Projeto Cache

Arquitetura de Computadores

Raissa Camelo, Mayara Castro, Jonas Freire, Giuseppe
Departamento de Computação
Universidade Federal Rural de Pernambuco
Recife PE Brasil

ABSTRACT

Este documento contempla os resultados obtidos no projeto 4 da disciplina Arquitetura de Computadores, que consiste na implementação de dois algoritmos de ordenação (Radix e Quick Sort) na linguagem Assembly Mips e sua simulação no ambiente MARS. A simulação visa demonstrar a eficiência de cada algoritmo em relação a configuração da memória cache e o tamanho do array ordenado.

1. ALGORITMOS

1.1 Radix Sort

O Radix sort se trata de um algoritmo de ordenação de itens representados por chaves únicas, compostas de cadeias de caracteres ou valores numéricos. Utiliza-se para a ordenação um array auxiliar (Bucket) cujas posições correspondem a cada dígito da base decimal.

A seguir encontra-se uma breve explicação do algoritmo Radix Sort implementado, junto a alguns trechos importantes do código. Utilizamos a filosofia LSD (Least Significant Digit).

1. Achar o maior valor do Array
2. Loop Principal:

Esse loop itera o número de vezes correspondente a quantidade de dígitos do maior valor. A cada iteração o valor do expoente é multiplicado por 10. O expoente indica qual casa decimal está sendo observada na iteração. Os demais trechos de código do Radix estão contidos neste loop.

1. loop_principal:
2. contador_de_ocorrencias:
3. reposicionador: (somador_bucket)
4. por_no_array

3. Loop contador de ocorrencias

Neste loop os valores do Array são lidos e a ocorrência de cada dígito na casa decimal lida é contada . É definido em qual "caixa" o valor será colocado, incrementando em +1, a posição equivalente ao dígito no array do bucket.

```
1.  contador_de_ocorrencias:
2.  lw $t4, array_to_sort($t0) #Pega o valor do array
3.  div $t1,$t4,$s1            #Divide pelo expoente
4.  div $t1,$t1, 10            #Mod 10
5.  mfhi $t1                   #Pega o resultado
6.  mul $t1,$t1,4              #(4Bytes por inteiro)
7.  lw $t4, bucket($t1)        #Salva na posição
8.  addi $t4, $t4,1
9.  sw $t4, bucket($t1)
10. addi $t0,$t0,4
11. bne $t0, $s0,contador_de_ocorrencias
```

4. Loop Reposicionador

Neste loop os valores contidos no bucket são atualizados, cada posição do buket recebe a soma dos valores contidos nas posições anteriores. Este procedimento determina o deslocamento que cada valor do array original irá sofrer, a fim de arranjá-lo em uma nova posição (ordenada).

```
1.  somador_bucket:
2.  lw $t4, bucket($t0)     #Bucket[i]
3.  lw $t5, bucket($t1)     #Bucket[i-1]
4.  add $t6, $t4,$t5        #B[i] += B[i-1}]
5.  sw $t6, bucket($t0)     #Salve
6.
7.  addi $t0, $t0, 4
8.  addi $t1, $t1, 4
9.  bne $t0, $s3, somador_bucket
```

1.2 Quick Sort

O Quick Sort é um algoritmo de ordenação que funciona com a técnica "dividir para conquistar", particionando o array a ser ordenado em metades e rearranjando os valores de acordo com um valor pivô, escolhido no início do código. O rearranjo é feito de forma que todos os valores menores que o pivô fiquem antes do mesmo e os maiores, depois. A seguir alguns dos principais trechos do código.

1. Setar ponteiros e determinar pivô
    1. lw $s0, begin # $s0 = begin

2. lw $s1, end # $s1 = end
3. sw $s2, pivot # pivot = array[begin]

A principio o pivô

2. Particionamento do Array
    1. addi $sp,$sp,-12 # Abre espaço na pilha
    2. sw $ra,0($sp) # endereço da chamada
    3. sw $s0,4($sp) # lado direito
    4. sw $s1,8($sp)
    5.
    6. sll $t0,$s0,2 # s0 é o indice do valor mais a esquerda
    7. add $t0,$t0,$s2 #
    8. lw $t0,0($t0) # Difine t0 como novo pivot
    9. addi $s1,$s1,1

Para particionar o Array inicialmente abre-se espaço na pilha (Stack Pointer) pois como a função é recursiva, faz-se necessário salvar os valores das chamadas anteriores em pilha. Em seguida, é calculado o novo pivot do Array particionado.

3. Swap entre valores do Array
    1. loop1:# Itera enquanto indice esq < indice dirt
    2. addi $s0,$s0,1 # indice esquerda++
    3. sll $t1, $s0, 2 # calcula posição em bytes
    4. add $t1,$t1,$s2 # Desloca o array pro indice s0
    5. lw $t1,0($t1) # $t1 = A[$s0]
    6. slt $t2,$t1,$t0 # comparando A[$s0] com o pivô
    7. bnez $t2, comparaindice # t2 == 0, compare o indice
    8. sub $t2,$t1,$t0 # A[$s0] == pivot?
    9. bnez $t2, loop2 # loop 2 caso igual
    10. comparaindice:
    11. slt $t2,$s0,$s1 # comparando es q e dir
    12. bnez $t2,loop1 # Rodar enquanto esq < dir
    13. sub $t2,$s0,$s1# indices iguais loop1
    14. beqz $t2,loop1
    15.

Este é o loop principal do código. Iterando até os ponteiros do valor mais esquerdo e mais direito array se tornarem iguais. Dentro desse loop é feito os swaps entre os valores menores que o pivot (para a esquerda) e os valores maiores que o pivot estabelecido (para a direita). Este loop chama também a segunda iteração de partição (segunda partição do array).

2.CONFIGURAÇÕES DA MEMÓRIA CACHE

Neste projeto optamos por utilizar uma memória cache de 512 Bytes, com blocos de 4 palavras e, associatividade 4-Way com política LRU. Espera-se diminuir a taxa de Miss em ambos os algoritmos, uma vez que com mais blocos por linhas na cache, maior será a taxa de hit.

Inicialmente o tamanho de array utilizado foi 30, dobrando o valor a cada nova iteração, durante oito iterações, a última com um array de tamanho 7680.

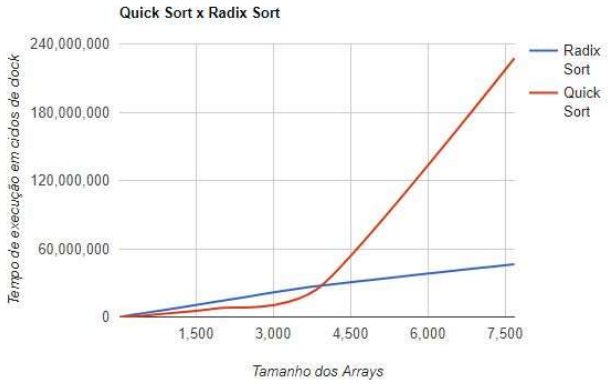
3. RESULTADOS

Dos testes realizados na memória cache configurada como dito no tópico anterior, abaixo alguns dos resultados:

Tabela 01 - Taxas de falta e acerto por tamanho de array

Array Size	RADIX		QUICK	
	HIT %	MiSS %	HIT %	MISS %
30	1576	19	738	20
60	2906	99	1503	21503
3840	15371	26949	374809	23577
7680	314713	46432	1070191	226214

Logo em seguida encontra-se um gráfico comparando o tempo que cada algoritmo gastou em ciclos de clock x o tamanho de array de cada execução.



4. CONCLUSÃO

O algoritmo quicksort apesar de inicialmente possuir o tempo de resposta semelhante ao do radix, perde gradualmente sua eficiência a medida que o tamanho do array aumenta. A partir de aproximadamente 3,000 valores inteiros por array, o quicksort começa a demonstrar seu comportamento de pior caso ($O(n^2)$). A taxa de miss entretanto só obteve uma melhora significativa para o Radix sort, em detrimento de uma Cache com mapeamento direto.