社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 二重キャッシングによるMemcached高速化の提案

福田エリック駿† 　定久　紀基† 　井上　浩明†† 　竹中　　崇†† 　浅井　哲也†
本村　真人†

† 北海道大学 情報科学研究科　〒 060–0814 北海道札幌市北区北 14 条西 9 丁目
†† 日本電気株式会社　〒 211–8666 神奈川県川崎市中原区下沼部 1753

**あらまし**　Memcached は多数のサーバのメモリ上にデータをキャッシングすることで Web サーバなどの応答を高速化する技術である。Memcached の処理は非常に単純である一方大きなメモリバンド幅を必要とするが、既存の汎用プロセッサでは低消費電力と高メモリバンド幅を両立することが難しいため、FPGA を用いて Memcached を高速化する研究が近年盛んに行われている。本研究では、Memcached の機能全体を FPGA に実装する従来のアプローチとは異なり、Memcached サーバ上に搭載した FPGA 搭載 NIC 上に Memcached の機能とデータの一部をキャッシングするアプローチを提案する。ソフトウェアシミュレーションによる評価の結果、本アプローチでは通常のソフトウェアによる Memcached サーバと比べて平均遅延が最大 6 倍改善することがわかった。

**キーワード**　Memcached、FPGA、NIC

## 1. はじめに

Web サービスを提供する多くの企業ではサーバの効率を上げるために様々な技術を取り入れている。そのような技術の中でも近年多くの大規模 Web サービス企業で使われているのが Key-value store (KVS) と呼ばれるデータベース技術である。KVS では Key と Value のペア (Key-value pair、KVP) を保存しておき、Value が必要となった際に Key を問い合わせることで、Key とペアになった Value を読み出す。KVS は従来の関係データベース管理システム（Relational database management system, RDBMS）と比較してデータの読み出しが速いことから、多くの大規模 Web サービス提供企業で採用されている。例えば Amazon では DynamoDB、GitHub や Digg などでは Redis、Facebook や Twitter などでは Memcached といった KVS 技術が使われている。Memcached は KVS の中でも多数のサーバの DRAM 上に分散してデータを保持する KV Cache と呼ばれる技術で、ハッシュテーブルを使ったシンプルなデータ構造と、シンプルな処理方式（後述）を持つことから多くの Web サービスで採用されている。

Memcached は内部で複雑な計算をあまり行わないため、一般的なサーバで用いられる Intel® Xeon®(注1) のようなプロセッサでは処理能力が余ってしまう [1]。その一方で Memcached はデータや命令のメモリからの転送頻度が高いため Intel® Atom™(注2) のような低電力プロセッサではメモリバンド幅が不足してしまう [2] [3] [4]。このため、Memcached を高い電力効率で実行することが難しかった。この問題を解決するため、専用のアーキテクチャを持ったプロセッサがいくつか提案され、FPGA で実装／評価されている [3] [4] [5] [6]。これらのプロセッサは高い電力効率を示すが、ソフトウェアによる Memcached が持つすべてのコマンドに対応することが難しい、サーバと同等のメモリ容量を搭載することが難しい [7]、などの問題が残っている。

以上の状況を踏まえ、本稿では

- 通常すべてホストコンピュータ上で行われている Memcached の一部の機能とデータを NIC 上にキャッシュする手法を提案する。
- 提案手法により、Facebook における Memcached データ [10] をモデルに生成されたデータに対して、ソフトウェアによる Memcached に比べてレイテンシを改善できることをソフトウェアシミュレーションにより示す。その結果、レイテンシを最大で 6 倍程度改善できる見込みを得た。

既存の研究では Memcached 専用のハードウェアによって従来の Memcached サーバを置換しようとしたきたのに対し、本稿で提案する手法の狙いは NIC 上で Memcached の機能とデータの一部をキャッシュすることによって従来の Memcached サーバを補完することである。従って本手法は既存の手法に比べて、ソフトウェアによる Memcached からハードウェアによる Memcached への移行をよりスムーズに行えるというメリットがある。

以降の章は次のような構成となっている。2. ではハードウェアによって Memcached を高速化する既存の研究を紹介する。3. では Memcached の仕組みと、処理やデータの特徴について説明する。4. では本研究で提案する手法を、5. では評価方法を紹介し、6. では提案手法の有効性をソフトウェアシミュレー

(注1)：Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.
(注2)：Intel and Intel Atom are trademarks of Intel Corporation in the U.S. and/or other countries.

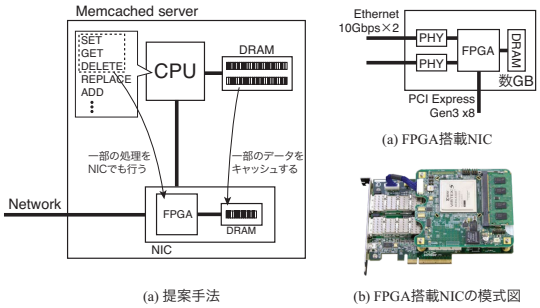



図 1　提案手法のイメージ

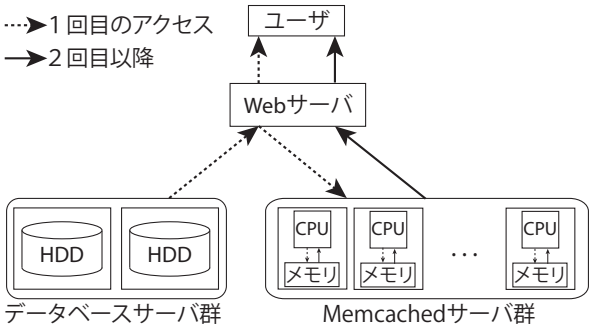



図 2　Memcached の利用イメージ

ションにより評価、考察する。

## 2. 関 連 研 究

Berezechi らは Tilera 社が提供する TILEPro64 プロセッサで Memcached を評価した [8]。TILEPro64 は 64 個のコアを搭載したプロセッサで、コア毎に処理を割り当てることができる。Memcached に必要となる Linux やパケット処理、ハッシュ計算などに割り当てるコア数を変えながら評価をした結果、Xeon とくらべて 2.4 倍の単位電力あたりのスループットが得られた。また、Memcached クライアントからの UDP パケットを使ったリクエストに対するレイテンシは 200-400$\mu$s であった。Xeon のレイテンシは 200-300$\mu$s 程度なので、この値は Xeon と同程度か若干悪化している。

FPGA を使った最初の Memcached プロセッサの研究は Chalamalasetti らによって行われた [5]。この研究では受け取ったパケットをすべてハードウェアで処理する。処理はネットワーク処理部と Memcached アプリケーション処理部の二つに分かれる。ネットワーク処理部では受け取ったパケットから必要な情報を取り出して FIFO 経由で Memcached アプリケーション部に渡す、あるいはその逆を行う。ネットワーク処理部からデータを受け取った Memcached アプリケーション部では、ハッシュ計算をしてデータのアドレスを算出し、メモリの書き出し・読み込みを行う。この研究で性能は飛躍的に向上し、単位電力あたりのスループットが Xeon の 4.3 倍となり、レイテンシが 2.4-12$\mu$s となった。

Blott らはこのシステムでボトルネックとなっていたネットワーク処理部の UDP オフロードエンジンとパイプライン処理を改善し、Xeon と比較して 15 倍以上の単位電力あたりのスループットと、3.5-4.5$\mu$s のレイテンシを達成した。この研究ではメモリのアロケーションを CPU に計算させることで高速化していることも特徴の一つである。

また、同時期に二つの研究が類似のアプローチで Memcached を高速化しようとしている [3] [4]。これらの研究では、ソフトウェアによる Memcached ではプロトコルスタックやカーネル、ライブラリコードの読み込みが頻発することによる命令キャッシュミスや予測分岐の失敗がボトルネックになっていること突き止め、ネットワーク処理と Memcached 処理の一部（GET リクエストの処理）をハードウェア化し、CPU コアと共に SoC にするという手法を提案した。この手法は CPU コアを搭載する FPGA にプロトタイプとして実装され、Xeon と比べて 2.3-6.1 倍の単位電力あたりのスループットが得られた。

[3] と [4] はネットワークインターフェースで一部の Memcached 処理をハードウェアで行い、対処できないものは CPU に任せるという点で本研究で提案する手法と似ているが、メモリ（データ）を Memcached 処理回路と CPU で共有している点で本研究とは異なる。

既存の Memcached サーバを置き換えることが可能な商用の Memcached アプライアンスも登場している [9]。この製品では CPU と多数の FPGA を用いる事により、GET リクエストのスループットを Xeon と比較して 9.7 倍まで向上させている。ただしレイテンシは 500$\mu$s-1ms と Xeon に比べて大きい。また、このシステムが高速化するのは GET リクエストのみで、単位電力あたりの性能は公開されていない。

## 3. Memcached

Memcached は多数の Memcaced サーバのメモリ上にデータをキャッシュする技術である。図 2 に示すように、Web サーバがユーザから受けるリクエストに素早く応じるために必要なデータを Memcached サーバ群に保存しておくのが一般的な使い方である。Memcached サーバは一台あたり数十 GB 以上のメモリをもち、数百台がクラスタとして運用されていることが多い。あるデータは最初の段階では Memcached サーバに保存されておらず、Web サーバはデータベースからデータを持ってくる必要がある。Web サーバはそのデータを使ってユーザに応答を返すと共に、Key（250 バイト以下）と Value（1MB 以下）のペアを持つ SET リクエストを送ることでデータを Memcached に保存しておく。次に Web サーバで同じデータが必要になった際には、Memcached サーバに GET リクエストを送ることでデータベースにアクセスするよりも速くデータを得ることができる。Memcached サーバの容量が一杯になってくると、あまり使われていないデータは Memcached から消去される。消去されたデータについて Web サーバが GET リクエストを送ると、Memcached サーバはキャッシュミスが発生したことを Web サーバに通知する。その様な場合には Web サーバは再度データベースにアクセスしてデータを得、それを Memcached サーバに保存し直す必要がある。

Memcached には表 1 に示すようなコマンドが用意されてい

表 1　Memcached のコマンド

| コマンド | 処理の説明 |
|---|---|
| SET | Memcached に KVP を格納する |
| ADD | ある Key が Memcached に存在しなければ KVP を格納する |
| REPLACE | ある Key が Memcached に存在していればそこに新たな Value を格納する |
| APPEND | ある Key に対応する Value の先頭に値を追加する |
| PREPEND | ある Key に対応する Value の末尾に値を追加する |
| CAS | ある KVP が前回読んでから変更されていなければ新たな Value を格納する |
| GET | ある Key に対応する Value を Memcached から取得する |
| GETS | GET を行いながら、CAS に用いる識別子を得る |
| DELETE | ある KVP を Memcached から削除する |
| INCR /DECR | ある Key に対応する Value の値をインクリメント/デクリメントする |
| STATS | Memcached の利用状況を取得する |

る。実際によく利用されるのは GET/SET/DELETE の三つのコマンドで、GET はその中でも特に利用頻度が高い。Facebook における Memcached の利用状況をまとめた文献 [10] では、あるデータ群における GET/SET/DELETE の割合が 30:1:14（DELETE については具体的な数字が示されていないため、グラフを目測で読み取った値）であることがわかっている。このような背景から 2. で紹介した関連研究では GET のみ [3] [4] [9]、GET と SET のみ [5] [8]、あるいは GET と SET と DELETE の三種類のコマンドのみ [6] に対応している。本研究でも NIC では GET/SET/DELETE のみを処理する。

## 4. 提案手法

本研究で提案する手法は、Memcached サーバで行う処理とデータの一部を NIC でハードウェア処理することにより、Memcached における処理の大部分を NIC で低遅延で処理し、あまり使われない処理は CPU に任せるシステムである。具体的には、NIC では表 1 に示した Memcached コマンドのうち、頻繁に使われる SET/GET/DELETE の三つのコマンドを処理する。各コマンドに対する挙動は次の通りである。

- SET: リクエストが持つ Value サイズが特定のサイズ（Chunk サイズ、後述）以下であれば図 3 のように KVP を Key テーブルと Value 保持領域に保持する。同時に CPU にリクエストを送り（ライトスルー）、Web サーバに成功／失敗を返す。ライトスルーを行うのは、表 1 に示した STATS コマンドに対応するためである。Memcached は各コマンドの受信回数やヒット率などの利用状況を保持し、STATS コマンドを受け取るとそれらの情報を送り返す。従って Memcached が正確な利用状況を把握するためには NIC は（GET、DELETE を含めた）全てのリクエストを CPU に送る必要がある。
- GET: リクエストが持つ Key が NIC で保持されていれば Value と共に Web サーバに返す。同時にリクエストを CPU に送る。Memcached でこの KVP が保持されていればそれが

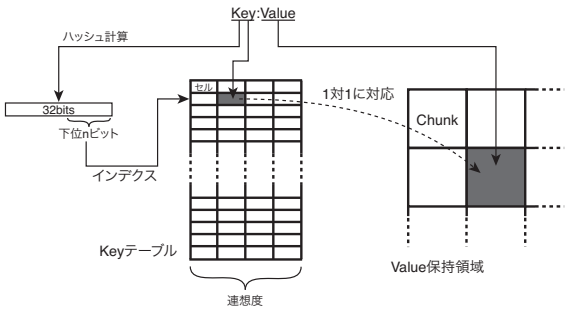


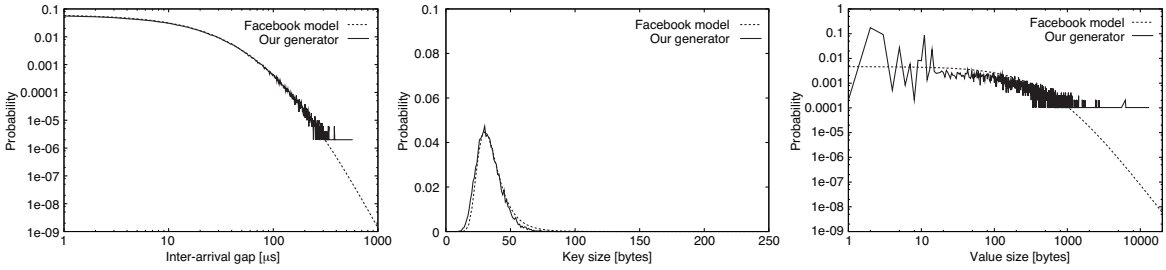

図 3　Key テーブルと Value 保持領域の対応

NIC に返ってくるが、もしすでに NIC が KVP を Web サーバに送り返していれば Memcached からのリプライを Web サーバに送り返さずに消去する。もしそうでなければ NIC にその KVP がキャッシュされていないということなので、NIC にキャッシュした上で Web サーバに KVP を送る。

- DELETE: リクエストがもつ Key が NIC で保持されていればそれを削除し、Web サーバに成功／失敗を返す。同時にリクエストを CPU に送る。

このようなシステムは図 1b に示すような FPGA 搭載 NIC を使用することを想定している。Memcached がサポートするすべてのコマンドに FPGA で対処することは難しいため、また実際の Memcached 運用でも使われるコマンドのほとんどが SET/GET/DELETE であるため、NIC ではこの三つのコマンドだけに対応し、他のコマンドを持ったリクエストが送られてきた場合には NIC で対応せず、そのリクエストをホスト CPU 上で動作する Memcached に処理を任せる。ただし、本稿の段階では実際に NIC に搭載された FPGA に回路を実装するのではなく、このようなシステムをソフトウェアシミュレーションで評価する。なお、シミュレータは NIC の動作だけをシミュレートし、Memcached には実際の Memcached プログラムを用いる。

シミュレータ内の仮想的な NIC には図 3 のような Key テーブルと Value 保持領域を用意する。CPU キャッシュではキャッシュのインデクスはメモリアドレスの下位数バイトから決定されるが、Memcached では KVP をメモリアドレスで管理するのではなく Key で管理するため、CPU キャッシュと同様の方法が使えない。そこで Key から 32 ビットのハッシュ値を計算し、その下位数バイトを Key テーブルのインデクスとする。Key テーブルの各セルには Key が保持されるが、その大きさは Memcached の仕様で定められている Key の最大長である 250 バイトとした。Key テーブルはセットアソシアティブ方式とし、同じハッシュ値をもつ複数の Key をテーブル上に保持することができる。あるインデクスにおいて Key テーブルがサポートする連想度を超える Key が SET された場合、LRU（Least Recently Used、最後に参照されてから最も時間が経っている Key を消去する）か FIFO（一番初めに SET された Key を消去する）で Key テーブルから Key を削除し、そこにあらたな Key を保持する。Key テーブルの各セルは対応する固定メモリ領域（Chunk）を持っており、セルに保持された Key に対応す




(a) リクエストの間隔の確率分布　(b) Keyサイズの確率分布　(c) Valueサイズの確率分布

図 4　テストデータの確率的性質（破線は [10] でモデル化された値）

る Value はこのメモリ領域に保持される。

## 5. 評価方法

### 5.1 テストデータの生成

キャッシュの性能を評価するには実際に使われているデータを用いるのが最も確実な方法だが入手が難しいため、Facebook で運用されている Memcached サーバの負荷を分析した文献 [10] で紹介されている値や性質を元に、テストデータ生成プログラムを作成した。このプログラムで作成したテストデータは次のような特徴を持つ。

- GET/SET/DELETE リクエストの比率は 30:1:14 である。
- ソフトウェアによる Memcached に対して 90%程度のヒット率を持つ。
- 1$\mu$s の時間窓内で一度しか現れない Key（ユニーク Key）の比率は 45%である。
- リクエストの間隔は図 4a の確率分布をとる。
- Key の長さは図 4b の確率分布をとる。
- Value の長さは図 4c の確率分布をとる。
- GET リクエストに対する応答がミスだった場合にのみ SET リクエストを送信する。

SET リクエストは GET リクエストに対して Memcached でミスが発生した際にのみ送られるため、SET リクエストに対して NIC キャッシュでは必ずキャッシュミスが発生する。また SET リクエスト数に対し DELETE リクエスト数が 14 倍存在するが、これはデータベースの書き換えの際にその値が Memcached に保持されている、いないに関わらず、Memcached から書き換え前の値を削除するためである。

### 5.2 シミュレーション手順

テストデータ生成プログラムを使ったシミュレーションは次の手順で行う。

（1） 図 4b の確率分布を用いて $k$ 個の Key を生成する。

（2） コールドミスを避けるためにウォームアップを行う。生成された Key の組から Key を選び、図 4c の確率分布を用いて生成した Value とペアにして SET コマンドと共に NIC シミュレータに送る。これを Memached で最初の Key の追い出しが発生するまで続ける。

（3） GET/SET/DELETE リクエストを 30:1:14 の比率で NIC シミュレータに送り始める。その際、Key は上で予め作成した Key から選ぶ。GET リクエストに対しミスが返ってきたら、その Key を新たな Value とペアにし、SET コマンドと共に NIC に送る。これを 10 万リクエスト（およそ 2 秒分に相当）行い、ヒット率を算出する。

ただし、この手順を NIC シミュレータに対して行う前に Memcached に対して行い、ヒット率が 90%程度になるような $k$ の値を見つけておく。こうすることで、Memcached の容量に合わせたテストデータを生成することが可能になる。本稿での評価ではシミュレーション時間短縮のため Memcached の容量をサポートされている最小の容量（8MB）とした。これは実際運用に使われる Memcached 容量と比べて非常に小さな値であるが、容量に応じて $k$ の値を設定するため、評価結果にはある程度の妥当性があると考えられる。

## 6. 結果と考察

### 6.1 LRU と FIFO

最初に Key テーブルと Chunk サイズに関する複数の設定について、LRU と FIFO の Key テーブル置換アルゴリズムを用いた際の GET コマンドに対するヒット率を比較した結果を図 5 に示す。これを見ると、LRU と FIFO の違いによるヒット率の差はほとんど見られないことがわかる。従って NIC における Memcached のキャッシュにおいては LRU と FIFO のどちらを用いても良いが、以降の実験では LRU を用いる（ただし 16 ウェイの LRU のハードウェア実装は非常に難しいため、擬似 LRU を用いることを仮定する）。またヒット率が 60%と CPU キャッシュに比べて低い。これは Chunk サイズを超える Value サイズを持つ KVP を NIC でキャッシュしないことと、保持される Value の多くが Chunk サイズよりもかなり小さいサイズを持つことからキャッシュ容量の大部分が無駄になっていることに起因すると考えられが、より詳しい調査を今後行う必要がある。

### 6.2 Chunk サイズ

次に Key テーブルのインデクス数が 4096 と 8192、また連想度が 1 と 16 の場合について、Chunk サイズ変化させた際のヒット率の変化を図 6 に示す。図 4c からもわかるとおり、1024 バイト以上の Value サイズをもつリクエストはほとんど存在しないため、Chunk サイズを 1024 バイトより大きくしてもはほとんどヒット率は向上しない。問題は 1024 バイト以下のどの値に設定するかであるが、図 6 を見ると 256 バイト以下で急激

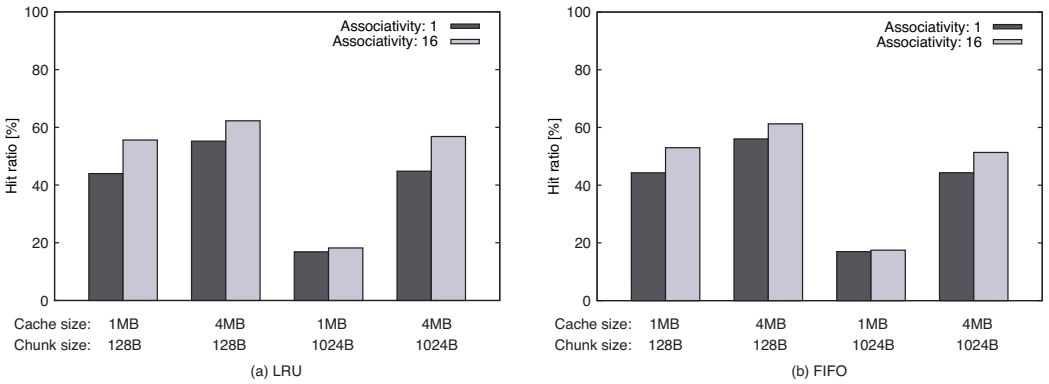



図 5　Key テーブル置換アルゴリズムによるヒット率の違い

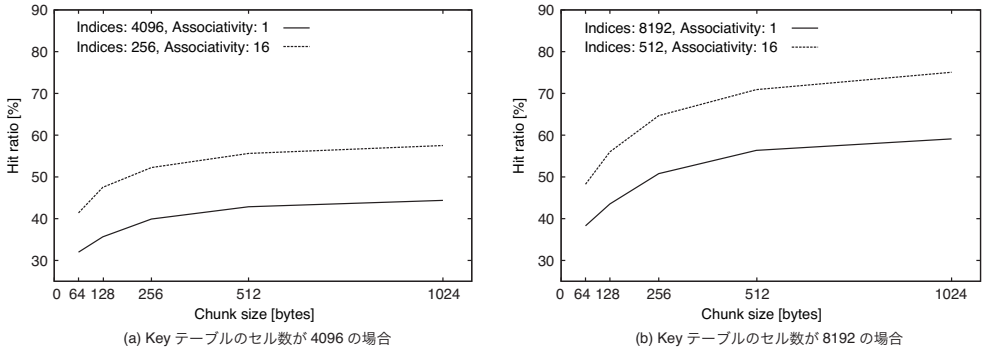



図 6　Chunk サイズによるヒット率の違い

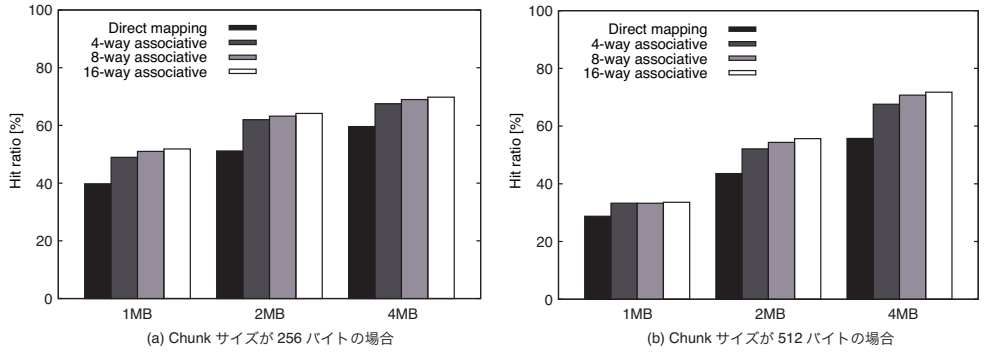



図 7　キャッシュサイズ（Value 保持領域サイズ）によるヒット率の違い

にヒット率が下がっている。逆に 512 バイトから 1024 バイトに増やしても 5%程度しかヒット率は上がらない。従って、全ての Chunk サイズを固定にする上で最も効率のよい Chunk サイズは 256 バイトか 512 バイトであると考えられる。

### 6.3　Value 保持領域の容量

図 7 にキャッシュサイズ（Value 保持領域サイズ）によるヒット率の違いを Chunk サイズが 256 バイト（a）と 512 バイト（b）の二つの場合について示した。これを見るとキャッシュサイズが小さい（1MB）時には Chunk サイズを小さくして保持できる Key の数を増やしたほうがヒット率を上がることがわかる。しかし、キャッシュサイズが 4MB 程度になると Chunk サイズによるヒット率の違いはほとんど見られなくなる。これはキャッシュサイズが大きくなると Chunk サイズが大きいことによる KVP の取りこぼしの少なさがヒット率を上げる要因として強く現れてくるためであると考えられる。また決められたキャッシュサイズの中では、Key テーブルの連想度を上げることでヒット率は上昇する。キャッシュサイズ、Chunk サイズに関わらずダイレクトマッピング方式はヒット率が低いので避けたほうが良いが、4 以上の連想度ではそれほど大きくヒット率は変わらないため、ハードウェア化する際のコストを勘案しながら適切なものを選ぶことが好ましい。

一方で、Memcached の容量 8MB に対し、NIC キャッシュのサイズが 4MB にもなるのは通常の CPU キャッシュと比べて考えるとかなりキャッシュの容量が大きいと言える。この原因は本実験で Chunk サイズを固定したことにある。表 2 に図 7a、b でキャッシュサイズが 4MB、16 ウェイ・アソシアティブの場合の保持されている合計 Value サイズを示す。ここからわかるように、用意されている Chunk はほぼ全て使用されているものの、各 Chunk の平均充填率が 20%程度にとどまっている。様々なサイズの Chunk を用意し、Value サイズに応じ

表 2　Chunk の使用状況

| Chunk サイズ | キャッシュサイズ | 使用キャッシュサイズ | 保持 Value サイズ | 平均 Chunk 充填率 |
|---|---|---|---|---|
| 256B | 4MB | 3.78MB | 782KB | 20.2% |
| 512B | 4MB | 3.97MB | 731KB | 18.0% |

表 3　ヒット率による遅延改善

| ヒット率 | A. 遅延改善 (GET) | B. 遅延改善 (全体) |
|---|---|---|
| 60% | 2.44 倍 | 4.78 倍 |
| 70% | 3.22 倍 | 6.25 倍 |
| 80% | 4.69 倍 | 9.09 倍 |

て Chunk を割り当てる仕組みを導入することによって、本実験で得られた結果と同様のヒット率をより小さいサイズの NIC キャッシュで実現できると考えられる。

### 6.4　遅延の改善

実際にレイテンシがどれくらい減少するかの見積もりには [6] の値を用いる。[6] では Xeon を使った際のレイテンシが 200-300$\mu$s、提案システムのレイテンシが 3.5-4.5$\mu$s としている。そこで本研究ではそれらの中間値（250$\mu$s と 4$\mu$s）をそれぞれ NIC キャッシュでミスした場合の遅延、NIC キャッシュでヒットした場合の遅延として利用する。表 3 に示した GET リクエストの遅延減少率と図 7 から、本提案手法によって GET リクエストの遅延を 3.2 倍程度改善できることがわかる。

また、SET/DELETE リクエストも 4$\mu$s の遅延で対応できると仮定すると、表 3 の B 列に示すように遅延を 9.1 倍程度改善できることになる。このような仮定は以下の理由により十分可能である。SET/DELETE リクエストに対して Memcached は成功／失敗を返すが、SET/DELETE が失敗するのは SET する Value のサイズが大きすぎたり Memcached の内部でエラーが起きた時である。Value サイズが大きすぎる場合については SET 失敗は NIC で容易に判断できる。Memcached の内部でエラーが起きた場合についても NIC から一旦成功を返しておいて、Memcached から失敗が返ってきた際に NIC キャッシュ上の当該の KVP を消去するなどしておけば、同じ KVP に対する次の GET リクエストに対して再度ミスは発生するものの、正しい値は Web サーバがデータベースから取ってくるのでシステム全体としては整合性が保たれる。

## 7. 結　　論

本稿では Memcached サーバに搭載した NIC 上で Memcached データをキャッシングすることで Memcached の平均遅延を削減する手法を提案した。Facebook の Memached データを模倣するテストデータ生成プログラムを作成し、それで生成したデータを用いて提案手法をソフトウェアシミュレーションで評価したところ、GET リクエストに対して最大 3.2 倍、リクエスト全体に対して最大 6 倍程度の遅延を削減できることがわかった。本手法の現在の問題点は、様々な大きさを持つ Value に対して固定サイズのメモリ領域を割り当てていることからメモリの使用効率が悪いことと、Memcached の容量が少ないことである。今後メモリ割り当てアルゴリズムを工夫することにより、より少ないメモリサイズで現在と同等、あるいはそれ以上の遅延削減を実現することと、シミュレーション時間を短縮しより大きな Memcached 容量でシミュレーションを行う必要がある。また、今回提案した手法の実装／評価を行っていく。

## 文　　献


[1] D.G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, “Fawn: A fast array of wimpy nodes,” Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, pp.1–14, SOSP ’09, ACM, New York, NY, USA, 2009.

[2] W. Lang, J.M. Patel, and S. Shankar, “Wimpy node clusters: What about non-wimpy workloads?,” Proceedings of the Sixth International Workshop on Data Management on New Hardware, pp.47–55, DaMoN ’10, ACM, New York, NY, USA, 2010.

[3] K. Lim, D. Meisner, A.G. Saidi, P. Ranganathan, and T.F. Wenisch, “Thin servers with smart pipes: Designing soc accelerators for memcached,” Proceedings of the 40th Annual International Symposium on Computer Architecture, pp.36–47, ISCA ’13, ACM, New York, NY, USA, 2013.

[4] M. Lavasani, H. Angepat, and D. Chiou, “An fpga-based in-line accelerator for memcached,” IEEE Computer Architecture Letters, vol.99, pp.1–4, 2013.

[5] S.R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, “An fpga memcached appliance,” Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp.245–254, FPGA ’13, ACM, New York, NY, USA, 2013.

[6] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, “Achieving 10gbps line-rate key-value stores with fpgas,” Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing, pp.1–6, USENIX, Berkeley, CA, 2013.

[7] A. Wiggins and J. Langston, “Enhancing the scalability of memcached”. http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0.

[8] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, “Many-core key-value store,” Proceedings of the 2011 International Green Computing Conference and Workshops, pp.1–8, IGCC ’11, IEEE Computer Society, Washington, DC, USA, 2011.

[9] “Convey computer memcached appliance”. http://www.conveycomputer.com/files/1813/7998/4963/CONV-13-046_MCD_Datasheet.pdf.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store.,” SIGMETRICS, eds. by P.G. Harrison, M.F. Arlitt, and G. Casale, pp.53–64, ACM, New York, NY, USA, 2012.