Sistemas Embebidos

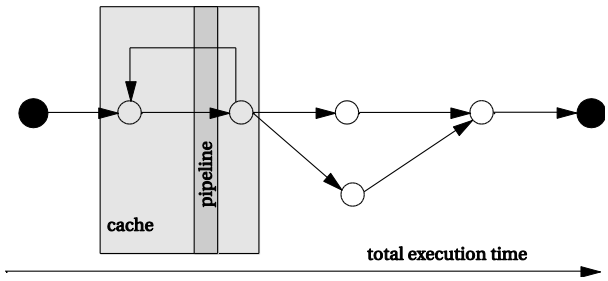
Análise e Desenvolvimento de Programas

Sumário

- Análise da performance ao nível do programa
- Otimização para:
  - Tempo de execução
  - Energia/Potência
  - Dimensão do programa
- Validação do programa e teste

Análise da performance ao nível do programa

- Requer a compreensão da performance em detalhe:
  - Comportamento em tempo real
  - Em plataformas complexas
- Performance do programa ≠ performance do CPU:
  - Pipeline e cache são "janelas" no programa
  - O programa deve ser analisado na sua globalidade



3

Complexidade da Performance do Programa

- Varia com os dados de entrada:
  - Diferentes *paths* (sequência de instruções executadas no programa)
- Efeitos da cache
- Variações da performance ao nível das instruções:
  - *Interlocks* do *pipeline* (mecanismo para detetar perigo e resolve-lo automaticamente)
  - Tempos de *fetch*

4

Como medir a performance de um programa?

- Simular a execução do CPU
  - Implica tornar o estado do CPU visível
- Medir num CPU real utilizando o *timer*
  - Requer alterar o programa para que controle o timer
- Medir num CPU real utilizando um analisador lógico
  - Requer eventos visíveis nos pinos do CPU

Timer – Função que pode ser ativada para extrair métricas do CPU

5

Métrica de medição de performance

- Tempo de execução do caso médio
  - Tipicamente utilizado na programação de aplicações genéricas!
- Tempo de execução do pior caso
  - Componente para satisfação das restrições temporais! Importante para sistemas em tempo real.
- Tempo de execução do melhor caso
  - Imprevisibilidade das interações ao nível das tarefas podem alterar que o melhor caso resulte no pior caso! Importante para sistemas multi-rate.

6

Elementos da performance do programa

- Fórmula básica do tempo de execução de um programa:
  - Tempo de execução = *program path + instruction timing*
- A resolução destes problemas de forma independente ajuda a simplificar a análise.
  - Mais fácil separar em *CPUs* mais simples
- Uma análise da performance detalhada requer:
  - Código *Assembly/Binário*
  - Plataforma de execução

7

Path é o caminho da sequência de instruções executadas pelo programa.

7

Paths de dados dependentes numa estrutura if

```
if (a) { /* T1 */
    if (b) { /* T2 */
          x = r * s + t; /* A1 */
          }
    else {
          y = r + s; /* A2 */
          }
    z = r + s + u; /* A3 */
    }
else {
    if (c) { /* T3 */
          y = r – t; /* A4 */
          }
    }
```

a	b	c	path
0	0	0	T1=F, T3=F: no assignments
0	0	1	T1=F, T3=T: A4
0	1	0	T1=F, T3=F: no assignments
0	1	1	T1=F, T3=T: A4
1	0	0	T1=T, T2=F: A2, A3
1	0	1	T1=T, T2=F: A2, A3
1	1	0	T1=T, T2=T: A1, A3
1	1	1	T1=T, T2=T: A1, A3

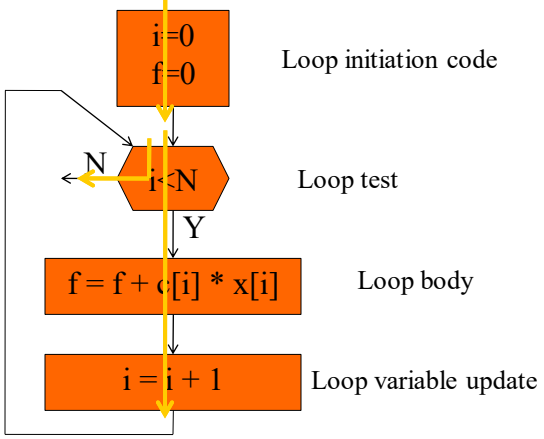
Os testes condicionais (Tx) e atribuições (Ax) são etiquetados dentro de cada **if** para tornar mais fácil a identificação dos *paths*

8

8

Paths num Loop (for)

```
for (i=0, f=0; i<N; i++)
    f = f + c[i] * x[i];
```



Mesmo otimizando o gráfico CDFG (Gráfico para controlo de fluxo de dados), o compilador pode alterar a estrutura do fluxo de controlo/dados para maior otimização

9

Instruction timing

- Nem todas as instruções necessitam do mesmo tempo de execução
  - Instruções multi-ciclo
  - *Fetches*
- Os tempos de execução das instruções não são independentes
  - *Pipeline interlocks*
  - Efeitos da Cache
- Os tempos de execução variam com o valor do operando
  - Operações de vírgula-flutoante
  - Operações inteiras multi-ciclo

10

Simulação do *CPU*

- Alguns simuladores são pouco precisos
- Simulador *cycle-accurate* permite temporização precisa do ciclo do relógio
  - O simulador modela os registos internos do CPU
  - O simulador deve conhecer o modo de funcionamento do CPU

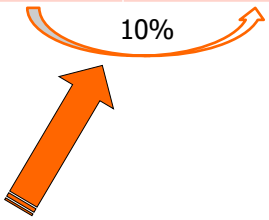
11

11

Simulação de um Filtro FIR (*SimpleScalar*)

```
int x[N] = {8, 17, … };
int c[N] = {1, 2, … };
main() {
   int i, k, f;
   for (k=0; k<COUNT; k++)
        for (i=0; i<N; i++)
             f += c[i]*x[i];
}
```

N	total sim cycles	sim cycles per filter execution
100	25854	259
1,000	155759	156
10,000	1451840	145



10% são valores razoavelmente próximos do tempo de execução real do filtro FIR

12

12

Programas e análise da performance

- Os melhores resultados derivam da análise de instruções otimizadas, não de código *HLL*:
  - Não se tratam de traduções obvias de código *HLL* em instruções
  - O código pode mover-se
  - Os efeitos da cache são difíceis de prever

HLL - High-level programming language 13

13

Otimização de Loops

- Os *loops* são bons alvos de otimização
- Otimizações básicas dos *loops*:
  - Movimento do código;
  - Eliminação das variáveis de indução (variável que aumenta ou decresce por uma quantidade fixa em cada iteração do *loop*);
  - Redução da "*strength*" (x*2 → x<<1)
    - Operações mais dispendiosas são trocadas por operações equivalentes mas de menor custo de processamento

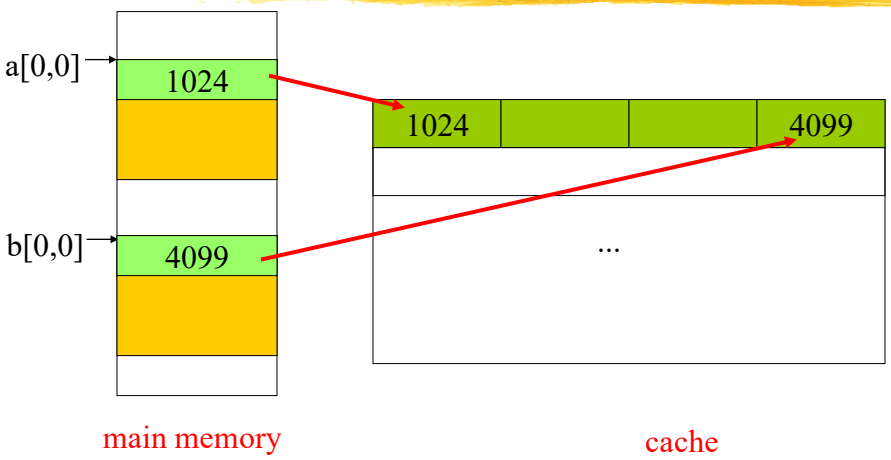
Strength – resistência 14

14

Análise da Cache

- *Loop nest*: conjunto de loops, dentro de outro loop.
- *Perfect loop nest*: sem estruturas condicionais no nest.
- Uma vez que os *loops* utilizam grandes quantidades de dados, os conflitos da cache são comuns.

Conflitos de *arrays* na *cache*



Não mapeiem para a mesma palavra na cache, mas mapeiam para o mesmo bloco, originando conflitos

Otimização de energia /potência

- Energia: capacidade de realizar trabalho
  - Mais importante em sistemas alimentados por bateria
- Potência: energia por unidade de tempo
  - Importante porque mesmo nos sistemas alimentados por corrente, a potência converte-se em calor (lei de *joule* P=RI^2)

*Potência (P) = Resistência elétrica (R) * a Corrente elétrica (I) ^2*

17

17

Fontes de consumo de energia

- Energia relativa por operação (ciclos relógio):
  - Transferência de memória: 33
  - I/O externo: 10
  - Escrita na SRAM: 9
  - Leitura na SRAM: 4.4
  - multiplicação: 3.6
  - soma: 1

18

18

Comportamento da Cache

- O consumo de energia varia com a dimensão das caches:
  - cache muito pequena: programa com performance reduzida - colapso, gasto de energia em acessos à memória principal.
  - cache muito grande: a cache gasta demasiada energia.

19

Otimização de energia

- Uso de registos eficientemente
- Identificar e eliminar conflitos da cache
- Moderação dos *loops*
- Eliminar os hiatos *(gaps)* do *pipeline*

20

Loops eficientes

- Regras gerais:
  - Não utilizar chamadas de funções
  - Manter o corpo do *loop* pequeno para permitir repetições locais (*branches*)
  - Utilizar *unsigned integer (range 0-65535,* não pode ser negativo*)* para o contador do *loop*
  - Utilizar o **<=** para testar o contador do *loop*
  - Fazer uso do compilador - otimização global, software *pipelining*

21

Otimização para o tamanho do programa

- Objetivo:
  - Reduzir o custo do hardware de memória
  - Reduzir o consumo de energia das unidades de memória
- Duas oportunidades:
  - Memória de dados
  - Memória de instruções

22

Validação e teste do programa

- O programa funciona?
- Focar a verificação funcional
- Estratégias de teste:
  - *Clear box (white box)* – analisar o código fonte
  - *Black box* – não analisar o código fonte

Teste *Clear-box*

- Examinar o código fonte para determinar se este funciona:
  - Consegue exercer um *path*?
  - Obtém o valor que espera ao longo do *path*?
- Procedimento de teste:
  - Controlabilidade: fornecer ao programa inputs
  - Execução
  - Observabilidade: examinar os outputs

Teste de *Black-box*

- Complementa o teste *clear-box*
  - Pode requerer um grande número de testes
- Testa o software em diferentes formas
  - *Random*
  - Regressão (com base no histórico)

Vectores de Teste do *Black-box*

- Testes *random*
  - Considera a distribuição dos dados baseada na especificação do software
- Testes de Regressão
  - Testes de versões anteriores, *bugs*, etc
  - Podem ser testes *clear-box* de versões anteriores

Quanto é necessário testar?

- Teste exaustivo é impraticável
- Uma importante medida da qualidade do teste – há sempre *bugs* que escapam para o mercado
- Grandes organizações testam o software de forma a obter taxas de bugs muito baixas (MS, Apple, Sun…)
- Injeção de erros mede a qualidade do teste:
  - Adiciona bugs conhecidos
  - Executa os testes pretendidos
  - Determina a % de bugs injetados que são detetados

Sistemas Embebidos

Resolução da FT5