Aula 08 – Otimização de Código

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- **Aulas anteriores**: geração de código, sem preocupações com eficiência, assumindo a existência de um número ilimitado de temporários.
- Para geração de código real, é necessário considerar mais restrições.
- **Estes *slides***: princípios de otimização de código.
- **Objetivos**: apresentar uma visão geral das técnicas de otimização.

Referências

Section 8.9 – A Survey of Code Optimization Techniques
K. C. Louden
Chapters 8, 9, 10, 11, 12 & 13
K. D. Cooper
Chapter 12 – Optimization
D. Thain

Introdução

- *Three-address code* (TAC) é uma representação intermediária (IR) muito comum do código fonte.
- TAC pode ser gerado diretamente pelo *parser* através de ações semânticas.
- Na prática: gerado por um caminhamento na *abstract syntax tree* (AST).
- O *back-end* do compilador recebe como entrada uma IR e gera como saída o código-alvo específico para uma dada arquitetura.

Técnicas de Otimização de Código

- Desde o surgimento dos primeiros compiladores na década de 1950, a qualidade do código gerado por um compilador foi levada em consideração.
- Medidas de qualidade do código: velocidade (tempo de execução das instruções) e tamanho (número de instruções e consumo de memória).
- Quais medidas são mais importantes depende da aplicação.
- Técnicas de otimização: série de transformações para a melhoria do código (segundo algum aspecto).
- Termo “otimização” não pode ser levado ao pé-da-letra: em geral não existe um código “ótimo” no sentido matemático.

Técnicas de Otimização de Código

- Pesquisa atual em compiladores foca nas otimizações.
- Otimização de código é a etapa mais demorada do processo de compilação.
- *Scanning* e *parsing*: tempo de execução linear em relação ao tamanho do programa de entrada.
- Métodos de otimização podem ser polinomiais (quadráticos, cúbicos) ou até pior.
- Desafio: encontrar boas soluções sem aumentar demais o tempo de compilação.
- $\Rightarrow$ Otimização geralmente usa heurísticas.

Alocação de Registradores

- Uso adequado de registradores é uma das características mais importantes de um código eficiente.
- *Approach CISC*: poucos registradores, muitas operações diretas na memória.
- Consequência: muito *register spill*, i.e., necessidade de guardar valores intermediários dos registradores em áreas temporárias de memória.
- Arquitetura CISC tenta minimizar o impacto do *register spill* tornando as operações na memória mais eficientes. Por exemplo, introduzindo vários níveis de cache.
- *Approach RISC*: muitos registradores, poucas operações diretas na memória.
- Alocação de registradores em máquinas RISC é mais crítica para o desempenho do código, mas também é mais fácil pois há mais registradores disponíveis.

Eliminação de Operações Desnecessárias

- Segundo ponto principal da otimização de código é evitar gerar comandos para operações que são redundantes ou desnecessárias.
- Análise pode ser simples, envolvendo algumas poucas instruções locais, ou bem complexa, levando em conta todo o código.
- Exemplos típicos: *common subexpression elimination* e *dead code elimination*.
- *Common subexpression elimination*: se uma expressão aparece repetida e o seu valor não muda, ela só precisa ser avaliada a primeira vez e o seu valor guardado em um temporário.
- A otimização dual é evitar armazenar o valor de uma variável ou temporário que não for mais usado depois.

Eliminação de Operações Desnecessárias

- Exemplos de *dead code elimination*:
  - Código morto devido à macros.

```
#define DEBUG 0
...
if (DEBUG)
{ ... }
```

  - Funções que são declaradas mas não são chamadas, ou são chamadas somente por código inalcançável.
- Eliminação de código morto não afeta muito o tempo de execução do programa mas pode reduzir consideravelmente o tamanho do executável.
- Outra otimização similar: *jump optimization*.
- Eliminação de código morto pode levar ao aparecimento de saltos consecutivos que podem então ser condensados.

Simplificação de Operações

- O gerador de código também deve tentar diminuir o custo das operações que devem ser emitidas.
- Exemplos típicos:
  - Multiplicação por 2 pode ser implementada como um *shift*.
  - Exponenciação a uma potência pequena pode ser implementada por multiplicações.
- *Constant folding*: substituir uma expressão constante ($2 + 3$) pelo seu resultado (5).
- *Constant propagation*: quando uma variável possui um valor constante em um trecho de código, substituir as ocorrências da variável pela constante.
- *Procedure inlining*: substitui a chamada da função pelo corpo, evitando o *overhead* da chamada.

Previsão de Comportamento do Programa

- Para efetuar algumas das otimizações descritas anteriormente, o compilador precisa coletar informações sobre o uso de variáveis, valores e funções nos programas fonte.
- Os compiladores comuns fazem uma estimativa conservadora dessas informações.
- Isto é, na dúvida, o compilador sempre assume o pior caso para evitar a geração de código incorreto.
- Alternativa: otimização com o uso de *profiling*.
- O código é executado várias vezes e uma estatística do seu comportamento é construída.
- Essa informação pode ser usada para ajustar a estrutura de saltos, *loops*, etc, para melhorar o tempo de execução dos casos mais comuns.

Interação entre Otimizações

- A maioria das otimizações são realizadas durante ou após a geração do IC ou do código alvo.
- *Source-level optimizations*: otimizações que não dependem das características da arquitetura alvo. Ex.: *constant folding*.
- *Target-level optimizations*: otimizações que usam as características específicas da máquina alvo.
- Muitas vezes é difícil se estabelecer uma sequência fixa para as operações de otimização, pois elas podem interagir entre si. $\Rightarrow$ Requer múltiplas passadas.
- *Exemplo*: considere o código abaixo.

```
x = 1; ...
y = 0; ...
if (y) x = 0; ...
if (x) y = 1;
```

Interação entre Otimizações

Uma primeira passada com propagação de constantes pode resultar no código abaixo.

```
x = 1; ...
y = 0; ...
if (0) x = 0; ...
if (x) y = 1;
```

Agora, o corpo do primeiro `if` é inatingível, e sua eliminação gera

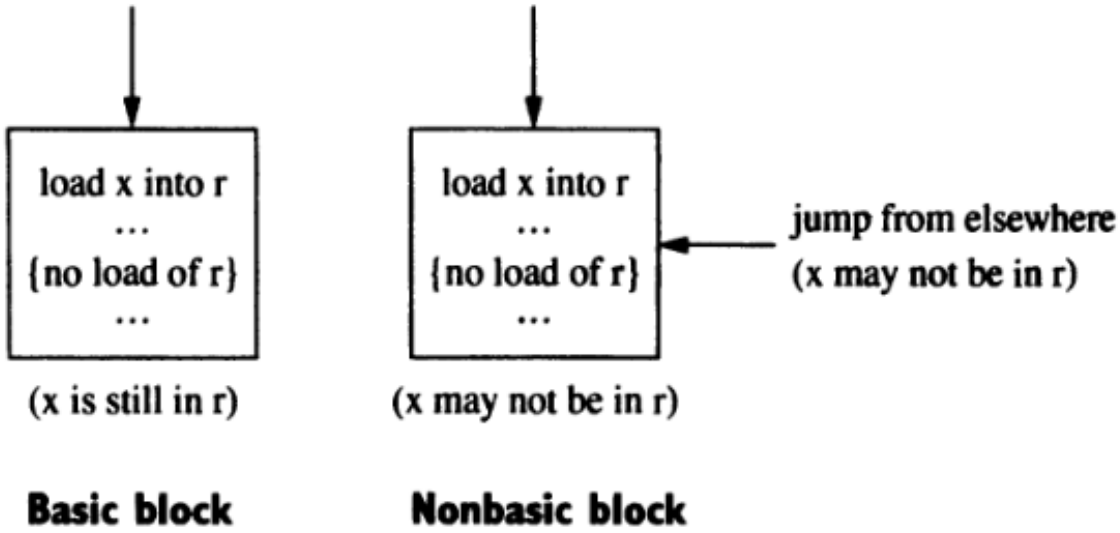
```
x = 1; ...
y = 0; ...
if (x) y = 1;
```

o que pode levar a novas oportunidades para propagação de constantes, etc.

Localidade das Otimizações

Considerando a área do programa sobre a qual atuam, as otimizações podem ser classificadas em:

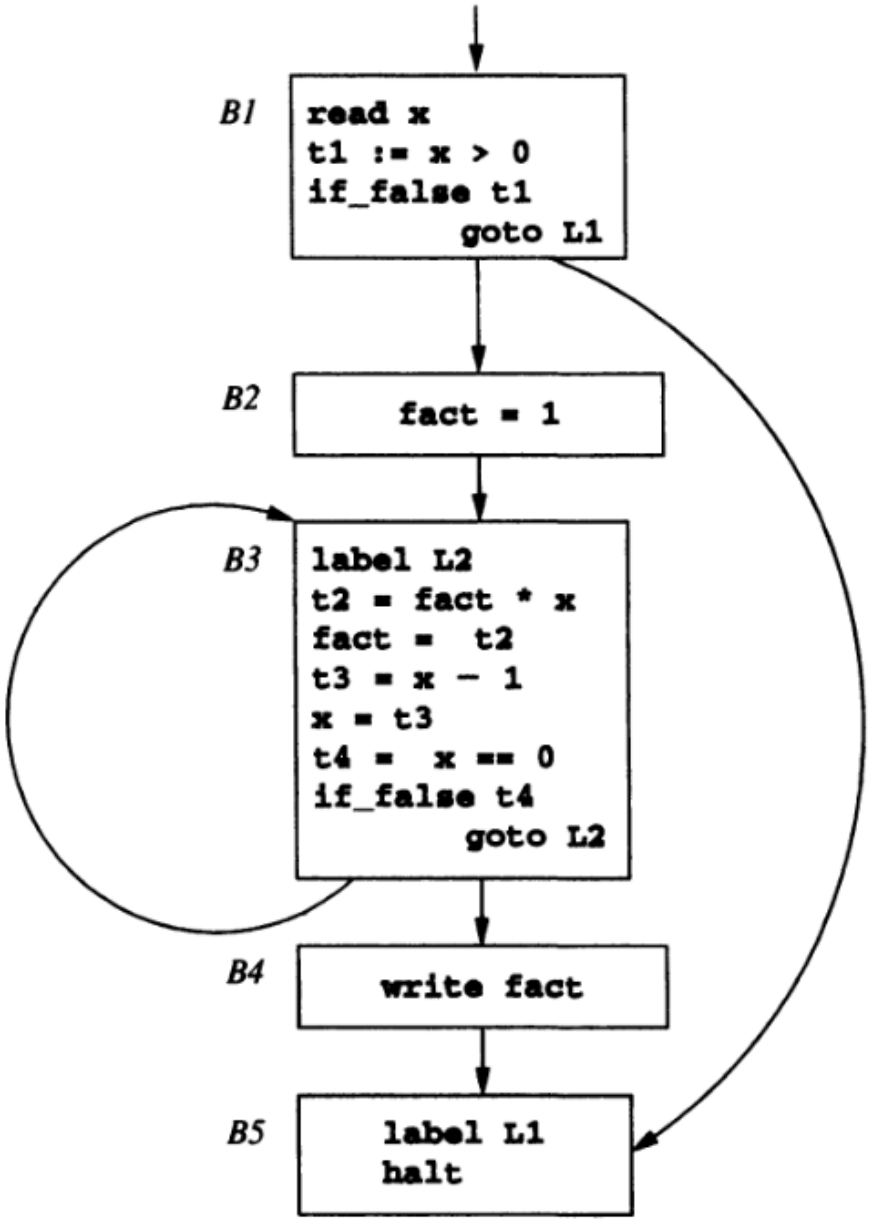
1. Locais: aplicadas para segmentos lineares de código (*basic blocks*).
2. Globais: aplicadas a uma função individual.
3. Inter-procedurais: aplicadas a todo o programa.



Estruturas e Técnicas para Otimizações

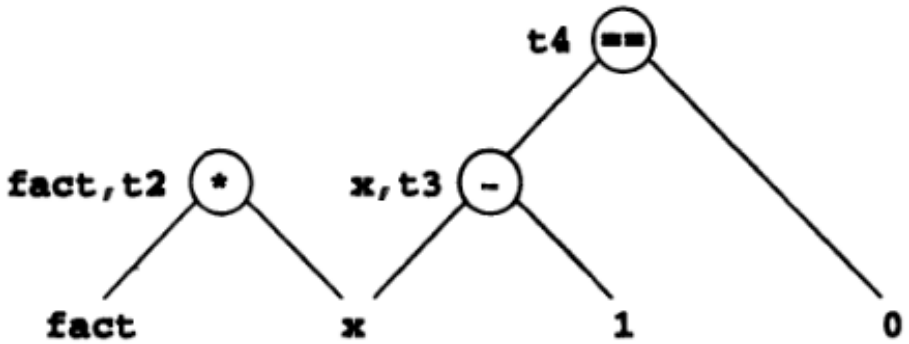
- Algumas otimizações como *constant folding* podem ser feitas diretamente na AST.
- No entanto, a AST na maioria dos casos não é uma estrutura adequada para a realização de otimizações.
- Um módulo que realiza otimizações globais constrói um *flow graph* a partir do IC de cada função.
- Os nós de um *flow graph* são os blocos básicos e os arcos representam saltos.
- Uma única passada pelo IC é suficiente para construir o *flow graph*.
- Cada novo bloco básico (BB) é identificado como a seguir:
  - A primeira instrução começa um novo BB.
  - Cada rótulo que é o alvo de um salto começa um novo BB.
  - Cada instrução que segue um salto começa um novo BB.

Flow Graph


B1
read x
t1 := x > 0
if_false t1
goto L1
B2
fact = 1
B3
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4
goto L2
B4
write fact
B5
label L1
halt

Estruturas e Técnicas para Otimizações

- Uma outra estrutura de dados que é frequentemente construída é o DAG de um bloco básico.
- O DAG acompanha a computação e atribuição de valores e variáveis em um bloco básico.
- Valores que são usados no bloco mas que vêm de outro local viram nós folha.
- Operações sobre os valores viram nós internos.
- Atribuição de um novo valor é representado associando-se o nome da variável alvo ao nó do DAG que contém o valor.
- Exemplo de DAG para o bloco B3 da figura anterior:

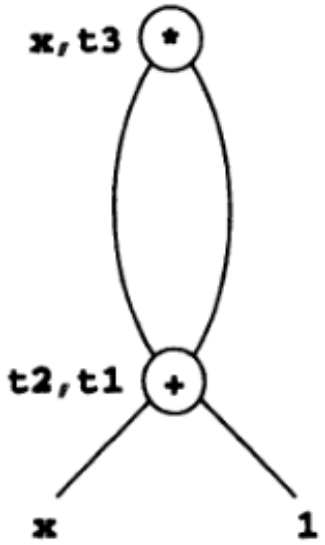


Repetições de valores também são representados no DAG.

Por exemplo, a atribuição em C, `x=(x+1)*(x+1)`, é traduzida nas seguintes instruções em código de três endereços.

```
t1 = x + 1
t2 = x + 1
t3 = t1 * t2
x = t3
```

O DAG do bloco ao lado mostra o uso repetido da expressão `x+1`.



Do DAG para Código Alvo

- O código alvo (ou uma versão revisada do IC) pode ser gerado a partir do DAG através de um caminhamento no DAG que respeite as dependências dos nós.
- Vários caminhamentos podem ser válidos. Nesse caso uma heurística escolhe o que deve gerar um menor número de temporários.
- Um caminhamento no DAG do slide anterior leva a:

```
t1 = x + 1
x = t1 * t1
```

- Usando um DAG para a geração de código alvo de um bloco básico, obtemos automaticamente uma otimização local de eliminação de sub-expressão.
- A representação em DAG também permite a eliminação de comandos de `store` redundantes, além de indicar quantas referências há para cada valor (útil na alocação de registradores).

Etapas de um *Back-end*

As ações realizadas por um *back-end* podem ser divididas em:

- **Seleção de instruções**: mapear IR (AST ou TAC) em código *assembly* real.
- **Escalonamento de instruções**: reordenar as operações levando em conta as latências para gerar código mais eficiente.
- **Alocação de registradores**: determinar quais valores ficam na memória e quais ficam nos registradores.

É interessante notar que:

- Todas as etapas são totalmente dependentes da arquitetura alvo.
- $\Rightarrow$ *Back-end* para uma arquitetura RISC pode implementar algoritmos muito distintos do que um para CISC.
- Problemas acima são NP-complete! $\Rightarrow$ Heurísticas.

Seleção de Instruções

- Problema: dada uma IR gerar um código *assembly* que leva o menor número de ciclos para executar.
- Solução usa alguma forma de casamento de padrões: se uma parte da IR tem um certo formato, emite código em um padrão associado.
- Essa solução pode fazer escolhas locais ótimas mas otimalidade global é NP-complete.
- Simplificação: assume que há registradores suficientes.
- Vamos assumir como linguagem alvo uma arquitetura RISC (próximo slide).
- Qualquer seleção de instruções minimamente realista deve levar em conta as latências das instruções.
- Latência de instrução: tempo entre o início da execução da instrução até a hora que os resultados estão disponíveis.
- Complicação: instruções podem ter latências diferentes – soma: 1 ciclo, multiplicação: 3-16 ciclos, etc.

Conjunto Básico de Instruções

```
load r1, @a      ; load register r1 with
                 ; the memory value for a
load r1, [r2]    ; load register r1 with the
                 ; memory value pointed by r2

mov r1, 3        ; r1=3

add r1, r2, r3 ; r1=r2+r3
(same for mult, sub, div)

shr r1, r1, 1    ; r1=r1>>1 (shift right; r1/2)

store r1         ; store r1 in memory

nop              ; no operation
```

Seleção de Instruções para Expressões Aritméticas

```
int expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case +, -, *, /:
      t1 = expr(left_child(node));
      t2 = expr(right_child(node));
      result = next_register();
      emit(op(node), result, t1, t2);
      break;
    case ID:
      t1 = offset(node) {
      result = next_register();
      emit(LOAD, result, t1);
      break;
    case NUM:
      result = next_register();
      emit(MOV, result, val(node));
      break;
  }
  return result;
}
```

Resultado para `x+y`:

```
load r1, @x
load r2, @y
add  r3, r1, r2
```

Comando `load` sempre carrega a partir de endereço base do *frame*, logo `@x` é o *offset*.

Código não leva em conta valores já carregados em registradores.

Número de Registradores vs. Desempenho

Considere os códigos abaixo gerados a partir da expressão `w = w*2*x*y*z`.

```
 1. load r1, @w              1. load r1, @w        ; load 5 cycles
 2. mov  r2, 2               2. load r2, @x
 6. mult r1,r1,r2            3. load r3, @y
 7. load r2, @x              4. load r4, @z
12. mult r1,r1,r2            5. mov  r5, 2         ; mov 1 cycle
13. load r2, @y              6. mult r1,r1,r5      ; mult 2 cycles
18. mult r1,r1,r2            8. mult r1,r1,r2
19. load r2, @z             10. mult r1,r1,r3
24. mult r1,r1,r2           12. mult r1,r1,r4
26. store r1                14. store r1          ; store 5 cycles
```

- Código à esquerda: 2 registradores, 31 ciclos.
- Código à direita: 5 registradores, 19 ciclos.
- Escalonamento de instruções (o problema de): dado um fragmento de código e as latências para cada operação, reordenar as operações para minimizar o tempo de execução.

Escalonamento de Instruções

Muitas operações têm latências para execução.

- Exemplos: `load`, `store`: *delay* de *N* ciclos de CPU (conforme arquitetura).
- Ao iniciar a operação, o resultado aparece *N* ciclos depois.
- Execução continua se o resultado não é referenciado.
- Referência prematura causa *hardware stall*.

Visão geral da solução:

- Mover `loads` para trás pelo menos *N slots* de onde eles são necessários.
- Problema: aumenta a pressão por registradores (*register pressure*), i.e., mais registradores podem ser necessários.
- Idealmente: minimizar *hardware stall* e *register pressure*. Nem sempre é possível pois são objetivos conflitantes.

Escalonamento de Instruções de um Bloco de Código

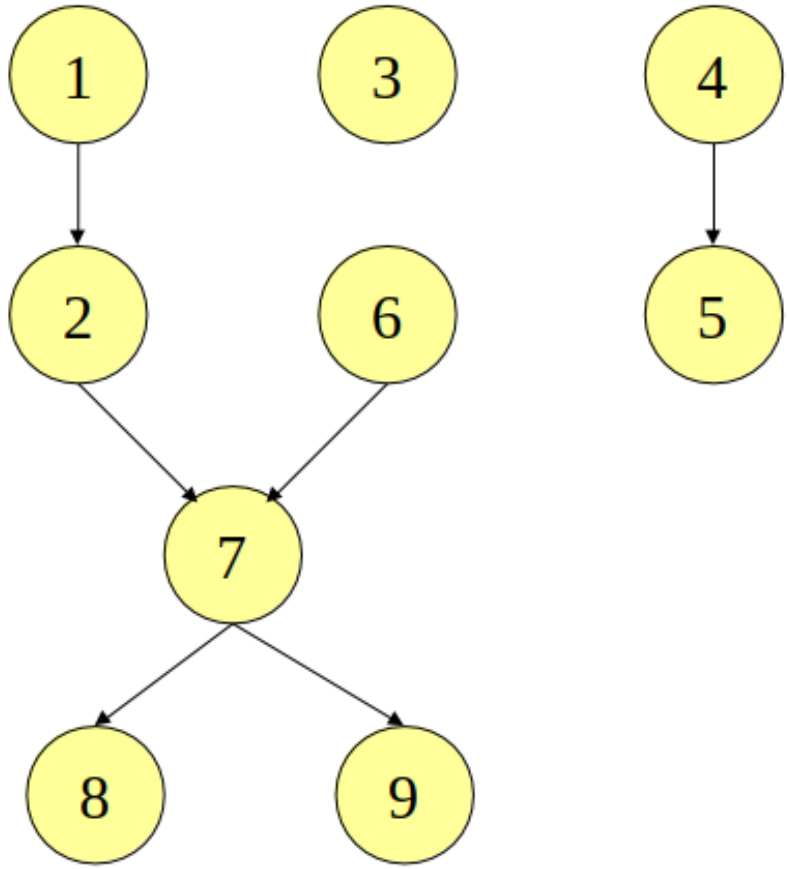
Sequência de passos para escalonamento de código de um bloco linear (sem saltos).

1. Construir um grafo de dependência dos dados.
2. Computar uma função de prioridade para cada nó do grafo.
3. Usar uma fila das operações que estão prontas para construir um escalonamento. A cada ciclo, faça:
   1. Escolha um operação pronta e escalone.
   2. Atualize a fila de operações prontas.

Algoritmo acima em geral é uma heurística gulosa, possível de variações.

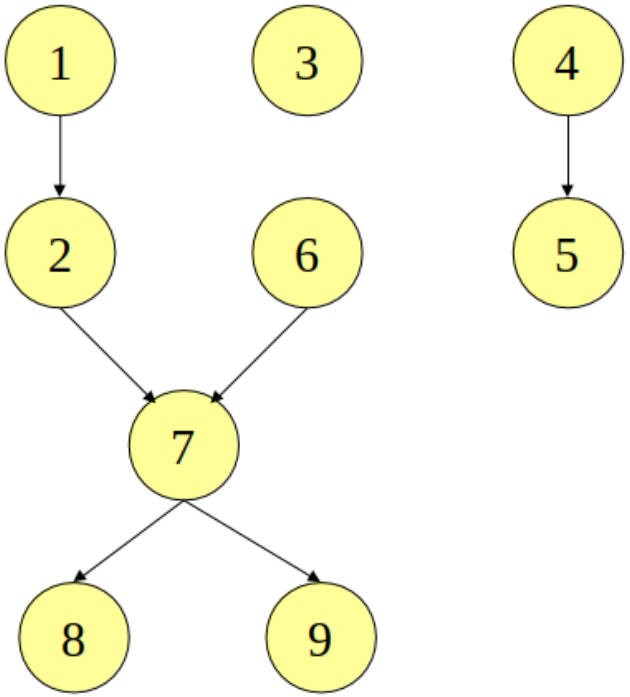
Construção do Grafo de Dependências

```
1. load r1, @x
2. load r2, [r1+4]
3. shr r3, r3, 2
4. mult r6, r6, 100
5. store r6
6. div r5, r5, 100
7. add r4, r2, r5
8. mult r5, r2, r4
9. store r4
```



Computando uma Função de Prioridade

- Para cada nó: associar um peso igual à latência máxima necessária para chegar em uma folha.
- Construção *bottom-up*: peso de uma folha é a latência.
- Peso de um nó interno é a sua latência mais o maior peso dos sucessores.
- Assumindo `div`: 4 ciclos, `mult`: 3 ciclos; demais: 1 ciclo.



Nó	Op	Peso
1	`load`	6
2	`load`	5
3	`shr`	1
4	`mult`	4
5	`store`	1
6	`div`	8
7	`add`	4
8	`mult`	3
9	`store`	1

Algoritmo Local de Escalonamento

```
cycle = 1;
Ready = set of available operations;
Active = {};
while (Ready U Active != {}) {
  if (Ready != {}) {
    remove an op from Ready (based on the weight);
    schedule(op) = cycle;
    Active = Active U {op};
  }
  cycle = cycle + 1;
  for each op in Active {
    if (schedule(op) + delay(op) <= cycle) {
      remove op from Active;
      for each immediate successor s of op {
        if (all operand of s are available) {
          Ready = Ready U {s};
        }
      }
    }
  }
```

Encontrando o Escalonamento

Cy.	Ready(Weight)	Schedule	Active
1	6(8), 1(6), 4(4), 3(1)	`div r5, r5, 100`	6
2	1(6), 4(4), 3(1)	`load r1, @x`	6, 1
3	2(5), 4(4), 3(1)	`load r2, [r1+4]`	6, 2
4	4(4), 3(1)	`mult r6, r6, 100`	6, 4
5	7(4), 3(1)	`add r4, r2, r5`	4, 7
6	8(3), 3(1), 9(1)	`mult r5, r2, r4`	4, 8
7	3(1), 9(1), 5(1)	`shr r3, r3, 2`	8, 3
8	9(1), 5(1)	`store r4`	8, 9
9	5(1)	`store r6`	5

Variações no Algoritmo de Escalonamento

Dois tipos distintos de escalonamento:

- *Forward*: começa com todas as operações disponíveis; trabalha avançando no tempo.
- *Backward*: começa das folhas; trabalha voltando no tempo.
- Usual: testar ambos e ficar com o melhor.

Variações da função de prioridade:

- Caminho máximo contendo nó (diminui o uso de registradores).
- Priorizar caminho crítico.
- Número de sucessores imediatos ou descendentes.
- Um longo etc...

Alocação de Registradores

- Assume uma IR linear: *three-address code*.
- TAC utiliza um número ilimitado de registradores virtuais (VR – *Virtual Registers*), AKA temporários.
- CPU possui um número limitado (por exemplo, k) de registradores físicos (PR – *Physical Registers*).
- Tarefa:
  - Produzir um código correto que use k registradores;
  - Minimizando o número de `loads` e `stores`;
  - Tomando o menor tempo possível para a alocação.

Alocação de Registradores – Definições

- **Bloco Básico (BB)**: segmento de código composto somente por comandos sequenciais (sem *branch*).
- **Alocação de Registradores Local (ARL)**: alocação feita dentro de um único bloco básico.
- **Alocação de Registradores Global (ARG)**: alocação feita dentro de uma função inteira (múltiplos BBs).
- **Alocação**: escolher o que manter nos registradores.
- **Atribuição**: escolher registradores específicos para certos valores.

Alocação de Registradores – Definições

- CPUs podem ter múltiplos tipos de registradores: propósito geral, ponto flutuante, etc.
- Problema: interações entre as classes.
- Em geral se considera uma alocação separada para cada classe. (Resolvido pelo sistema de tipos.)
- Qual é a complexidade do problema de alocação?
- Somente casos simples de alocação local podem ser resolvidos em tempo linear.
- Todo os demais casos (incluindo ARG e demais sub-problemas) são NP-complete.
- $\Rightarrow$ São necessárias heurísticas.

Tempo de Vida de uma Variável

- Problema: Quantos registradores são necessários para um BB?
- Solução simples: alocar todas as ocorrências de uma variável para o mesmo registrador.
- Solução realista: calcular o tempo de vida de cada variável.
- Uma variável é considerada viva entre a sua definição (atribuição) e o seu último uso.
- Pode-se representar o tempo de vida como um intervalo $[i, j]$ no BB, aonde i é a definição e j é último uso.
- Para cada BB, MAXLIVE é o número máximo de variáveis vivas considerando todas as instruções do bloco.
- Seja k o número de registradores físicos disponíveis.
  - Se MAXLIVE $\leq k$, alocação é trivial.
  - Se MAXLIVE $> k$, alguns valores precisam ser derramados (*spilled*) na memória.

Tempo de Vida de uma Variável – Exemplo

- Considere o código baixo, que utiliza 9 registradores.
- O tempo de vida de cada um está indicado na tabela.
- Qual o valor de MAXLIVE?

```
1. load r1, @a
2. load r2, @y
3. mult r3, r1, r2
4. load r4, @x
5. sub  r5, r4, r2
6. load r6, @z
7. mult r7, r5, r6
8. sub  r8, r7, r3
9. add  r9, r8, r1
```

r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*

MAXLIVE = 5: número máximo de células marcadas em uma dada coluna. Conclusão?

Alocação Local *Top-Down*

- Alocador precisa reservar f registradores para garantir que o problema é solucionável (viável).
- Esses f registradores são usados nas computações que envolvem valores alocados na memória.
- Em geral, $f = 2$ ou 4 na maioria das arquiteturas.
- Ideia (algoritmo de contagem de frequência): manter $k - f$ valores mais usados do BB nos registradores, usar f para os demais valores.
  1. Contar o número de usos de cada VRs.
  2. Associar $k - f$ VRs a PRs.
  3. Reescreve código:
     - Se VR passou para PR, substitui.
     - Senão, *spill*: use os PRs reservados para `load` antes do uso e `store` depois da definição.
- Problema: um valor muito usado na 1a. parte do BB e não usado na 2a. parte fica prendendo um registrador.

Alocação Local *Top-Down* – Exemplo

Assuma 3 PRs, com $f = 2$ para viabilidade.

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r3 //spill r3
5. load r3,@x
6. sub  r3,r3,r2
7. load r2,@z
8. mult r3,r3,r2
9. load r2, // spilled value
10.sub  r3,r3,r2
11.add  r3,r3,r1
```

r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*

- `r1` é alocado para o valor mais utilizado (há um empate entre `a` e `y`).
- `r2` e `r3` são usados para viabilidade.
- Algoritmo *bottom-up*: Sheldon Best (1955).

Alocação de Registradores – Considerações Finais

- BBs raramente existem isoladamente:
  - BB1: `store r17, @a` é seguido por:
  - BB2: `load r12, @a`.
  - Uso de grafo de controle de fluxo permite trocar `load` por `move`.
- Múltiplos BBs aumentam a complexidade:
  - Alocação Global de Registradores.
  - Geralmente resolvido com um algoritmo de coloração de grafos.
  - Fora do escopo desta disciplina.

Aula 08 – Otimização de Código

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**