前回:
パイプライン - 命令実行をステージごとに重ねることで,
時間あたりの命令の実行数を増やして高速化をはかる

今回:
記憶階層 - 高速小容量と低速大容量の記憶素子を組み合わせることで,
高速大容量の記憶を実現しようとする

キーワード:
時間的局所性, 空間的局所性, ヒットとミス, キャシュ, ブロック,
マッピング, ダイレクトマッピング, フルアソシアティブ, セットアソシアティブ,
連想度, ライトスルー, ライトバック, ダーティビット

第12回 記憶階層 - キャシュメモリ

7. 記憶階層

目的: フォンノイマンボトルネックの解消
  CPUとメモリの間の通信が多すぎて性能を制約 (授業 第4回)

7.1 記憶階層構造

7.1.1 メモリのアクセス時間
メモリに対してデータの読み出し/書き込みにかかる時間
定義にはいろいろある:
(1)アドレスを指定してから読み出し/書き込みが開始するまでの時間
(2)読み出し/書き込みが完了するまでの時間 (アクセスレイテンシ)
(3)次に読み出し/書き込みができるまでの時間 (サイクルタイム)

CPUのクロック
2.5 GHz だと 1サイクルは 0.4ns
メモリのアクセス時間 - たとえば60ns
  150サイクルかかる計算

7.1.2 メモリの速度と価格(したがって容量)はトレードオフの関係にある
高速なメモリ - 小容量
低速なメモリ - 大容量

CPUの近くに高速小容量なメモリを置き, 遠くに低速大容量のメモリを置く
  CPUのレジスタ
  キャシュメモリ (new!)
  主記憶
  二次記憶(ハードディスク) (new!)
この構造を記憶階層構造という

7.2 プログラムが持つメモリアクセスの性質

メモリへのアクセスは均一ではない
  「かたより」がある

7.2.1 時間的局所性
最近アクセスした命令/データが再度アクセスされやすい

7.2.2 空間的局所性
アクセスした命令/データの(アドレス空間上で)近くの命令/データが
アクセスされやすい

7.2.3 局所性が成り立つ理由

7.2.3.1 命令
命令はアドレス順に実行される - 空間的局所性
  読み出す命令の位置は(分岐がなければ)順次すすむ
プログラムでの繰り返し(ループ)の存在 - 時間的局所性
  ループ内の命令は何度も繰り返して読み出される

7.2.3.2 データ
同一データの読み書きが多い - よく使われる変数とか - 時間的
配列要素など隣接データを順次アクセスすることが多い - 空間的

7.3 局所性の利用
アイデア:
アクセスされる可能性の高いデータを高速なメモリに格納する
→プログラムの平均的なメモリアクセス時間が減少して高速化
(ボトルネックが太くなったことに相当)

7.3.1 命令/データのアクセスのしかた

上位(高速)メモリにあった(hitという) - OK

なかった(missという)
  下位(低速)メモリをアクセス

ヒット率とミス率 (足すと1)

ヒット時間とミスペナルティ

上位メモリにあるかどうか判断する時間も必要

8. キャッシュ / キャシュ
(cacheのこと, 現金(cash)とは違う)

CPU(高速)と主記憶(低速)の差を解消するために
  間に高速のメモリをはさむ (キャッシュメモリ)

8.1 ブロック (あるいはラインともいう)
キャッシュと主記憶とのやりとりの単位

8.2 マッピング
アドレスの対応付け

CPUが要求した主記憶のアドレスが
- キャッシュにあるのかどうか
- あるとするとどこにあるか
を判断する必要がある

いろいろな方式がある.
  ダイレクトマップ方式
  フルアソシアティブ方式
  セットアソシアティブ方式

8.3 ダイレクトマップ方式 (direct mapping)

図7.5

主記憶のアドレスから直接キャシュのブロック番号を計算

ブロック番号 = 主記憶アドレス mod ブロック数
(図ではインデクスと書いてある)

ブロック数が2のべき乗なら, ブロック番号 = アドレスの下位xビット
  除算を行わなくてよい

8.3.1 異なるアドレスが同一のブロックに対応する可能性
下位xビットが同じならそうなる

そのチェックのために「タグ」
  「本当は何番地」
主記憶アドレスのブロック番号より上の部分

8.3.2 そもそもそのブロックに有効なデータが収容されているか?
有効ビット

模式図(図7.5)
シナリオ(表)
キャッシュの内容の変化(図7.6)

しくみ(図7.7)
ブロックサイズは1語(4バイト)
キャシュのブロック数 1024 (10bit)
タグ: そのブロックに収容されているデータの本来のアドレス(の上位ビット) - (32-10-2 = 20bit)

8.3.3 ブロックサイズを大きくする
ここまでブロックサイズは1語
  これでは空間的局所性は活かせない

ブロックサイズを大きくする
  ミスした後、1ブロック分の語をキャシュに入れる
    遅くならないか？
  アクセスした語の近隣の語もキャシュに乗る
  空間的局所性を利用 - hit しやすくなる
図7.10

8.3.4 ダイレクトマップ方式の特徴

短所
同じブロック番号に対応するアドレス同士が競合
  ミスにつながる

長所
ヒットしたかの判定が高速
  タグの比較一回で済むから

8.4 フルアソシアティブ方式 (fully associative)

メモリブロックをキャシュのどこに置いても構わない

短所
キャシュのどこにあるか検索の必要
  比較回路を多数持たないといけない
    キャシュのブロック数と同数
  大容量のキャシュには不向き

長所
アドレスの競合がない(容量がゆるせば)
  ミスが起こりにくい

8.5 セットアソシアティブ (set associative)
前述のふたつの方式の中間

メモリブロックを置いてよい場所が複数 (nとして)
  n: 連想度 associativity
  n-way set associative

  そのブロックの組をセットという

キャシュのどこにあるか
メモリアドレスの下位xビットでセットが定まる
セット中の n箇所のどこか, はn個の比較回路で判断

図7.19
4ウェイ - 1セットに4ブロック
セット数は256
1ブロックは1語
図7.7と同じ容量(256セット×4ブロック=1024語)

タグ比較回路が4つ

8.6 キャシュのあれこれ

8.6.1 命令用のキャシュとデータ用のキャシュ
キャシュを命令用とデータ用に分けるのが一般的
命令実行のパイプラインにおいて,
  命令フェッチ
  データのメモリアクセス

の競合(構造ハザード)が回避できる

8.6.2 ミスの際の処理
(読み出しの場合)
主記憶から読み出してきてキャシュにも収容
そのあいだCPUは止めておく(ストールする)のが普通

8.6.3 書き込みの処理
キャシュにヒットして書き込んだとき
主記憶と内容が食い違うことになる
  一貫性(consistency)がなくなる

主記憶にいつ書き戻すか

8.6.3.1 ライトスルー write through
キャシュに書くときは同時に主記憶にも書く
(キャシュミスのときは主記憶だけに書く)

一貫性の問題は単純
メモリアクセスによる性能低下が問題

8.6.3.2 ライトバック write back
書き込みはキャシュだけ
キャシュから追い出されるときに主記憶に書き込み
  つまり, 別のアドレスブロックと競合して置き換わるとき
キャシュに「書き込みがあった」ことを示すビットが必要
  変更してないなら書き戻す必要ないから
  ダーティビット, 変更ビット
(ミスの場合)
前にあったブロックを追い出して, キャシュにだけ書く
  変更ビットもセットする

メモリが低速でも性能が低下しない
書き込みがブロック単位なので効率よい
制御回路は複雑

8.6.4 置き換えアルゴリズム
ミス発生でキャシュに新しく収容するとき
連想度が2以上の場合(つまりダイレクトマップ以外)
候補となるブロックは複数ある
  どれを置き換えるか

(a)ランダム

(b)FIFO (first-in first-out)
もっとも古くからあるブロックを選ぶ
輪番に選択すればよい (0->1->2->...->0->1->...)
ラウンドロビン(round robin)ともいう

(c)LFU (least frequently used)
もっともアクセス回数の少ないブロックを選ぶ
回数をかぞえる回路が必要

(d)LRU (least recently used)
最終アクセスがもっとも古いブロックを選ぶ
アクセス時刻を記録する回路が必要

どの方式を用いるか?
連想度があがると回路が複雑化
-> ハードウェアが単純な方式を一般に採用
キャシュ容量の増大でどの方式でもミス率に大差がなくなる

8.6.5 多層のキャシュ
一例: Intel Core i7-960
L1: (32KB(命令)+32KB(データ)) * 4 (コア数)
L2: 256KB * 4 (コア数)
L3: 8MB