キャッシュの書き込みポリシーと仮想記憶

天野英晴

今回は前回の続きでキャッシュの書き込みポリシー、性能の検討をやって、仮想記憶を紹介します。

キャッシュ

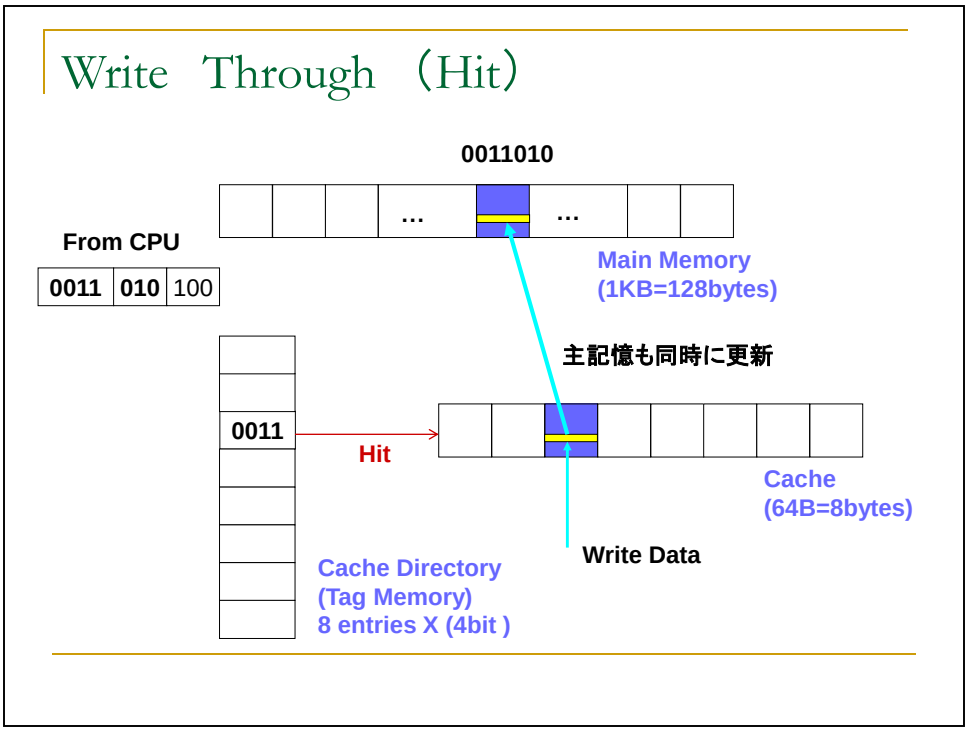
- 頻繁にアクセスされるデータを入れておく小規模高速なメモリ
  - CacheであってCashではないので注意
  - 元々はコンピュータの主記憶に対するものだが、IT装置の色々なところに使われるようになった
    - ディスクキャッシュ、ページキャッシュ..etc..
- 当たる(ヒット)、はずれる(ミスヒット)
  - ミスヒットしたら、下のメモリ階層から取ってきて入れ替える(リプレイス)
- マッピング(割り付け)
  - 主記憶とキャッシュのアドレスを高速に対応付ける
  - Direct map ⇔ Full associative cache
- 書き込みポリシー
  - ライトスルー、ライトバック
- リプレイス(追い出し)ポリシー
  - LRU (Least Recently Used)

ではメモリの基本がわかったところでキャッシュの話をしましょう。キャッシュとは頻繁にアクセスされるデータ(命令もデータの一種と考える)を入れておく小規模高速なメモリを指します。小銭のCashではなく、Cache(貴重なものを入れておく小物入れ)なのでご注意ください。この言葉はコンピュータの世界で大変有名になったので、IT機器の色々なところで使われるようになりました。ディスクキャッシュやページキャッシュとかがこの例です。キャッシュ上にデータが存在する場合は、ヒットと呼び、はずれるとミスヒット(ミス)と呼びます。ミスヒットしたら、下のメモリ階層から持ってきて入れ替えます。この処理をリプレイスと呼びます。キャッシュを理解するには三つのポイントがあります。一つはマッピングです。主記憶とキャッシュのアドレスを高速に対応付ける方法です。二つ目は書き込みポリシー、三つ目はリプレイスポリシーです。これを順に紹介しましょう。

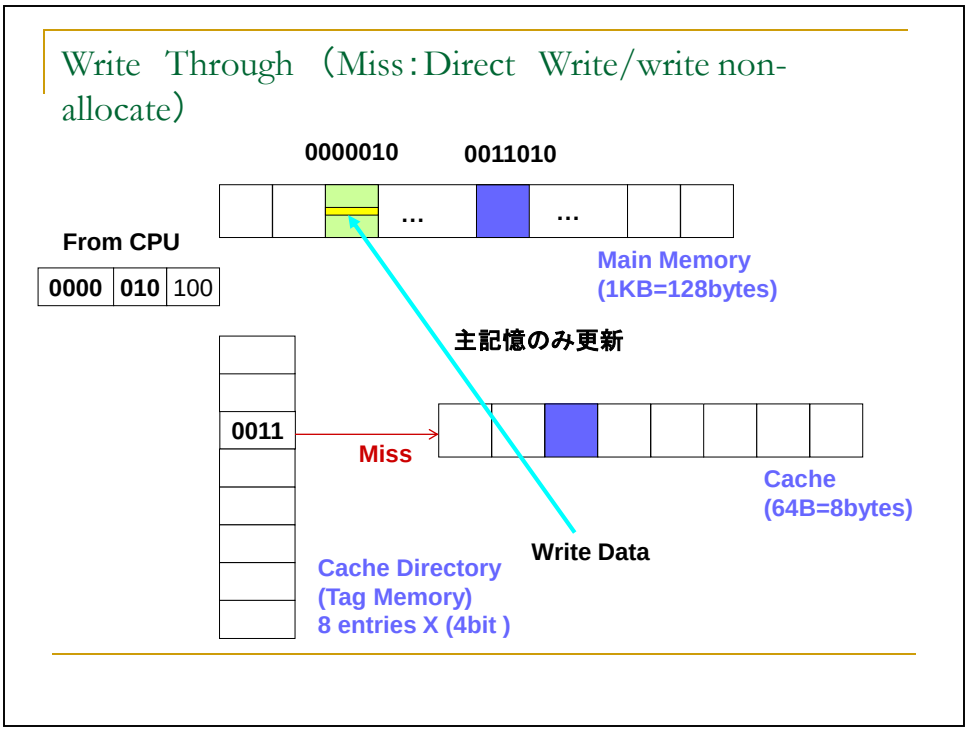
書き込みポリシー

- Write Through
  - 書き込み時に主記憶にもデータを書く
  - Direct Write:ミス時は主記憶だけに書く
  - Fetch-on-write:ミス時はリプレイスしてから書く
  - 主記憶に合わせると性能ががた落ち（Verilogの設計はそうなっている）だが、Write bufferがあれば性能がさほど落ちることはない
- Write Back
  - 書き込みはキャッシュのみ
  - キャッシュと主記憶が一致：Clean、違う：Dirty
  - Dirtyなキャッシュブロックは書き戻し（Write Back)をしてからリプレイス

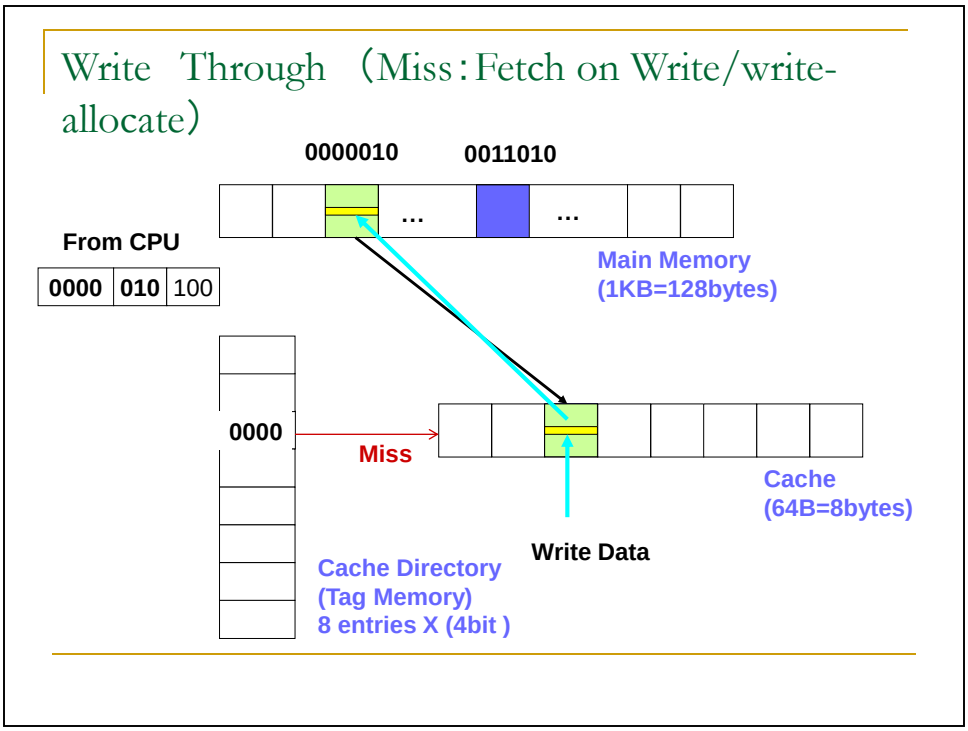
次のポイントである書き込みポリシーについて説明します。キャッシュから読み出す場合、ヒットすれば直接読み、ミスヒットすれば主記憶からブロックを取ってきて（リプレイス）してから読み出します。しかし書き込みの際どのようにするかには二つの方法があります。ライトスルーは、キャッシュに書き込む時に主記憶にもデータを書いてしまう方法で、キャッシュ上のデータと主記憶上のデータが常に一致するようにします。一方、ライトバックは書き込みはキャッシュだけにして、キャッシュ上のデータと主記憶のデータが一時的に異なった状態にすることを許しています。以下、順に説明します。



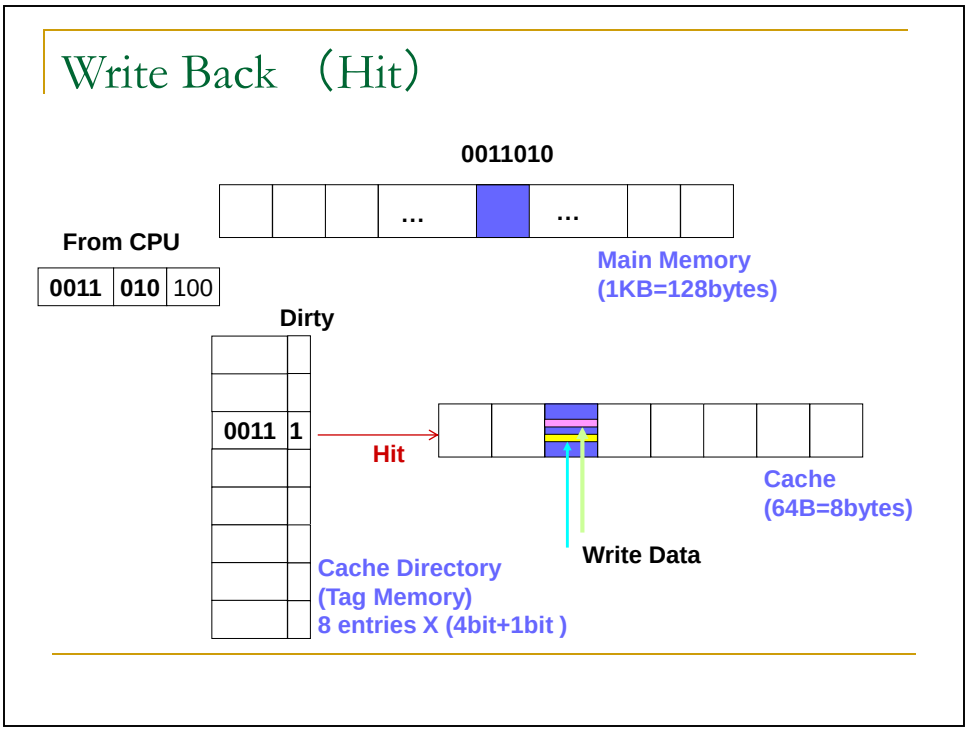
ライトスルーキャッシュは、図に示すようにヒットした場合は、キャッシュに書いたデータをそのまま主記憶に串刺しで書き込みます。



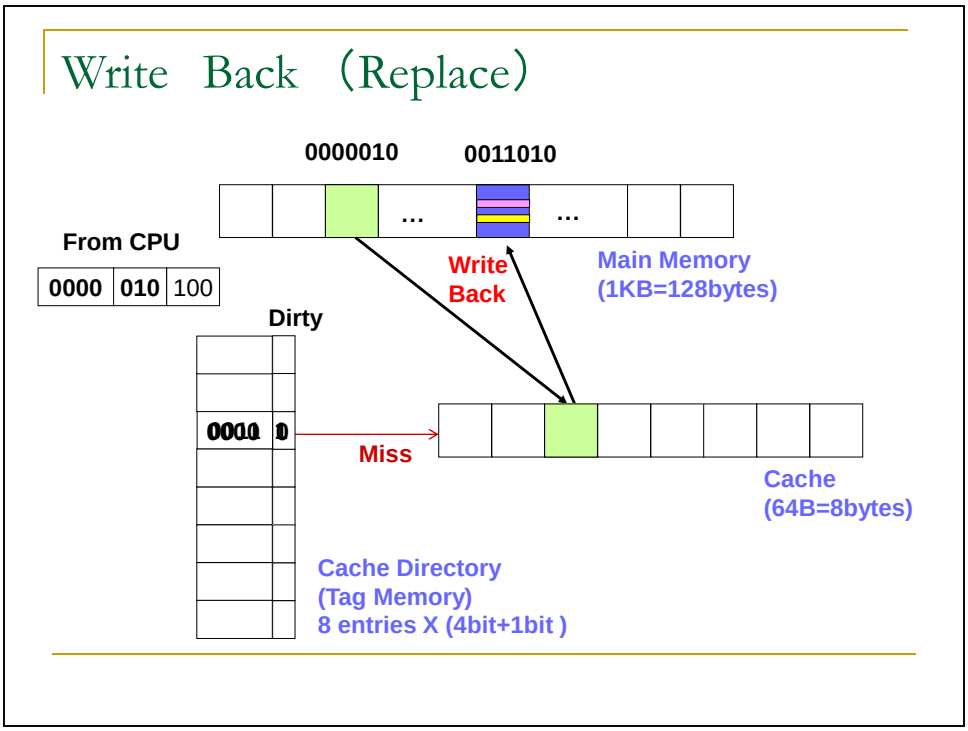
ではミスした場合はどうなるでしょう？この時の処理によりライトスルーキャッシュは二つの方法に分かれます。一つはダイレクトライトと呼び、ミスした場合、キャッシュをすっとばしてデータを直接主記憶に書いてしまう方法です。キャッシュへの書き込み信号をストップするだけなので、実装が簡単な利点があります。



もう一つの方法では、書き込みミスの場合も、読み出しミス同様、まず主記憶からブロックを取ってきてキャッシュに入れ（リプレイス）てやり、それから書き込みヒットと同様にキャッシュと主記憶に同時にデータを書き込みます。これをフェッチオンライトと呼びます。フェッチオンライトは、ダイレクトライトに比べて実装がやや複雑（リードミスとライトヒットを順に行えばよいのでそんなに複雑というほどではない）ですが、局所性の原則により書いたブロックには次には読み出しが予想されるので、この際にヒットする可能性が高く、ヒット率が若干改善されるという利点があります。



一方、ライトバックキャッシュでは、キャッシュにだけデータを書き込み、主記憶には書き込みません。このため、キャッシュの内容と主記憶の内容が違ってしまいます。この状態をダーティ(汚れちゃった)と呼び、主記憶と一致している状態をクリーンと呼びます。キャッシュディレクトリにこの状態を示すダーティビットを付けておき、最初に書いたときにこのビットをセットします。



ライトバックキャッシュはキャッシュにヒットしつづける限り、そこに書いて読めばよいので問題ないです。問題はこのキャッシュブロックがキャッシュから追い出されるときに生じます。今、キャッシュがミスしてブロックのリプレイスが起きる際に、今までのように単純に主記憶からブロックを持ってきて上書きすると、書いたデータが消えてしまいます。そこで、まず、ダーティなブロックを主記憶に書き戻し(ライトバックし)、それから新しいキャッシュブロックを取って来ます。ディレクトリを更新するとともにダーティビットを0にします。この書き戻しはダーティビットがセットされているブロックだけに必要です。クリーンなキャッシュに対しては今まで同様、単にキャッシュブロックを取ってくれば良いです。ダーティビットの存在によりこの部分で効率化を行っています。

ライトスルーとライトバック

- 「ライトスルーは主記憶を待たなければならないので非効率」というのは嘘
  - ちゃんとライトバッファを装備すれば性能的に悪くはならない
  - しかし、シングルライトが必要→DRAMに合わない
  - 常にデータの一致が取れるのがメリット、観測性が高い、I/Oで有利
- ライトバック
  - 常にデータ転送がブロック単位→DRAM、高速バスに適合
  - バスの利用率が下がる→マルチコアに適合

大体世の中はライトバックになりつつある

さて、ライトスルーとライトバックを比較してみます。「ライトスルーは遅い主記憶を待たなければならないので非効率」と書いてあるテキストもありますが、これは半分嘘です。書き込みの場合、CPUは終了を待たずに次の命令の実行に入れるので、キャッシュと主記憶の間にきちんとした中間記憶（ライトバッファ）を設けておけばライトスルーの性能はライトバックに比べてさほど落ちません。しかし、ライトバックは性質上、シングルワードの書き込みが必要です。先ほどメモリで紹介したように最近のDRAMはブロックライトしか受け付けないので、シングルワードの書き込みを行うためには1ブロック読み出して、この一部を変更して書き込む操作が必要になります。これは非効率です。したがってライトスルーはL1キャッシュとL2キャッシュの間などキャッシュ間のみで使われます。ライトスルーの良い点は常にデータの一致が取れることです。観測性が良く、来年やりますが入出力を行う場合に有利です。

一方、ライトバックは主記憶との転送が常にブロック単位なので、DRAMやブロック転送の得意な高速バスに良く合っています。またバスの利用率が下がるので、最近のマルチコアに適合します。世の中のマルチコア化が進むにつれ、ライトバックはライトスルーを圧倒して使われるようになっています。

シミュレーション

- c2kai/wthにライトスルー
- c2kai/wbackにライトバック
- それぞれctest.asmをシミュレーションしてみよう
  - make
  - make ctest
  - ./test >| tmp　で実行可能
- メッセージを見やすくした
- キャッシュの動きを観測しよう

ではキャッシュの書き込みについての動作を確認しましょう。c2kai.tarを取ってきてこのwth,wbackのそれぞれで単純なテストプログラムctest.asmを動かして様子を見ましょう。メッセージは多少見やすくなっていると思います。

リプレイスポリシー

- リプレイスの際、どのWayを選ぶか？
  - Direct map以外のキャッシュで問題になる
- LRU (Least Recently Used)
  - 最近もっとも使っていないwayを選ぶ
  - 2-wayならば簡単→ Verilog記述参照
  - 4-way以上は結構面倒→　擬似的なLRUでも大体OK
- 他にランダム、FIFOなどが考えられるが実際上あまり用いられない

最後のポイントは、リプレイスポリシーです。これはリプレイス、すなわち主記憶からブロックを持ってきて、キャッシュに入れる際に、セット内のどのウェイに入れるのかを決める方法のことです。セット内に1ウェイしかないダイレクトマップは入れる場所が一つに決まるので悩みがありません。セット内に複数のウェイがある場合は、どれかのウェイのブロックを選んで、ここに新しいブロックを入れる必要があります。このリプレイスポリシーで最も良く使われるのがLRU（Least Recently Used)

で、対象ブロックの中で、使われた時間が最も古いものを選ぶ方法です。つまり最近使われていないものを選びます。最近使われたものは、また使われる可能性が高いので、追い出しの対象にするのは良くないです。LRUは最後に使われたのが最も古いものを選ぶので、局所性の原則に適っています。これは2ウェイの場合、最後に使われた方のビットをセットしておけば良いので簡単です。Verilogのコードを見てもとても簡単なことが分かります。しかし4ウェイ以上は履歴を取っておく必要があって結構面倒です。実際は、最近使われたもの、その次に使われたもの、を除外して後は適当に選んでもさほど性能は低下しないので、擬似的なLRUで済ませる場合が多いです。

他にもランダムに選んだり、入ってきた順に選んだり（FIFO：First In First Out)する方法もあるのですが、実際上はLRU以外には用いられません。

キャッシュの性能

キャッシュオーバーヘッド付きCPI(Clock cycles Per Instruction)＝
理想のCPI ＋
命令キャッシュのミス率×ミスペナルティ ＋
データキャッシュの読み出しミス率×読み出し命令の生起確率×ミスペナルティ

- この式の問題点
  - ミスペナルティは書き戻しを伴うかどうかで違ってくる（Write Back)
  - ライトバッファの容量、連続書き込み回数によっては書き込みミスでもCPUは停止する
  - 書き込み直後に読み出しをするとキャッシュが対応できないでペナルティが増えることもある→ノンブロッキングキャッシュ
  - 実際は階層化されているのでそれぞれの階層を考えないといけない
  - プロセッサがOut-of-order実行可能ならば読み出し時にストールしないかもしれない（この話は3年生で）
- ちゃんと評価するにはシミュレータを使うしかない、、、、

理想のキャッシュを使った場合のCPI(Clock cycles Per Instruction)は、キャッシュミスが起きると延びてしまいます。キャッシュの性能は、キャッシュのオーバーヘッドを含むCPIの値で示すことができます。命令を一つ読み出す度に命令キャッシュがアクセスされます。このため、命令キャッシュのミス率×ミスペナルティでミス時のオーバーヘッドが表されます。ちなみにミスペナルティはミス時に増加するクロック数で表します。命令の中でデータを読み出す命令についてはデータキャッシュがミスすると、そのペナルティだけCPIが延びます。すなわち、データキャッシュの読み出しミス率×読み出し命令の生起確率×ミスペナルティがこれに加わります。この式ではデータキャッシュへの書き込みミスについては無視しています。これはCPUはミスが起きた場合でも次の命令を実行することができるからです。

ただし、この式は問題があります。まずミスペナルティは一定ではないです。Write Backキャッシュでは書き戻しを伴うかどうかで2倍くらい違ってきます。次にこの式では書き込みミスでもCPUは次の命令を実行できるとしましたが、書き込みが続いたり、書き込み命令がミスした直後に読み出しを行う時などその読み出しがヒットしてもキャッシュが使えない場合がでてきます。（これを防ぐには後に示すノンブロッキングキャッシュを使います）

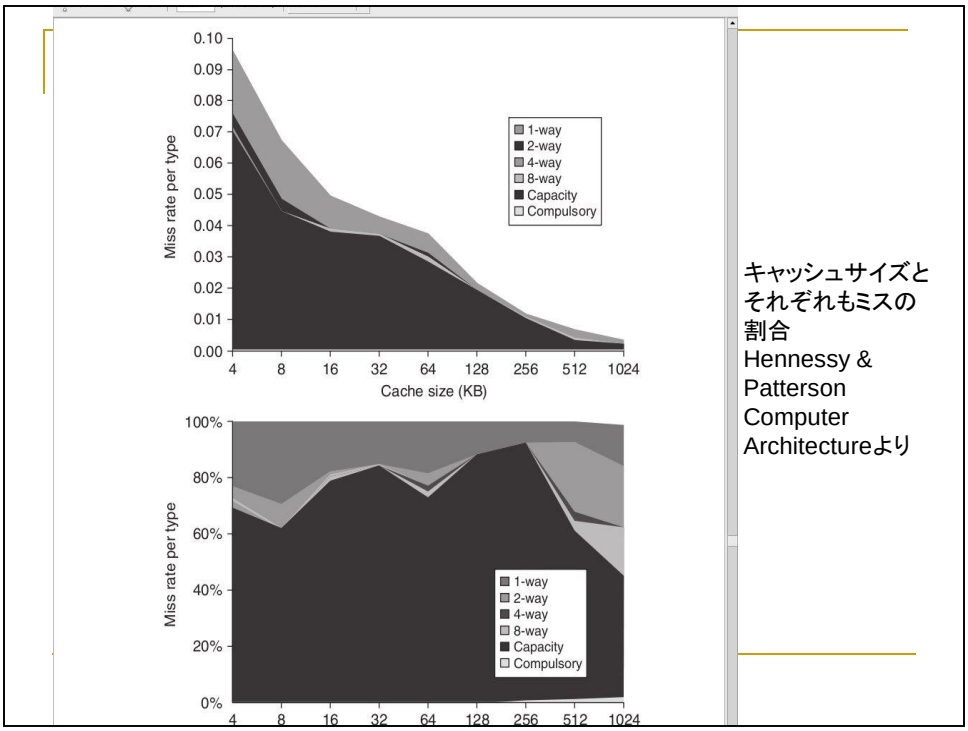
実際の記憶の階層は、1階層ではなくてもっと深いので、これも考えないといけません。最後にCPUがアウトオブオーダー実行可能（この話は来年やります）な場合、ミスが起きてもそのままオーバーヘッドにならない場合があります。

というわけで、キャッシュの性能をきちんと評価したかったらシミュレータを用いるしかありません。ここに示す式は、単純なものの中ではマシな方とお考え下さい。

ミスの原因：3つのC

- Capacity Miss：容量ミス
  - 絶対的な容量不足により起きる
- Conflict Miss:競合（衝突）ミス
  - 容量に余裕があっても、indexが衝突することで、格納することができなくなる
- Compulsory Miss (Cold Start Miss) 初期化ミス
  - スタート時、プロセス切り替え時に最初にキャッシュにブロックを持ってくるためのミス。避けることができない

先ほどの式がどの程度正確かどうかは疑問の余地があるとはいえ、キャッシュの性能がミス率とミスペナルティによって決まることは間違いないです。すなわち、キャッシュの性能を上げるには、ミス率を減らすか、ミスペナルティを小さくすれば良いのです。まずミスについて検討しましょう。ミスは、容量ミス、競合ミス、初期化ミスの三つに分けて考えることができます。英語の頭文字をとって3つのCと呼びます。容量ミスは、キャッシュの絶対的な容量不足により生じるミス、競合（衝突）ミスは、インデックスが衝突することによって格納できなくなってしまう問題、最後の初期化（Compulsory:強制、必須という意味です）ミスは、スタート時、プロセス切り替え時など最初にキャッシュにブロックを持ってくるためのミスです。これは避けることができません。



キャッシュサイズとそれぞれもミスの割合
Hennessy & Patterson Computer Architectureより

このグラフは、キャッシュの原因を分類したもので、横軸にキャッシュ容量、縦軸にミス率を取っています。1-way(ダイレクトマップ)、2-way…とウェイ数が増えていくにつれ、競合ミスが減っていきます。ウェイ数を無限に増やしても減らすことができない部分が容量ミスになります。初期化ミスは下のほうに見える非常に細い筋です。下のグラフは上のグラフと同じデータですが、ミス率全体を100%と考えて、この中のミスの成分を示しています。

キャッシュの基本的なパラメータ

- 容量を増やす
  - ○容量ミスはもちろん減る。競合（衝突）ミスも減る。
  - ×コストが大きくなる。ヒット時間が増える。チップ（ボード）に載らない
- Way数を増やす
  - ○競合（衝突）ミスが減る
    - キャッシュ容量が小さいと効果的、2Wayは、2倍の大きさのDirect Mapと同じ位のミス率になる
    - キャッシュ容量が大きい場合、残った不運な競合（衝突）ミスを減らす効果がある
  - ×コストが大きくなる。ヒット時間が増える。4以上はあまり効果がない。
- ブロックサイズを大きくする
  - ○局所性によりミスが減る。
  - ×ミスペナルティが増える。（ブロックサイズに比例はしないが、、）
    - キャッシュ容量が小さいと衝突ミスが増える
  - 容量に応じて適切なブロックサイズを選ぶ。32byte-128byte

ミス率を減らすのに最も効果的な方法は容量を増やすことで、このことで容量ミス、競合ミスの両方が減ります。しかし、容量が増えるとコストが大きくなり、ヒット時間が増えます。さらに物理的にチップやボードに搭載できる量は制限されます。

次にWay数を増やすと競合（衝突）ミスが減ります。先の図を見ると、キャッシュ容量が小さいとき、2wayは2倍の容量のダイレクトマップとほとんど同じくらいのミス率になります。Way数を増やす効果は4，8と大きくするほど小さくなってしまい、4以上にしてもほとんど効果がなくなります。Way数を増やす効果はキャッシュ容量が小さいときに大きいですが、逆に容量が非常に大きい場合にも、不運な競合ミスを減らしてミス率を非常に小さくするために有効です。前のページの図をご覧下さい。Way数を増やすと比較器やマルチプレクサのコストが大きくなり、ヒット時の遅延が増えます。このため8より大きいものはほとんど使われません。

最後にブロックサイズを大きくする手があります。これについては次のページにグラフが載っています。

ブロックサイズと
ミスの割合
Hennessy &
Patterson
Computer
Architectureより

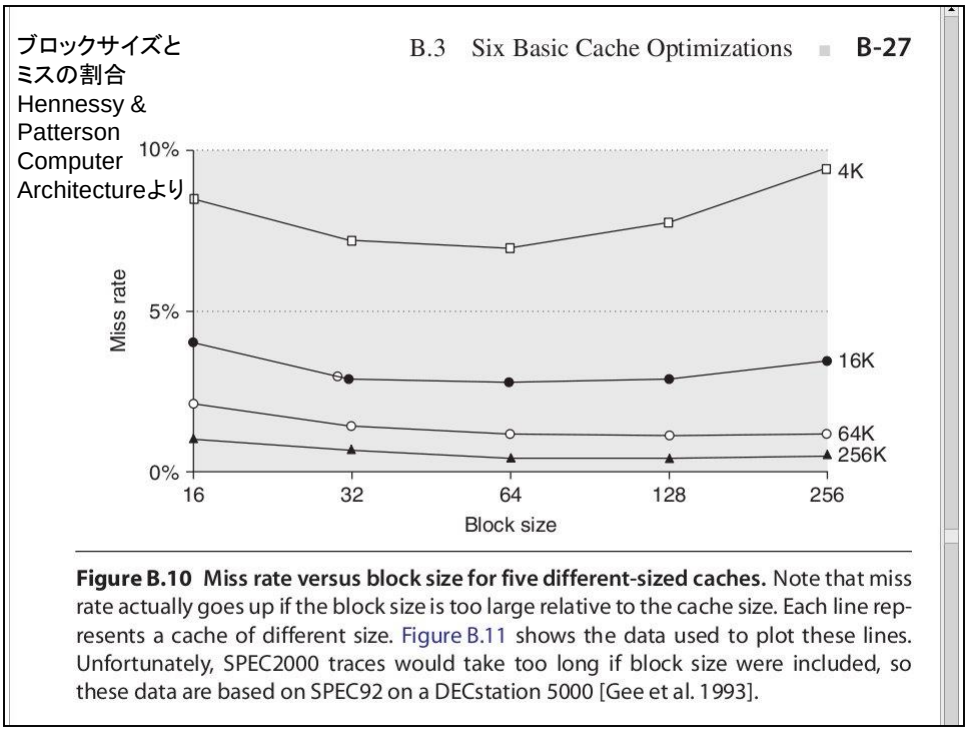
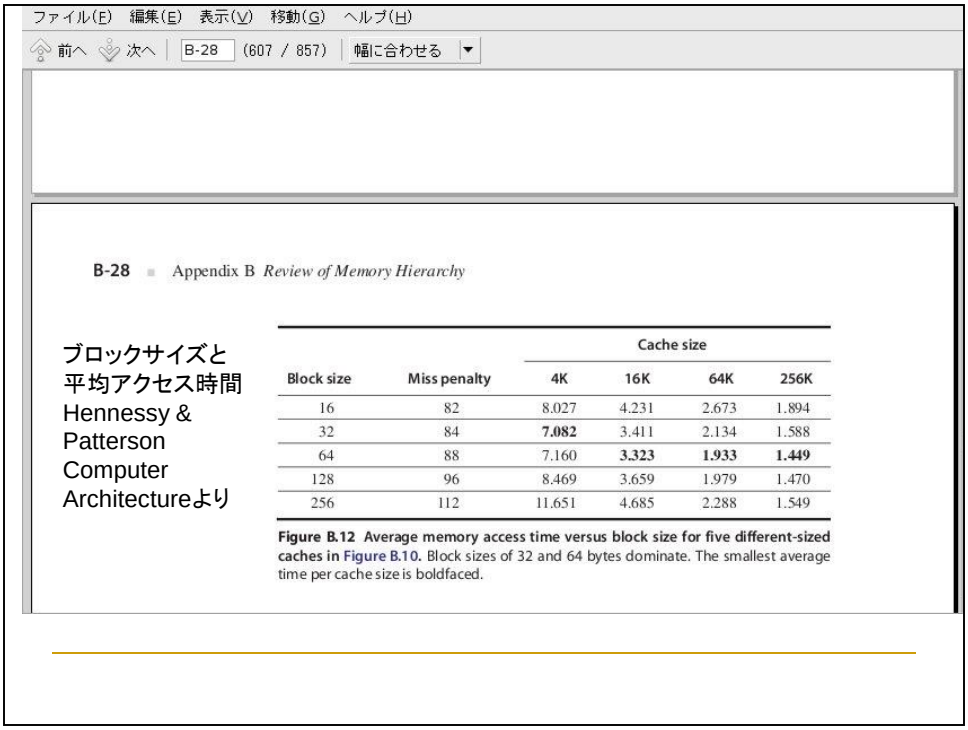


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 [Gee et al. 1993].

ブロックサイズを増やすと、一度に周辺のデータを取って来ることができるので、局所性の原則からミス率を減らすことができます。しかし、キャッシュ容量自体が小さいときにブロックサイズを大きくすると、インデックスが重なる可能性が増えるため、競合ミスが増えてしまいます。この図はサイズをパラメータに取っているので、一番小さい4Kでこの傾向がはっきり出ています。64K以上のサイズならばブロックサイズを増やしてもミス率は上がりません。とはいえ下がることもないです。

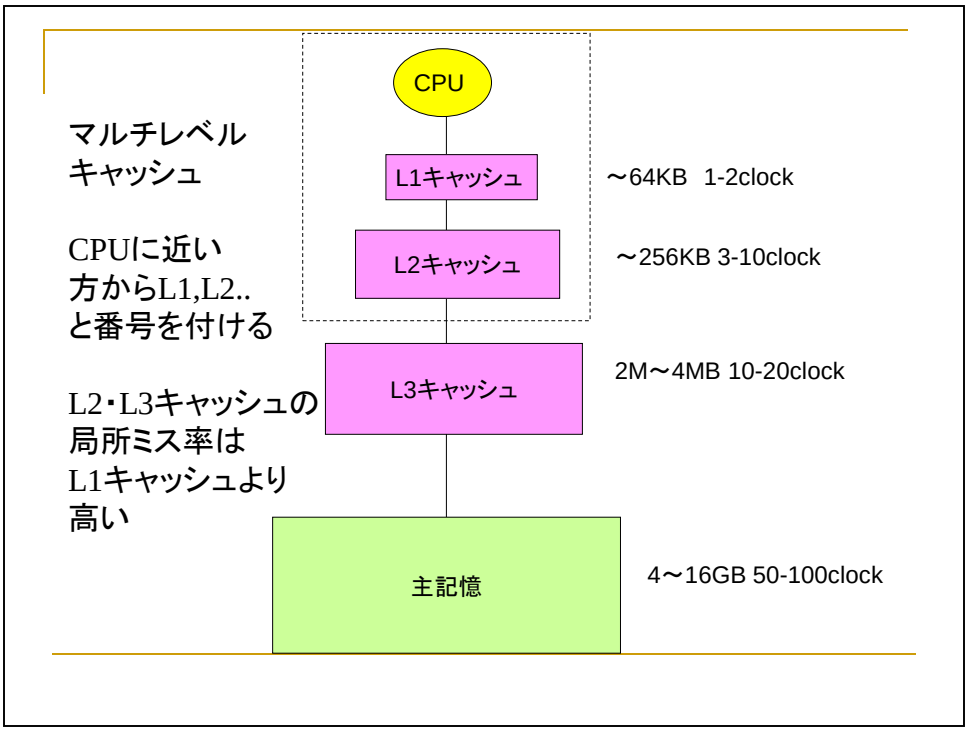


ブロックサイズを増やす問題点は、ミスペナルティが大きくなることです。大きいサイズのデータを動かすのでこれは当然です。しかしDRAMやバスの性質上、サイズに比例して増えるのではなく、増え方はずっとおだやかなものになります。この表はひとつの例であり実装でいろいろ変わりますが、ブロックサイズとミスペナルティ(クロック数)を示しています。キャッシュサイズのところに示してる数値はミス率とペナルティを掛けたものです。太字がもっとも小さい値です。これを見るともっとも小さくなるのは、ブロックサイズが32-64バイトであることがわかります。実際のキャッシュのブロックサイズもこの程度の値を取ります。

キャッシュの性能向上法

- マルチレベルキャッシュ（階層キャッシュ）
  - CPU－Memory間に複数のキャッシュを設ける
- ノンブロッキングキャッシュ
  - ミス処理の間にも次のアクセスを受け付ける
- Critical Word FirstとEarly Restart
  - CPUに対して可能な限り早くアクセスされたデータ（命令）を渡す
- プリフェッチ
  - あらかじめ早めにキャッシュへの要求を出しておく
- コンパイラによる最適化
  - キャッシュに入るようにデータ構造を変換

では、キャッシュの性能を向上する代表的な手法を紹介します。ざっと概念だけ理解しましょう。



階層キャッシュは主記憶とCPUの間にアクセス時間と容量の違った複数のキャッシュを置く方法です。まずCPUの直近に小容量でもできるだけ高速なキャッシュを置きます。これがL1キャッシュです。次にCPUと同じチップ内に容量が大きいL2キャッシュを置きます。最近のマルチコアプロセッサではさらに次のL3キャッシュも同一チップ内に置く場合が多いです。この図では、次にCPUチップ外で同じボード上にオンボードキャッシュを置きます。オンボードキャッシュは高速SSRAMが用いられます。この次のレベルがDRAMの主記憶になります。

マルチレベルキャッシュの制御

- Multi-level Inclusion
  - 上位階層のキャッシュが下位階層の内容を全て含む
  - 階層間のやり取りは、キャッシューメモリ間と同じ
  - メモリシステム中にデータの重複が数多く存在
- Multi-level Exclusion
  - 上位階層のキャッシュと下位階層のキャッシュの内容が重なることはない
  - 階層間のやり取りは、リプレースというよりはスワップ

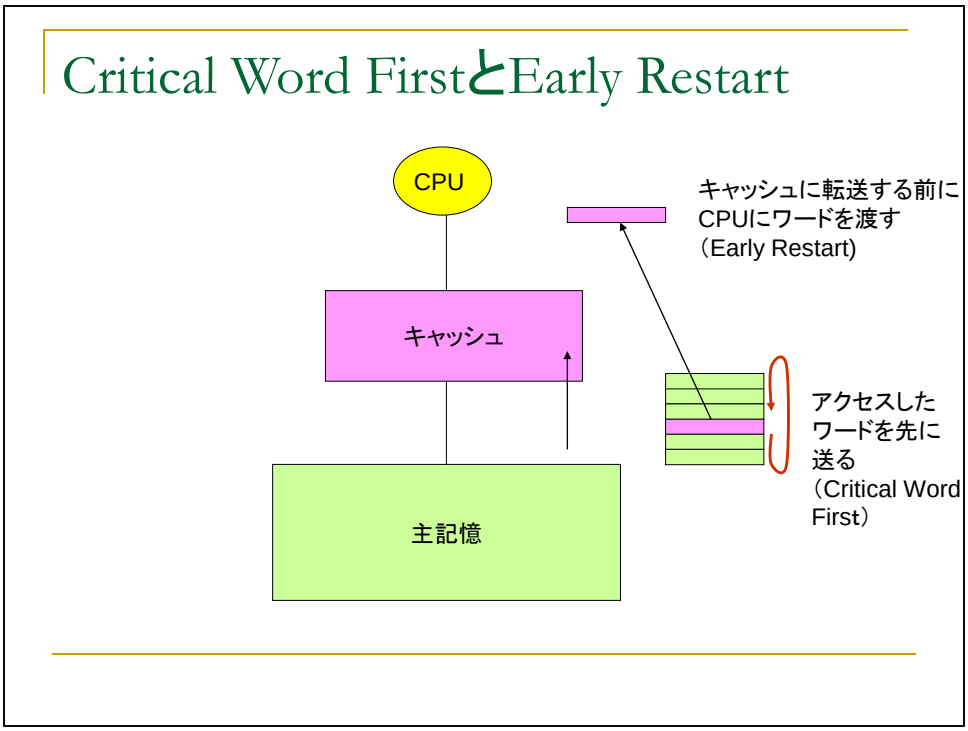
マルチレベルキャッシュの制御法にはマルチレベルインクルージョンとマルチレベルエクスクルージョンがあります。マルチレベルインクルージョンは、上位階層のキャッシュがそれより低い階層の内容を全て含んでいます。したがって階層間のやり取りはキャッシューメモリの場合と同じで、それぞれの階層で今まで紹介してきた構成にすれば良く、一度リプレイスされたキャッシュブロックに再びアクセスがあった場合、一つ深い階層に存在する利点があります。しかしメモリシステム全体として、データのコピーが複数個所に存在することになり、無駄が多いといえます。

一方でマルチレベルエクスクルージョンは、上位階層のキャッシュと下位階層のキャッシュを入れ替えてしまい、内容が重ならないようにする方法です。ミスヒットが起きたら、キャッシュブロックは上位階層に移動し、その場所にあったブロックが下位階層に移動します。リプレースというよりもスワップを行います。マルチレベルエクスクルージョンは、メモリ全体の利用効率は良いのですが、ミスが起きた際、深い階層までとりに行かなければならない場合が増えます。

ノンブロッキングキャッシュ

- キャッシュが動作中にも次のアクセスを受け付ける
  - キャッシュの操作をパイプライン化（来年やる）する
  - メモリアクセスを強化しないとノンブロッキングキャッシュにはできない
  - 実際はミス中のヒットを1回許せば大体OK
- CPUがアウトオブオーダ実行可能でないと効果が小さい→来年

キャッシュの書き込みミスが起きても、CPUは引き続き命令を実行することができます。この場合、再びCPUが読み出し命令を実行したらキャッシュはどうすれば良いでしょう。キャッシュコントローラが単純なものだと、書き込みミスヒット時の処理中は次のアクセスが受け付けられず、例えヒットしても値を返すことができません。このようなことを防ぐためにキャッシュ自体をパイプライン化（これも3年生でやります）して、連続した要求を処理できるようにするのがノンブロッキングキャッシュです。ノンブロッキングキャッシュは、次々に到着した要求を待たせることなしに次々と受け付けるのが理想ですが、実際にはミスの間に起きた一つのヒットを扱えれば、十分な性能向上が得られます。また、この方法はCPUがアウトオブオーダ実行できないとあまり効果が上がりません。このため今年はこの程度の説明にとどめたいです。



クリティカルワードファーストとアーリーリスタートはもっと簡単な実装上のアイディアです。ミスをしたキャッシュを主記憶から取って来る際に、通常はまずブロックを取ってきてキャッシュに転送し終わってから、CPUにそのことを知らせてそのワードを読みこんでもらいます。しかし、CPUはブロック全体の転送が終わるのを待っている必要はなく、自分が欲しいワードが主記憶から読み出されてきたら、すぐそれを受け取って次の処理に移ればペナルティが減ります。CPUが動くのと並行してキャッシュの転送も引き続き行われます。さらにこの考え方を進め、ブロックの先頭からではなく、CPUが要求したワードから先に読み出して1ブロック分をぐるっとまわって読む方法をクリティカルワードファーストと呼びます。これはメモリの方が対応する必要がありますが、最速で必要なワードをCPUに渡してスタートさせることができます。

プリフェッチ

- アクセスする前にキャッシュに取って来る
  - (問題点) 使うかどうか分からないデータ(命令)のために他のブロックを追い出していいのか？？

  →プリフェッチバッファを使う場合が多い
  - 本当にアクセスされたらキャッシュに入れる
- ハードウェアプリフェッチ
  - 命令キャッシュで用いる。一つ(二つ)先のブロックまで取って来る
    - 命令キャッシュは局所性が高いので効果的
- ソフトウェアプリフェッチ
  - プリフェッチ命令を使う:データキャッシュ
  - コンパイラが挿入
  - 命令実行のオーバーヘッドを伴う

プリフェッチは、キャッシュ上に存在しないブロックを、アクセスされることを予測して、それがアクセスされる前にキャッシュに取って来る方法です。予測が当たればペナルティを場合によってはゼロにすることができます。しかし、これには問題点があり、予測がはずれた場合、不要なブロックによって使うかもしれないブロックが追い出される可能性が出てきてしまいます。そこで、プリフェッチしたブロックは、まず小規模なプリフェッチバッファに入れておき、本当にアクセスされた場合にキャッシュに移すのが標準的な方法になっています。プリフェッチは、ハードウェアプリフェッチとソフトウェアプリフェッチがあり、

コンパイラによる最適化

- ループ構造の最適化
  - ループの入れ子を入れ替える

```
for(j=0; j<100; j=j+1)
  for(i=0; i<5000;
i=i+1)
    x[i][j] = a * x[i][j];
```

```
for(i=0; i<5000; i=i+1)
  for(j=0; j<100; j=j+1)
    x[i][j] = a * x[i][j];
```

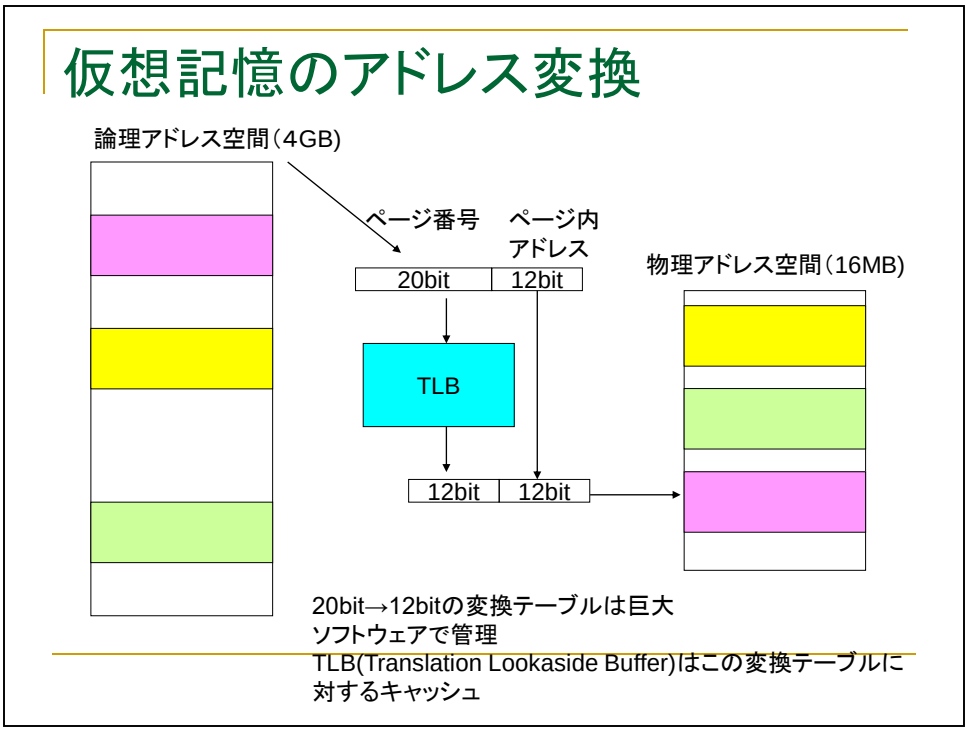
  - ループをくっつける
- ブロック化
  - キャッシュにうまく入るようにデータ構造を変更する
- 科学技術計算には効果的

プリフェッチ以外でも、コンパイラががんばることで、キャッシュのヒット率を上げることができます。例えば左の入れ子構造を見てください。iが0から5000まで変化するので、ループの内部で何回もミスヒットが起きます。これを外側のループと内側のループを入れ替えれば、内側のループで扱う構造がキャッシュの中に入ってしまうので、ミス数が減ります。これをループ交換と呼びます。他にもループをくっつけたり、扱う配列をキャッシュに入るようにブロック化する方法が知られています。これらは行列のように静的なデータ構造を扱う科学技術計算では効果的です。

仮想記憶（Virtual Memory）

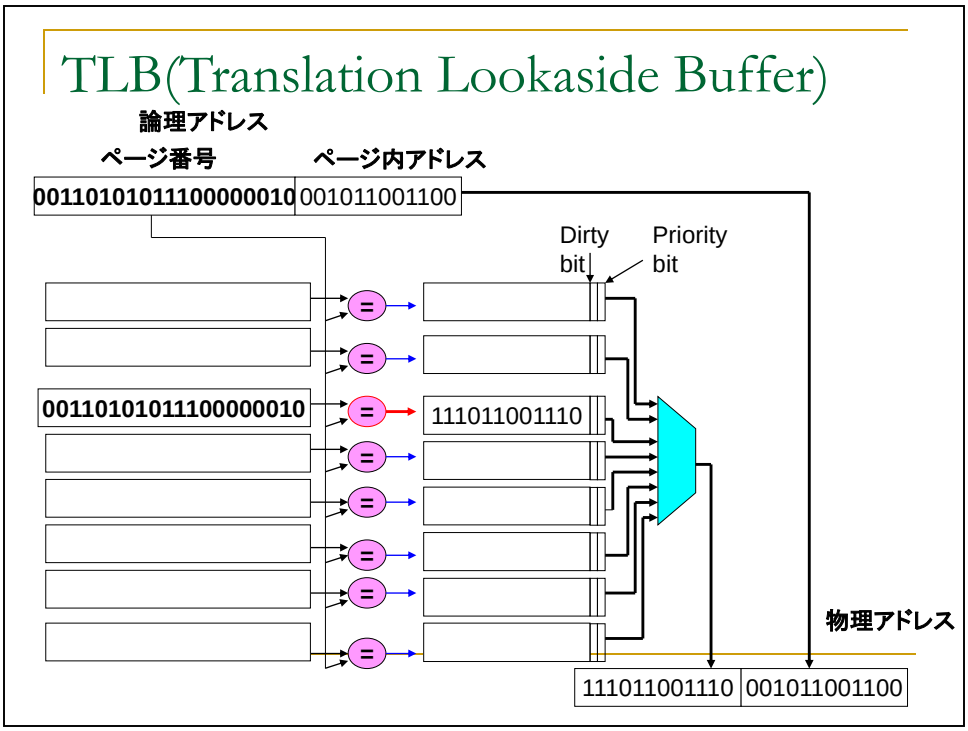
- プロセッサから見たアドレス（論理アドレス）と実際のメモリ上のアドレス（物理アドレス）を分離する
  - 実メモリよりも大きいメモリを扱うことができる
  - 複数のプロセスを互いのアドレスを気にせずに並行実行可能
  - 管理単位で記憶の保護
- ページ：固定サイズ(4K-16KB)　vs. セグメント：可変サイズ→ページを用いる場合が多い
- 概念はキャッシュに似ているがOSが管理、用語も違う
  - ブロック(ライン)：32-128B　⇔　ページ:4KB
  - リプレイス　⇔　スワップイン
  - ライトバック　⇔　スワップアウト
- ページの割り付けはOSが管理
- リプレイスはLRU(Least Recently Used)
- 書き込み制御は当然ライトバック

最後に仮想記憶について紹介します。実はこの仮想記憶はOSの守備範囲です。最近のプロセッサの多くはプロセッサから見たアドレス（論理アドレス、あるいは仮想アドレス）と、実際のメモリ上のアドレスを分離しています。このことで、実メモリよりも大きいメモリを扱うことができ、複数のプロセスを互いのアドレスを気にしないで実行させることもできます。さらに管理単位で記憶領域の保護もできます。この管理単位は固定サイズのページ（これはキャッシュブロックよりも大きく4KBから16KBくらいです）と可変サイズのセグメントがありますが、最近のOSではページを用いる場合が多いです。この仮想記憶は、記憶の階層の最後の主記憶と補助記憶間のやりとりであり、概念としては今まで紹介してきたキャッシュに似ていますが、ハードウェアではなくOSが管理する点が大きく違います。用語も違っており、キャッシュブロックに対応するのがページ、リプレイスはスワップイン、ライトバックはスワップアウトと呼びます。主記憶と補助記憶（ディスク）とのアドレスのマッピングはOSが管理するので、ダイレクトマップのようなキャッシュで使った方法よりもずっと高度な方法が用いられます。しかし、スワップの方法はLRUで同じです。書き込みの制御は、補助記憶とのやり取りを減らすために当然ライトバックです。



論理アドレスと物理アドレスの変換の例を図に示します。この例では仮想アドレス空間は32ビット分すなわち4GBで、物理アドレス空間は16MBを想定しています。実際はもっと大きいですが、原理は同じです。両者ともに4KBのページで区切られていますので、アドレスの対応は論理ページと物理ページの対応表により行います。すなわち、20ビットの論理ページ番号で参照すると12ビットの物理ページ番号が出てくる表を用意すればいいです。しかし、このような表は巨大になってしまうため、実際は主記憶上でOSにより管理されます。

変換の度に巨大な表を引いていては大変なので、小規模高速なメモリを設けて変換テーブルの内容をキャッシュします。このメモリをTLB（Translation Lookaside Buffer)と呼びます。ページは4KB程度の大きさがあるので、局所性の原則からプログラムが利用するページのアドレスがTLBに入ってしまえば、実行中ほとんどミスをすることはなくなります。



TLBは、小規模のメモリを効率良く利用するため、キャッシュではほとんど用いられることはないフルマップ方式を用いることが多いです。ここでは先のスライドのTLBに相当する実装例を示します。論理ページ番号は、一度にTLBの全ての値と比較し、マッチすれば、対応する物理ページ番号が出力されます。物理ページ番号のほかにもそのページに書き込みがあったかどうかを示すDirty bit、そのページを普通のユーザがアクセスして良いのかどうかを示すPriorityビットなど、ページの管理に必要なデータをも持たせておきます。ページ内アドレスは変わらないので、これを物理ページ番号にくっつけて物理アドレスができあがりです。

ページフォルト(Page Fault)の発生

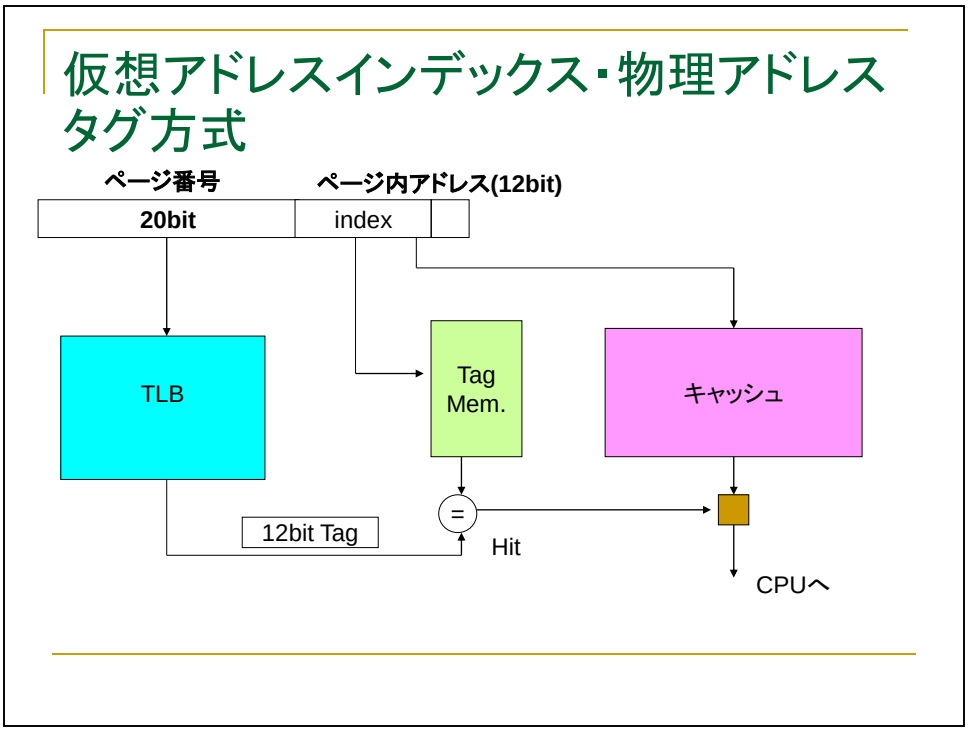
- OSの授業で学ぶ例外処理の一つ
- TLBミス
  - ページ自体は主記憶中に存在→TLBの入れ替え
  - ページ自体が主記憶中にない→スワップイン＋TLBの入れ替え
- ヒットしたがDirty bitが0のページに書き込みを行った
  - Dirty bitのセット
- ヒットしたが特権命令でないのに特権ページを扱った
- いずれのケースもOSで処理する

CPUからのアドレスがTLBにマッチしなかったらどうなるでしょう？この時にはページフォールトと呼ばれる例外処理が発生します。この話は、3年のコンピュータアーキテクチャ、OSの授業で学びます。ページフォールトは、ページ自体は主記憶に存在するけれど、そのコピーがTLBに存在しない場合と、ページ自体が主記憶中に存在しない場合の両方で生じます。前者はTLBを入れ替えだけで済みますが、後者はディスクなどの補助記憶からページをスワップインしてきて、さらにTLBを入れ替えます。この処理はいずれもOSが行います。他にもTLB自体にヒットしたけれど、Dirty bitが0のページに書き込みを行った場合も、ページフォルトが生じます。これは、Dirty bitをセットする必要があるからで、ライトバックキャッシュ同様、Dirty bitがセットされているページはスワップアウトの際、補助記憶にライトバックしなければなりません。さらに特権違反などでもページフォルトが生じ、全てOSにより処理されます。

TLB変換時間の短縮

- 仮想アドレスキャッシュ
  - キャッシュは仮想アドレスで参照する
  - 違ったプロセスでアドレスが重複する問題（シノニム問題）の解決が難しい
- 仮想アドレスインデックス-物理アドレスタグ方式
  (Virtually indexed, Physically Tagged)
  - 変換を行わないページ内アドレスをキャッシュのインデックスに使う
  - タグ参照、キャッシュ参照、TLB変換が同時に可能
  - Direct Mapだとキャッシュサイズが4KBに制限される
    - 2 way だと8K、4 wayだと16K、8 wayだと32K
    - １次キャッシュだけの話なので、多少小さくてもいいか。。。。

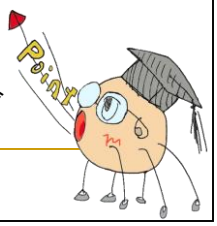
TLBでやっかいな点は、キャッシュは物理アドレスでアクセスされるのが普通で、このため、論理アドレスと物理アドレスの変換は、キャッシュをアクセスする前に行なわれなければならないです。これは論理アドレスでキャッシュをアクセスすると、違った　プロセスでアドレスが重複してしまう問題（シノニム問題）が発生してしまうためです。しかし、TLBで変換してからキャッシュをアクセスすると、時間が掛かってしまい、折角のキャッシュの効果が台無しになりかねません。そこでよく使われるのが仮想アドレスインデックス一物理アドレスタグ方式です。これは、論理アドレスと物理アドレスの変換の対象外のページ内アドレスをキャッシュのインデックスにつかうことで、TLB参照と、タグ参照、キャッシュ参照を同時に行なってTLB変換による時間的ロスを防ぐ方法です。ページサイズが4KBの場合は、インデックスも12ビットまでの範囲で収めなければならないため、ダイレクトマップだとキャッシュサイズは4KBに制限されます。2wayならば8K,4wayならば16Kまでになります。しかし、TLBの変換時間が問題になるのはL1キャッシュまでの話なので、サイズは小さくても問題は少ない場合が多いです。



仮想アドレスインテックス、物理アドレスタグ方式の図です。ページ番号の部分でTLBを引き、残りの部分はインデックスとしてタグメモリ、キャッシュをアクセスします。同時にTLBにより得られた物理アドレスのタグと、タグメモリから出力されるタグを比較します。この方法は三つの記憶要素を同時にアクセスすることで、TLBの変換時間を隠蔽することができます。

本日のまとめ

- 書き込みポリシー
  - ライトスルーは、書いたデータをそのまま主記憶に書き込む
  - ライトバックは、キャッシュにだけ書き込む
    - 書き込んだブロックにはDirty bitをセット
    - Dirty bitがセットされているブロックは追い出しの際に書き戻しを行なう
- リプレイスポリシー
  - LRU(Least Recently Used)
- キャッシュの性能評価
  - CPIの増加として考える。ミス率とミスペナルティ
- キャッシュのミスの原因は3つのC
  - キャッシュ容量、ブロックサイズ、way数に依存
- キャッシュのミスペナルティ
  - 階層キャッシュ、ノンブロッキングキャッシュなど
- 仮想記憶
  - OSの領域だがTLBの周辺だけ理解して



インフォ丸が教えてくれる今日のまとめです。

演習1

- 0x00番地からサイズ8の配列A[i]の総和を求めた答SをA[0]に書いた後、0x100番地からのサイズ8の配列B[i]の各要素に加算するプログラムを作れ。

int i、S=0;

for(i=0;i<8;i++) S=A[i]+S; A[0]=S;

for(i=0;i<8;i++) B[i]=B[i]+S:

S=44(16進数）

B[i] = 45,46,47…に加えると、89,8a,8b…になるはず

上記のアセンブラプログラムenshu.asmを書け

これをライトスルーキャッシュとライトバックキャッシュで実行した際、終了するまでのクロック数を求めよ

提出物は、enshu.asmと終了までのクロック数

演習2

- アドレスA,Bは、ダイレクトマップ方式において互いにブロックが競合するアドレスである。これに対して以下のアクセスを順に行う。
  1. Aから読み出し
  2. Bから読み出し
  3. Aに書き込み
  4. Aから読み出し
  5. Bに書き込み
  6. Aから読み出し
  7. Aに書き込み
- ダイレクトライト型のライトスルーキャッシュ、ライトバックキャッシュについて、それぞれのアクセスがミスするかヒットするかを示せ。また、各アクセスによってメモリに対してどのような操作（リプレイスR、ライトバックWB、ライトスルーの書き込みWTH）が必要か？ライトバックについてはブロックはC、Dのうちどちらの状態になるか？