Prática Avulsa Memória Cache – Versão para Beta-Test

Nesta prática, faremos a compilação de programas em linguagem C para assembly do MIPS, usando o *cross compiler* Lcc. Usaremos o MARS para os testes, com a ferramenta de simulação de cache de dados.

Não custa lembrar que, num dia-a-dia de programador orelha seca, a preocupação com a cache é desnecessária. Isso muda apenas quando trabalhamos com um volume muito grande de dados ou quando um alto desempenho é crucial para o sistema.

Teremos duas semanas para trabalhar à vontade nesta prática no laboratório. É preciso escolher uma das três opções para entregar: a) um ***EXTRA*** qualquer; b) um "▸" que não seja com mergesort ou quicksort; ou (c) duas perguntas do questionário. Mas façam a prática inteira, claro. Não precisa relatório, só as respostas e dados, mas obviamente vocês devem entendê-los.

Dois lembretes: antes de mais nada, estamos com exemplos pequenos pra simulação ir rápido; na vida real, com outra escala, alguns problemas são minimizados. E vários problemas de acesso podem passar desapercebidos com uma cache multinível, que diminui o custo das falhas.

Compilando Linguagem C para o MIPS

Tenho três sugestões pra vocês:

1. Servidor de e-mail em c2mips@gmail.com[1]

Basta enviar um e-mail com o assunto contendo a palavra *compilar* e o arquivo em C em anexo. Se não houver algum anexo *.c, o servidor procura um texto simples contendo a palavra *main* e finalmente um texto em Html. O programa compilado para assembly e adaptado ao MARS deve surgir na sua caixa de entrada em poucos segundos (clique no reload). Não funciona com respostas ou encaminhar. Tenho um script pra automatizar isso, se quiserem[2].

Infelizmente, você provavelmente deverá fazer um filtro *impedindo explicitamente* os e-mails do endereço c2mips@gmail.com de irem parar na sua caixa de SPAM. Lembre-se disso.

2. Script instalado no Linux do laboratório

Como salvaguarda, instalei o script *c2mips.py*, que converte a saída do compilador Lcc para o formato do MARS. Para usá-lo para compilar um arquivo *prog.c*, digite `c2mips.py prog.c` no terminal.

OPÇÃO 2 INDISPONÍVEL
lab em estado precário

Se você quiser usar o compilador diretamente, o comando é `lcc -S -Wf-g1,# prog.c`, que gera uma saída `prog.s`. Esta saída tem que ser adaptada ao MARS (é bem fácil mas é chato, veja a descrição se quiser[3]).

3. Instalação customizada

Se você quer usar seu laptop, instale no seu Linux o python, o Lcc e o meu script c2mips.py, seguindo instruções da minha página[4]. É pra ser tranquilo. Aí basta usar como descrito acima.

Agora, se você é fã do Mr Gates, uma opção ao Lcc é aprender a instalar o gcc cross para MIPS[5]. Isso também funciona pra Linux, mas exige a compilação do gcc-mips, que talvez dê trabalho. Instruções não testadas estão na página[6].

---

1 "E-mail? Mas por que e-mail?" Porque eu sou manco pra caramba e isso é um jeito mais seguro de vocês não hackearem. Mesmo assim, tiveram que me avisar pra cuidar com o Bobby Tables (https://xkcd.com/327/)

2 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/compilaremoto.py

3 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#lcc2mars

4 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#install

5 Há o lcc-win32 que infelizmente *não é um cross compiler*, o que significa que só produz código para x86. Pode ser usada a versão "normal" do Lcc que produz código pra MIPS e tem versão Windows, só que requisita a presença do Visual C++.

6 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#installgcc

Simulação no MARS

Para o primeiro teste, usaremos um quicksort não otimizado para levantar algumas estatísticas sobre o comportamento da cache. Os passos são os seguintes:

1. Obtenha o arquivo quicksort.c da minha página[7] (ou ache outro e retire as bibliotecas)
2. Invoque o script no terminal de comandos com ./c2mips.py quicksort.c
3. Abra no MARS o arquivo quicksort.asm e compile com F3
4. Acesse no menu Tools => Data Cache Simulator
5. Na janela do simulador, conecte ao MIPS (botão conectar, claro)
6. Execute na velocidade máxima com F5 e observe o comportamento da cache

O script de compilação avisará sobre erros ou sucesso do processo. Note que a simulação da cache é só de dados, não simulando o acesso à memória para busca das instruções. Para uma cache única com dados e instruções juntos, é preciso uma simulação mais completa.

É importante notar que os exemplos são reduzidos em relação à vida real (cache só de 512 bytes?! Claro que não existe), pra que a simulação não mate a todos de tédio.

Observe que o código em assembly não é muito fácil de ser lido: os nomes de funções e variáveis globais permanecem, mas os de variáveis locais são perdidos. Na seção final desta prática há uma explicação de como criar programas próprios e botar pra rodar.

▶	Simule o quicksort[8] com a configuração *default* da cache (mapeamento direto, 8 blocos com 4 words por bloco), observando os resultados e anotando o *hit rate*. Aumente o número de blocos para 16 e anote a *hit rate*. Com 16 blocos ainda, aumente cada bloco para 8 words por bloco. Compare a taxa de acertos para cada uma das três situações e explique a variação. Esta variação é significativa?

Agora, com qualquer configuração de cache, ligue a ferramenta de estatísticas de instruções executadas (Tools=>Instruction Statistics – não confunda com o Instruction *Counter),* conecte-a ao MARS e anote os números de total de instruções e de instruções de memória para o quicksort. Feche esta janela quando não estiver em uso pois ela deixa a execução lenta – cerca de um minuto na minha máquina. A seguir, obtenha o mergesort.c da minha página[9] e repita todo o processo.

▶	Comparando o quicksort e o mergesort, podemos observar vários fatos interessantes. Em especial, responda: • Será que o quicksort é realmente mais rápido que o mergesort? Por quê? Verifique os números apresentados. • Se o número de acessos à memória deles é praticamente igual, o desempenho final depende do sistema de memória ou não? • Por que a variação do número de blocos afeta tanto a taxa de acertos? • A variação do tamanho do bloco produz uma melhora significativa na *hit rate*? • Comparando o quicksort e o mergesort, a *hit rate* é melhor para o quicksort de forma consistente? Explique.

EXTRAS:

(a) Simule o quicksort para 2000 elementos com as três configurações de cache propostas e compare com o de 200 (demora alguns minutos[10]). Agora altere a inicialização do vetor (linha 39) para ao invés de usar números entre 0 e 99, usar qualquer inteiro e simule com 2000 elementos, comparando com o anterior. Explique o que se observa.

---

7 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/fontes/quicksort.c

8 Minha versão gera 200 números randômicos a ordenar, o que produz um "caso médio"; se isso não te convencer, simule umas três vezes pra ver que a variação é pequena

9 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/fontes/mergesort.c

10 Qualquer atualização da tela deixa o processo bem mais lento, então reduza isso, talvez até minimizando a janela

(b) Ache um radixsort em C, compile e simule para 200 elementos nas três configurações e compare com o quicksort. Explique as diferenças observadas (quem é mais rápido, quem acessa melhor a cache)[11].

Testes de Associatividade

Primeiramente, altere o tamanho do vetor do mergesort para exatos 256 elementos (basta alterar a primeira linha de código para `#define SIZE 256`). Compile e refaça a simulação anterior, comparando as taxas de acerto.

▶	Descubra uma explicação para os números obtidos.

Mantendo a última configuração (16 blocos de 8 words cada), altere *"Placement Policy"* de *"Direct Mapping"* para *"N-way Associative"*. Agora podemos alterar a associatividade em *"Set Size (blocks)"*. Podemos variá-la desde 1 (igual ao mapeamento direto) até 16 (igual a totalmente associativa).

▶	Meça a taxa de acertos para o mergesort de 256 inteiros variando a associatividade de 1 a 16 vias. Explique a variação, em especial a melhora brutal do mapeamento em *n* vias para o mapeamento direto, para qualquer n>1.

Se quiser comprovar a melhora praticamente insignificante do algoritmo LRU *(Least Recently Used)* em relação ao randômico para a escolha do bloco a descartar, altere a *"Block Replacement Policy"* e verifique isso.

▶	Para o quicksort original de 200 inteiros, mantendo a configuração de 16 blocos de 8 words cada, varie a associatividade de 1 a 16 vias. Repita a experiência para 256 inteiros. Explique por que razão a melhora foi nitidamente inferior à que pôde ser observada no caso do mergesort.

EXTRAS: Simule o radixsort com 200 elementos usando as configurações: a) 8 blocos de 4 words e associatividade de 2 vias; b) 16 blocos de 8 words e associatividade de 2 vias e 4 vias; e c) 16 blocos de 16 words e associatividade de 2 vias e 4 vias. O que se pode concluir a respeito do radixsort com estas medições? Compare-as entre elas e com o mapeamento direto.

Checksum

O programa a seguir faz um checksum de um vetor de 100 elementos: é apenas a soma de todos os valores armazenados, para fins de conferência de integridade dos dados. Ele está pronto pra ser baixado lá na minha página.[12] As listagens seguintes contém o essencial; os arquivos online possuem pequenas variações.

```
int vetor[100];
// permutacao dos numeros de 0 a 99  para acesso randomico
int mapa[100] = {19, 45, 16, 21, 0, 87, 69, 20, 66, 2, 56, 96, 1, 92, 47, 72, 98,
27, 35, 9, 3, 43, 26, 71, 4, 48, 82, 65, 88, 8, 55, 10, 61, 46, 99, 93, 12, 83,
74, 58, 77, 54, 33, 13, 39, 7, 85, 25, 95, 11, 90, 64, 17, 24, 14, 29, 70, 91, 81,
23, 44, 89, 6, 37, 34, 52, 50, 49, 63, 60, 18, 5, 32, 31, 76, 84, 79, 68, 51, 38,
75, 53, 28, 78, 94, 42, 57, 36, 67, 80, 41, 62, 59, 86, 30, 15, 73, 40, 97, 22};

void linear_checksum(void)
{
        int i, soma, pos;
        soma=0;
```

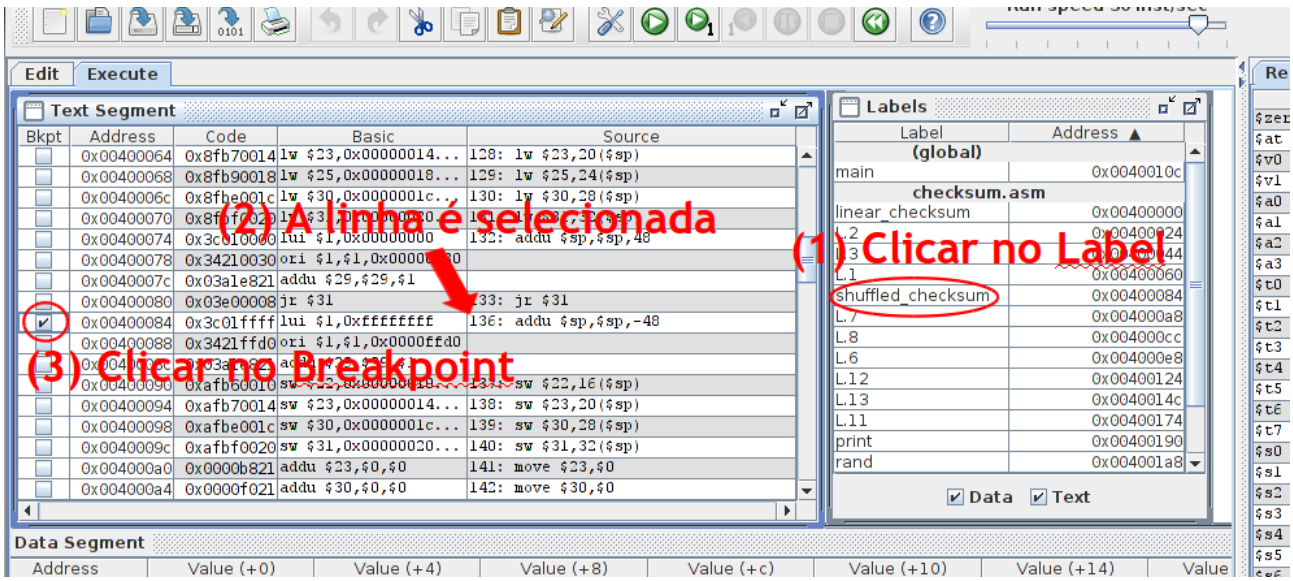
11 Note, em especial, que o desempenho do radixsort depende radicalmente do *número de dígitos dos elementos a ordenar* (é diretamente proporcional). Sorteando elementos entre 0 e 99 faz o algoritmo ser muito mais rápido do que se sortearmos inteiros de 31 bits sem sinal. Pode ser legal você dar uma testada nisso.

12 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/fontes/checksum.c

```
        for(i=0;i<100;i++)
        {
                // faz este acesso a mapa[] para ter numero de acessos igual
                // ao shuffled_checksum
                pos = mapa[i];
                soma = soma + vetor[i];
        }
        print(soma);
}

void shuffled_checksum(void)
{
        int i, soma, pos;
        soma=0;
        for(i=0;i<100;i++)
        {
                pos = mapa[i];
                soma = soma + vetor[pos];
        }
        print(soma);
}

int main()
{
        int i;
        for(i=0; i<100; i++)
                vetor[i] = rand()%10;

        linear_checksum();
        shuffled_checksum();

        exit(-1);
        return 0;
}
```

Resolvi fazer de duas formas: o linear e o desordenado, que pula de um lado para o outro. Vamos fazer a simulação *lentamente*. Preparação:

1. Compila o negócio, gere o checksum.asm e abra no MARS
2. Compile no MARS e vá para a aba *Execute*
3. Localize o endereço do label `linear_checksum`: na janelinha de labels, basta achar ele e dar um duplo-clique que o MARS vai até ele na janela de código (se a janela de labels não estiver visível: Settings => Show Labels Windows)
4. Insira um *breakpoint* nele, clicando no quadradinho à esquerda na linha da instrução: um sinal de check vai aparecer (deve ser no endereço 0x0040 0000)
5. Idem para `shuffled_checksum` (deve ser 0x0040 0084)
6. Abra o simulador de cache (Tools => Data Cache Simulator) e pressione "Connect to MIPS"; use a configuração padrão (mapeamento direto, LRU, 8 blocos de 4 words cada)



Simulação, em três etapas. Primeiro, inicialização do vetor randômico:

1. Execute passo a passo até fazer três acessos à memória; como a cache está vazia, o primeiro acesso a qualquer bloco causa uma falha, pois ele deve ser puxado da memória principal
2. Deixe a velocidade de execução no máximo
3. Execute ("Run", F5) até o *breakpoint*
4. Anote a taxa de acertos ("Cache Hit Rate")

Observe que as falhas iniciais, inevitáveis porque a cache ainda está vazia, são denominadas **Falhas Compulsórias**. Em segundo lugar, faça o cálculo do checksum com varredura linear:

1. Pressione "Reset" na janela do simulador de cache: isso limpa a cache e zera os contadores
2. Execute a partir do primeiro *breakpoint* até o outro *breakpoint* e anote a taxa de acertos

Finalmente, faça o cálculo do checksum com varredura aleatória.

1. Pressione "Reset" na janela do simulador de cache
2. Execute a partir do segundo *breakpoint* e anote a taxa de acertos

▶	Com a simulação acima, temos uma diferença na taxa de acertos das três seções do programa: a inicialização (1), o checksum linear (2) e o checksum aleatório (3). Por que a pequena diferença entre 1 e 2? Por que 3 é *tão* pior?

Falhas de cache que ocorrem porque a memória não é grande o suficiente são chamadas **Falhas de Capacidade**. Se aumentamos o tamanho, potencialmente até o infinito, elas desaparecem.

▶	(A) Meça a taxa de acertos *sem resetar a cache*, isto é, execute o programa inteiro e anote a *hit rate* final, sem usar breakpoints. Agora aumente o tamanho da cache para 32 blocos de 4 words, possuindo um total de 512 bytes. (B) Refaça a experiência com o reset da cache. (C) Refaça a experiência sem resetar a cache. Explique os números obtidos, em especial a diferença entre B e C, entre A e C e entre B e a simulação original. Quais as causas destas variações?

Completando o "Modelo dos 3 C's", as **Falhas de Conflito** são devidas ao mapeamento de dois ou mais dados para o mesmo endereço da cache. Estas falhas são reduzidas aumentando-se a associatividade ou o tamanho da cache.

Otimização de Código

Dá um bico no programa de multiplicação de matrizes abaixo[13]. A ordem do loop (k-i-j) está otimizada.

```
#define TAMANHO 100
#define MAX_VALOR 10
int matriz_a[TAMANHO][TAMANHO];
int matriz_b[TAMANHO][TAMANHO];
int matriz_c[TAMANHO][TAMANHO];

int main()
{
    int i,j,k;

    // inicializa com valores inteiros randomicos
    for(i=0;i<TAMANHO;i++)
        for(j=0;j<TAMANHO;j++)
        {
            matriz_b[i][j] = rand() % MAX_VALOR;
            matriz_c[i][j] = rand() % MAX_VALOR;
        }
```

13 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/fontes/multmat.c

```
        // faz matriz_a = matriz_b * matriz_c
        for(k=0;k<TAMANHO;k++)
            for(i=0;i<TAMANHO;i++)
                for(j=0;j<TAMANHO;j++)
                    matriz_a[i][j] = matriz_a[i][j] + matriz_b[i][k] * matriz_c[k][j];
    }
```

Simule o programa com a configuração default (mapeamento direto com 8 blocos de 4 words cada) anotando a *hit rate*. Agora faça a seguinte alteração no loop: troque de posição o `for i` pelo `for j` e vice-versa, como abaixo. O programa, obviamente, produz o mesmo resultado.

```
    for(k=0;k<TAMANHO;k++)
        for(j=0;j<TAMANHO;j++)
            for(i=0;i<TAMANHO;i++)
                matriz_a[i][j] = matriz_a[i][j] + matriz_b[i][k] * matriz_c[k][j];
```

▶	Simule o programa com os loops na ordem k-j-i. Observe o que ocorre com as falhas de cache em relação à ordem k-i-j. **Explique** com detalhes.[14]

Existem vários truques como este para otimizar acessos a memória feitos por loops. Você pode ler mais a respeito pesquisando sobre *High Performance Computing*. Esta área pode incluir também detalhes de compilação, arquitetura, multiprocessamento e análise numérica (!).

EXTRA: Simule essas situações no seu PC para ter um *feeling* dos tempos envolvidos. Veja a descrição à parte[15].

Padrões de Acesso / Padding

Padrões de acesso à memória podem ser bastante prejudiciais. Para visualizar isso, insira no programa anterior os dois vetores padding1 e padding2 *obrigatoriamente entre as matrizes A e B e entre B e C*, como abaixo, usando tamanho 100x100.

```
int matriz_a[TAMANHO][TAMANHO];
int padding1[112];
int matriz_b[TAMANHO][TAMANHO];
int padding2[112];
int matriz_c[TAMANHO][TAMANHO];
```

▶	Escolha qualquer ordem dos loops e anote a *hit rate* para o caso original, usando mapeamento direto com 8 blocos de 4 words cada. Simule o programa com estas novas variáveis de *padding*. Observe o que ocorre com as falhas de cache. **Explique.** "Dica": observe no simulador os endereços das três matrizes e onde eles são mapeados na cache.

Se você desejar uma visualização gráfica deste tipo de problema, pode simular o programa do checksum usando a ferramenta *Memory Reference Visualization* (menu Tools do MARS) ligada junto com o *Cache Simulator*, o que leva 2 ou 3 min. Pra ver a tragédia acontecendo, insira um `int padding[28];` entre as declarações `int vetor[100];` e `int mapa[100];` e simule na velocidade máxima até o *breakpoint* do `linear_checksum`; daí simule a 30 instruções/s para poder observar o alinhamento causando falhas.

```
int vetor[100];
int padding[28]; // <= Exatamente neste local
// permutacao dos numeros de 0 a 99  para acesso randomico
int mapa[100] = {19, 45, 16, 21, 0, 87, 69, 20, 66, 2, 56, (...)
```

14 *Dica:* se não está claro para você, utilize a ferramenta experimental Tools => Memory Reference Visualization para as versões k-j-i e k-i-j do programa. NOTA: para isso, mude *"Unit Height in Pixels"* para 4 para caber tudo na tela da ferramenta, senão ela derruba o MARS. E não simule até o fim, demora eras geológicas...

15 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#matrizpc

EXTRA: Aprenda sobre o *padding* de estruturas feito pelos compiladores de alto nível[16]. Entenda os *tradeoffs* de desempenho-memória envolvidos nesta escolha.

Última Seção: Propostas de Prática Adicional

Dou três sugestões aqui:

1. ***EXTRA:*** Testar comportamento de um algoritmo de filtro de imagens passa-baixa com frame de (a) 3x3 vs (b) 5x5. Trata-se de um filtro linear de suavização utilizando a média dos pontos, muito simples mesmo.
   O roteiro da prática, as explicações e todos os arquivos necessários estão na minha página[17].
2. ***EXTRA:*** Escolha um método de ordenação diferente do quicksort, insertion/selection sort, bubblesort e mergesort. Implemente ou arranje um código em C deste algoritmo (mas *teste* e garanta que *funciona*) e adapte ele para simular no MARS e comparar com o quicksort.
   Para trabalhar usando meu script siga o modelo do quicksort. Lembre-se que não há bibliotecas, mas você pode construir as suas[18].
3. ***EXTRA:*** Proponha ao professor um ensaio com algum algoritmo interessante e tente demonstrar algum efeito de configuração de cache sobre a execução (localidade é sempre bacana). Não precisa *conseguir,* é só propor e fazer, mesmo que não dê nada (eu tentei rapidamente ensaiar o algoritmo de Djikstra e não consegui achar nada interessante).
   Para trabalhar usando meu script siga o modelo do quicksort. Lembre-se que não há bibliotecas, mas você pode construir as suas[19]. Dá até pra usar alocação dinâmica; se quiser, converse comigo.

Para o 2 e o 3, alguns detalhes devem ser observados a respeito da programação em C para MARS via Lcc:

- **Não podemos usar bibliotecas**, pois elas dependem do sistema rodando no processador; isso significa que não temos `printf`, `scanf` e `#include`'s...
- ...mas pra quebrar um galho, na conversão são criadas funções assembly simples e úteis para imprimir um inteiro *i* (`print(i);`), para terminar a simulação (`exit();`) e para obter um número aleatório (`rand()`). Basta chamá-las em C normalmente: `print(14);` mostra o número 14 etc. Você pode criar as suas se quiser: veja o material extra[20].
- No caso específico de funções indefinidas, o Lcc posterga a resolução dos nomes; o erro será visto apenas no MARS
- O script de conversão é necessário para adaptar o assembly gerado (a sintaxe do RISCompiler é ligeiramente diferente da do MARS) e está em ***alpha testing***; quaisquer artefatos inesperados deverão ser corrigidos à mão e informados ao professor.

Dados adicionais sobre o processo de compilação e sobre o assembly gerado podem ser vistos em http://pessoal.utfpr.edu.br/juliano/arqcomp/cache, incluindo detalhes sobre a alocação das variáveis feita pelo Lcc, já que vocês talvez queiram monitorá-las[21].

Variáveis Locais são por padrão *armazenadas na pilha* ao invés de utilizarem registradores, portanto toda entrada e saída de uma função mexem com $sp e o acesso às variáveis é feito com `lw $reg,Δ($sp)` e `sw $reg,Δ($sp)`, sendo Δ uma constante. Contudo, variáveis simples (um contador inteiro num loop, por exemplo) podem ser designadas a registradores.

Variáveis Globais são armazenadas numa área de memória à parte (*static memory*, identificada por `.data` ou `.rdata`) e portanto podem ser acessadas diretamente com `lw` e `sw`, sem necessidade de um ponteiro como $sp.

---

16 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#padding
17 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/pdi.html
18 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#funcasm
19 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#funcasm
20 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#funcasm
21 http://pessoal.utfpt.edu.br/juliano/arqcomp/cache/index.html#alocacao

Perguntas sobre os conceitos (★!)

1. O quicksort e o mergesort foram feitos com 200 `int`'s de 4 bytes cada. Se ao invés disto usarmos 200 `char`'s de um byte cada, o Lcc vai produzir o mesmo número de acessos à memória, só que vai usar instruções `lb` e `sb` (1 byte por vez) ao invés de `lw` e `sw` (4 bytes duma vez). As instruções levam o mesmo tempo para executar no MIPS. Pergunta: o desempenho total dos programas melhora? Sim ou não? Por quê?
2. O mesmo raciocínio pode ser feito para as variantes linear e aleatória do checksum. Preveja e explique as alterações de desempenho ao trocar os dados de `int`'s para `char`'s.
3. Pode-se dizer que o mínimo de Falhas de Conflito num sistema será obtido se usarmos associatividade total, e que o máximo ocorrerá se for usado mapeamento direto? Explique.
4. Na multiplicação de matrizes, suponha que por qualquer razão podemos escolher apenas entre duas ordens de loops: k-j-i e j-k-i. Quem vai decidir a disciplina de escrita (*write-back* ou *write-through)* é o gerentão. Pode haver diferença significativa de desempenho entre as ordens conforme a decisão do gerente? Ou seja, os loops k-j-i e j-k-i serão mais ou menos equivalentes entre si, independente da disciplina? Caso contrário, qual loop é mais adequado a qual disciplina? Justifique.
5. Determinado sistema possui uma *cache line* (ou seja, tamanho do bloco da cache) de 8 bytes. Rodamos nele um programa que realiza milhões de operações sobre uma matriz de *double's* (fracionários de precisão dupla, com 64 bits cada). Fará alguma diferença se executarmos varreduras na ordem linha-coluna ou coluna-linha? Justifique.
6. Você deve projetar um sistema microprocessado e há três opções disponíveis para acesso aos dados: (a) RAM lenta com *waitstates* (ver abaixo); (b) cache com linha curta; (c) cache com linha longa. Desejamos saber qual deverá fornecer a melhor performance, considerando simplificadamente que na média 30% das instruções executadas pelo nosso sistema são de acesso à memória e que o padrão de acesso aos dados tipicamente possui uma hit rate de 85%.

   Dados: sistema com pipeline e CPI_{ideal}=1; f_{clock}=1GHz; tempo médio de acesso a uma word de 4 bytes na RAM de 3ns (portanto há dois clocks "*idle*" – *waitstates* em que o uP não faz nada – se não houver cache); cache de linha curta com leitura de 8 bits por vez a 0,5 ns, usando mapeamento direto e blocos de 8 bytes; cache de linha longa com leitura de 16 bits por vez em 0,7ns, usando mapeamento direto e blocos de 16 bytes.

   Referência: baseado nos cálculos da seção 7.3 do Patterson (pág. 373), não vistos em aula. Não cai na prova, mas é bacana.
7. Numa matriz gigantesca de $1*10^6$x$1*10^6$ dados inteiros de 32 bits, queremos contar o número total de submatrizes distintas 3x3 com todos os elementos positivos. Usamos o algoritmo ingênuo, gerando todas as submatrizes possíveis e fazendo as comparações (se souber de outro melhor, me conte).

   Aponte, nas situações abaixo, os possíveis problemas de desempenho, se existirem. Em que sistema o programa provavelmente rodará melhor e por quê? Considere uma word com 32 bits.

   a) Cache com 256 blocos de 1 word cada com associatividade total

   b) Cache com 256 blocos de 8 words cada com associatividade de 1 via

   c) Cache com 512 blocos de 4 words cada com associatividade de 4 vias
8. Na questão anterior, suponha que alteramos o tamanho da matriz para 20x20 e garantimos que ela inicie num endereço de memória que é um número primo (ou seja, não alinhado com nada) tal como $1000\ 0003_{16}$[22]. Reavalie as situações apresentadas.

22 http://www.wolframalpha.com/input/?i=what+is+the+smallest+prime+above+0x10000000