単一レベル記憶のための高精度キャッシュ制御

前田 賢一[1,a)]

概要：単一レベル記憶は，大がかりなキャッシュシステムと見なすことができる．ここでキャッシュ制御は重要な課題であり，効率的な制御法について述べる．未来のキャッシュの動作予測は困難な問題であり，一般的な条件下で完璧に未来を予測できる方法は存在しない．ここでは，過去の履歴からキャッシュアクセス頻度を予測し，最適な制御を，簡単な処理によって行う方法を提案する．

High-Precision Cache Control for Single-Level Store

Ken-ichi Maeda[1,a)]

1. はじめに

ネットワーク透過な単一レベル記憶（Single-Level Store: SLS） [1] と，ネットワーク透過なスレッドをベースとした，ネットワーク上の仮想計算機の構成法が提案されている [2]．単一レベル記憶では，全てのデータが仮想的な主記憶に存在することになる．それは，いわば大がかりなキャッシュシステムであり，物理的なストレージやネットワークの向こう側に存在するデータをローカルなメモリにキャッシュしたものとみることができる．単一レベル記憶においてキャッシュという見方は，いろいろなレベルで存在する．

- 主記憶（通常 DRAM）の CPU 内部（一次）キャッシュ，二次キャッシュ
- ローカルなストレージの主記憶へのキャッシュ
- ネットワークの向こう側のデータのローカルなストレージへのキャッシュ

ただし，CPU 内部キャッシュ，二次キャッシュは，既に標準的なハードウェア制御の元にあるため，以下の議論から除いておく．

一般的に大容量の記憶装置は速度が遅く，高速な記憶装置は容量が限られているという問題がある．どのデータを高速な記憶装置に保持しておくべきかというのが，キャッシュ制御の問題の一つとして知られている．本稿の目的は，比較的アクセスが遅いストレージ以下のデータのキャッシュ制御方法を提案することである．

特殊なケースは別として，「未来は予測できない」というのが一般的な制約事項である．その制約の下で，過去の状況からなるべく合理的な予測をすることが求められる．

CPU のキャッシュは CPU に組み込まれたハードウェアによって制御されている．これは，一般的に第三者が変更することはできない．しかし，それ以外のレベルのキャッシュ機構は，OS を中心にソフトウェアで実現されている．もちろん，必要によっては CPU の外部に簡単なハードウェアを追加することも可能である．

いずれにしても，処理速度を考えると，なるべく簡単な演算でその予測を実現することが重要である．

本報告では，データのアクセス頻度がべき乗分布で表現されるという仮定のもと，まず，その分布の推定方法について述べる．さらに，それを応用した簡単な演算で実現可能なキャッシュ制御に関して述べる．

2. 単一レベル記憶とキャッシュ問題の位置付け

まず，前提条件を明確にするため，簡単に過去に提案されたネットワーク透過な単一レベル記憶 [2] のサマリーを述べる．

通常の計算機では，主記憶（Main memory）と二次記憶

[1] フリーランス・コンサルタント
Freelance Consultant, Japan
[a)] ken.maeda@m.ieice.org

表 1 明示的階層メモリとキャッシュとの比較

種類	明示的階層	キャッシュ
メリット	高速	使いやすい
CPU	スクラッチパッド	CPU キャッシュ
記憶	主記憶＋二次記憶	単一レベル記憶

(Storage) とがあり，前者は CPU の `load/store` 命令を使ってアクセスし，後者は `read/write` のシステムコールを使ってアクセスする．

これに対して，単一レベル記憶というのは，全て CPU の `load/store` 命令を使ってアクセスするものである．アクセスした際に，対象が実際に主記憶にない場合には，トラップを使って二次記憶から主記憶に転送される．

単一レベル記憶が最初に導入されたのは Atlas [3] であり，大々的に実用になったのは Multics [4] である．

ネットワーク透過な単一レベル記憶 [2] は，これをネットワーク上に拡張したものである．それは，ネットワークの向こう側の記憶も，ローカルの主記憶と等価に見せようとするものである．すなわち，ネットワーク上の全ての記憶対象に対して，`load/store` 命令でアクセスすることになる．結果として，先に述べたように，全体を大規模なキャッシュシステムと見做すことができる．

ネットワークの向こう側の記憶を直接主記憶にキャッシュするのか，間にローカルの二次記憶のキャッシュが入るのかは，両方の可能性があるが，ここでは議論しない．速い記憶と遅い記憶があり，遅い記憶を速い記憶にキャッシュするという，抽象的なモデルで以下の議論を進める．

ここで，プログラマが明示的に区別する二階層（一般的には複数階層）のメモリとキャッシュとの違いを，より明確にしておく．CPU キャッシュと類似した目的に対して，スクラッチパッドメモリという解決法が存在し，これはプログラマから見ることができる．プログラマは，どのデータを速いスクラッチパッドメモリに持つべきかを意識しながらコーディングする必要がある．プログラムの挙動がわかっている場合には，その負担をしても高速な実行速度を得ることができるのである．

CPU キャッシュは，通常のプログラマからは見えない．したがって，プログラマはどのデータがキャッシュされているかを意識せずにコーディングすることが可能である．その簡便さと引き替えに，後述するような問題点が知られている．ここで，明示的な階層メモリとキャッシュとの比較を，表 1 にまとめる．

キャッシュの問題は計算機分野の重要問題のひとつとして，長年にわたって議論されてきた．一般に高速のキャッシュは容量が限定されているため，全ての必要なデータ（あるいはプログラム）を収容することはできない．どのデータをキャッシュし，どのデータをキャッシュから追い出すかというのがキャッシュ制御の問題である．

制御の方法としても，FIFO（First In, First Out） や LRU（Least Recently Used）やランダム方式など，いろいろな方法が比較検討されてきた [5]．しかし，一般的に最適な方法というのはなく，どの方法にも問題点があった [*1]．

FIFO というのは，キャッシュされた時刻が古いものから追い出すというアルゴリズムである．しかし，キャッシュされた時刻が古いといっても，将来使われる可能性が低いとは限らない．

LRU は，最後に使われた時刻が最も古いものから追い出すというアルゴリズムである．使われた時刻を使うとうことで，キャッシュされた時刻を使う FIFO よりは，もっともらしいが，これも将来使われる可能性という意味では，まだ不十分である．

キャッシュに入りきらない大きな画像を処理するケースが，問題を把握するために典型的な例となる．簡単のために，キャッシュのサイズが画像のサイズのちょうど半分であったとしてみる．処理は，画像のビットを反転することだとする．FIFO でも LRU でも，最初はキャッシュに入ってないため，キャッシュはミスして遅い記憶からデータを読んでくる．画像の半分までは遅いメモリから読んできてキャッシュし，それをビット反転して同じキャッシュに書き込む．半分まで来るとキャッシュがいっぱいになるため，先ほど書き込んだデータを古い方から遅いメモリに追い出して，後半のデータをキャッシュする．ビット反転したデータを使って，次の処理をする場合には，後半のデータも古い方から遅いメモリに追い出す必要がある．

このアクセスパターンを考えると，遅いメモリのアクセス回数はキャッシュがない場合と同じになるため，キャッシュを持つことが無駄であることがわかる．実際には，キャッシュした後でキャッシュへのアクセスが追加されるので，無駄というよりかえって遅くしてしまっている．

あらかじめわかっているのであれば，画像の前半だけをキャッシュして後半は遅いメモリを直接アクセスするようにすれば良い．そうすれば，２個目の処理以降はキャッシュが効いて高速化できる．強制的に前半だけをキャッシュするために，容量がいっぱいになるまでキャッシュするという方法はある．しかし，それでは使わなくなった後まで最初のデータが残ってしまう．そうならないような，キャッシュ制御が必要である．

これを何とか自動化したいというのがモチーフである．CPU キャッシュは速度が速いため，あまり凝った制御方法を採用することは難しい．しかし，単一レベルメモリのようにストレージが遅いメモリである場合には，ある程度凝った制御アルゴリズムも採用可能である．本稿では，それを前提に議論する．

[*1] プログラマがあらかじめ挙動を知っている場合には，ソフト的にプリフェッチできれば有効である．しかし，複雑なプログラムで挙動を完璧に知っていることはほどんど期待できない．

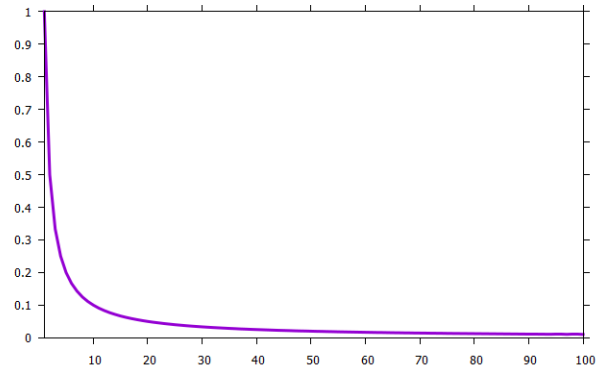

図 1 ジップの法則（べき乗分布）

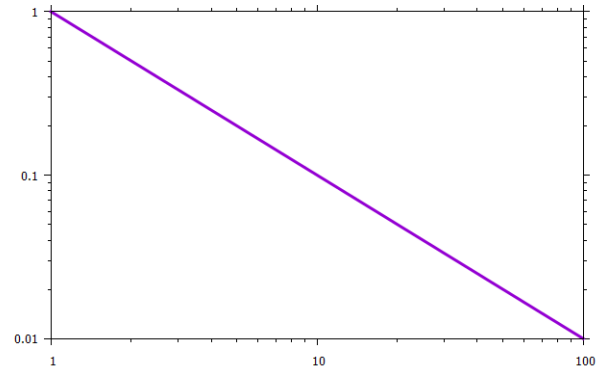

図 2 ジップの法則（べき乗分布）log スケール

3. アクセス頻度のべき乗分布

世の中のいろいろな分布は，ガウス分布（正規分布）をしていると思われがちである．しかし，実際には正規分布であることはむしろまれであり，べき乗分布（パレート分布）をしていることが多い [6]．この分布は，統計力学ではボルツマン分布として知られており，外部とのエネルギの出入りがない状態で，エントロピーが最大になることが特徴である．一方で，データ（NAND のページ単位，など）へのアクセス頻度の分布がべき乗分布になることが知られている [7]）．

実世界ではべき乗分布が多いと述べたが，他の実例をあげてみる．たとえば，使われる単語の頻度，富の分布 [8]，都市の人口分布，地震の規模分布，など，がべき乗分布になる．

この法則は，「ジップの法則」という名前でも知られている [9]．k 番目に大きい要素が全体に占める割合が $1/k$ に比例するという法則である．横軸に k 番目，縦軸に頻度をプロットすると，図 1 のように，$1/k$ で下がる双曲線（の一部）になる．それぞれ log 値をプロットすると，図 2 のように，右下がりの直線になる．

次に，どんな状況がべき乗分布を発生させるのかを簡単に説明する．ひと言で言うと，「履歴を引きずるランダム」がべき乗分布になる．

たとえば，ランダムなホワイトノイズを考える．（フーリエ変換すると，スペクトルはフラット．）一つ前の値に次々にそのノイズを加えていくと，そのスペクトルはべき乗分布をするようになる．（フーリエ変換すると，スペクトルは $1/f$ の形．）「ホワイトノイズ」に対して，上記のようにして作ったノイズを「ピンクノイズ」と呼ぶ．

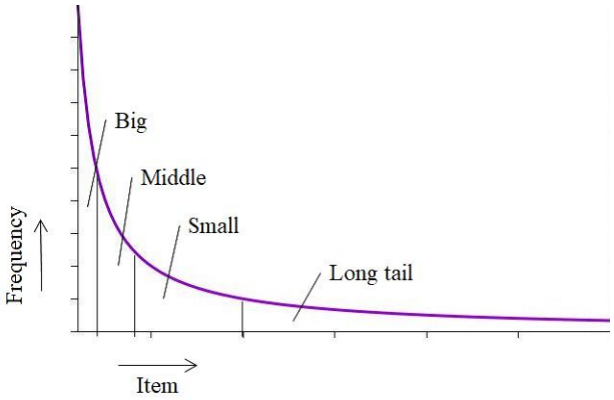

図 3 ページのアクセス頻度（べき乗分布）

気体分子の運動も，個人の富の蓄積 [*2] も，「履歴を引きずるランダム」であるため同様にべき乗分布となる．

この考え方をデータのアクセスについて適用してみる．今，個々のデータのアクセスがランダムに発生すると仮定する．そうすると，そのアクセスを加算したアクセス分布は，上で説明した理由によりべき乗分布をするようになる．

4. キャッシュ制御法への応用

4.1 キャッシュすべきデータ

キャッシュシステムとしては複数階層でも良いのだが，ここでは簡単のために 2 階層しかないとする [*3]．キャッシュは，高速であるが，容量が限定されている．たとえば，遅いメモリの容量は 128 GB であるが，キャッシュの容量は 128 MB というような状況である．遅いメモリは NAND メモリで，ページサイズを 4 KB としよう．キャッシュは DRAM でも MRAM でもよい [*4]．

そうすると，キャッシュには 32 K 個のページを入れることができる．ページ全体は，32 M 個あるので，キャッシュできるのは 0.1 % ということになる．この状況で，どのページをキャッシュすべきか，新たにキャッシュする場合，どのページを追い出すべきか，というのが解きたい問題である．ここでは，簡単のためにページ単位のキャッシュを考えるが，複数ページ単位のキャッシュや，セグメント [10] 単位のキャッシュも，ほぼ同様の扱いが可能である．

[*2] 一見，社会法則であると思われる富の分布が，物理法則としてべき乗分布になるのは興味深い．
[*3] 実際のケースで，複数階層ある場合には，本稿での提案にしたがって多重に同じメカニズムを採用することとする．
[*4] 先に述べたように，そういう構成に限定される必要はなく，「遅いメモリはインターネットの向こう側でキャッシュはローカルのストレージ」という構成でもよい．

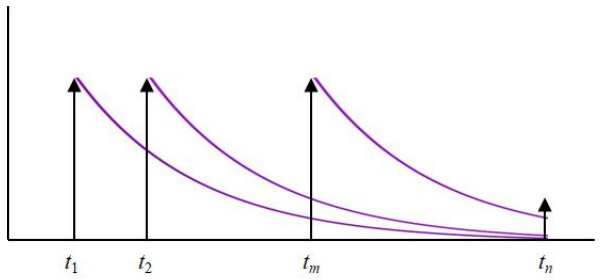

図 4 忘却を考慮した t_n 時のカウンタの値

アクセス頻度はべき乗分布になるという前提なので，分布を見ると，図 3 のようになるはずである．

図には無限に続くように書かれているが，今考えているメモリシステム構成では，横軸の長さは 32 M ページ（128 GB） までである．その 0.1 % の 32 K ページ（128 MB）が，たとえば，Big と書かれているところであれば，そこまでをキャッシュすれば最適なキャッシュページの選択となる．キャッシュのヒットは Big の面積で，全体の面積に占める割合はその比率となる（キャッシュのヒット率）．もう少しキャッシュ容量が大きいのであれば，Middle までキャッシュすることができるだろう．なお，この例に限らず現実の世界では，Longtail は無限に続くわけではない．

4.2 分布の推定法

次に問題となるのは，特定のページがグラフ上のどこに来るかを正しく推定することである．ここでは，その方法を議論する．

基本的に「未来を予測することはできない」と言われている．したがって，過去のアクセス状況からグラフを作らざるを得ない．しかし，処理される内容によってアクセスパターンは変化すると予想される．グラフは固定的なものではなく，時々刻々変化する．特に，特定のページがどの程度アクセスされるかが変化する．

これを記録する手段として，ページごとにカウンタを用意することにする．ページがアクセスされるたびに，カウンタをインクリメントする．しかしそのままでは，どのページもカウンタの値が大きくなる一方で，いつかはあふれてしまう．また，過去に多くアクセスされたものが，いつまでもアクセス数が多いとしてランキングされてしまう．

この問題を解決するために，「忘却」を導入する．最も単純な方法としては，カウンタをリセットすれば良いのであるが，それでは突然過去の履歴を無視することになる．そこで，ここでは，よく知られた忘却曲線を使うことにする．それは，$\exp(-t/T)$ の形で，時間 t の増加と共に急激に減少する曲線である（ここで T は，「時定数」）．

ここで，a_m を時刻 t_m に対応する係数とする．時刻 t_m にアクセスがあった際には，$a_m = 1$ [*5] であり，アクセスがない場合には，$a_m = 0$ とする．現時点を t_n とすると，忘却を考慮した現在のカウンタ値 $f(t_n)$ は，

$$f(t_n) = \sum_{m \leqq n} a_m \exp(-\frac{t_n - t_m}{T}) \qquad (1)$$

となる．ここで，重要なのは相対的な大きさなので，正規化は無視するものとする．この式が表す忘却のイメージを図 4 に示す．

ある程度高速に計算したいので，厳密に $\exp(-t/T)$ を計算するのはたいへんである．そこで，指数関数の底の変換を使って，$2^{-t/T'}$ で表す．ただし，$T' = T \ln 2$ である．具体的な値としては，$\ln 2 = 0.693...$ であるから，約 0.7 倍と考えて良い．

さらに，t が T' 以下の中途半端なところも無視することにし，T' ごとに 1/2 にするものとする．具体的には，T' ごとにカウンタを右に 1 bit シフトすれば良い．これはハード的に極めて簡単である．あるいは，T' での除算も，T' を 2^k にしておけば，k bit のシフトで代用できる．

カウンタはページごとに必要なので，基本は図 5 のような構成になる．カウンタの個数が多いため，カウンタの実体は遅いメモリに持たざるを得ない．ページをアクセスするたびに，遅いメモリ内のカウンタにアクセスするのは，時間がかかってしまう．そこで，カウンタもキャッシュすることにする．

カウンタもキャッシュした構造を，図 6 に示す．これでカウンタへのアクセスは，かなり高速化される．しかし，ページ数が $32K$ 個（$128MB$ 分）あるので，これを全部操作するのはたいへんである．

そこで，次の工夫は，毎回操作しなくても済むようにすることである．そのために，カウンタに最後にアクセスした時刻も記録しておくことにする．したがって，ページごとに用意すべき情報は，従来のカウンタと最後にアクセスした時刻ということになる．アクセス時刻も追加した構成を，図 7 に示す．

アクセスがあった場合には，次のような操作を行う．

(1) 記録された時刻 t_l を読み出す．
(2) 現在時刻を t とすると，$(t - t_l)/T'$（実際には，k bit の右シフト）を計算し，その値だけカウンタを右にシフトする．
(3) カウンタをインクリメントする．

この操作は，遅いメモリからキャッシュに持ってくる際にも行う．アクセスがないページに対しては，最終アクセス時刻が記録されているため，原理的にはカウンタを更新する必要はない．

[*5] 任意の正の値で良いが，簡単のために 1 とする．

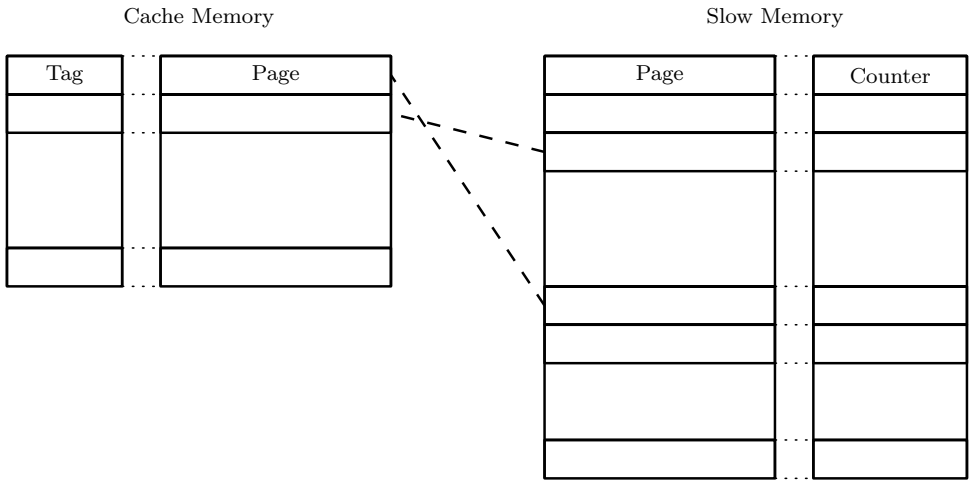

図 5 アクセス頻度のカウント（基本）

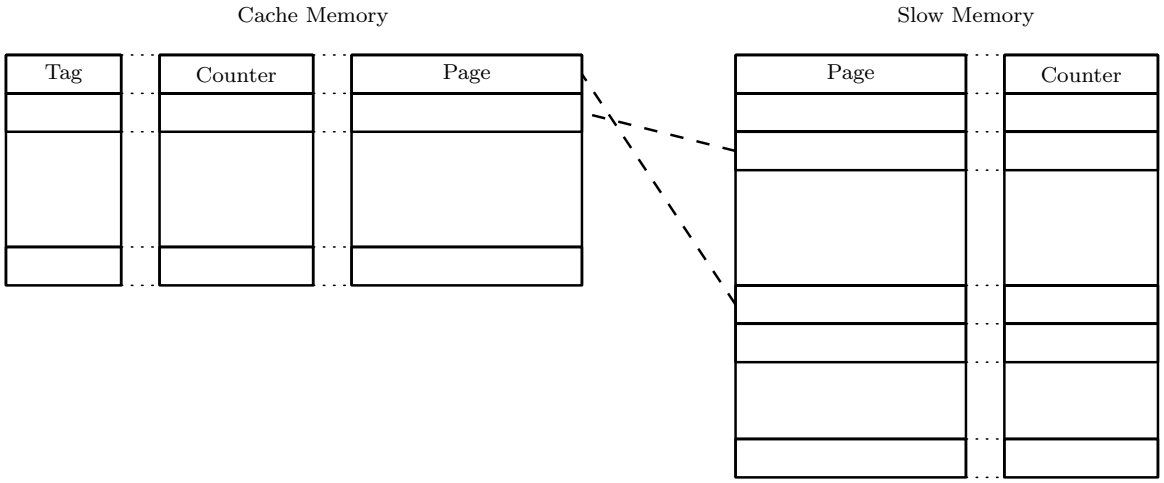

図 6 アクセス頻度のカウント（カウンタもキャッシュ）

遅いメモリがネットワークの向こう側にある場合には，実際にデータを持ってくるためには，かなりの時間がかかる．そのため，ここに述べた操作は，OS のようなシステムソフトウェアで実現しても大きなオーバーヘッドにはな

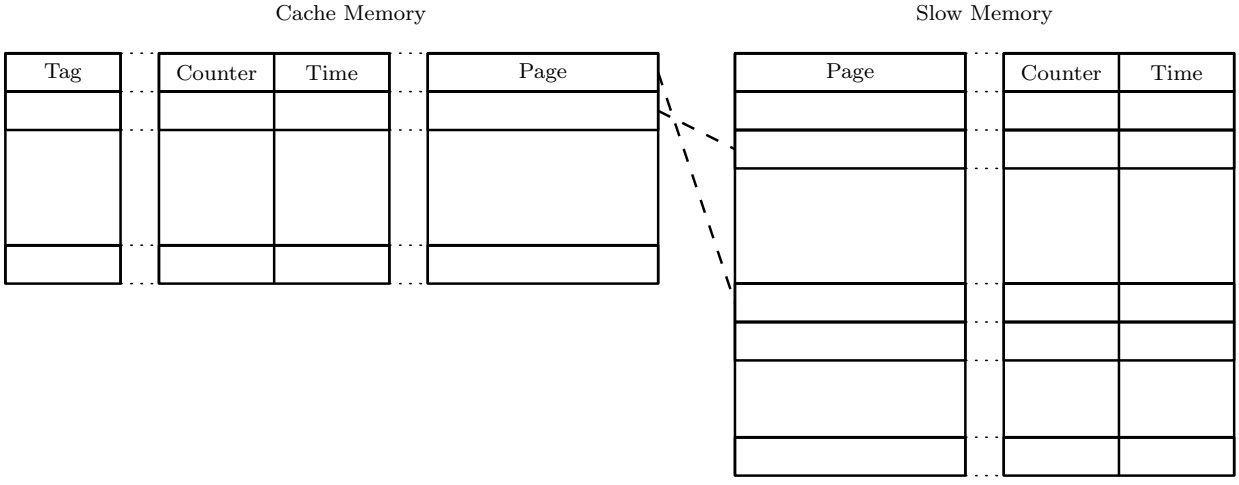

図 7 アクセス頻度のカウント（カウンタに時刻をプラス）

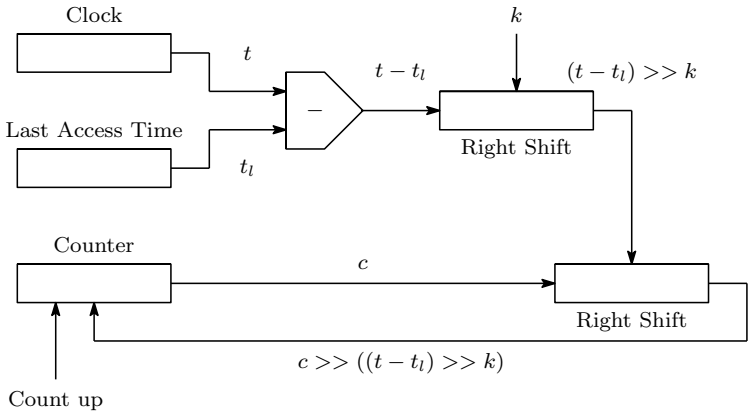

図 8 カウンタの更新ハード

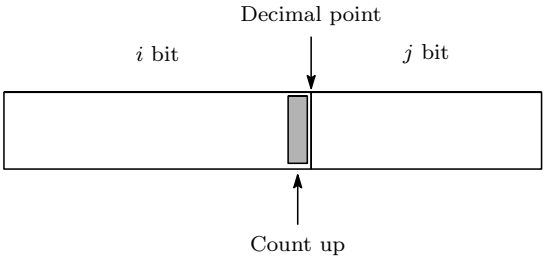

図 9 カウンタの構造（固定小数点）

らない．しかし，遅いメモリが，DRAM の 1000 倍程度しか遅くない場合には，ハードウェアで制御する方が良いこともある．想定されるカウンタの更新ハードのブロック図を，図 8 に示す．ただし，カウンタやアクセス時刻の領域を出し入れする部分は省略してある．

4.3 カウンタの構成

前節では，簡単のために単に「カウンタ」という名称で呼んでおり，その構成は議論してこなかった．しかし，単純なカウンタでは以下の問題があることがわかる．

時定数 T' と比較して長い時間間隔で，少ない回数アクセスされるデータがあったとする．カウンタは，T' ごとに右にシフトされるので，アクセスごとにカウントアップしたデータはのカウンタは，LSB だけにビットがあるかないかという状態になる恐れがある．右シフトすると 0 になってしまうため，そういうデータ同士の頻度比較は，必然的に高精度では行えない．

そこで，固定小数点表示を導入することにする．図 9 に提案するカウンタの構造を示す．カウンタは i bit の整数部と j bit の小数部とからなる．カウンタをカウントアップするときには，整数部の LSB に 1 を加える．右シフトは，整数部と小数部とをまとめてシフトする．大小比較も整数部と小数部とをまとめて行う．そうすることにより，整数部から右側にあふれたビットも小数部に残るため，比較することが可能となる．

なお，ソフトウェアで実現する場合には，カウントアップする際に 1 を加えるのではなく，たとえば，8 や 16（一般的には 2^n）を加えるというようにすれば簡単である．

4.4 キャッシュの制御法

最後に必要なのは，キャッシュの制御法である．ここまでで過去のアクセス頻度の推定ができているので，後は簡単である．

今，遅いメモリ中のキャッシュされていないデータへのアクセスがあったとする．キャッシュコントローラは，対応する遅いメモリ中のカウンタに対して，下記の操作を行う．ただし，t は現在時刻である．

(1) 記録された時刻 t_l を読み出す．
(2) $(t - t_l)/T'$ を計算し，その値だけカウンタを右にシフトする．T' が 2^k で表されるような場合には，$(t - t_l) >> k$ だけカウンタを右シフトするだけで良い．（理論的には，$\exp(-(t - t_l)/T)$ 倍を計算する．）
(3) カウンタをインクリメントする．

ここで，キャッシュは全部使われているとし，さらに頻度順にソートされているとする．（実際のソートは比較的たいへんなので，簡易法を考える必要があるかも知れない．たとえば，順番を記録しておいて実際には順番を入れ替えない，など．）キャッシュに対して，下記の操作を行う．

(1) 一番カウンタ値の小さいキャッシュの記録された時刻 t_m を読み出す．
(2) $(t - t_m)/T'$ を計算し，その値だけカウンタを右にシフトする．T' が 2^k で表されるような場合には，$(t - t_m) >> k$ だけカウンタを右シフトするだけで良い．（理論的には，$\exp(-(t - t_m)/T)$ 倍を計算する．）
(3) その値を前記遅いメモリのカウンタ更新結果と比較する．
(4) その値の方が大きい場合には，アクセスしたデータをキャッシュしない．
(5) その値の方が大きくない場合には，最低頻度のキャッシュデータと入れ替える．
(6) さらに，一つ上の頻度のキャッシュとも（時刻を含め

て演算した結果を）比較し，値が大きくない場合には入れ替え，値が大きくなくなるまで繰り返す．

なお，元々キャッシュされているデータに対しては，上記の遅いメモリの操作は不要で，キャッシュの操作のみを実行する．具体的には下記の操作である．

(1) アクセスされたデータのキャッシュの記録された時刻 t_l を読み出す．
(2) $(t-t_l)/T'$ を計算し，その値だけカウンタを右にシフトする．T' が 2^k で表されるような場合には，$(t-t_l) >> k$ だけカウンタを右シフトするだけで良い．（理論的には，$\exp(-(t-t_l)/T)$ 倍を計算する．）
(3) カウンタをインクリメントする．
(4) 一番カウンタ値の小さいキャッシュの記録された時刻 t_m を読み出す．
(5) $(t-t_m)/T'$ を計算し，その値だけカウンタを右にシフトする．T' が 2^k で表されるような場合には，$(t-t_m) >> k$ だけカウンタを右シフトするだけで良い．（理論的には，$\exp(-(t-t_m)/T)$ 倍を計算する．）
(6) その値を前記遅いメモリのカウンタ更新結果と比較する．
(7) その値の方が大きい場合には，アクセスしたデータをキャッシュしない．
(8) その値の方が大きくない場合には，最低頻度のキャッシュデータと入れ替える．
(9) さらに，一つ上の頻度のキャッシュとも（時刻を含めて演算した結果を）比較し，値が大きくない場合には入れ替え，値が大きくなくなるまで繰り返す．

以上のように操作することによって，常に上位がキャッシュされることとなる．また，キャッシュの「上の方」に頻度が高いものが来る．

4.5 パラメータの意味

実際に応用するためには，具体的なパラメータを決定する必要がある．具体的には，時定数 T'（理論的には T）の値を決めなくてはならない．結論から言うと，これは実験で決めるのが良いと思われる．ここでは，詳細に検討するために，時定数をある値にした際の挙動を調べてみる．

- $T \gg 1$ の場合：時定数 T が無限大だとすると，$\exp(-t/T)$ は 1 なので，忘却がないことになる．この場合には，一旦多くのアクセスがあったデータは，いつまでもキャッシュに残り続けることとなる．
- $T = 1$ の場合：時定数 $T = 0$ はあり得ないので，$T = 1$ が一番小さい場合である．このときは，毎回 $\exp(-1)$ 倍になるので．カウンタの値は常に 0 である．したがって，結果は LRU と同じことになる．（毎回同じものが入れ替えられないように，キャッシュのソートで，「大きくない」という条件とした．）

いずれもカウンタを設けた意味が全くない．根拠はないが，クロックに対して，1500 倍ぐらいの T を考えてみよう．$T' = T \ln 2$ であったので，$T' \fallingdotseq 1050 \fallingdotseq 1024$ である．この場合は，$k = 10$ とすれば良い [*6]．しかし，節の冒頭に述べた結論のように，実際には適当な時定数は実験で決める必要がある．

とりあえずの方針としては，「カウンタがオーバーフローしない範囲で，適当な大きさ」ということになるのだが，「一つの処理にかかる時間に対しては長く，処理が変更された状況では短くなるような値」というのが指標になる．また，実時間クロックは，文字通り実時間だと bit 数が多くなるため，実時間の一桁とか二桁とか長い時間で刻む方が良い可能性がある．

同様に，カウンタの整数部と小数部のビット数（i, j）も決める必要がある．あまり少なすぎてはオーバーフローやアンダーフローが心配であるが，多すぎるのはハードウェアがもったいない．まず時定数を決めた後に，これも実験的に余裕を持った値に決めるのが良い．

5. おわりに

本稿では，データアクセスの頻度がべき乗分布にしたがうという仮定の下，キャッシュ化の方針と，実際に分布を測定しながらキャッシュするアイディアを述べた．実際には厳密にべき乗分布をしている必要はなく，単調に減少する分布であれば良い．

研究の目的は，ネットワーク透過な単一レベル記憶を効率的に運用するためのキャッシュ制御方法を確立することであった．しかし，得られた結論自体は当初の目的に限定されずに，他のいろいろな場面で応用が可能である．たとえば，DRAM より少し遅い不揮発メモリを主記憶として持つ CPU のキャッシュ機構への応用が考えられる．さらに，インターネットデータのローカルキャッシュの制御にも応用可能である．これら別の応用の詳細な検討は，今後に残された問題である．

先の述べたように，未来は予測できないので，このようなキャッシュ制御法を採用したとしても，実際に効率が良くなるかどうかは未知である．別の残された問題点としては，提案方式を LRU やランダム選択と多くの実データを用いて，実験的に比較することが必要である．

参考文献

[1] Wikipedia: Single-level store, available from ⟨https://en.wikipedia.org/wiki/Single-level_store⟩, (2016-09-05).
[2] 前田 賢一: ネットワーク上の仮想マルチコア計算機構成法, *ComSys2016* **予稿集**, (2016-11-29), pp. 72–79.
[3] Kilburn, T., et al: One-Level Storage System, *IRE Trans. Electronic Computers*, Vol. 2 (1962), pp. 223–235.
[4] Organick, E.I.: *The Multics System*, MIT Press (1972).

[*6] $T' = 2^k$ というのは必須ではなく真面目に計算しても良い．

[5] Tanenbaum, A.S.: *Modern Operating Systems, 4th Edition*, Pearson (2014).
[6] 増井 俊之: 第 6 回なんでもフラクタル, available from ⟨http://www.pitecan.com/articles/WiredVision/wv06/index.html⟩, (2007-08-03).
[7] 市川 裕介, 小林 透: ユーザの Web アクセス履歴のべき乗分布傾向に着目した属性推定手法の提案, **情報処理学会論文誌**, Vol. 52 (2011), No. 3, pp. 1195–1203.
[8] トマ・ピケティ, (山形, 他 (訳)): *21* **世紀の資本**, みすず書房, (2014).
[9] Wikipedia: ジップの法則, available from ⟨https://ja.wikipedia.org/wiki/ジップの法則⟩, (2017-08-16).
[10] Wikipedia: Memory segmentation, available from ⟨https://en.wikipedia.org/wiki/Memory_segmentation⟩, (2016-09-05).