bem-sucedidas em reduzir o tamanho do código gerado. No entanto, deduplicação de `movs`, uso de `inc-decs`, e reuso de temporárioos falharam em melhorar tempo de execução, ainda que a otimização das potências de dois tenha sido eficiente para tal.

Recuperação de Erros

Para a recuperação de erros sintáticos, atacaram-se alguns padrões:

1. Ponto-e-vírgula faltante em declarações de variáveis globais;
2. Ponto-e-vírgula faltante na separação de *statements*;
3. Falta do sinal de igual em atribuições e inicializações;
4. Falta de nome de variável em declaração de variáveis escalares;

5. Falta de nome de variável em declaração ou acesso de vetores;
6. Falta do lado direito da atribuição;
7. Falta de colchetes ou colchetes desbalanceados em declaração e acesso a vetores;
8. Falta de parêntesis ou parêntesis desbalanceados em declaração de parâmetros de funções;
9. Falta de lista de parâmetros em declarações de funções;
10. Falta de chaves ou chaves desbalanceados em corpo de *statements*;
11. Parêntesis desbalanceados em expressões;
12. Falta de parêntesis ou parêntesis desbalanceados em condições de `se` e `enquanto`;
13. Falta de condição de `se` e `enquanto`.

Como princípio, todos os casos de erros tem uma mensagem de erro diferente. Realizando-se alguns testes, pode-se ver que a recuperação foi sucessiva, detectou-se vários erros sem gerar falsos erros. Por exemplo, ao compilar-se o anexo II (`sample-bad-syntax.txt` no projeto), encontramos as seguintes mensagens:

```
syntax error at line 11
     missing assigned vector variable
syntax error at line 12
     missing semicolon separating statements
syntax error at line 39
     missing opening parenthesis in `enquanto` condition
syntax error at line 45
     missing parameter list
```

Conclusão

Considera-se que o resultado final foi um êxito, pois ao menos uma otimização melhorou tempo de execução, todas otimizações reduziram tamanho da seção de código, e a recuperação de erros consegue emitir vários erros corretamente. Possíveis melhorias seriam lidar com mais casos de erro de sintaxe, para evitar erros confusos, e aperfeiçoar a qualidade do código.

Gostaria-se de implementar mais otimizações, mas devido ao tempo de entrega, não foi possível. Gostaria-se também de implementar mudanças na linguagem, tais qual parâmetros locais e variáveis locais.

ANEXO I – MD5

```
inte k[64];
inte s[64];
inte bloco[16];
inte hash[4];
inte h[4];
inte n = 0;
inte i = 0;
inte j = 0;
inte l = 0;
inte m = 0;
inte f = 0;
inte g = 0;
inte yes = 0;
inte and.i = 0;
inte and.k = 0;
inte and.c = 0;
inte or.i = 0;
inte or.k = 0;
inte or.c = 0;
inte xor.i = 0;
inte xor.k = 0;
inte xor.c = 0;
inte lrot.i = 0;

inte main()
{
    yes = 1;
    {
       escreva "processar hash (y=1 / n=0)? ";
       yes = entrada;
       entaum
            md5()
       se (yes != 0);
    } enquanto (yes != 0);
}

inte md5()
{
    init_s();
    init_k();
    init_hash();
```

```
escreva "tamanho da mensagem: ";
n = entrada;

escreva "palavras da entrada:\n";

i = 0;
{
   j = 0;
   {
         bloco[j] = entrada;
         j = j + 1;
   } enquanto (j < 16);
   processar_bloco();
   i = i + j;
} enquanto (i < (n / 16 * 16));

j = 0;
{
   bloco[j] = entrada;
   j = j + 1;
} enquanto (j < (n - i));

entaum
{
   {
         bloco[j] = 0;
         j = j + 1;
   } enquanto (j < 16);
   j = 0;
   processar_bloco();
} se (j > 14);

{
   bloco[j] = 0;
   j = j + 1;
} enquanto (j < 14);

bloco[j] = n * 4294967296 / 4294967296;
bloco[j + 1] = n / 4294967296;
processar_bloco();

escreva "palavras da hash:\n";
```

```
    i = 0;
    {
       hash[i] = i64_to_u32(hash[i]);
       escreva hash[i] '\n';
       i = i + 1;
    } enquanto (i < 4);
}

inte processar_bloco()
{
    l = 0;
    {
       h[l] = hash[l];
       l = l + 1;
    } enquanto (l < 4);

    m = 0;
    {
       entaum
       {
            f = or32(and32(h[1] h[2]) and32(0 - h[1] - 1 h[3]));
            g = m;
       }
       senaum
            entaum
            {
                  f = or32(and32(h[1] h[3]) and32(0 - h[3] - 1
h[2]));
                  g = (5 * m + 1) * 1152921504606846976 /
1152921504606846976;
            }
            senaum
                  entaum
                  {
                        f = xor32(xor32(h[1] h[2]) h[3]);
                        g = (3 * m + 5) * 1152921504606846976 /
1152921504606846976;
                  }
                  senaum
                  {
                        f = xor32(h[2] or32(h[1] 0 - h[3] - 1));
                        g = 7 * m * 1152921504606846976 /
1152921504606846976;
```

```
                }
                se (m < 48)
          se (m < 32)
       se (m < 16);

       f = f + h[0] + k[m] + bloco[g];
       h[0] = h[3];
       h[3] = h[2];
       h[2] = h[1];
       h[1] = h[1] + leftrotate(f, s[m]);

       m = m + 1;
    } enquanto (m < 64);

    m = 0;
    {
       hash[m] = hash[m] + h[m];
       m = m + 1;
    } enquanto (m < 4);
}

inte init_k()
{
    k[0] = 3614090360;
    k[1] = 3905402710;
    k[2] = 606105819;
    k[3] = 3250441966;
    k[4] = 4118548399;
    k[5] = 1200080426;
    k[6] = 2821735955;
    k[7] = 4249261313;
    k[8] = 1770035416;
    k[9] = 2336552879;
    k[10] = 4294925233;
    k[11] = 2304563134;
    k[12] = 1804603682;
    k[13] = 4254626195;
    k[14] = 2792965006;
    k[15] = 1236535329;
    k[16] = 4129170786;
    k[17] = 3225465664;
    k[18] = 643717713;
    k[19] = 3921069994;
```

```
k[20] = 3593408605;
k[21] = 38016083;
k[22] = 3634488961;
k[23] = 3889429448;
k[24] = 568446438;
k[25] = 3275163606;
k[26] = 4107603335;
k[27] = 1163531501;
k[28] = 2850285829;
k[29] = 4243563512;
k[30] = 1735328473;
k[31] = 2368359562;
k[32] = 4294588738;
k[33] = 2272392833;
k[34] = 1839030562;
k[35] = 4259657740;
k[36] = 2763975236;
k[37] = 1272893353;
k[38] = 4139469664;
k[39] = 3200236656;
k[40] = 681279174;
k[41] = 3936430074;
k[42] = 3572445317;
k[43] = 76029189;
k[44] = 3654602809;
k[45] = 3873151461;
k[46] = 530742520;
k[47] = 3299628645;
k[48] = 4096336452;
k[49] = 1126891415;
k[50] = 2878612391;
k[51] = 4237533241;
k[52] = 1700485571;
k[53] = 2399980690;
k[54] = 4293915773;
k[55] = 2240044497;
k[56] = 1873313359;
k[57] = 4264355552;
k[58] = 2734768916;
k[59] = 1309151649;
k[60] = 4149444226;
k[61] = 3174756917;
k[62] = 718787259;
```

```
    k[63] = 3951481745;
}

inte init_s()
{
    i = 0;
    {
      s[i] = 7;
      s[i + 1] = 12;
      s[i + 2] = 17;
      s[i + 3] = 22;
      i = i + 4;
    } enquanto (i < 16);

    {
      s[i] = 5;
      s[i + 1] = 9;
      s[i + 2] = 14;
      s[i + 3] = 20;
      i = i + 4;
    } enquanto (i < 32);

    {
      s[i] = 4;
      s[i + 1] = 11;
      s[i + 2] = 16;
      s[i + 3] = 23;
      i = i + 4;
    } enquanto (i < 48);

    {
      s[i] = 6;
      s[i + 1] = 10;
      s[i + 2] = 15;
      s[i + 3] = 21;
      i = i + 4;
    } enquanto (i < 64);
}

inte init_hash()
{
    hash[0] = 1732584193;
    hash[1] = 4023233417;
```

```
    hash[2] = 2562383102;
    hash[3] = 271733878;
}

inte bit0(inte bit0.i)
{
    entaum
      retorne 0
    senaum
      retorne 1
    se (bit0.i / 2 * 2 == bit0.i);
}

inte and1(inte and1.a inte and1.b)
{
    entaum
      retorne 1
    senaum
      retorne 0
    se ((bit0(and1.a) == 1) & (bit0(and1.b) == 1));
}

inte and32(inte and32.a inte and32.b)
{
    and.c = 0;
    and.i = 0;
    and.k = 1;
    {
      and.c = and.c + (and1(and32.a and32.b) * and.k);
      and.k = and.k * 2;
      and32.a = and32.a / 2;
      and32.b = and32.b / 2;
      and.i = and.i + 1;
    } enquanto (and.i < 32);

    retorne and.c;
}

inte or1(inte or1.a inte or1.b)
{
    entaum
      retorne 1
    senaum
```

```
      retorne 0
    se ((bit0(or1.a) == 1) | (bit0(or1.b) == 1));
}

inte or32(inte or32.a inte or32.b)
{
    or.c = 0;
    or.i = 0;
    or.k = 1;
    {
      or.c = or.c + (or1(or32.a or32.b) * or.k);
      or.k = or.k * 2;
      or32.a = or32.a / 2;
      or32.b = or32.b / 2;
      or.i = or.i + 1;
    } enquanto (or.i < 32);

    retorne or.c;
}

inte xor1(inte xor1.a inte xor1.b)
{
    entaum
      retorne 1
    senaum
      retorne 0
    se (bit0(xor1.a) != bit0(xor1.b));
}

inte xor32(inte xor32.a inte xor32.b)
{
    xor.c = 0;
    xor.i = 0;
    xor.k = 1;
    {
      xor.c = xor.c + (xor1(xor32.a xor32.b) * xor.k);
      xor.k = xor.k * 2;
      xor32.a = xor32.a / 2;
      xor32.b = xor32.b / 2;
      xor.i = xor.i + 1;
    } enquanto (xor.i < 32);

    retorne xor.c;
```

```
}

inte leftrotate(inte lrot.a inte lrot.b)
{
    {
      lrot.a = lrot.a / 2 + (bit0(lrot.a) * 4294967296);
      lrot.b = lrot.b - 1;
    } enquanto (lrot.b > 0);

    retorne lrot.a;
}

inte i64_to_u32(inte i64)
{
    i64 = i64 * 4294967296 / 4294967296;
    entaum
      i64 = i64 + 4294967296
    se (i64 < 0);
    retorne i64;
}
```

ANEXO II – Sintaxe Inválida

```
inte ivec[5] 1 2 3;
cara cvec[6] 'h' 'e' 'l' 'l' 'o' '\n';
real rvec[2] 2.56 3.7;
inte i = 0;

inte main()
{
    i = 0;
    {
       escreva cvec[i];
       [i] = cvec[i] + 1
       i = i + 1;
    } enquanto (i < 6);

    i = 0;
    {
       escreva cvec[i];
       i = i + 1;
    } enquanto (i < 6);
    escreva '\n';

    i = 0;
    {
       ivec[i] = ivec[i] + entrada;
       i = i + 1;
    } enquanto (i < 5);

    i = 0;
    {
       escreva ivec[i];
       escreva '\n';
       i = i + 1;
    } enquanto (i < 5);

    i = 0;
    {
       escreva rvec[i] '\n';
       i = i + 1;
    } enquanto i < 2);

    retorne 0;
```

```
}

inte do_stuff
{
    escreva x;
}
```