Universidade Federal do Rio de Janeiro
Pós-Graduação em Informática
DCC/IM - NCE/UFRJ

Arquiteturas de Sistemas de Processamento Paralelo

Modelos de Consistência de Memória em Multiprocessadores

Gabriel P. Silva

Introdução

- Um modelo de consistência de memória define um *contrato* entre o software e o sistema de memória, ou seja, se o software obedecer certas regras, o sistema de memória funcionará corretamente.
- Como forma de aumentar o desempenho, as técnicas de "pipelining" e de "bufferização" dos acessos são usualmente empregadas em sistemas multiprocessadores com caches.
- Para que estas técnicas possam ser empregadas sem problemas, é necessário que algumas condições e técnicas sejam consideradas no projeto de multiprocessadores.

Introdução

- Um modelo de consistência de memória define um *contrato* entre o software e o sistema de memória, ou seja, se o software obedecer certas regras, o sistema de memória funcionará corretamente.
- Como forma de aumentar o desempenho, as técnicas de "pipelining" e de "bufferização" dos acessos são usualmente empregadas em sistemas multiprocessadores com caches.
- Para que estas técnicas possam ser empregadas sem problemas, é necessário que algumas condições e técnicas sejam consideradas no projeto de multiprocessadores.

Consistência Seqüencial

- Um sistema é seqüencialmente consistente se o resultado de qualquer execução é o mesmo que seria obtido se as operações de todos os processadores fossem executados em uma dada ordem seqüencial, e as operações de cada processador individualmente aparecessem, nessa seqüência, na ordem especificada por seu programa.
- Ou seja, a execução paralela de um programa é seqüencialmente consistente se qualquer uma de suas execuções é equivalente a uma execução entrelaçada em um único processador.

Consistência Seqüencial

- A condição necessária e suficiente para que um sistema com acesso de memória indivisível (atômico) seja seqüencialmente consistente é que os acessos de memória sejam feitos na ordem especifica no programa.
- Sistemas que obedecem a essa condição são chamados de **fortemente ordenados.**
- Um acesso de memória é atômico se seus efeitos são observáveis para cada processador do computador paralelo ao mesmo tempo.

Consistência Seqüencial

- Quando caches são adicionados aos processadores de um computador paralelo, os acessos à memória podem se tornar não atômicos.
- Sistemas multiprocessadores com barramento único possuem acessos de memória atômicos por causa de capacidade de "broadcast" simultâneo do barramento.
- Sistemas multiprocessadores com protocolos de coerência por diretório não possuem acessos de memória atômicos porque os efeitos do acesso de memória se espalham com diferentes velocidades para processadores diferentes.

Consistência Seqüencial

- **Exemplo:**

Processador P1	Processador P2
A = 0;	B = 0;
.....	.....
A = 1;	B = 1;
if (B==0) ...	if (A ==0) ...

- **Hipótese:**
  - A e B estão inicialmente nos caches dos dois processadores com valor 0.
- **Conclusão:**
  - Se a operação de invalidação das caches sofre um atraso longo e o processador que realizou a escrita pode prosseguir o processamento, torna-se possível que tanto P1 como P2 não tenham ainda percebido as escritas em A e B antes da execução dos testes.

Consistência Seqüencial

- **O modelo de consistência de memória seqüencial é o mais simples de ser entendido pelos programadores, mas impõe sérias limitações ao desempenho do sistema, como por exemplo:**
  - **Não se pode fazer uso de write-buffers com bypass de operações de leitura => quebra da seqüência store -> load; (região crítica).**
  - **Em arquiteturas baseadas em redes de interconexão diferentes de barramento, não se pode ter sobreposição de operações de escrita=> quebra da seqüência W -> W (stolen data).**
  - **Em arquiteturas com cache e que usam redes de interconexão diferentes de barramento é necessário manter-se a atomicidade das operações de escrita, incluindo-se o processo de invalidação/atualização das caches, que não é uma operação atômica.**

Consistência Seqüencial

- Ou seja, novos mecanismos em "hardware" deveriam ser inseridos para garantir a ordenação forte dos eventos.
- Contudo, a questão que se coloca é esta: a consistência seqüencial é realmente necessária?
- O uso de programas sincronizados oferece uma alternativa ao modelo de consistência seqüencial que preserva sua simplicidade e impacta menos no desempenho do sistema.

Alternativas de Modelos de Consistência de Memória

- Um programa é dito sincronizado se todo o acesso a dados compartilhados é ordenado por operações de sincronização, ou seja, se entre uma escrita de uma variável por um processador e sua leitura por outro há uma operação de "release" (liberação) e uma operação de "acquire" (aquisição):

```
Write (x)
.....
Release (S)
.........
Acquire (S)
........
Read (X)
```

Alternativas de Modelos de Consistência de Memória

- Vamos definir os ordenamentos possíveis entre leituras e escritas realizados por um único processador. Há quatro ordenamentos possíveis:
  - R → R: uma leitura seguida de uma leitura.
  - R → W: uma leitura seguida de uma escrita, que são sempre preservados se são para o mesmo endereço, já que existe aí uma antidependência
  - W → W: uma escrita seguida de uma escrita, que é sempre preservada se são para o mesmo endereço, já que existe uma dependência de saída.
  - W -> R: uma escrita seguida de uma leitura, que é sempre preservada se são para o mesmo endereço, já que existe uma dependência verdadeira.

Alternativas de Modelos de Consistência de Memória

- Se houver uma dependência entre uma leitura e uma escrita, então a semântica do programa requer que as operações sejam ordenadas. Se não houver dependência, então o modelo de memória determina quais ordenamentos devem ser preservados.
- Quando uma ordenação é relaxada, isto apenas significa que permitimos que uma operação executada posteriormente pelo processador complete em primeiro lugar
- Um modelo de consistência, na realidade, não restringe as ordens dos eventos, mas apenas a ordem em que os eventos *parecem* ter sido executados.

Alternativas de Modelos de Consistência de Memória

- Definimos "cercas" como pontos fixos na computação que asseguram que nenhuma leitura ou escrita será movida além da "cerca".
- Uma "cerca" de escrita executada pelo processador P assegura que:
  - Todas as escritas realizadas por P que ocorrerem antes de P executar a operação de "cerca" de escrita foram completadas, e
  - Nenhuma escrita que ocorrer depois da "cerca" em P será iniciada antes de P.
- O efeito prático para a "cerca" de escrita é causar a parada da execução do programa até que todas as escritas pendentes tenham sido completadas.

Alternativas de Modelos de Consistência de Memória

- As "cercas" de leitura podem ser definidas de maneira similar. O efeito prático para as "cercas" de leitura é marcar o ponto mais recuado (cedo) na computação em que uma leitura pode ser realizada.
- Os modelos de consistência de memória mais relaxados, a serem apresentados a seguir, oferecem o potencial de esconder as latências de leitura ou de escrita definindo um número menor de "cercas" de leitura e de escrita, já que no modelo seqüencial todas as leituras e escritas são "cercas".

Total Store Ordering

- Se relaxarmos o ordenamento entre uma leitura e uma escrita (para diferentes endereços), eliminando a ordem W→ R.
- Esse modelo permite que uma leitura posterior a uma escrita seja realizada antes que essa escrita seja vista por todos os processadores.
- Uma operação de sincronização S (uma "cerca") deve ser realizada antes da leitura para garantir que a escrita se realize antes da leitura.
- Este modelo de consistência é conhecido como *consistência de processador* ou como *total store ordering* (TSO).

Partial Store Ordering

- Se permitirmos que escritas não conflitantes possam completar fora de ordem, relaxando a ordem W→W, obteremos o modelo denominado "partial store ordering" (PSO).
- Esse modelo permite que o sobreposição de operações de escrita ou o uso da técnica de "pipelining", ao invés de obrigar uma operação de escrita terminar antes da outra.
- Uma operação de escrita deve causar uma parada apenas quando uma operação de sincronização, que ocasiona uma operação de "cerca" de escrita, for encontrada.

Ordenação Fraca

- Outra grande classe de modelos relaxados elimina as ordenações do tipo R→R e R→W, além das outras duas. Este modelo, que é chamado de "ordenação fraca", não preserva a ordenação entre referências, exceto pelo seguinte:
  - Uma leitura ou escrita é completada antes que qualquer operação de sincronização executada na ordem do programa pelo processador depois da leitura ou escrita.
  - Uma operação de sincronização é sempre completada antes de quaisquer leituras ou escritas que ocorrem na ordem do programa depois da operação.

Ordenação Fraca

- As únicas ordenações impostas pela ordenação fraca são aquelas criadas pelas operações de sincronização.
- Ou seja, são necessárias operações de sincronização explícitas para garantir a ordem em que as operações de leitura e escrita se realizarão.
- Apesar de termos eliminado todas essas restrições, o processador só poderá tirar alguma vantagem se puder fazer leituras não bloqueantes e a primeira leitura for um “miss” na cache, de modo a que possamos prosseguir com a computação.

Release Consistency

- Um modelo mais relaxado de consistência pode ainda ser obtido se fizermos uma distinção entre as operações de sincronização que são usadas para *adquirir* um acesso a uma variável compartilhada (Sa) e aquelas que *liberam* um objeto para permitir que um outro processador adquira o acesso (Sr).
- Se decompormos a sincronização S do modelo fraco em Sa e Sr, esse modelo, denominado de "Release Consistency", não possui as seguintes restrições: W→Sa, R→Sa, Sr→R e Sr→W.
- Ou seja, uma leitura ou escrita que precede uma "aquisição" não precisa completar antes da "aquisição", e que uma escrita ou leitura posteriores a uma "liberação" não precisa esperar a "liberação" terminar.

Release Consistency

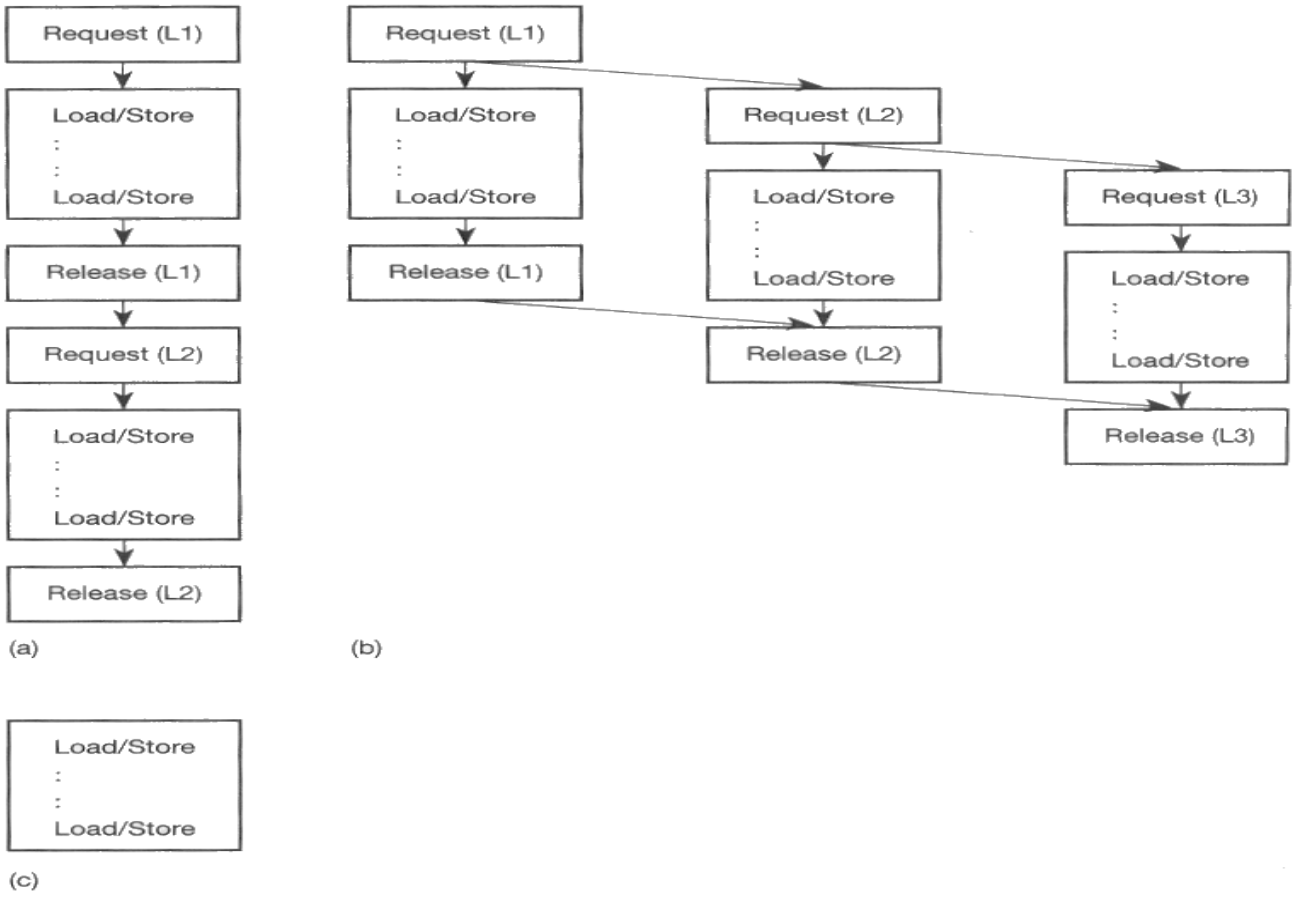

Figure 18.36 Comparison of the weak and release consistency models (Gharachorloo et al., 1990). (a) Weak consistency; (b) release consistency; (c) load and store operations can be executed in any order. © 1990 IEEE

Lazy Release Consistency

- Relaxa ainda mais o modelo Release Consistency através da observação que operações de acesso em variável compartilhada realizadas entre um "acquire" e um "release" só precisam estar concluídas antes do próximo "acquire", quando um novo processador pode vir a fazer acesso novamente essas variáveis.

Sincronização

Sincronização

- Operações de sincronização são necessárias em programação paralela.
- Duas operações de sincronização são largamente empregadas:
  - Exclusão Mútua: garante o acesso de um único processo ou "thread" a uma região crítica do programa que, em geral, modifica valores em uma estrutura de dados global.
  - Barreira: força que a execução de um conjunto de processos ou "threads" fique bloqueada até que todos os processos ou "threads" daquele conjunto tenham atingido um determinado ponto do programa (barreira).

Sincronização

- As operações de sincronização são implementadas com o uso de instruções do tipo "read-modify-write" (test-and-set) atômicas ou de um conjunto de instruções capaz de ler e modificar um dado na memória de forma atômica.
- A latência de operações de sincronização representa comumente um "gargalo" no desempenho de aplicações paralelas em multiprocessadores em larga escala.
- A implementação das operações de sincronização deve buscar minorar seu efeito negativo no desempenho das aplicações.

Sincronização por Exclusão Mútua

- Se dá através de uma operação de aquisição de um lock (*acquire*) e posterior liberação desse lock (*release*).
- Normalmente implementada com base em instruções atômicas do tipo: swap, test-and-set, fetch-and–increment.
- Exemplo de implementação (SPIN LOCK):

```
loop:   ld      r2, #1;
        swap    r2, (r1);   /* acquire */
        bne     loop
                .
                .           ( Região Crítica )
                .
        sto     #0,(r1) /* release  */
```

Sincronização por Exclusão Mútua

- Apenas um dos processos, tentando fazer a aquisição do **lock**, será bem sucedido na operação de **swap** pelo fato dessa operação ser atômica → só um processo entra na região crítica de cada vez.
- Usa-se **spin lock** ("espera ocupada") somente quando é sabido que o tempo para liberação de um **lock** é curto. Em caso contrário, o processo aguardando o lock é colocado para "dormir". A liberação do **lock** desperta os processos que estão aguardando.

Sincronização por Exclusão Mútua – Aspectos de Implementação

- É desejável que o *loop* para realizar o *acquire* de um lock seja feito numa cópia local "cacheável" da posição de memória contendo o lock para melhorar o desempenho e evitar "hot spots" na rede de interconexão.
- A solução anterior é inadequada, pois o loop contém operações de escrita na memória. Com múltiplos processadores tentando fazer o *acquire,* várias dessas escritas resultarão em "write miss" (caches invalidados pelas escritas de outros processadores) ➔ degradação do desempenho.

Sincronização por Exclusão Mútua – Aspectos de Implementação

- Solução modificada (apenas operações de leitura no loop):

```
Loop:   ld      r2, (r1);
        bne     loop;
        swap    r2, (r1)
        bne     loop;           /* acquire */
        .
        .                       (Região Crítica)
        .
        sto     #0, (r1)        /* release */
```

Observação:

- A instrução **swap** só é executada se a instrução **ld** revelar que o lock está livre.

Sincronização por Exclusão Mútua – Aspectos de Implementação

- O esquema apresentado, contudo, não escala bem para arquiteturas DSM com grande número de processadores devido ao tráfego gerado quando o lock é liberado (*released*).
- Princípio básico da solução (queueing lock):
  - Se o *lock* estiver ocupado ao tentar se fazer um *acquire*, o processo se inscreve em uma lista informando o endereço da variável "cacheável", onde estará realizando o "spin lock".
  - O processo que faz o "release do lock" busca na lista o próximo processo que deseja fazer o *acquire* do *lock* e coloca em zero a variável sobre a qual ele faz o "spin lock" para avisar da liberação.

Sincronização por Exclusão Mútua – Aspectos de Implementação

- Uma solução proposta no NYU UltraComputer é o uso de uma nova primitiva de "read-modify-write" chamada "fetch-and-add".
- Para a implementação eficiente dessa primitiva, todos os módulos de memória são equipados com um circuito somador.
- Para evitar a ocorrência de "hot-spots", a rede de interconexão omega pode ter somadores em cada um das suas chaves, de modo que os resultados vão se combinando até a memória e de volta para cada processador.

Sincronização por Barreira

- A implementação típica usa 2 spin-locks com as seguintes funções:
  - Proteger o incremento do contador que conta quantos processos já atingiram a barreira.
  - Reter o processo até que todos os processos atinjam a barreira.
- Exemplo de implementação básica de barreira:

```
acquire (conta_lock);
if (contador == 0) lib_barreira = 0;
contador++;
release (conta_lock);
if (contador == total)
   {      contador  = 0;
          lib_barreira = 1;
   }
   else
   {      while (lib_barreira == 0);
   }
```

Sincronização por Barreira

- Implementação básica pode apresentar problemas quando a barreira é usada dentro de um "loop".
- Se um processo A foi suspenso pelo SO quando executava a operação de SPIN (não saiu barreira) e um outro processo B volta à barreira, todos os demais processos que voltarem à barreira vão esperar indefinidamente nesse ponto, pois o processo A não conseguirá sair do "spin".
- Soluções:
  - Contar também os processos que saem da barreira e não permitir que nenhum processo re-entre na barreira até que todos tenham saído → aumenta a latência da barreira.

Sincronização por Barreira- Aspectos de Implementação

- **Soluções:**
  - **Sense-reversing barrier: uso de variável privada por processo inicializada com valor 1 e complementada a cada re-entrada na barreira:**

```
local_sense =  ~local_sense;
acquire (conta_lock);
if (contador==0) lib_barreira= ~local_sense;
contador++;
release (conta_lock);
if (contador == total)
{
        contador = 0;
        lib_barreira = local_sense;
}
else
        while (lib_barreira != local_sense);
```

Memória Virtual Compartilhada

Memória Virtual Compartilhada

- Conceito introduzido através da tese de doutorado de Kai Li em 1986 (Yale University).
- Objetiva oferecer um espaço de endereçamento virtual global compartilhado por processadores interligados em uma arquitetura baseada no modelo de troca de mensagens.
- A unidade de coerência de memória é uma página
- Cada página pode estar definida no sistema de gerência de memória como inválida, read-only ou read/write.
- Cada página possui um "home node" onde é armazenado o diretório informando que processadores possuem cópia daquela página.

Memória Virtual Compartilhada

- Quando ocorre um "page fault" em operações de leitura (acesso a uma página no estado inválido), o processador obtém uma cópia da página que passa a estar liberada apenas para leituras (read-only). Caso não haja um "page frame" vago para a nova página, uma página antiga terá que ser descartada.

- Quando ocorre um "page fault" em operações de escrita (página inválida ou read-only), uma cópia da página é trazida para aquele processador se necessário (página inválida) e todas as cópias daquela página em outros processadores (segundo o diretório no "home node") são invalidadas.

Implementações de Memória Virtual Compartilhada

- Ivy (Kai Li, 1986)
  - Usa um modelo de consistência de memória seqüencial.
  - Mensagens de invalidação de páginas são enviadas a cada operação de escrita.
- Munin (Carter, Bennett e Zwaenepoel, 1995)
  - Usa o modelo "release consistency".
  - Mensagens de invalidação de páginas são enviadas apenas no momento da operação de "release" para todos os processadores que compartilham páginas escritas após o "acquire".

Implementações de Memória Virtual Compartilhada

- Threadmarks (Amza, Cox et ali., 1996)
  - Usa o modelo "lazy release consistency".
  - Mensagens de invalidação só são enviadas do processador que faz o último "release" para o processador que faz o próximo "acquire".
  - Suporta múltiplos escritores na mesma página.
- Midway (Bershad, Zekauskas e Sawdon, 1993)
  - Usa o modelo "entry consistency".
  - Utiliza a atualização de páginas, ao invés da invalidação, quando ocorrem operações de escrita.

O Sistema "Threadmarks"

- Implementa o modelo de memória virtual compartilhada em uma rede de workstations (Computer, fev. 96).
- Utiliza o modelo "lazy release consistency" e suporta múltiplos escritores na mesma página.
- Quando ocorre um "page fault" em algum processador, uma cópia da página é gerada para aquele processador com permissão apenas de leitura.
- Na primeira tentativa de escrita da página, ocorre um "page fault" por violação da permissão de acesso e uma cópia gêmea (twin) da página é criada. A cópia original é desprotegida para a escrita.

O Sistema Threadmarks

- Na operação de "release", é gerado o conjunto de "diffs" (diferenças entre a página que foi escrita e a cópia "twin").
- Todos os outros processadores que possuem cópia dessa página são marcados como tendo cópia desatualizada.
- Quando um processador realiza uma operação de "acquire" para acessar essa mesma página, ocorre um "page fault" e uma cópia atualizada da página é gerada fazendo o uso dos "diffs".
- Pode haver múltiplos escritores na mesma página desde que os blocos de memória lidos/modificadas por cada um deles não se sobreponham ("false sharing"). Cada processador gera seu conjunto de "diffs".
- O sistema threadmarks trabalha com processos e não com "threads".

Técnicas para Ocultar Latências

- Em arquiteturas DSM de grande porte, a latência em operações de acesso remoto à memória compartilhada é ocultada do processador com base nos seguintes princípios:
  - Disparo antecipado das operações de leitura/escrita na memória.
  - Ocupação do processador com trabalho útil enquanto as operações de leitura/escrita não são concluídas.

Técnicas para Ocultar Latências

- As técnicas normalmente empregadas são as seguintes:
  - Uso de modelos de consistência de memória relaxada
    - Em operações de escrita o processador não necessita de nenhuma resposta → processador não precisa esperar o fim da escrita em modelos de consistência relaxada → write-buffers desacoplam o processador da latência de operações de escrita.
  - Prefetching
    - Busca mover antecipadamente para perto do processador dados que serão previstos para serem usados em breve.

Técnicas para Ocultar Latências

- As técnicas normalmente empregadas são as seguintes:
  - Multithreading
    - Utiliza processadores especiais que realizam rapidamente a troca de contexto entre threads quando uma operação de memória demorada é disparada.

Prefetching Binding x Non-Binding

- Binding prefetch ocorre quando o valor do dado buscado antecipadamente é definido no momento da operação de prefetch (ex: operação de load em um registrador), não sendo afetado por operações de manutenção de coerência das caches.
- O uso de binding prefetch em um processador fica limitado, pois só pode ser aplicado quando se tem certeza de que nenhum outro processador irá alterar o dado entre o instante do prefetch e o efetivo uso do dado pelo processador.

Prefetching Binding x Non-Binding

- Com o non-binding prefetch , o dado permanece visível para as operações de manutenção de coerência de cache e é, portanto, mantido consistente até o seu uso real pelo processador (ex: prefetching de dado para o cache do processador).
- Non-binding prefetch não tem efeito sobre o modelo de consistência de memória e as operações de prefetch podem ser realizadas em qualquer lugar do código da aplicação.

Prefetching Hardware x Software

- Exemplos de prefetching em hardware são as buscas antecipadas de linhas longas de cache e a busca em avanço de instruções.
- A eficiência do prefetching em hardware é limitada pelo pouco conhecimento que o hardware pode ter sobre os padrões de acesso da aplicação.
- Operações de prefetching em software são feitas através da inserção explícita de instruções que executam essas operações no código da aplicação pelo programador ou compiladores inteligentes.

Prefetching Hardware x Software

- As vantagens do prefetching em software são:
  - O uso seletivo desse tipo de operação.
  - O intervalo entre a operação de prefetch e o momento real do uso da informação pode ser consideravelmente estendido, permitindo que latências muito grandes sejam ocultadas.
- As desvantagens do prefetching em software são:
  - Utilização de parte da largura de banda de instruções com as instruções de "prefetch".
  - Complexidade da programação ou necessidade de se ter um compilador sofisticado.