社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2009-ARC-181／2009-EMB-11 (19)
2009/1/14

マルチコアにおけるオーバーヘッド解析を用いたキャッシュコアの最適化

森 洋介[†] 森谷 章[††]
藤枝直輝[††] 吉瀬謙二[††]

プロセッサのコア数が増加してゆく中，複数のコアが同時にメインメモリへアクセスすることで，通信オーバーヘッドが増加してしまう懸念がある．このメモリアクセスの集中を緩和するために，マルチコアプロセッサにおけるコアの利用法としてデータ供給の支援を目的とするキャッシュコアがある．キャッシュコアにはソフトウェアのキャッシュを実装する．本稿では Cell/B.E. の SPE にキャッシュコアを実装する．キャッシュコアのソフトウェアオーバーヘッドや通信オーバーヘッドを最適化する．また，キャッシュコアの性能を詳しく評価する．

The Cache-Core optimization on Multi-CoreProcessors considering several overheads

Yosuke Mori,[†] Akira Moriya,[††] Naoki Fujieda[††]
and Kenji Kise[††]

The number of cores in a processor increases. If several cores access to the main memory at the same time, the memory access latency increases. The Cache-Core architecture is proposed to reduce the centralized access to the main memory on multi-core processors. The Cache-Core aims to support other cores supplying data as well as a hardware cache. The Cache-Core is software-implemented cache. In this article, we optimize the Cache-Core implemented on SPE of Cell/B.E. considering the software overheads and the communication latency. We report the evaluation of its performance.

1. はじめに

プロセッサの性能向上を目指し，チップに複数のコアを搭載したマルチコアプロセッサが多く登場している．今後，半導体技術の向上により多くのコアを載せたメニーコアプロセッサへと向かうと考えられる．

マルチコアプロセッサではアプリケーションの並列性を高め，複数のコアに上手にタスクを配分することで処理の高速化を望むことができる．しかし，この方法ではコアの増加にアプリケーションの並列性が追いつかなければ，期待される速度向上を得ることはできない．またメインメモリへのデータアクセスの集中がボトルネックとなり，性能がコア数の増加と共に性能が線形に増加しない．

このような背景から，我々はマルチコアプロセッサのコアの利用法として，他コアへのデータ供給支援を目的とする**キャッシュコア**[1)] の研究を進めている．キャッシュコアはソフトウェア実装のキャッシュプログラムを実行する．また，アプリケーションを実行するコア（計算コア）の L2 データキャッシュとして機能する．すなわち，キャッシュコアは計算コアに対してデータを供給することで，通信レイテンシの大きいメモリアクセスを減少させ，処理速度の向上を目指す．

本稿では，Cell/B.E. の SPE に計算コアとキャッシュコアを実装する．アプリケーションを実行させたときのキャッシュコアのソフトウェアオーバーヘッドと通信オーバーヘッドを解析する．その解析結果からキャッシュコアを最適化する．また，キャッシュコアの性能を詳しく評価する．

本稿の構成を述べる．2 章では，キャッシュコアの初期実装について述べる．3 章では新たな試みとして行ったキャッシュコアの最適化について述べる．4 章でキャッシュコアの評価と考察をする．5 章でまとめる．

† 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
†† 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

```
1  void vec_add()
2  {
3     for (j = 0; j < N/M; j++){
4         DMA(a+j*M, MEM_ADR, M*sizeof(int), GET);
5         DMA(b+j*M, MEM_ADR, M*sizeof(int), GET);
6         wait_dma();
7
8         for (i = 0; i < M; i++)
9             c[j*M + i] = a[j*M + i] + b[j*M + i];
10
11        DMA(c+j*M, MEM_ADR, M*sizeof(int), PUT);
12        wait_dma();
13    }
14 }
```

図 1 ローカルストア（LS）を意識したベクトル加算の疑似コード

```
1  int mem_ld(int addr)
2  {
3     int buf;
4     DMA(&buf, addr, sizeof(int), GET);
5     wait_dma();
6     return buf;
7  }
8
9  void mem_st(int addr, int data)
10 {
11    DMA(&data, addr, sizeof(int), PUT);
12    wait_dma();
13 }
14
15 void vec_add()
16 {
17    for (i = 0; i < N; i++) {
18        int a_i = mem_ld(a + i*sizeof(int));
19        int b_i = mem_ld(b + i*sizeof(int));
20        mem_st(c + i*sizeof(int), a_i + b_i);
21    }
22 }
```

図 2 本研究のプログラミングモデルを採用するベクトル加算の疑似コード

2. キャッシュコアアーキテクチャ

2.1 プログラミングモデル

キャッシュコアを利用するプログラミングモデルは，アプリケーションを記述する際，汎用のプロセッサとできるだけ同様に書けることとする．

図 1 に Cell/B.E. におけるローカルストア (LS) を意識したプログラムの疑似コードを示す．通常，LS の容量の許す限りデータをまとめて DMA 転送でロード（4,5 行目）し，ロードしたデータに対し処理が終わるとまとめてメインメモリへ DMA 転送（11 行目）を行う．これは Cell/B.E. を意識した記述であり，汎用性に欠けている．

図 2 に本研究のプログラミングモデルを用いる擬似コードを示す．ここでは，メインメモリのデータを 1 つの配列と見なして，1 要素のデータアクセスに対して 1 回の DMA 転送（4,11 行目）を行う．このため Cell/B.E. を意識したプログラミングモデルで記述されたコードと比較すると性能が低下する．しかし，アプリケーションプログラマにとっては，コード記述時に LS の容量を気にする必要がないという利点がある．

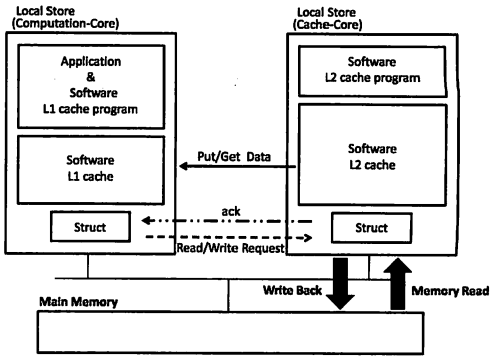

図 3 キャッシュコアアーキテクチャ

2.2 キャッシュコアアーキテクチャの概要

図 3 にキャッシュコアアーキテクチャを示す．図左に示すように，アプリケーションを動作させるコアのことを計算コア（Computation-Core）と呼ぶ．ただし，計算コアではソフトウェア L1 キャッシュを動作させることがある．

図右に示すキャッシュコアは計算コアのデータ供給の支援を目的とする．キャッシュコアを利用したアプリケーションはデータアクセスにおいて，まず計算コア内の L1 キャッシュを参照する．L1 キャッシュでミスが起こるとキャッシュコアの L2 キャッシュを参照する．

計算コアは L1 キャッシュと，データ転送用の構造体を持つ．キャッシュコアは L2 キャッシュと，データ転送用の構造体を持つ．この構造体はフラグ，データアドレス，データを格納し，互いに転送することで処理の要求や同期を実現する．

2.3 キャッシュコアの初期実装

キャッシュ方式はソフトウェア処理の簡単化のため，ダイレクトマップを採用する．リプレース方式はメインメモリアクセスの削減のために，L1，L2 キャッシュ共にライトバック方式を採用する．

図 4 にキャッシュコアの初期実装におけるデータ転送の流れを示す．図上の L2 キャッシュの読み出しでは，計算コアは構造体のフラグを READ にし，必要なデータのアドレスを構造体に書き込み，キャッシュコアに DMA 転送する．キャッシュコアは要求を受け取った後，L2 キャッシュにデータがヒットした場合は該当ラインを L1 キャッシュに DMA 転送する．ミスした場合はキャッシュコアはメインメモリからデータを取り，L1 キャッシュへ DMA 転送する．また，L2 キャッシュは ack として構造体を転送する．この構造体を L1 キャッシュが受け取るとキャッシュコアからデータが転送されたことを知り，処理を再開する．

図下の書き込み時は計算コアが構造体のフラグを

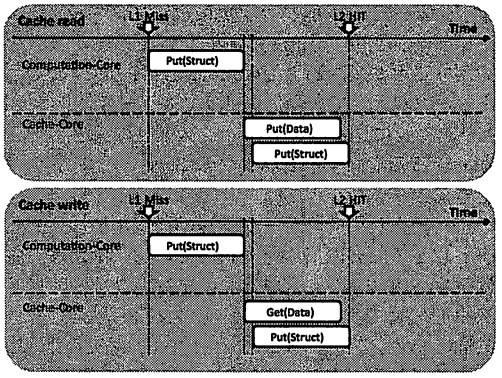

図 4 キャッシュコア初期実装におけるの読み書きでのデータ転送

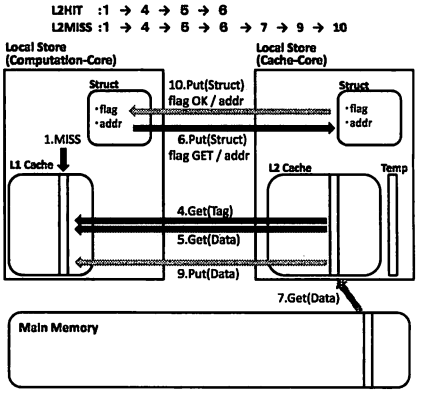

図 5 今回最適化したキャッシュコア使用時におけるデータ転送の流れ

WRITE にし，必要なデータアドレスを構造体へ書き込みキャッシュコアへ DMA 転送する．キャッシュコアは構造体を受け取ると L2 の該当するラインがダーティならばメインメモリへ書き戻す．そして，L1 から要求されたラインを受け取ると読み込み時の処理と同様，計算コアへ ack を返す．計算コアは ack を受け取ると処理を再開する．

この実装では通信レイテンシのオーバーヘッドが非常に大きいため，キャッシュコアの使用による速度低下がおきてしまう．実測値によると Cell/B.E. ではメインメモリからデータをロードする時のレイテンシが約 500 サイクルである．これと比較して，コア間のレイテンシは約 230 サイクルと小さくない．キャッシュコアの L2 キャッシュにヒットしても通信レイテンシだけで約 460 サイクルを必要とする．さらにソフトウェアオーバーヘッドを考慮すると，メインメモリから直接データをロードするよりも遅くなる．

なお，キャッシュコアの初期実装に関する詳細は 1) を参照のこと．

3. キャッシュコアの最適化

3.1 L2 キャッシュヒット時の転送の最適化

本節では L2 キャッシュにヒットした時のオーバーヘッドを小さくすることを重視して初期実装を最適化する．

図 5 に最適化を行った実装のデータ転送の流れを示す．ここではラインを入れ替える際，ラインがダーティでないと仮定し，ライトバックの処理は考えない．

キャッシュコアからデータを読み込む処理は以下のようになる．各番号は図 5 の番号に対応している．

- L1 キャッシュでキャッシュミスが発生 (1).
- 計算コアはアドレスから L2 キャッシュのタグとインデックスを計算し、キャッシュコアからデータとタグを DMA 転送で取得 (4, 5) する．また、必要なデータのアドレスを構造体に書き込み、キャッシュコアへ DMA 転送 (6) する．計算コアは取得したタグとデータのアドレスから計算したタグを比較し、タグが一致していれば L2 キャッシュヒットとなる．
- 構造体を受け取ったキャッシュコアは送られたアドレスからタグとインデックスを計算する．キャッシュコアのキャッシュにデータがあることが分かれば計算コアに送ったデータは正しいので、キャッシュコア処理を終了する．
- 計算コアが要求したデータがキャッシュコアになければキャッシュコアはメインメモリからデータを取得 (7) し，そのデータを計算コアの該当ラインへ DMA 転送 (9) する．またキャッシュコアは構造体を ack として転送 (10) する．計算コアは ack を受け取るとキャッシュコアからデータが送られたことを知り処理を再開する．

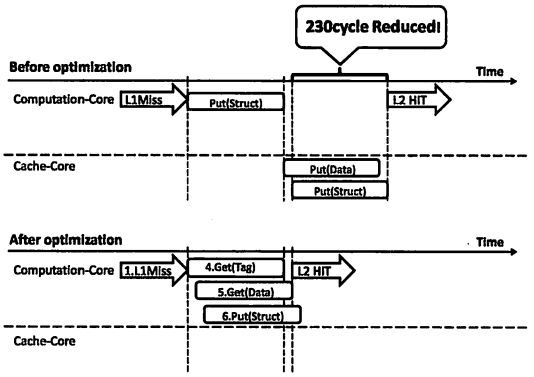

図 6 最適化前との L2 キャッシュヒット時のデータ転送の比較

図 6 に最適化前と最適化後の L2 キャッシュヒット時のデータ転送の流れを示す．この実装ではキャッシュ

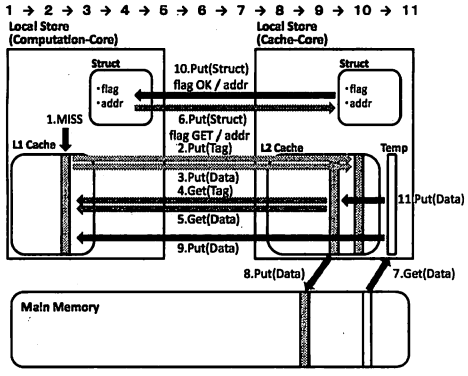

図 7 今回最適化したキャッシュコア使用時におけるライトバック処理を含めたデータ転送の流れ

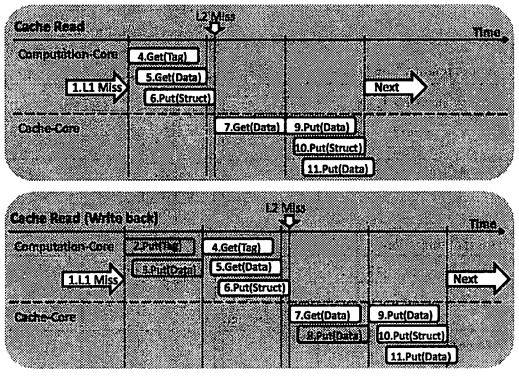

図 8 今回最適化したキャッシュコア使用時における L2 キャッシュミス時のデータ転送の流れ

コアに必要なデータがある場合、要するオーバーヘッドを一回分の DMA 転送の約 230 サイクルを削減できる.

また，この実装では L1 キャッシュのデータは必ず L2 キャッシュに存在し，データ書き込み時は必ず L1 キャッシュに書き込まれる．そのため，L1 キャッシュでライトバック処理が起きても，該当ラインの L2 キャッシュでのタグとインデックスを計算し，そこへ直接 DMA 転送できる.

3.2 その他の最適化

前節ではライトバックが起こらないことを仮定している．しかし，ライトバックが起こるとオーバーヘッドはさらに大きくなる．前節の実装を元に，ライトバック処理のオーバーヘッドを削減する最適化を行う.

図 7 と**図 8** にライトバック処理を含めた実装のデータ転送の流れを示す．図 8 の上図はライトバック処理がない場合，下図は L1，L2 ともにライトバック処理がある場合のデータ転送の流れを示している．L2 キャッシュのライトバック処理のオーバーヘッド削減のためにキャッシュコアに受信バッファ(Temp) を設けている．この受信バッファによって，キャッシュコアはデータを取得すると同時に，メモリへデータを書き戻すことができる．そのため，L2 キャッシュでライトバックが起きても受信バッファにデータを取得し L1 キャッシュに送る間に，メインメモリに書き戻すことができるので，ライトバックのオーバーヘッドを隠蔽できる.

また，これまでのデータ転送の最適化に加えて，ソフトウェア部分の最適化も行っている．例えば，従来は L1,L2 キャッシュそれぞれのエントリ数に応じてマスクを変えてタグやインデックスを計算していたが，共通のマスクを用いることで L1 キャッシュで計算したタグやインデックスを L2 キャッシュでも使用できる.

4. 評　　価

4.1 評価環境

Cell/B.E. 上で最適化を行ったキャッシュコアを評価するための環境を示す．評価を行う実機として PLAYSTATION3 を用いる．キャッシュコアを評価する基準として，計算コア上に L1 ソフトウェアキャッシュのみを実装する版（利用するコアは 1 個）を利用する．この版ではデータアクセスはコア内の L1 キャッシュを参照し，ヒットしない場合はメインメモリを参照する.

また，キャッシュコアを評価するプログラムとして次の 3 種類を用いる．3 種類のうちの 1 つ目は初期実装によるキャッシュコア（Cache-Core Ver1）である．2 つ目は今回オーバーヘッド解析を行って最適化したキャッシュコア (Cache-Core Ver2) である．3 つ目は最適化を行ったキャッシュコアをさらにアセンブラレベルで Cell/B.E. 用に最適化したもの (Cache-Core Ver2.1) である.

L1 キャッシュの容量は共通して 2KB である．また，L2 キャッシュの容量は 128KB である．キャッシュのラインサイズは 128B とする.

評価のためのアプリケーションにはクイックソート，行列積，マージソート，ダイクストラ法，姫野ベンチマーク，FFT(高速フーリエ変換) の 6 種類を使う．姫野ベンチマークと FFT の要素は 4 バイトの単精度浮動小数点型データとする．それ以外のアプリケーションの要素は 4 バイトの整数型データである.

4.2 各アプリケーションの性能比

図 9 に各アプリケーションのキャッシュコアを使用する場合の相対性能を示す．それぞれ L1 キャッシュのみを搭載した版の性能を 1 としている．性能は，各々のアプリケーション毎に設定したデータサイズにおける調和平均である．各アプリケーションにおいて，Ver1 と Ver2 の比較により，今回の最適化によってキャッシュコアの性能が大きく向上することが分かる．また，

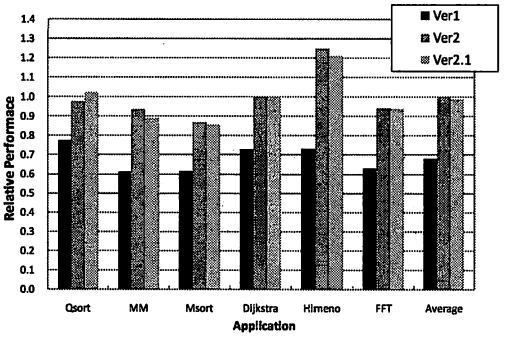

図 9 実機における各アプリケーションの L1 キャッシュのみを使用する版に対する各キャッシュコアの相対性能の平均

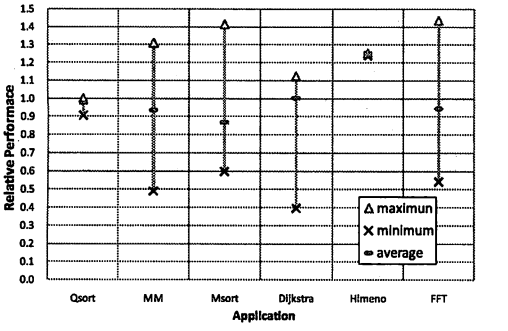

図 10 実機における各アプリケーションの L1 キャッシュのみを使用する版に対するキャッシュコア Ver2 の相対性能の最大値，最小値，平均値

Cell/B.E. 用に最適化された Ver2.1 はクイックソートのような L1 ヒット率が高いアプリケーションで性能が上がっている．

図 10 に Cache-Core Ver2 での各アプリケーションで設定したデータサイズ内での性能の最大値と最小値と平均値を示す．最大で性能が L1 キャッシュのみ使用する場合よりも 1.4 倍程向上する．このようなアプリケーションでは，ヒット率がキャッシュの容量に依存する傾向にある．また，キャッシュの容量を超えてしまうとスラッシングが起きてしまいヒット率が大きく下がる行列積，マージソート，FFT では L1 キャッシュのみ版と比べて性能が低下する．しかし，図 9 から分かるように最適化する前よりも性能は向上している．

4.3 L1 キャッシュ容量を変えたときの性能比

図 11 に L1 キャッシュのサイズを 2KB，4KB，8KB と変えたときのキャッシュコアを使用する場合の相対性能を示す．L1 キャッシュのみを搭載した版の性能を 1 としている．このキャッシュコアは今回最適化を行った Ver2 を使用している．L1 キャッシュの容量を

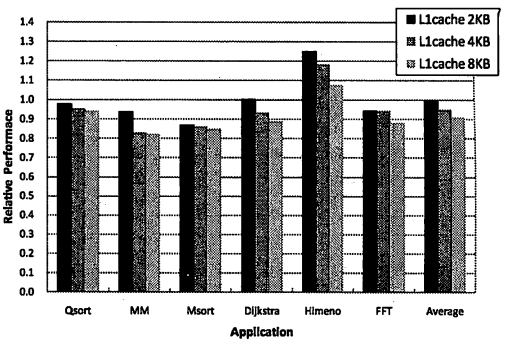

図 11 L1 キャッシュの容量を変えた時の実機における各アプリケーションの L1 キャッシュのみを使用する版に対するキャッシュコア Ver2 の相対性能の平均

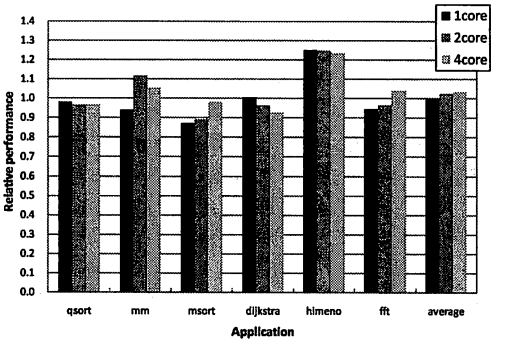

図 12 キャッシュコアの数を変えた時の実機における各アプリケーションの L1 キャッシュのみを使用する版に対するキャッシュコア Ver2 の相対性能の平均

増やしていくとキャッシュコアの有効性が薄れてしまい，性能は低下する．しかし，姫野ベンチマークのようなキャッシュコアに有利なアプリケーションでは依然として性能が向上している．

4.4 キャッシュコアの数を変えたときの性能比

図 12 にキャッシュコアの数を変えた時のキャッシュコアを使用する場合の相対性能を示す．L1 キャッシュのみを搭載した版の性能を 1 としている．このキャッシュコアは今回最適化を行った Ver2 を使用している．L1 キャッシュの容量は 2KB とする．どのコアへデータ転送するかはタグの有効ビットの下位 2 ビットのマスクにより判断しているため，2 個のコアをキャッシュコアとして使う場合（2core）は容量 256KB のダイレクトマップのキャッシュと見なせる．同様に，4 個のコア（4core）では容量 512KB のダイレクトマップのキャッシュと見なせる．

データを保持するコアを計算するオーバーヘッドや計算終了時のライトバック処理のオーバーヘッドにより，キャッシュの容量の影響が少ないアプリケーショ

表 1 各アプリケーションにおけるキャッシュコア Ver2 の最大性能時におけるヒット率

Application	L1 only	L1 + L2
Qsort	97.65	99.57
MM	46.50	99.20
Msort	44.04	99.02
Dijkstra	89.17	98,93
Himeno	65.45	97.77
FFT	10.84	99.90

表 2 クイックソートにおけるメモリアクセス回数

Data size	L1 only	L1 + L2	reduction rate(%)
64KB	15,099	512	96.61
128KB	35,444	1,024	97.11
256KB	82,175	6,407	92.20
512KB	193,166	33,795	82.50
1024KB	413,914	71,747	82.66
2048KB	935,842	205,112	78.08
4096KB	2,064,810	512,994	75.16

ンでは性能が低下するが，キャッシュの容量が性能に大きく影響するアプリケーションは性能が向上する．

4.5 ヒット率とメモリアクセス数の変化

表 1 に，キャッシュコア Ver2 の最大性能時における L1 キャッシュのみ使用する場合と，1 つのキャッシュコアを使う場合のヒット率を示す．L1 キャッシュは 2KB とする．

表 2 に，クイックソートにおけるメモリのアクセス回数を示す．データサイズが小さい場合にはメモリアクセス数を 90%以上削減でき，4MB の場合においては，約 1/4 に抑えることができる．今後コア数が増えていく場合，キャッシュコアの利用によってメモリアクセスの増加を抑え，アクセス集中によるレイテンシの増大防止を見込むことができる．

4.6 考　　察

表 3 に，キャッシュの各動作状況において処理に要するレイテンシ (cycle) を示す．各レイテンシは Cell/B.E. の SPE の機能である SPU デクリメンタを用いて計測する．これは最適化をした Ver2 を用い，各々のアプリケーション毎に設定したデータサイズにおける調和平均の値である．CLEAN はキャッシュラインの値の変更がなくライトバック処理が起きなかったことを示し，DIRTY はライトバック処理が起きたことを示す．表 3 を見ると，今回の実装が L2HIT 時に非常に有利に働いていることが分かる．また，受信バッファにより L2 ライトバックの際のレイテンシを低く抑えられていることも確認できる．

表 3 キャッシュの各動作状況の処理に要するレイテンシの平均

Case	L1 only	L1 + L2
L1:HIT	75	77
L1:MISS-CLEAN L2:HIT	582	383
L1:MISS-DIRTY L2:HIT	929	646
L1:MISS-CLEAN L2:MISS-CLEAN	582	1282
L1:MISS-DIRTY L2:MISS-CLEAN	929	1547
L1:MISS-CLEAN L2:MISS-DIRTY	582	1324
L1:MISS-DIRTY L2:MISS-DIRTY	929	1556

5. ま と め

本稿ではメニーコアプロセッサにおけるデータ供給支援を目的とするキャッシュコアのソフトウェア処理と通信レイテンシのオーバーヘッドを解析して最適化を行った．また，データサイズがキャッシュコアのキャッシュ容量に収まる場合において L1 キャッシュのみを用いる場合よりも最大 1.4 倍の高速化を達成した．

現在，キャッシュの方式をソフトウェア処理の簡単化のためダイレクトマップ方式で実装している．そのため，行列積やマージソートや FFT においてスラッシングが起きて，キャッシュヒット率が大きく低下する．さらなる性能向上のため，プリフェッチや，SIMD を用いたセットアソシアティブを導入することで，ソフトウェアオーバーヘッドを抑えつつヒット率の向上を目指す．また，ソフトウェア実装であることを利用し，アプリケーションごとに，より性能の出る実装に動的に切り替えることも今後の課題となる．

謝　　辞

本研究の一部は，財団法人北九州産業学術推進機構 (FAIS) の支援による

参 考 文 献

1) 森谷 章, 藤枝 直輝, 佐藤 真平, 吉瀬 謙二: メニーコアプロセッサに向けたデータ供給を支援する多機能キャッシュコア, SACSIS2008, pp.421-430(2008).
2) 佐藤芳紀, 神酒 勤: Cell Broadband Engine への SPE ソフトウェアデータキャッシュの実装, 情報処理学会研究報告, HPC-110, pp.13-18(2007).
3) J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'brien, K. O'brien: A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor, The 20th International Workshop on Languages and Compilers for Parallel Computing(2007).
4) 林徹生, 今里賢一, 井上弘士, 村上和彰: 演算/メモリ性能バランスを考慮した CMP 向けオンチップ・メモリ貸与法の提案, 情報処理学会研究報告, ARC-176, pp.59-64(2008)
5) Kistler, M., Perrone, M. and Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed, *IEEE micro*, Vol.26, No.3, pp.10-23(2006).