Introdução ao PAD

Airam J. Preto e Stephan Stephany

- Motivação

- Microprocessadores x supercomputadores;
- CISC x RISC;
- Desempenho x benchmarks;
- O que é PAD?

- Objetivos

- O que é desempenho computacional?
- O que torna uma computação mais rápida?
- Como medir e comparar desempenho?
- O valor do **tempo** e do **espaço**.

- Programa

1. Preliminares e bibliografia.
2. Arquitetura CISC.
3. Arquitetura RISC.
4. Classes de processadores.
5. Introdução à tecnologia de memórias.
6. Tecnologia da Memória Virtual.
7. Aspectos funcionais de compiladores.
8. Otimizações clássicas na compilação.
9. Clareza e portabilidade de código.
10. Temporização e profiling.

11. Noções de paralelismo de código.
12. Otimização de código.
13. Otimização de loops.
14. Transformações associativas.
15. Otimização de acesso à memória.
16. Espaços de endereçamento.
17. Extensões de linguagens com suporte ao PAD.
18. Avaliação de desempenho - benchmarking.
19. Processamento paralelo.
20. Multiprocessadores.

- Arquitetura CISC

- Economia de tempo e espaço (recursos computacionais eram limitados).
- Tecnologia de compiladores era pouco desenvolvida (programação em assembler).
- Menor esforço de programação e economia de memória e no tempo de execução.
- IBM model 704 (32 Kwords, válvulas, 1956): operações de ponto flutuante, divisão, indexação de registradores, manipulação direta na memória.
- PHILCO 2000 (32 Kwords, transístor, 1959): instrução envolvendo sequência de operações com decremento de contador.
- Memória mais lenta que CPU (instruções que realizavam várias operações reduzia tempo de carregamento de instruções da memória).
- Até os anos 80, maior instruction set significava maior desempenho.

- Modos de endereçamento

Linguagens de alto nível permitem organizar os dados de maneiras convenientes: arrays, structures, etc.

Na programação em assembler a memória é apenas um sequência de endereços.

Por Exemplo, em Fortran:

```
      SUBROUTINE BETA (A,N)
      REAL A(*)
      DO 10 I=1,N
         A(I) = 0.0
10    CONTINUE
```

A é passado como um ponteiro para o endereço onde o primeiro elemento está armazenado. Para recuperarmos A(I) é necessário calcular:

```
address(A(I)) = address(A(0)) + (I-1)*sizeof(A)
```

ou

```
address(A(I)) = address(A(I-1)) + sizeof(A)
```

O suporte de hardware para o cálculo de endereço é chamado **modo de endereçamento**. Por ex.:

- **base register offset** - um registrador contém o endereço de início da área de dados.
- **indexed addressing** - requer um registrador de índice.

Cada tipo de processador suporta um número de modos de endereçamento, que é maior no caso de processadores CISC.

Máquinas com conjunto sofisticado de instruções e modos de endereçamento são muito complicadas?

Processador com 200 instruções, 100 delas com 5 modos de endereçamento poderia ter
$100 + 100 * 5 = 600$ combinações legais de instrução/endereçamento. Algumas destas instruções podem operar com endereçamento indireto e ter tamanho variável.

- Microprogramação

A CPU é um conjunto de **unidades lógicas** (adders, shifters, registradores) interligadas (data paths).

Instruções são executadas pela ativação das unidades lógicas numa ordem definida pela seção de controle da CPU.

Em circuitos "hardwired" a seção de controle é composta por portas lógicas interligadas que convertem instruções em sinais de controle.

A topologia das conexões na seção de controle definem o set de instruções.

Nos anos 50 Maurice V. Wilkes (UK) inventa a **microprogramação**.

O controle hardwired é substituído por por um conjunto de **microinstruções** e um **sequenciador** que aciona unidades lógicas e data paths numa certa ordem.

Cada instrução de máquina é então quebrada num conjunto ainda mais detalhado de instruções chamado **microprograma**.

- Benefícios da Microprogramação

- Facilita o projeto e manutenção do instruction set. Instruções são simuladas e não executadas diretamente.
- A complexidade das instruções pode ser aumentada sem a necessidade da criação de novo hardware na CPU.
- Instruções podem ser emuladas por qualquer arquitetura microprogramada.
- Criação de família de arquiteturas que compartilham o mesmo instruction set (IBM S/360, 1964).
- Permite a criação de instruções mais complexas reduzindo o tráfego entre memória e CPU.

- Pipelines

Há um limite para a frequência do relógio: tecnologia de semicondutores, empacotamento, interconexão de unidades lógicas, etc.

Muitas instruções necessitam vários ciclos de relógio para serem executadas e é necessário extrair o máximo de cada ciclo.

A redução do número de ciclos para executar uma instrução é limitado pela sua complexidade.

A solução é executar mais de uma instrução simultaneamente.

As operações são divididas em estágios que são executados concorrentemente formando um **pipeline**.

Três áreas onde se aplica pipelining:

- Processamento de instruções.
- Acesso à memória.
- Operações de ponto flutuante.

Processamento de Instruções

A execução de uma instrução pode envolver vários passos:

- **Instruction fetch**;
- **Instruction decode**;
- **Operand fetch**;
- **Execute**;
- **Writeback**.

Quando **instrução 0** está no estágio **operand fetch**, a **instrução 1** está em **instruction decode**, e a **instrução2** está na fase **instruction fetch**. Se uma instrução necessita de 5 ciclos de relógio, seria possível executar 1 instrução por ciclo.

Perturbação devido a stalls e desvios condicionais.

Pipelines foram introduzidas no final dos anos 50 no UNIVAC LARC e IBM Strech. Seu uso se estabeleceu a partir de 1964 com o CDC 6600 e o IBM S/360.

- Arquitetura RISC

Reduzir a complexidade do instruction set.

Trade-off **complexidade x velocidade**.

1964, CDC 6600 - Seymour Cray; 1975, IBM 801.

Fatores que permitiram a evolução para RISC:

- Caches e instruction pipelines para acelerar instruction fetch;
- Diminuição de custo e aumento da memória;
- Melhor pipelining;
- Avanços em compiladores com otimização.

A evolução na tecnologia de compiladores mostrou que era melhor ter instruções mais simples.

Programação em assembler caiu em desuso.

Mais memória permitiu executar programas que ficaram maiores devido a instruções simplificadas.

Pipelining é mais adequado para instruções simplificadas.

- Características (RISC)

Functional pipelines, sistemas de memória sofisticados, execução de mais de 1 instrução por clock mostram a complexidade de processadores RISC.

A complexidade removida do instruction set é transferida para o compilador, que influencia o desempenho da máquina (capacidade de otimização).

Tamanho de instrução uniforme

O processador reconhece o tamanho de uma instrução apenas na sua decodificação.

Caso a instrução tenha vários bytes, pode ser necessário recuperar bytes restantes da memória causando um stall no pipeline.

Se todas instruções tiverem o mesmo número de bytes, fica mais fácil manter o fluxo no pipeline.

Instruction set compacto

O instruction set contém apenas as instruções executadas mais frequentemente.

Instruções mais simples podem ser hardwired e as mais complexas são emuladas.

Modos de endereçamento simplificados

Cálculos de endereço complicados causam stall do pipeline.

Aritmética de endereço é mais fácil de ser otimizada pelo compilador.

Arquitetura load/store

Referências à memória são restritas a instruções **load** e **store** (em CISC instruções podem conter referência à memória).

Tamanho fixo de instrução limita o número de operações com endereços.

Referências complexas exigiriam complexidade no pipeline (uma instrução pode calcular um endereço ou efetuar uma operação).

Número grande de registradores

Manter o maior número de dados próximo da CPU (16 a 64 registradores).

Branch delay slot

Instrução que é executada independentemente do resultado num desvio condicional.

Classes de Processadores

O objetivo inicial do RISC era 1 instrução por ciclo.

Atualmente 2 ou mais instruções são executadas simultaneamente em pipelines separados.

O número de operações paralelas depende do processador e do programa (desafio para projetistas de hardware e de compiladores).

Superescalar

Executa operações simultâneas em vários pipelines (até 8 operações por ciclo): RS/6000, UltraSPARC, PA RISC, Alpha, MIPS.

Escalonamento em tempo de execução.

Por ex. RS/6000 executa 4 operações por ciclo:

- Ponto flutuante.
- Ponto fixo (ou acesso à memória).
- Manipulação de condição.
- Desvios.

Superpipelined

Os estágios do pipeline são divididos em sub-estágios capazes de executar operações ainda mais simples (deep pipeline).

Very Long Instruction Word (VLIW)

Escalonamento em tempo de compilação.

O compilador funde 2 ou mais instruções RISC em uma única palavra.

Todas operações na palavra são executadas em paralelo.

Requer compiladores sofisticados (análise de dependência de dados, fluxo de instruções, execução fora de ordem e desvios muito complexa).

Outras Características

O objetivo é manter os pipelines cheios e evitar distúrbios no fluxo de instruções. Isto exige muita coordenação no hardware, compilador ou ambos.

Register Bypass

Instruções RISC operam em dados nos registradores. Caso a instrução seguinte opere na mesma variável que a instrução corrente, este valor pode ser copiado para outro registrador de onde a variável será recuperada.

Renomeação de Registradores

Atrasos podem ser causados no pipeline pela maneira que o programa usa recursos.

Por ex.: uma instrução que reutiliza registradores que ainda não foram liberados pelas instruções precedentes.

O compilador pode fazer um rodízio de registradores, mas ainda podem ocorrer conflitos.

Alguns processadores (eg. RS/6000) detecta o conflito (uso independente) e substitui por um outro registrador durante a operação.

Esta característica permite que processadores com número diferente de registradores executem a mesma aplicação compilada para aquela família.

Reduzindo Penalidades de Desvios

Desvios causam as maiores perturbações no fluxo do pipeline.

Pipeline flushing depois de cada desvio custa muito tempo.

Branch delay slot só funciona se existirem instruções que possam ser colocadas no slot.

A técnica se torna mais difícil se o código contém pouco paralelismo ou o processador pode executar mais de uma instrução por ciclo.

Slots de mais de 2 instruções podem causar perda de desempenho.

Operações Anuladas

Uma alternativa é usar o slot com opção de cancelamento da instrução.

É uma forma de **execução especulativa**. O compilador preenche o slot com a instrução e pode anulá-la antes do estágio de execução.

Processadores superescalares não usam branch delay, ao invés, usam um processador de desvio e carregam as instruções do novo endereço.

Contudo mantém a habilidade de anular instruções (SPARC, RS/6000, etc.). Mas somente instruções inofensivas podem ser anuladas. Instruções que causam exceção (ponto flutuante, memória) invocam **exception handler** e não podem ser descartadas.

Atribuição Condicional

Substitui a sequência **teste-desvio**. Há a comparação, mas não o desvio.

```
a = b < c ? d : e;
```

Alguns processadores implementam atribuição condicional para ponto flutuante (Alpha, SPARC).

Branch Target Buffers

Blocos de instruções são armazenadas antecipadamente no **prefetch buffer** que alimenta o pipeline de instruções. Um desvio causa um stall até que o **prefetching** comece no novo endereço (**branch target**).

Para evitar o travamento é necessário antecipar o branch target.

Uma solução é usar um cache chamado **branch target buffer** que contém informação sobre os desvios prévios e as instruções contidas nos endereços alvo (Alpha).

Se ocorre um desvio, o processador imediatamente alimenta o pipeline com as instruções do **btb** enquanto faz o prefetching do novo endereço.

É necessário que já tenha ocorrido um desvio para o mesmo endereço fazendo que o **btb** esteja carregado com as instruções corretas.

Hardware Branch Prediction

A CPU armazena os destinos dos últimos desvios e usa algoritmos para estimar futuros endereços para prefetching.

Também é possível usar **static branch prediction** junto com heurísticas adequadas (a probabilidade de sair do loop é menos de 50%, um desvio para a frente é mais provável, etc.).

- Memória

Há uma disparidade crescente entre a velocidade da CPU e da memória.

(CPU a 500 MHz tem ciclos de 2 ns, cache com acesso na ordem de 10 ns, e memória 50 ns).

Possíveis soluções:

- Transferir um grande número de bytes por acesso.
- Hierarquizar a memória em porções rápidas e lentas, de maneira que as porções rápidas sejam usadas mais frequentemente.
- Segregar a armazenagem de instruções e de dados.

Random Access Memory

Existem 2 categorias de memória: dynamic random access memory (**DRAM**) e static random access memory (**SRAM**).

Na **DRAM** cada bit é representado por uma carga elétrica que se dissipa com o tempo, necessitando um **refresh** de tempos em tempos.

SRAMs mantém os dados enquanto houver alimentação do chip.

DRAMs são mais baratas, porém com tempo de acesso na ordem de 50 ns (**SRAM**s 10 ns).

Os dados na **RAM** são armazenados na forma matricial, e um bit pode ser localizado pelos endereços da linha e da coluna da sua posição.

Por ex., 1MB de RAM necessita de 20 bits de endereço para localizar as 1.048.576 posições de memória.

O **tempo de acesso** compreende:

- recuperar endereço da linha;
- recuperar endereço da coluna;
- trocar dado.

O **ciclo de memória** compreende o tempo que os chips demoram para se recuperar de um acesso.

Por ex., uma memória com tempo de acesso de 70 ns pode ter um ciclo de 200 ns.

Como compensar a diferença entre velocidade da CPU e da memória?

- Memória Cache

Princípio da Localidade de Referência: "É muito provável que o dado usado mais recentemente será usado num futuro próximo."

Em H/W, **smaller is faster** (propagação de sinal).

Memórias pequenas são usadas próximas da CPU para armazenar os dados usados mais recentemente.

Quando a CPU encontra o dado no cache ocorre um **cache hit**, caso contrário, ocorre um **cache miss**.

Hit rate abaixo de 95% começa a causar perda de desempenho.

Quando a CPU requisita um dado, um **bloco** de dados é carregado no cache.

Localidade temporal: provavelmente o dado será usado de novo.

Localidade espacial: provavelmente os outros dados do bloco também serão usados.

Além de ser usado para leitura, o cache também é usado para escrita na memória. Modos de escrita:

- **Write-back**: uma escrita num endereço armazenado no cache ocorre apenas no cache. Quando um dado num outro endereço precisar ser armazenado na mesma linha, o dado atualizado é salvo na memória e a linha é liberada.
- **Write-through**: toda escrita no cache causa atualização na memória. O dado continua no cache para uso da CPU.

Write-back permite maior desempenho evitando escritas desnecessárias.

Direct Mapped Cache

Um bloco de memória é sempre copiado nas mesmas linhas do cache, mapeado por:

(Block address) mod (No. cache blocks)

Conflitos ocorrem quando 2 endereços são mapeados para a mesma linha do cache. Uma referência substitui a anterior e assim por diante (**thrashing**).

```
      REAL*4 A(1024), B(1024)
      COMMON /DADOS/ A,B
      DO 10 I=1,1024
         A(I) = A(I) * B(I)
10    CONTINUE
      END
```

Os vetores **A** e **B** ocupam 2 áreas adjacentes de 4 KB cada. Num **direct mapped cache** de 4 KB as mesmas linhas reservadas por A(i) são reservadas para B(i), causando cache miss repetitivo.

A solução seria interpor outras variáveis entre **A** e **B** na área comum.

Fully Associative Cache

Um bloco de memória pode ser copiado em qualquer linha do cache.

Tem o melhor hit rate (conflito zero).

Desempenho é prejudicado pela necessidade da busca do dado em todos os acessos (via S/W ou H/W).

Lógica necessária para determinar linha a ser ocupada por novos dados (algoritmo **Least Recently Used**).

Set Associative Cache

Um bloco de memória pode ser copiado num conjunto (set) restrito de linhas do cache. Primeiro o bloco é mapeado para um set, e depois pode ser copiado para quaisquer linhas do set, mapeado por:

$$\text{(Block address) mod (No. cache sets)}$$

Se há n blocos no set, o cache é chamado **n-way set associative**.

Set associative é mais imune a thrashing que **direct mapped**, mas não totalmente.

Por ex., num **2-way set associative cache**:

```
      REAL*4 A(1024), B(1024), C(1024)
      COMMON /DADOS/ A,B,C
      DO 10 I=1,1024
         A(I) = A(I) * B(I) + C(I)
10    CONTINUE
      END
```

3 variáveis contendem pelas linhas de cache mapeadas para 2. Novamente, é necessário interpor uma outra variável no bloco.

Também é necessário o uso de algoritmos de busca (**LRU**) para determinar linhas livres, porém espaço de busca é reduzido ao set.

Harvard Memory Architecture

Demanda de dados é segregada da demanda de instruções.

A memória é única, mas há um cache para dados e outro para instruções (possivelmente de tipos diferentes).

Por ex., RS/6000 usa **2-way set associative cache** para **instruções** e **4-way set associative cache** para **dados**.

A taxa de informação vinda da memória aumenta e a interferência entre os 2 tipos de referência é minimizado.

Máquinas com **Harvard Memory Architecture** tem maior desempenho e preço.

Desempenho da Memória Cache

$$T_{acesso} = T_{hit} + Miss\_rate \times Miss\_penalty$$

1. Redução de Cache Miss Rate

- **Compulsório**: no primeiro acesso o bloco tem que ser carregado da memória (**cold start miss**).
- **Capacidade**: O cache não consegue conter todos os blocos referenciados durante a execução.
- **Conflito**: blocos tem que ser recarregados muitas vezes devido a muitos blocos mapeados para o **set**.

1.1. Maior Tamanho de Bloco

Blocos maiores reduzem o cache miss compulsório (aumenta localidade espacial).

Blocos maiores causam número reduzido de blocos aumentando cache miss por conflito.

1.2. Maior Associatividade

Regra 2:1 - **direct mapped cache** de tamanho N tem o mesmo miss rate de um **2-way set associative cache** de tamanho $N/2$.

1.3. Caches Vítimas

Pequeno cache fully associative é usado para reter blocos recentemente substituídos no cache principal (devido a miss - **vítimas**).

A procura é feita em paralelo com o cache principal. Funciona melhor para caches direct mapped pequenos.

1.4. Hardware Prefetching

Instruções ou dados podem ser pré-carregados diretamente no cache ou em buffers externos mais rápidos que a memória.

Num cache miss o bloco requisitado é carregado no cache e blocos próximos são carregados no buffer.

Se o bloco requisitado já estiver no buffer, o carregamento do cache é cancelado e o bloco é lido do buffer juntamente com um novo prefetch.

1.5. Compiler-controlled Prefetching

Alternativamente, o compilador pode inserir instruções para prefetch requisitando dados antes que forem necessários:

- **Register prefetch** carrega valores no registrador;
- **Cache prefetch** carrega dados somente no cache.

O prefetch mais efetivo é **semanticamente invisível** para o programa: não altera conteúdo dos registradores ou memória que estão sendo lidos.

O prefetch só faz sentido se não bloquear as outras referências ao cache, permitindo o overlap da execução de operações com o prefetch.

1.6. Otimizações na Compilação

1.6.1. Fusão de arrays

Alguns programas referenciam elementos de vários arrays na mesma dimensão com os mesmos índices ao mesmo tempo. Estes acessos causam conflito no acesso ao cache.

A fusão melhora a **localidade espacial** e reduz o conflito, pois os elementos desejados estarão no mesmo bloco.

```
/* Antes */
int val1[SIZE];
int val2[SIZE];

/* Depois */
struct merge {
   int val1;
   int val2;
};
struct merge merged_array[SIZE];
```

1.6.2. Troca de índice

Evita acesso não-sequencial em arrays bi-dimensionais (melhorando **localidade espacial**).

```
/* Antes */
for (j=0; j<100; j++)
  for (i=0; i<5000; i++)
    x[i][j] = 2 * x[i][j];

/* Depois */
for (i=0; i<5000; i++)
  for (j=0; j<100; j++)
    x[i][j] = 2 * x[i][j];
```

1.6.3. Fusão de loops

Alguns programas referenciam os mesmos arrays em partes diferentes do código em loops.

A fusão destes loops permite a reutilização de dados que já estão no cache (**localidade temporal**).

A versão original sofre todos os **cache misses** para carregar os arrays **a** e **c** 2 vezes. Na versão concatenada, a segunda operação utiliza os dados já disponíveis no cache.

```
/* Antes */
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    d[i][j] = a[i][j] + c[i][j];

/* Depois */
for (i=0; i<n; i++)
  for(j=0; j<N; j++)
  { a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j]; }
```

1.6.4. Blocagem

Tenta melhorar **localidade temporal** quando arrays multi-dimensionais são acessados por colunas e linhas.

Neste caso, algoritmos operam em sub-matrizes ou **blocos**, maximizando o acesso aos dados carregados no cache antes de serem substituídos.

Por ex., multiplicação de matrizes:

```
/* Antes */
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  { r=0;
    for (k=0; k<N; k++) {
     r = r + y[i][k]*z[k][j]; };
  x[i][j] = r; };
```

Para garantir que elementos acessados caibam no cache usa-se um algoritmo **blocado** que opera em sub-matrizes B $\times$ B, onde B é o **blocking factor**.

```
/* Depois */
for (jj=0; jj<N; jj=jj+B)
 for (kk=0; kk<N; kk=kk+B)
  for (i=0; i<N; i++)
   for (j=jj; j < min(jj+B-1,N); j++)
   { r = 0;
     for (k=kk; k < min(kk+B-1,N); k++) {
      r = r + y[i][k]*z[k][j]; };
     x[i][j] = x[i][j] + r; };
```

2. Redução de Penalidade do Cache Miss

O aumento do desempenho da CPU em relação à memória causa um aumento relativo na **penalidade** do **cache miss**.

2.1. Prioridade para Read Miss

Caches **write-through** usam **write buffers** que podem conter um dado necessário num **read miss**.

A solução de esperar que o **write buffer** esvazie causa perda de desempenho.

A alternativa é verificar o buffer e se não há conflitos deixar o **read miss** continuar.

No caso de **write-back**, quando um **read miss** substituirá um **dirty block**. Ao invés de escrever o bloco e então ler a memória, copia o bloco para um buffer, lê a memória e então escreve o bloco.

2.2. Alocação de Sub-blocos

Um bit de validade é acrescentado em unidades menores que o bloco chamadas **sub-blocos**.

Num **cache miss** somente um **sub-bloco** precisa ser transferido entre a memória e o cache.

2.3. Early Start e Critical Word First

A CPU necessita de apenas 1 palavra por vez.
Portanto as palavras podem começar a ser enviadas para a CPU enquanto o **bloco** é carregado.

- **Early start**: assim que a palavra requisitada chegar, enviá-la à CPU, e continuar carregando o resto do bloco.
- **Critical word first**: carregar a palavra requisitada primeiro, enviá-la à CPU, e continuar carregando o resto do bloco.

2.4. Non-blocking Caches

Continua a suprir **hits** enquanto resolve um **miss** sem causar stall.

Aumenta a complexidade do controlador de cache que tem que coordenar acessos múltiplos na memória.

2.5. L2 Caches

Adicionar outro nível de cache na hierarquia da memória.

Desempenho em 2 níveis:

$$\mathrm{T_{acesso}} = \mathrm{T_{hit_{L1}}} + \mathrm{Miss\_rate_{L1}} \times \mathrm{Miss\_penalty_{L1}},$$

$$\mathrm{Miss\_penalty_{L1}} = \mathrm{T_{hit_{L2}}} + \mathrm{Miss\_rate_{L2}} \times \mathrm{Miss\_penalty_{L2}}.$$

$$\mathrm{T_{acesso}} = \mathrm{T_{hit_{L1}}} + \mathrm{Miss\_rate_{L1}} \times (\mathrm{T_{hit_{L2}}} + \mathrm{Miss\_rate_{L2}} \times \mathrm{Miss\_penalty_{L2}})$$

- **Local miss rate**: falhas no cache dividido pelo no. de acessos nesse cache ($\mathrm{Miss\_rate_{L2}}$ no L2).
- **Global miss rate**: falhas no cache dividido pelo no. total de acessos ($\mathrm{Miss\_rate_{L1}} \times \mathrm{Miss\_rate_{L2}}$).

A velocidade de L1 afeta o **desempenho** da CPU.

A velocidade de L2 afeta o **miss penalty** de L1.

L2 sofre menos hits que L1, portanto seu **miss rate** deve ser minimizado (**fast hits** $\times$ **few misses**).

L2 deve ser bem maior que L1 (reduzindo **miss rate** por capacidade).

3. Redução de Hit Time

Hit time limita a taxa de clock da CPU, mesmo em máquinas que usam vários ciclos para acessar o cache.

A essência está em balancear **hits** rápidos com poucos **misses**.

Endereço é composto por:

- **Tag**: endereço do bloco mais um bit de validade.
- **Índice**: seleciona o set.
- **Block offset**: aponta para o dado dentro do bloco.

Na leitura, só o **tag** é comparado para ver se o bloco com o dado desejado está no cache (**hit**).

Quanto **maior** a **associatividade**, **menor** o número de **sets**, portanto:

decresce o no. de bits do **índice** e aumenta o **tag**.

3.1. Caches Simples e Pequenas

Caches pequenos podem ser encapsulados na CPU.

Caches pequenos reduzem o tempo de leitura da memória associativa e comparação de endereços.

Direct mapped permite transmissão do dado durante a comparação do **tag**.

3.2. Evitar Endereçamento Virtual

O espaço de endereçamento é mapeado diretamente no cache.

Mudança de contexto exige flushing do cache (**tag** deve conter **PID**).

PID distingue se o dado no cache pertence ao programa em execução.

Aliases (2 end. virtuais mapeados para o mesmo end. físico) podem causar 2 cópias do mesmo dado. Se uma cópia é alterada a outra se torna inválida.

É necessário usar **anti-aliasing** em H/W e S/W.

Por ex., no Solaris, os 18 **lsb**s dos endereços são iguais nos **aliases** (**page colouring**).

3.3. Dividir Tradução de Endereço

Tradução de endereço e acesso ao cache em estágios separados no pipeline.

Tenta reduzir atrasos devido à tradução do endereço.

Maior no. de estágios para acesso à memória aumenta a penalidade em desvios não previstos e o no. de ciclos entre o disparo do **load** e o uso do dado.

3.4. Pipelining Writes

Write hits demoram mais que **read hits** devido a checagem do **tag** antes da escrita (na leitura o check pode ocorrer em paralelo).

Alpha usa pipelines para **writes**.

Primeiro **tags** e dados são separados. Após a comparação do **tag** a primeira escrita é feita. A próxima comparação pode ser feita durante esta escrita.

A comparação do **tag** na escrita atual ocorre em paralelo com a escrita prévia sem bloquear a CPU.

- Tecnologia da Memória

Organização para Aumento de Desempenho

É mais fácil aumentar a bandwidth da memória do que reduzir sua latência.

Caches se beneficiam do **aumento** de **bandwidth** por permitir **aumento** do **block size** sem aumentar a penalidade das falhas.

Ex. de desempenho num cache miss:

- 4 ciclos para enviar endereço;
- 24 ciclos para tempo de acesso por palavra;
- 4 ciclos para enviar a palavra (4 Bytes).

Considerando um bloco de 4 palavras, a penalidade é:

$$4 \times (4 + 24 + 4) = 128 \text{ ciclos}$$

Bandwidth de 1/8 Byte/ciclo (4×4 Bytes/128).

Como aumentar o **bandwidth**?

1. Aumento da Largura do Bus

Caches L1 tem largura de 1 palavra (largura do registrador).

Sistemas sem L2 tem memória compatível com largura do cache.

Dobrando a largura do bus (cache/memória) duplica o bandwidth.

No ex., a penalidade passa a ser $2 \times 32 = 64$ ciclos e **bandwidth** de 1/4 Byte/ciclo.

A CPU utiliza uma palavra por vez, necessitando um multiplexador na conexão com o cache (**penalidade**).

Usando L2, o multiplex fica entre o L1 e L2, fora do caminho crítico com a CPU.

No caso de expansão da memória, o incremento tem que ser dobrado ou quadruplicado de acordo com a largura (**custo**).

No caso de memória **ECC**, a correção de erro é feita em toda a largura da memória, aumentando a frequência das operações "ler-modificar-escrever" (escritas se tornam parciais no bloco).

2. Acesso Intercalado

Memória é organizada em bancos que permitem a leitura simultânea de várias palavras.

Memória intercalada tenta tirar proveito do **bandwidth** de todos bancos.

Os bancos tem largura de 1 palavra não sendo necessário alterar largura do cache e do bus.

O envio de endereços a todos os bancos permite a sua leitura em paralelo.

Por ex., enviando um endereço para 4 bancos, a penalidade passa a ser:

$$4 + 24 + (4 \times 4) = 44 \text{ ciclos}$$

Bandwidth de 0.36 Bytes/ciclo (4×4 Bytes/44).

Bancos permitem sucessão de 1 escrita/ciclo desde que não sejam para o mesmo banco.

Memória intercalada otimiza acesso sequencial às palavras no bloco. Palavras cujo end. mod $4 = 0$ ficam no banco 0, end. mod $4 = 1$ ficam no banco 1, etc.

Ideal: No. Bancos $\geq$ No. Ciclos p/ Acessar Palavra.

3. Bancos de Memória Independentes

Na memória intercalada os bancos compartilham as linhas de endereço com o controlador de memória.

Uma generalização deste esquema permite acessos múltiplos, usando vários controladores de memória que suportam operação independente dos bancos.

Cada banco utiliza uma linha de end. e um **data bus**.

Por ex., pode ocorrer um **cache write** em um banco e um **read** em outro.

Non-blocking caches necessitam de vários bancos de memória, caso contrário, leituras múltiplas são servidas por um único **port** de memória prejudicando o overlap do acesso com a transmissão.

Multiprocessadores com **memória compartilhada** também necessitam de bancos de memória.

4. Evitando Conflitos entre Bancos

A **eficácia** de memórias que suportam acessos independentes é função da **frequência** do acesso aos diferentes bancos.

Acesso sequencial funciona bem no esquema intercalado. Porém, se a diferença entre acessos é um número par, há conflito.

Alguns sistemas utilizam inúmeros bancos (NEC SX/3 tem 128 bancos). Mesmo neste caso pode haver conflito:

```
int x[256][512];
  for(j=0; j<512; j++)
    for(i=0; i<256; i++)
      x[i][j] = 2*x[i][j];
```

Como 512 é múltiplo de 128, os elementos de uma coluna estarão sempre no mesmo banco ocorrendo o **cache miss** independentemente da sofisticação da memória.

Uma solução seria a troca de índice pelo compilador, ou alterar a dimensão do array no programa.

O mapeamento do endereço para o banco de memória e dado por:

$$\text{Bank No.} = (\text{Add.}) \bmod (\text{No. of Banks})$$

$$\text{Add. within Bank} = (\text{Add.})/(\text{No. of Banks})$$

Sistemas tradicionais usam número de bancos e quantidade de memória por banco que são potência de 2 para facilitar o mapeamento.

Uma solução de H/W para evitar conflitos é usar números primos de bancos. Se o número primo puder ser representado por $2^N - 1$, o mapeamento é:

$$\text{Add. within Bank} = (\text{Add.}) \bmod (\text{No. Words/Bank})$$

Usando uma seleção de bits no lugar de uma divisão por número primo.

5. DRAM Interleaving

O acesso à DRAM é dividido em acesso a linhas (**RAS**) e colunas (**CAS**).

A **DRAM** armazena um linha de bits num buffer para o **CAS** (tamanho do buffer é **sqrt** do tamanho da **DRAM**, por ex. 16 Kb para 256 Mb).

Para melhorar desempenho, **DRAM**s usam sinais de temporização que permitem acessos repetitivos ao buffer sem incorrer numa penalidade de **RAS**:

- **Nibble mode** - **DRAM** supre bits extras de localizações sequenciais para cada **RAS**.
- **Fast page mode** - Através da mudança de endereço de coluna, bits aleatórios são acessados no buffer até o próximo **RAS** ou **refresh**.
- **Static column** - Similar ao **fast page**, porém sem a necessidade de trocar a linha de **CAS** para toda mudança de endereço de coluna.

Estas operações aceleram o **ciclo de memória** sem adicionar muito **custo** ao sistema.

O **ciclo de memória** otimizado é o mesmo para qualquer um dos modos.

Por ex., em **nibble mode**, o chip lê 4 bits ao mesmo tempo internamente, suprindo 4 bits externamente durante 4 ciclos otimizados (similar à memória intercalada).

Novas Tecnologias

RAMBUS usa uma interface fazendo que 1 chip atue como um sistema de memória.

Não usa **RAS/CAS**, substituídos por um bus que permite acessos entre o envio de endereços e o retorno de dados (**packet switched bus**).

1 chip funciona como um banco de memória, podendo retornar um no. variável de dados e fazer seu refresh.

RAMBUS usa uma interface com largura de **1 Byte** e um sinal de clock que sincroniza com o clock da CPU. Quando o pipe de endereços está cheio, um chip fornece 1 Byte a cada 2 ns.

A maioria dos sistemas usa técnicas como **fast page mode**, que não trazem desvantagem quando se aumenta a DRAM (ao contrário de memória intercalada). **RAMBUS** poderá ser o novo padrão.

RAMBUS também poderá substituir **VRAM** (memória de vídeo com linha de saída serial rápida).

- Memória Virtual (VM)

Objetivos

- Compartilhar uma quantidade pequena de memória física entre vários processos através do remapeamento de endereços **lógicos** para **físicos**.
- Restringir o acesso de um processo aos blocos que lhe pertencem (**proteção**).
- Permitir a alocação de programas em qualquer parte da memória física (**realocação**).

Paginação × Segmentação

VM pode utilizar blocos de tamanho fixo (**paginação**) ou variável (**segmentação**).

Páginas tem tamanho fixo entre 4 KB a 64 KB.
Segmentos podem variar de 2^0 a 2^{32} Bytes.

Endereçamento paginado requer 1 palavra por endereço contendo **no. da página** e **offset**.

Segmentos tem tamanho variável e exigem 1 palavra para **no. do segmento** e 1 palavra para **offset**.

Segmentos paginados (segmentos contém no. inteiro de páginas) simplificam substituição de blocos e não é necessário que o segmento inteiro esteja na memória.

Alpha 21064 utiliza tamanhos de páginas múltiplos: 8 KB, 64 KB ($2^3 \times$ 8 KB), 512 KB ($2^6 \times$ 8 KB), e 4096 KB ($2^9 \times$ 8 KB).

Memória Virtual $\times$ Caches

- Substituição de blocos no cache é controlada pelo H/W; na VM é controlada pelo OS (**penalidade** maior exige **taxa de falhas** menor).
- Tamanho de endereço na CPU determina tamanho da VM (tamanho do cache é independente da capacidade de endereçamento).
- Memória secundária (disco) também é usada pelo file system, que não faz parte do espaço de endereçamento.

Translation Look-aside Buffer (TLB)

Tabelas de páginas são grandes e precisam ser armazenadas na memória (as vezes paginadas).

O resultado é que todo acesso à memória é dobrado: um para recuperar o endereço físico e outro para recuperar o dado.

Solução é evitar novo mapeamento armazenando o end. da página acessada mais recentemente.

Solução mais geral é armazenar as **traduções de endereço** num cache (**TLB**).

O **tag** do TLB contém parte do end. virtual e o **campo de dados** contém frame no., campo de proteção, bit de validade, dirty bit, etc.

Para alterar um campo na tabela de páginas (frame no., proteção, etc.) o OS tem que remover a entrada correspondente no TLB.

O TLB armazena 8 a 1024 posições (**fully assoc.**)

Hit time = 1 ciclo; taxa de falhas $\leq$ 1%; penalidade do miss = 10 a 50 ciclos.

Tradução de end. no Alpha 21064:

- Endereço virtual é enviado p/ todos **tag**s que estão marcados se estão válidos ou não (1, 2).
- Violação de proteção é verificada (1, 2).
- O **page offset** não está incluído no TLB.
- O **tag** que contém o end. envia o **end. físico** da pg. correspondente pelo multiplex (3).
- O **page offset** é concatenado ao **end. físico** da pg. formando um end. físico de 34 bits (4).

Tamanho de Página

Motivos para páginas grandes:

- Tabela de pgs. menores ocupam menos memória.
- Transferência de pgs. da memória secundária (disco) é mais eficiente.
- No. menor de pgs. diminui o conflito no TLB.

Motivos para páginas menores:

- Menor fragmentação interna.
- Inicialização de processos mais rápida.

Proteção

Não pode haver interferência entre processos compartilhando a VM.

Seus espaços de endereçamento devem ser protegidos.

Mecanismo simples: registradores **Base** e **Bound**.

O endereço é válido se: $\mathbf{Address} + \mathbf{Base} \leq \mathbf{Bound}$

Processos não tem permissão par alterar estes reg.

O OS deve alterar estes registradores para mudar contexto. Portanto H/W tem que prover:

- 2 modos de operação: **user** e **kernel**.
- Parte do estado da **CPU** que processos no modo **user** podem ver mas não alterar.
- Mecanismos para mudança de modo.

Processos do OS são executados no modo **kernel**.

Processo no modo **user** tem acesso de apenas leitura a registradores **base/bound**, **user/kernel** mode bits e **exception** enable/disable bit.

A mudança da CPU de modo **user** para **kernel** é feita por uma **system call** que aciona um trecho de código do kernel após salvar o PC. O retorno ao modo **user** ocorre como o retorno de uma subrotina.

O end. fornecido a CPU resulta de um mapeamento de end. virtual para físico. Este mapeamento permite verificação de proteção através de **flags** de permissão adicionados a cada página ou segmento.

Flags incluem **read/write** e **user/kernel**.

Processos são protegidos cada um tendo sua próprias **page tables** cada uma apontando para páginas distintas da memória.

Outros esquemas mais sofisticados de proteção incluem **rings** (acesso hierárquico) e **capabilities** (verificação de acesso).

VM Paginada - Alpha 21064

Alpha usa combinação de paginação e segmentação.

Espaço de endereçamento de 64 bits é dividido em:

- **seg0** (bit 63=0): user (texto e heap).
- **seg1** (b62=1, b63=1): user (pilha).
- **kseg** (b62=0, b63=1): kernel.

kseg é reservado para o OS com proteção uniforme.

seg0 e **seg1** são usados por processos no modo **user** mapeados em páginas com proteção individual.

Segmentação divide o espaço de endereçamento e economiza espaço para page tables.

Paginação provê VM, realocação e proteção.

A tabela de páginas é dividida em 3 níveis para reduzir o seu tamanho.

A tradução de end. começa com a soma do campo de end. L1 com o **base reg.** da tabela de pág. para ler nesta locação o **base address** da tabela de pág. L2.

O campo de end. L2 é somado ao end. recuperado para determinar o **base address** da tabela L3.

Finalmente, o campo L3 é somado com com o **base address**, e esta posição é lida para se obter o endereço físico da página referenciada.

Este endereço é concatenado com o offset para se obter o endereço completo.

Cada page table é limitada para caber numa única página, assim todos os **base address** de page tables são endereços físicos que não precisam de tradução.

Alpha usa 64-bit **page table entry** (PTE) em cada tabela de página. Primeiros 32 bits contém frame no. e os demais contém 5 campos de proteção:

- **Valid** (frame no. válido para trad.)
- **Kernel (page) read enable**
- **User (page) read enable**
- **Kernel write enable**
- **User write enable**

VM Segmentada - Pentium

8086 usava segmentação sem provisão para VM ou proteção.

Segmentos tinham registrador **base** mas não **bound** e nem verificação de acesso.

Já o P5 tem 4 níveis de proteção. Nível 0 corresponde ao **kernel** e nível 3 ao **user** mode.

O P5 usa pilhas separadas para cada nível para evitar quebra de proteção. Também usa "tabelas de segmentos" e verificações para tradução de endereços.

Processos no modo **user** podem chamar rotinas do OS dentro do seu espaço de endereçamento mantendo proteção total (ação não trivial, pois OS e processos tem suas próprias pilhas). O OS ainda mantém proteção para os parâmetros da rotina chamada, evitando acessos indiretos (**Trojan horses**).

O suporte para proteção e compartilhamento é mantido com confiança mínima no OS.

Mapeamento de memória

Ao invés de **base address** os regs. de segmento do P5 contém um índice para uma **descriptor table** (que funciona como uma **page table**):

- **Present bit** - equivalente ao bit de validade.
- **Base field** - equivalente ao **page frame add.**, contém o end. físico do 1o. byte do segmento.
- **Access bit** - similar ao **use bit** usado em algoritmos de substituição como LRU.
- **Attributes field** - especifica operações válidas e níveis de proteção para o segmento.
- **Limit field** - limita offsets para o segmento.

P5 ainda suporta um sistema de paginação onde a porção superior do end. de 32 bits seleciona o **segment descriptor** e a porção média é um índice para a tabela de pág. selecionada pelo **descriptor**.

Multiprocessadores

Razões para se ter máquinas paralelas:

- Maneira econômica de se aumentar o desempenho além do proporcionado por uniprocessadores.
- Não parece claro que a taxa de aumento de desempenho de uniprocessadores poderá ser mantida indefinidamente.
- Tem ocorrido progresso na área que representa maior obstáculo para a popularização de máquinas paralelas: **software**.

Taxonomia de Flynn

- **SISD** (Single Instruction stream, Single Data stream)
- **SIMD** (Single Instruction stream, Multiple Data streams)
- **MIMD** (Multiple Instruction, Multiple Data streams)

Arquitetura **SISD** é o uniprocessador.

Na **SIMD** a mesma instrução é executada por todos processadores em partições de dados diferentes.

- Cada processador tem a sua memória de dados.
- Há uma única memória de instruções e processador de controle.
- Usa processadores com propósito especial, sem flexibilidade.

Na **MIMD** cada processador carrega suas instruções e opera em seus dados.

- A memória pode ser compartilhada ou distribuída entre os processadores.
- São mais flexíveis, podem ser usadas como multiprogramadas/multitarefa, ou single user.
- Melhor relação custo/benefício (microprocessadores **off-the-shelf**).

Máquinas **MIMD** são classificadas em 2 categorias conforme a organização da memória:

1. Memória compartilhada centralizada.
    - **Bus** conecta processadores e memória.
    - **Escalabilidade baixa**. Com caches grandes, bus e memória única conseguem atender um número limitado de demandas de processadores (**bus bandwidth**).
    - **UMA** (uniform memory access). O tempo de acesso é o mesmo para todos processadores.
2. Memória distribuída fisicamente.
    - **Rede de interconexão** conecta processadores e memória (**nós**).
    - **Escalabilidade maior**. Maioria das demandas dos processadores são atendidas pelas memórias locais.
    - **Interconnetion bandwidth** alta para viabilizar acesso à memória não local.

Organização do Espaço de Endereçamento

1. **Espaço de Endereçamento Único**
   - Memórias fisicamente separadas são acessadas num espaço de endereçamento compartilhado logicamente.
   - Qualquer processador pode fazer referência a qualquer endereço.
   - **DSM** (Distributed Shared Memory). Endereços físicos em 2 nós podem ser mapeados a mesma locação de memória.
   - **NUMA**. Tempo de acesso depende da localização do endereço na **DSM**.
   - Nós trocam dados via load/store.
2. **Espaços de Endereçamento Múltiplos**
   - Espaços de endereçamento disjuntos.
   - Endereços físicos em 2 nós são mapeados a locações em memórias diferentes.
   - Nós se comunicam por mensagens.

Vantagens dos Mecanismos de Comunicação

1. Memória compartilhada
    - Compatibilidade com os mecanismos usados em uniprocessadores.
    - Programabilidade quando padrões de comunicação entre processadores são complexos ou dinâmicos durante a execução.
    - Menor overhead de comunicação e melhor uso de bandwidth para transmitir poucos dados (acesso à memória com proteção por H/W).
    - Habilidade para usar cache controlado por H/W para reduzir acesso remoto.
2. Comunicação por mensagens
    - H/W mais simplificado (se comparado com suporte para coerência de cache).
    - Comunicação explícita obriga considerar os custos de comunicação.

Métricas de Desempenho

1. **Communication bandwidth**
    - Bisection bandwidth é determinada pela rede de interconexão.
    - I/O Bandwidth de um nó é determinada pela arquitetura do nó e pelo mecanismo de comunicação.
2. **Communication latency**
    - Latência de transporte é determinada pela rede de interconexão.
    - Overheads de envio/recepção dependem do mecanismo de comunicação.
3. **Latency hiding**
    - É dependente da aplicação.
    - Pode ser quantificada por comparação entre máquinas com mesma latência porém com diferente suporte para latency hiding.

Desafios do Processamento Paralelo

- Paralelismo limitado nos programas. Lei de Amdahl:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}} + (1 - \text{Fraction}_{\text{parallel}})}$$

- Grande latência do acesso remoto.

Características das Aplicações Paralelas

- Balanceamento de carga.
- Sincronização.
- Sensitividade a latência da memória.
- Distribuição de dados na memória.
- Estrutura do algoritmo paralelo.
- Padrões de acesso aos dados (temporal/espacial).

Memória Compartilhada Centralizada

Multilevel caches reduzem demanda por bandwidth de memória, assim vários processadores podem compartilhar a mesma memória.

A arquitetura suporta caching de **dados privados** (usados por um único processador) e **dados compartilhados** que provê comunicação entre processadores através de read/write nesses dados.

Dados privados são carregados apenas no cache local e o programa se comporta como no uniprocessador.

Dados compartilhados são replicados em vários caches, reduzindo a latência e a contenção entre processadores disputando o mesmo dado. Porém, um novo problema é introduzido: **coerência de cache**.

Coerência de Cache

Definição simplista:

"Um sistema de memória é **coerente** se qualquer leitura de um item retorna o valor mais recentemente escrito deste item."

A definição acima abrange dois aspectos:

- Coerência (**quais** valores são lidos);

- Consistência (**quando** um valor escrito será retornado por uma leitura).

Um sistema de memória é **coerente** se:

1. Uma leitura pelo processador **P** numa posição **X** feita após uma escrita de **P** em **X**, sendo que nenhuma escrita de **X** feita por outro processador ocorre entre a escrita e leitura feitas por **P**, sempre retorna o valor escrito por **P**.

2. Uma leitura pelo processador **P** numa posição **X** feita após uma escrita por outro processador em **X**, retorna o valor escrito se a leitura e escrita estão suficientemente separados e nenhuma outra escrita em **X** ocorre entre os dois acessos.

3. Escritas na mesma posição são serializadas: duas escritas na mesma posição feitas por quaisquer dois processadores são vistas na mesma ordem por todos processadores.

Coerência e consistência são complementares:

- Coerência define o comportamento de leitura e escrita na **mesma** posição de memória.
- Consistência define o comportamento de leitura e escrita relativo a acessos a **outras** posições de memória.

Protocolos para Coerência de Cache

Caches coerentes provêm:

- **Migração** de dados compartilhados: dados são movidos para o cache local e usados localmente de maneira transparente (reduz latência).
- **Replicação** de dados compartilhados: na leitura, os dados são copiados para o cache local (reduz latência e contenção de leitura).

Migração e replicação são críticos para o desempenho do acesso a dados compartilhados.

Multiprocessadores de pequena escala adotam uma solução de H/W através do uso de protocolos de manutenção de caches coerentes:

- **Directory based** - cada bloco da memória física tem um status de compartilhamento que é mantido em um local apenas (**directory**).
- **Snooping** - cada cache que tem cópia de dados de um bloco da memória física também tem cópia do status de compartilhamento do bloco (não há estado centralizado). Todos caches monitoram (**snoop**) o bus para determinar se possuem cópia do boco requisitado no bus.

Protocolo Write-Invalidate

- Assegura que cada processador tem acesso exclusivo a um item antes de escrever neste item.
- as outras cópias do item são invalidadas numa operação de escrita.

Protocolo Write-Updade (ou Broadcast)

- Atualiza todas as cópias quando um item é escrito.
- O broadcast deve ser feito somente se a palavra é compartilhada.

Desempenho de Write-Invalidade $\times$ Write-Update:

1. Escritas múltiplas na mesma posição requer múltiplos broadcasts, mas apenas uma invalidação.
2. Em blocos com múltiplas palavras, é necessário um broadcast para cada palavra escrita, mas apenas uma invalidação inicial para a primeira palavra.
3. O atraso entre escrita num processador e leitura em outro é menor no write-update. Invalidação causa cahe read miss.

Técnicas de Implementação

Algoritmo de invalidação na escrita:

- O processador adquire acesso ao bus e difunde o endereço a ser invalidado.
- Todos processadores monitoram o bus verificando se os endereços inválidos estão no cache local. Neste caso o dado no cache é invalidado.
- A serialização do acesso ao bus força a serialização das escritas.

Localização de dado num cache read miss:

- **write-through**: dado está na memória.
- **write-back**: como todos processadores monitoram o bus, se um processador tem uma cópia **dirty** (atualizada) do bloco, essa cópia é disponibilizada e o acesso à memória é cancelado.

Distributed Shared Memory

Esquema mais simples exclue caches coerentes:

- Cray T3D.
- Dados compartilhados não são carregados em cache (só os privados).
- Coerência é controlada por S/W (copiando dados compartilhados para locações privativas).
- Exige pouco suporte de H/W.
- Mecanismos na compilação para gerenciamento do cache são limitados (paralelismo em loops bem estruturados).
- Localidade em blocos de dados compartilhados não pode ser aproveitada.

No caso de esquemas com caches coerentes, protocolos **snooping** não são **escaláveis** (exigem comunicação com todos os caches num cache miss).

A alternativa é um protocolo baseado em diretórios.

Protocolos Directory-Based

No tratamento de read miss e escrita em blocos compartilhados, uma lista de registros (**directory**) monitora os estados dos blocos nos caches:

- **Shared** - Um ou mais processadores tem cópia do bloco no cache e o valor na memória está atualizado.
- **Uncached** - Nenhum processador tem cópia do bloco no cache.
- **Exclusive** - Exatamente um processador (**owner**) tem cópia do bloco no cache que acabou de ser atualizado, e a cópia na memória está desatualizada.

Além dos blocos é necessário monitorar os processadores que têm cópia do bloco que será invalidado numa escrita.

Um **bit vector** é mantido para cada bloco indicando quais processadores têm cópia ou qual é o **owner**.

Sincronização

Mecanismos de sincronização são construídos com rotinas baseadas em instruções de sincronização suportadas pelo H/W.

Em situações de **baixa contenção** (poucos nós), a característica básica é uma sequência de instruções capazes de **operações atômicas** de leitura e escrita numa posição de memória.

Em situações de **alta contenção** (muitos nós), a sincronização se torna o gargalo (devido a atrasos por **contenção** e maior **latência**).

Primitivas de Hardware

São recursos básicos usados na construção das operações de sincronização a nível de usuário (como **locks** e **barreiras**).

Estas instruções primitivas não são acessíveis ao usuário, mas são usadas pelos programadores do sistema na geração de bibliotecas de sincronização.

Instruções Típicas

- **Atomic Exchange** - Troca um valor de um registrador por outro na memória (tentativas de troca simultâneas são serializadas).
- **Test-and-set** - Testa um valor e o modifica se o valor passa no teste.
- **Fetch-and-increment** - Retorna um valor da memória e o incrementa (similar ao **atomic exchange**).
- **Load Linked & Store Conditional** - Par de instruções executado como se fosse atômico:
  - se o conteúdo do endereço especificado pelo **load linked** é alterado antes do **store condition**, este falha.
  - se há mudança de contexto entre as 2 instruções, o **store conditional** falha.
  - **store conditional** retorna um valor indicando sucesso ou não.

Spin Locks

- **Locks** que um processador tenta adquirir continuamente executando dentro de um loop.
- Usados quando é esperado que o **lock** seja mantido por pouco tempo.
- Usados quando se deseja que o processo de locking tenha baixa latência.
- Consomem tempo de CPU.

Barreiras

- Força que todos os processos esperem até que todos alcancem a barreira, então são liberados.
- Implementação típica envolve 2 spin locks: um para proteger um contador de processos que chegam na barreira e outro para segurar os processos até que o último chegue.
- Escalabilidade limitada (contenção e latência).

Modelos de Consistência de Memória

O problema da consistência:

- Quando um processador deve ver um valor que foi atualizado por outro processador?
- Em que ordem um processador deve observar escritas feitas por outros processadores?
- Quais propriedades devem ser garantidas entre leituras e escritas em endereços diferentes feitas por processadores diferentes?

Consistência Sequencial

"Requer que o resultado de qualquer operação seja o mesmo como se os acessos executados por cada processador fossem mantidos em ordem e os acessos entre processadores diferentes fossem intercalados."

Implementação mais simples requer que qualquer acesso à memória seja atrasado até que todas as invalidações estejam concluídas.

A Visão do Programador

Programas **sincronizados** permitem implementação eficiente de modelos de consistência.

Todo acesso a dados compartilhados é ordenado por sincronizações.

"Uma referência é ordenada por sincronização se, para todas execuções, uma escrita numa variável por um processador e um acesso à mesma variável por outro estão separados por um par de operações de sincronização: o processador que escreve executa uma sincronização **após** a escrita e o segundo processador executa a outra **antes** do acesso."

Atualizações de dados sem ordenação por sincronização são chamadas **data-races**.

Programas sincronizados são **data-race-free**.

Estas sincronizações garantem exclusão mútua no acesso aos dados.

A ação de bloquear o dado é chamada **acquire**, e a ação de liberar **release**.

Ordenação de Operações na Memória

Além da sincronização, é necessário ordenar as operações na memória (**write** e **read fences**).

Fences são pontos fixos numa computação que impedem reads/writes através do fence.

Write fence executado por **P** garante que:

- Todos os writes executados por **P** antes do write fence já teminaram.
- Nenhum write que ocorre após o fence é iniciado.

Na consistência sequencial, todos os **reads** são **read fences** e todos os **writes** são **write fences** (limita otimizações por H/W, a ordem é estrita).

Para melhor desempenho **reads** devem ocorrer o **mais cedo** e **writes** o **mais tarde**.

Write fence pode causar stall até que todos writes sejam completados (inclusive invalidações).

Read fence é usada para determinar o mais cedo que um read pode ocorrer.

Modelos Relaxados de Consistência

Modelos relaxados criam oportunidade para read e write **latency hiding** definindo menos fences.

Modelos relaxados devem manter a semântica do programa como se fosse consistência sequencial.

Possíveis ordenações de reads e writes no uniprocessador:

- **R**→**R**: read seguido por read.
- **R**→**W**: read seguido por write, sempre é preservado em operações no mesmo endereço (antidependência).
- **W**→**W**: write seguido por write, sempre é preservado em operações no mesmo endereço (dependência de output).
- **W**→**R**: write seguido por read, sempre é preservado em operações no mesmo endereço (dependência).

Consistência sequencial preserva todas ordenações.

Ordenação relaxada permite a antecipação do término de uma operação que vem após outra.

Por ex., relaxando **W→R**, um read pode ocorrer antes que outro write esteja concluído.

Modelo de consistência diz que ordem é **observada**.

O modelo também define ordenações entre sincronizações (fences): **S→W**, **S→R**, **W→S**, **R→S**, **S→S**.

TSO (Total Store Ordering): elimina **W→R** (para endereços diferentes), esconde latência de write (writes são write fences).

PSO (Partial Store Ordering): similar ao **TSO** eliminando **W→W** (não conflitantes) permitindo pipelining de writes.

Weak Ordering: também elimina **R→R** e **R→W** e mantém apenas ordenações relacionadas com as sincronizações.

Release Consistency: similar a **weak ordering** distinguindo entre sincronizações $\mathbf{S}_\mathrm{A}$ (acquire) e $\mathbf{S}_\mathrm{R}$ (release).