演算/メモリ性能バランスを考慮したCMP向けヘルパースレッド実行方式の提案と評価

今里, 賢一
九州大学大学院システム情報科学府

福本, 尚人
九州大学大学院システム情報科学府

井上, 弘士
九州大学大学院システム情報科学研究院

村上, 和彰
九州大学大学院システム情報科学研究院

出版情報：電子情報通信学会技術研究報告. 108 (28), pp.75-80, 2008-05. IEICE
バージョン：
権利関係：

演算/メモリ性能バランスを考慮したCMP向け ヘルパースレッド実行方式の提案と評価

今里　賢一†　　福本　尚人†　　井上　弘士†　　村上　和彰†

† 九州大学大学院 システム情報科学府/科学研究院 〒 819–0395 福岡市西区元岡 744
E-mail: †{imazato,fukumoto}@c.csce.kyushu-u.ac.jp, ††{inoue,murakami}@i.kyushu-u.ac.jp

あらまし　複数のプロセッサコアを 1 チップに搭載するチップマルチプロセッサ (CMP) が現在注目されている．チップ内スレッドレベル並列処理により高い演算性能を得ることができるためである．しかしながら，メモリバンド幅の制約や複数コア搭載によるメモリアクセス頻度の増加により，メモリウォール問題が深刻化する．その結果，多くのメモリ参照を必要とする並列プログラムの実行においては実効性能が低下するといった問題が生じる．そこで本稿では，CMP の性能向上を目的として，演算性能とメモリ性能のバランスを考慮したヘルパースレッド実行方式を提案する．従来の方式では，スレッドレベル並列性を高めるため，搭載された全てのプロセッサコアを利用して並列プログラムを実行する．これに対し，提案方式では，一部のプロセッサコアをプリフェッチを行うヘルパースレッドに割当てる．ヘルパースレッドの最適な数が既知であると仮定して提案方式の性能を評価した結果，従来方式と比較して，最大で 47%の性能向上を得ることができた．

キーワード　チップマルチプロセッサ，並列処理，プリフェッチ，ヘルパースレッド

Performance Balancing: An Efficient Helper-Thread Execution on CMPs

Kenichi IMAZATO†, Naoto FUKUMOTO†, Koji INOUE†, and Kazuaki MURAKAM†

† Graduate school / Faculty of Information Science and Electrical Engineering, Kyushu University
Motooka 744, Nishi-ku, Fukuoka-shi, 819–0395 Japan
E-mail: †{imazato,fukumoto}@c.csce.kyushu-u.ac.jp, ††{inoue,murakami}@i.kyushu-u.ac.jp

Abstract　Conventional CMPs attempt to exploit the thread-level parallelism (TLP) by using all of the cores integrated in a chip. However, this kind of straightforward way does not always achieve the best performance. This is because the memory-wall problem becomes more critical in CMPs, resulting in poor performance in spite of high TLP. To solve this issue, we propose an efficient thread management technique, called performance balancing. We dare to throttle the TLP to execute software prefetchers as helper-threads. Our experimental results show 47% speed up in the best case compared with a conventional parallel execution.

Key words　chip multiprocessor, parallel processing, prefetching, helper thread

1. はじめに

複数のプロセッサコアを 1 チップに搭載するチップマルチプロセッサ (CMP) が注目されている．CMP は，複数コアで並列処理することにより，高い演算性能を達成することができる．しかしながら，オフチップメモリバンド幅の制約や複数コア搭載によるメモリアクセス頻度の増加によりメモリウォール問題が深刻化する．その結果，多くのメモリ参照を必要とする並列処理においては実効性能が低下するといった問題が生じる．このような場合，単純に全てのコアで並列実行しても十分な性能向上を得られない．

そこで本研究では，CMP における並列処理の性能向上を目的として，演算性能とメモリ性能のバランスを考慮したヘルパースレッド実行方式を提案する．従来方式とは異なり，並列プログラムを実行するプロセッサコア数を減らす一方，メモリ性能向上を目的としたヘルパースレッドを実行するプロセッサコア数を増加させる．このように，演算性能をある程度犠牲にしてでも，性能向上阻害要因であるメモリボトルネックを解消することにより CMP 全体の性能向上を目指す．

本稿の構成は以下の通りである．第 2 節で CMP における問題点について議論し，従来の解決策について述べる．第 3 節では，提案方式とそのアーキテクチャ・サポートについて説明する．さらに性能モデル式を用いて提案方式の効果を示す．第 4 節では，ベンチマークプログラムを用いた定量的評価を行い，

提案方式の有効性を明らかにする．最後に第 5 節でまとめる．

2. チップマルチプロセッサにおける問題点と従来の解決策

2.1 性能向上阻害要因

CMP 上の並列処理では，実行コア数に比例した性能向上を実現できない場合が多くある．この 1 つ要因として，コア数の増加に伴う相対メモリ性能の低下が挙げられる．図 1 は CMP における実行コア数と性能向上の関係を示している (実験環境は第 5.2 節を参照)．横軸は実行コア数，縦軸は L2 キャッシュサイズ 1MB のときのシングルスレッド実行を基準とした性能向上率である．凡例の”L2-1MB” は L2 キャッシュサイズを 1MB とした場合，“L2-perfect” はオフチップメモリアクセス時間ゼロの理想メモリを想定した場合を示す．

図 1(d) の *Barnes* に関しては，実行コア数の増加に対してほぼ比例した性能向上を得ることができている．しかしながら，図 1(a)(b)(c) に示すようにその他 3 つのプログラムにおいては十分な性能向上を達成していない．これは，主に以下 2 つの問題に起因する．

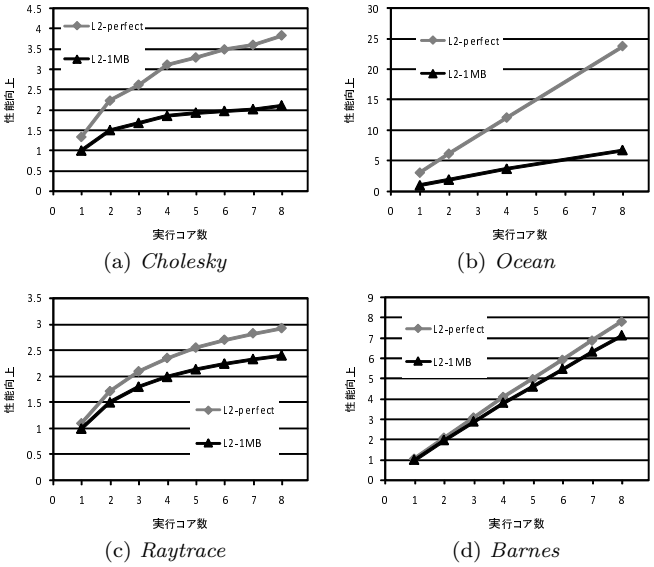
- 低いスレッドレベル並列性：図 1(a)(c) に示す *Cholesky* と *Raytrace* では，理想的なメモリシステムを想定した L2-perfect においても，実行コア数が 4 以上になると十分な性能向上を実現できていない．実行コア数の増加に伴い，その性能改善率が徐々に低下している．これは，1) 並列化不可能な処理部分の顕在化や，2) コア間通信オーバーヘッドの増大が原因である．

- メモリウォール問題：図 1(a)(b) に示す *Cholesky* や *Ocean* において，L2-1MB モデルではコア数の増加に伴う性能向上が低い．これに対し，L2-perfect では極めて高い性能となっている．特に *Ocean* については，実行コア数に対する性能スケーラビリティは高いものの，メモリ性能の違いにより大きな差が存在している．これは，演算性能とメモリ性能差の拡大 (いわゆるメモリウォール問題) が顕在化したためである．

2.2 従来技術による解決策

CMP において高い性能を実現するためには，第 2.1 節で示した問題点を解決する必要がある．低いスレッドレベル並列性による性能向上の阻害を解決する手法として，並列化コンパイラによる並列性の抽出がある．また，実行スレッド数を制限する手法などが提案されている [1].

一方，CMP のメモリ性能を向上させる手法として，共有型キャッシュにおける競合ミス削減法が提案されている．例えば，文献 [2] では，総 L2 ミス回数の削減を目的としたキャッシュ領域割当て方式が提案されている．また，CMP においてアイドル状態のコアが存在する場合，それを利用してヘルパースレッド（ソフトウェア・プリフェッチャ）を実行する方法がある．プリフェッチの方法としては，周囲のコアのメモリ参照アドレス計算を先行して行う方法や，周囲のコアのキャッシュミス情報からメモリ参照アドレスを予測する方法などが提案されている [3] [4].



(a) *Cholesky* (b) *Ocean*

(c) *Raytrace* (d) *Barnes*

図 1　実行コア数の増加による性能向上

3. 演算/メモリ性能のバランスを考慮したヘルパースレッド実行方式

3.1 基 本 概 念

第 2.2 節で説明した従来技術では，CMP の演算性能やメモリ性能の向上についてそれぞれ個別のアプローチを取っている．しかしながら，より高い性能を得るためには演算性能とメモリ性能の両方を考える必要がある．そこで本稿では，演算/メモリ性能のバランスを考慮したヘルパースレッド実行方式を提案する．図 2 に，提案方式の基本概念を示す．通常の実行方式とは異なり，メインスレッドを実行するメインコアと，メモリ性能を高めるためにヘルパースレッドを実行するヘルパーコアが存在する．ここで，メインスレッドとはアプリケーション・プログラムを実行するスレッドである．ヘルパーコアはすべてのメインコアのメモリ性能を高めるために，メインコアからキャッシュミス情報を収集しプリフェッチを行う．これにより，メインコアでのキャッシュミスが減少する．プログラム実行において，演算性能が必要な場合はメインコア数を，メモリ性能が必要な場合はヘルパーコア数を増加する．こうすることで，演算性能とメモリ性能のバランスをとり，CMP 全体でより高い性能を実現する．ヘルパースレッドを実行することによるメモリ性能の向上が，メインコアの減少による演算性能の低下を上回れば，CMP 全体の高性能化を実現できる．

3.2 アーキテクチャ・サポート

本稿では，オンチップに共有 L2 キャッシュを持つ CMP アーキテクチャを対象とする．したがって，ヘルパースレッドにより L2 キャッシュにプリフェッチされたデータに対しメインコアからアクセス可能となる．

ヘルパーコアは，周りに存在する複数のメインコアに対してプリフェッチを行う必要がある．したがって，ヘルパーコアはメ

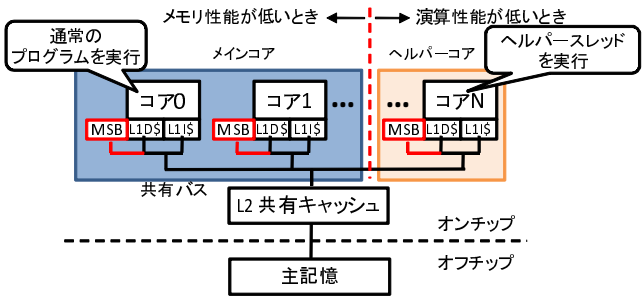


図 2　提案方式の全体図

インコアとキャッシュミス情報を共有しなければならない．これを可能にするために，各プロセッサコアに対してキャッシュミス情報を格納する専用バッファを追加する (図 2)．これを Miss Status Buffer(以下 MSB) と呼ぶ．MSB では，各キャッシュミスを発生したメモリ参照アドレス，プロセッサコア番号，当該メモリ参照命令の PC 値を保持する．これらの情報は，キャッシュミスが発生したときに生じるコヒーレンス維持のためのブロードキャストをスヌープすることで取得する．ヘルパーコアが複数存在する場合は，それぞれのヘルパーコアが担当するメインコアを決定する．各ヘルパーコアの MSB には担当するメインコアのキャッシュミス情報のみが格納される．ヘルパーコアは MSB に収められたキャッシュミス情報を参照し，メインコアのキャッシュミスアドレスを予測して L2 共有キャッシュにプリフェッチを行う．

3.3　性能モデリング

本節では，提案方式の性能モデリングを行う．並列処理の実行クロックサイクル数は，最も実行時間の長いスレッドの実行クロックサイクル数で表される．スレッド数 i における実行クロックサイクル数 CC_i は，演算実行に要するクロックサイクル数 $CC_{exe,i}$，メモリアクセスによりストールした実行クロックサイクル数 $CC_{mem,i}$ によって以下のように表される．

$$CC_i = CC_{exe,i} + CC_{mem,i} \tag{1}$$

ただし，ここでは簡単化のため演算とメモリアクセスのオーバーラップ実行は考慮しない．ここで，$CC_{exe,i}$ と $CC_{mem,i}$ は以下のように表すことができる．

$$CC_{exe,i} = (1-f) \times CC_{exe,1} + \frac{f \times CC_{exe,1}}{i} \tag{2}$$

$$CC_{mem,i} = AC_i \times \{HCC_{L1} + MR_{L1,i} \times (HCC_{L2} + MR_{L2,i} \times MML)\} \tag{3}$$

各項の定義は以下の通りである．

- f：並列化できる演算の割合
- AC_i：スレッド数が i のときのメモリ参照回数
- HCC_{L1}：L1 キャッシュアクセスに要するクロックサイクル数
- $MR_{L1,i}$：スレッド数が i のときの L1 キャッシュミス率
- HCC_{L2}：L2 キャッシュアクセスに要するクロックサイクル数
- $MR_{L2,i}$：スレッド数が i のときの L2 キャッシュミス率
- MML：主記憶アクセスに要するクロックサイクル数

なお，L1-L2 キャッシュ間，L2-主記憶間のバスアクセスに要する時間はスレッド数に対し一定であると仮定し，L2 キャッシュアクセス時間ならびに主記憶アクセス時間に含めている．さらに，AC_i はスレッド数 i と並列化できるメモリアクセスの割合 f_{AC} を用いて，

$$AC_i = (1-f_{AC}) \times AC_1 + \frac{f_{AC} \times AC_1}{i} \tag{4}$$

と表すことができる．$f = f_{AC}$ を仮定すると，式 (1)〜(4) より CC_i は，

$$CC_i = \left\{(1-f) + \frac{f}{i}\right\} \times \left[CC_{exe,1} + AC_1 \times \left\{HCC_{L1} + MR_{L1,i} \times (HCC_{L2} + MR_{L2,i} \times MML)\right\}\right] \tag{5}$$

と表すことができる．従来方式の実行クロックサイクル数は，コア数を N とした場合，式 (5) の i に N を代入することにより導かれる．

次に，N コア CMP 上で m 個をヘルパーコアとして動作させた場合の提案方式の実行クロックサイクル数を求める．$N-m$ スレッド実行時の実行クロックサイクル数 CC_{N-m} を式 (5) より求め，式変形を行うと式 (6) を得る．

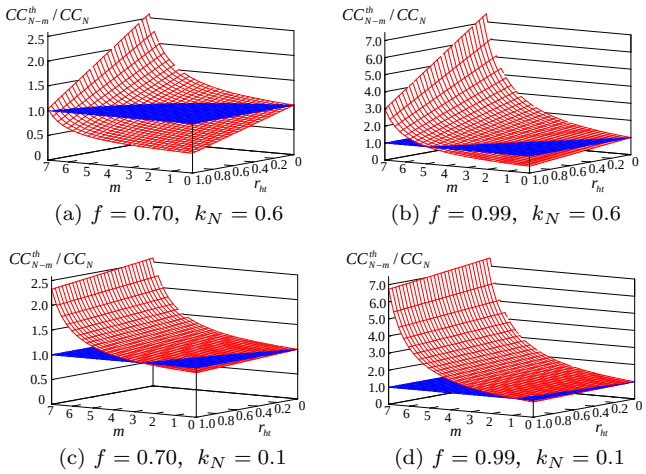
$$CC_{N-m} = \frac{1-f+\frac{f}{N-m}}{1-f+\frac{f}{N}} \times (1-r_{MR_{L2},N,m} \times k_N) \times CC_N \tag{6}$$

$$k_N = \frac{AC_N \times MR_{L1,N} \times MR_{L2,N} \times MML}{CC_N} \tag{7}$$

ここで，$r_{MR_{L2},N,m}$ は N スレッド実行から $N-m$ スレッド実行に変更したときの L2 キャッシュミス率の減少率である．k_N はスレッド数 N で実行したときの，全実行時間に占める主記憶アクセスによるプロセッサストール時間の割合である．また，本稿ではプライベート L1 キャッシュを前提としており，以下のように仮定した．

- $MR_{L1,N-m} = MR_{L1,N}$：スレッド数 N とスレッド数 $N-m$ とでは L1 キャッシュミス率はほぼ同一と考えられる．メモリアクセス回数が変化するため，厳密には上記の値は異なるが，アクセスパターンはほぼ同じと考えられるためミス率の変化は小さいと推測できる．
- $MR_{L2,N-m} = (1-r_{MR_{L2},N,m}) \times MR_{L2,N}$：スレッド数 N とスレッド数 $N-m$ とで，L2 キャッシュミス率は $r_{MR_{L2},N,m}$ の割合だけ減少する．スレッド数の減少により，あるキャッシュセットに対してアクセスされる間隔が長くなると考えられる．そのため競合性のミスが減り，ミス率は減少する．

ヘルパースレッドを実行させた場合，プリフェッチ効果によ



(a) $f = 0.70$, $k_N = 0.6$　(b) $f = 0.99$, $k_N = 0.6$

(c) $f = 0.70$, $k_N = 0.1$　(d) $f = 0.99$, $k_N = 0.1$

図 3　提案方式の実行クロックサイクル数 CC^{ht}_{N-m} の変化 (コア数 $N = 8$)

り L2 キャッシュミス率が減少する．m 個のプロセッサコアでヘルパースレッドを実行した場合に式 (6) の L2 キャッシュミス率の減少率 $r_{MR_{L2},N,m}$ が r_{ht} に変化したとする．ここで，r_{ht} は従来実行方式を基準としたときの提案実行方式の L2 キャッシュミス率の減少率である．このときの実行クロックサイクル数 CC^{ht}_{N-m} は，$r_{MR_{L2},N,m}$ を r_{ht} に置き換えることによって導かれ，式 (8) で表せる．

$$CC^{ht}_{N-m} = \frac{1 - f + \frac{f}{N-m}}{1 - f + \frac{f}{N}} \times (1 - r_{ht} \times k_N) \times CC_N \quad (8)$$

3.4　従来方式と提案方式の比較

式 (8) を用いて提案方式と従来方式の性能を比較する．図 3 に，f と k_N の値の大小の組み合わせについて，m と r_{ht} が変化したときの実行クロックサイクル数の変化を示す．縦軸は，N スレッド実行時のクロックサイクル数 CC_N を 1 としたときの提案方式による実行クロックサイクル数 CC^{ht}_{N-m} の比を示している．傾きのある曲面が提案方式による実行クロックサイクル数の変化を表しており，この曲面が高さ 1 の平面 (従来の全コア並列実行方式) を下回れば，提案方式により性能向上が実現できることを示す．

提案方式が最も有効なのは，f が小さく，かつ k_N が大きい場合である (図 3(a))．これは，プログラムを実行するスレッドの減少によるクロックサイクル数の増加が小さく (f が小さい)，さらに，L2 キャッシュミス率を改善することによるクロックサイクル数の減少が大きい (k_N が大きい) ためである．逆に，f が大きく，かつ k_N が小さい場合 (図 3(d)) は，提案方式によって性能向上を実現することは難しい．

4.　ベンチマークを用いた定量的評価

4.1　評 価 環 境

提案方式の有効性を明らかにするため，ベンチマークプログラムを用いた定量的評価を行う．ここで，本稿では最も高い性能を達成できるヘルパーコア数は既知と仮定する．また，プログラム実行中にはヘルパーコア数を変更しない．最適なヘルパーコア数の決定法ならびに，動的なヘルパーコア数の変更は今後の課題である．ヘルパースレッドの動作としては，以下の 3 種類を実装した．

表 1　プリフェッチアルゴリズムの設定

アルゴリズム	設定
local stride	テーブルの総エントリ数 2048, degree 8
global stride	ミス履歴バッファのエントリ数 8/core, テーブルのエントリ数 8/core, degree 8
delta correlation	インデックステーブルの総エントリ数 2048, GHB の総エントリ数 2048,

- local stride プリフェッチ [5]：stride プリフェッチの一種である．stride プリフェッチとは，連続したメモリ参照アドレスの差分 (以下ストライド値) が一定であるようなメモリアクセスパターンを検出し，プリフェッチを行う手法である．現在のミスアドレスを a，ストライド値を s とした場合，アドレス $a+s$，$a+2s$，…，$a+ds$ に対しプリフェッチを行う．ただし，d は prefetch degree と呼ばれる値で，キャッシュミス発生時に発行するプリフェッチ数を表す．local stride プリフェッチでは，命令別にミスアドレスのストライド値を計算する．キャッシュミスを起こした命令のプログラムカウンタ値，ミスアドレス，ストライド値を記録するテーブルが必要となる．

- global stride プリフェッチ [4]：stride プリフェッチの一種である．global stride プリフェッチでは現在のミスアドレスと最近 n 個のミスアドレスとの差のうち最も絶対値が小さいものをストライド値として算出する．最近 n 個のミスアドレスを記録するミス履歴バッファとストライド一定のアドレスストリームを記録するテーブルが必要となる．

- delta correlation プリフェッチ [6] [7]：時間的に連続するキャッシュミスアドレスの差分がマルコフ情報源になっていると仮定し，プリフェッチを行う．マルコフモデルの構築はミスアドレスの履歴を記録しておくことにより疑似的に行われる．プリフェッチは現在のマルコフモデル上での状態ノードを特定し，その後マルコフモデルのエッジを辿ることで実現する．本稿では文献 [6] で示された Global History Buffer(GHB) を用いる手法を用いる．

本実験で使用したプリフェッチアルゴリズムの設定を表 1 に示す．本実験では，マルチプロセッサシミュレータ M5 [8] に Miss Status Buffer を実装し評価を行った．実験に用いたシミュレータの設定を表 2 に示す．また，評価対象モデルは以下の通りである．

- BASE：すべてのコアで並列プログラムを実行する従来モデル
- PB-LS：ヘルパースレッドの動作として local stride プリフェッチを実行する提案モデル
- PB-GS：ヘルパースレッドの動作として global stride プリ

表 2　シミュレータの設定

コアの構成	8 コア, イン・オーダ
L1 命令キャッシュ	32KB, 2-way, 64B lines, 1 clock cycle, MSHR 8
L1 データキャッシュ	32KB, 2-way, 64B lines, 1 clock cycle, MSHR 32
L2 キャッシュ	1MB, 8-way, 64B lines, 12 clock cycles, MSHR 92
L1-L2 間共有バス幅	64B
L2-主記憶間バス幅	16B
主記憶レイテンシ	300 clock cycles
miss status buffer	エントリ数：20, レイテンシ：1 clock cycles

表 3　各ベンチマークプログラムの入力と f，k_N の値

プログラム	入力	f	k_N
Barnes	32K particles	1.0017	0.0897
Cholesky	tk29.O	0.7294	0.4519
FMM	64K particles	0.9869	0.2900
LU	1024 × 1024 matrix	0.8818	0.1740
Ocean	258 × 258 ocean	0.9946	0.7155
Radix	16M integers	0.9997	0.6098
Raytrace	teapot	0.7061	0.1811
WaterSpatial	4096 molecules	0.9871	0.0532

f：並列化できる演算の割合
k_N：全てのコアを通常実行させる場合の実行時間にしめる主記憶アクセスによるストール時間の割合

フェッチを実行する提案モデル

- PB-DC：ヘルパースレッドの動作として delta correlation プリフェッチを実行する提案モデル

ただし，PB-LS，PB-GS，PB-DC では，ヘルパーコアの数は 1 以上とする．

ベンチマークプログラムは Splash2 [9] から複数のプログラムを選択した．各ベンチマークプログラムの入力を表 3 に示す．各ベンチマークに関して，第 4 節で用いたパラメータ f と k_N の値は表 3 の通りである(注1)．第 4 節で述べたように，提案方式は f が小さく，また k_N が大きい場合に有効である．従って，k_N が大きい *Cholesky*，*Ocean*，*Radix* といったプログラムや f が小さい *Cholesky*，*LU*，*Raytrace* では，提案方式による効果が期待できる．

4.2 評価結果

4.2.1 提案方式の性能

従来方式と比較した提案方式の性能向上を表す実験結果を図 4 に示す．縦軸はすべてのコアで並列プログラムを実行した場合の性能を 1 としたときの相対性能を示している．それぞれのベンチマークプログラムと評価モデルの組について最適なヘルパーコア数で実行したときの性能を示している．また，表 4 には，そのときコアの割当てを示す．ただし，*Ocean*，*Radix* についてはスレッド数の制約 (2 の累乗にしかスレッド数を設定できない) のため，ヘルパーコアが 4 個の場合しか評価しておらず，最適なヘルパーコア数が 4 となっている．

(注1)：f は，第 2.1 節の実験結果から得られた実行命令数から最小自乗法を用いて近似的に求めた値である．

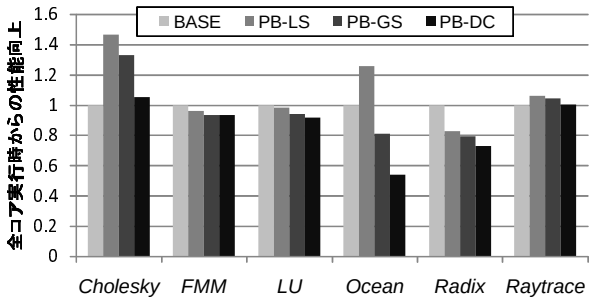


図 4　ヘルパースレッドによる性能の変化

表 4　最適なコアの割当て (メインコア:ヘルパーコア)

評価モデル	*Cholesky*	*FMM*	*LU*	*Ocean*	*Radix*	*Raytrace*
PB-LS	6:2	7:1	7:1	4:4	4:4	7:1
PB-GS	7:1	7:1	7:1	4:4	4:4	7:1
PB-DC	7:1	7:1	7:1	4:4	4:4	7:1

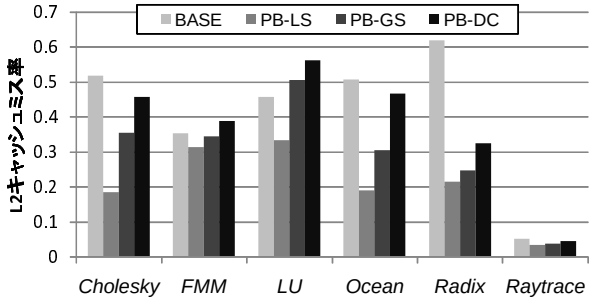


図 5　ヘルパースレッドによる L2 キャッシュミス率の変化

図 4 から *Cholesky*，*Ocean*，*Raytrace* では提案方式による性能向上が得られている．さらに，すべてのベンチマークプログラムにおいて提案手法の評価モデルの中では PB-LS が最も高い性能を示している．*Cholesky*，PB-LS の組で最大 47% の性能向上を達成した．

図 5 には，各モデルに関する L2 キャッシュミス率を示す．ただし，L2 キャッシュミス率はメインコアの L2 キャッシュアクセスのみについて算出している．提案方式により性能向上が得られていた *Cholesky*，*Ocean*，*Raytrace* では，L2 キャッシュミス率の削減が大きい．一方，提案方式による効果が低かった *FMM* では，ほとんど L2 キャッシュミス率を削減できていない．*Radix* では，L2 キャッシュミス率の減少が大きいが，図 4 では性能向上が得られていない．これは，スレッド数の制限のために通常実行を行うコア数を半分にしたことによる性能低下が大きい (第 2.1 節の実験より性能は 34.5% 低下) ためと考えられる．

4.2.2 並列実行部分の提案方式の性能

並列処理では，一つのスレッドのみが動作する逐次実行部分と複数のスレッドが同時に動作する並列実行部分が存在する．本実験において，実際に通常実行を行うコア数を減らし，そのコアをヘルパーコアとして動作させているのは並列実行部分である．

図 6 に並列実行部分のみの提案方式の性能を示す．プログラム全体で性能向上が得られていた *Cholesky* と *Raytrace* におい

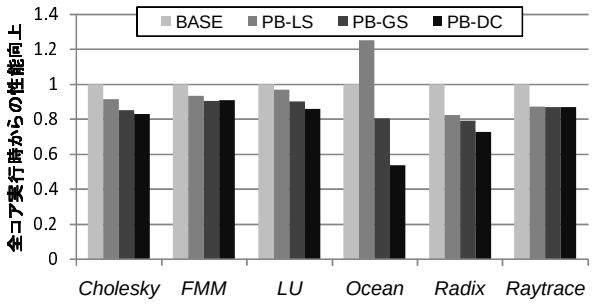


図 6　ヘルパースレッドによる性能の変化 (並列実行部分)

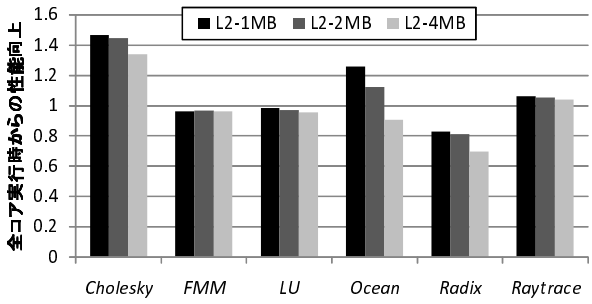


図 7　L2 キャッシュサイズの変化による提案方式の性能の変化 (PB-LS)

ては，並列実行部分では性能向上が得られていない．このようなプログラムでは，プログラム全体を通してあるコアをヘルパーコアとして動作させるのではなく，逐次実行部分のみヘルパーコアとして動作させ，並列実行部分ではメインコアとして動作させることにより，より高い性能向上が得られるのではないかと考えられる．一方，*Ocean* については，並列実行部分でも提案方式による性能向上が得られている．これは，従来実行方式での主記憶メモリアクセスによるストール時間の割合が大きく (k_N が大きく)，さらに図 5 よりヘルパースレッドを実行することによる L2 キャッシュミス率の減少が特に大きいためだと考えられる．

4.2.3　L2 キャッシュサイズを変化させた場合の提案方式の性能

図 7 に L2 キャッシュサイズを 1MB，2MB，4MB としたときの提案方式による性能向上を示す．この図では，PB-LS モデルの性能を示している．縦軸は，それぞれの L2 キャッシュサイズにおける全コア並列実行時の性能を 1 とした時の相対性能を示している．すべてのプログラムにおいて L2 キャッシュサイズの増加に伴い提案方式による性能向上が低くなっている．これは，従来実行方式の主記憶メモリアクセスによるストール時間が少なく (k_N が小さく) なり，ヘルパーコアによるメモリ性能向上の効果が得られにくくなったためと考えられる．

5. おわりに

本稿では，ヘルパースレッドの新しい実行方式について提案とその評価および考察を行った．その結果，すべてのコアで並列プログラムを実行する従来の実行方式と比較して，提案方式は最大で 47%の性能向上を得ることができた．

今後は，ヘルパースレッドを実行するコア数を決定する機構を考える．また，実験結果の詳細な解析および考察を行う．具体的には，ベンチマークプログラムの変更，メモリ関連のパラメータの変更，ハードウェアプリフェッチャと組み合わせた場合の提案方式の効果の検証について行う．

謝辞　日頃から御討論頂いております九州大学安浦・村上・松永・井上研究室ならびにシステム LSI 研究センターの諸氏に感謝します．また，本研究は主に九州大学情報基盤研究開発センターの研究用計算機システムを利用しました．

文　献

[1] M. A. Suleman, M. K. Qureshi and Y. N. Patt: “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps”, SIGARCH Comput. Archit. News, **36**, 1, pp. 277–286 (2008).

[2] M. K. Qureshi and Y. N. Patt: “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches”, IEEE MICRO, **0**, pp. 423–432 (2006).

[3] J. A. Brown, G. C. Hong Wang, P. H. Wang and J. P. Shen: “Speculative precomputation on chip multiprocessors”, Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation (2001).

[4] I. Ganusov and M. Burtscher: “Efficient emulation of hardware prefetchers via event-driven helper threading”, Proceedings of the 15th international conference on Parallel architectures and compilation techniques, pp. 144–153 (2006).

[5] J. L. Baer and T. F. Chen: “Effective Hardware-Based Data Prefetching for High-Performance Processors”, IEEE Trans. Comput., **44**, 5, pp. 609–623 (1995).

[6] K. J. Nesbit and J. E. Smith: “Data Cache Prefetching Using a Global History Buffer”, IEEE Micro, **25**, 1, pp. 90–97 (2005).

[7] K. J. Nesbit, A. S. Dhodapkar and J. E. Smith: “AC/DC: An Adaptive Data Cache Prefetcher”, Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 135–145 (2004).

[8] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi and S. K. Reinhardt: “The M5 Simulator: Modeling Networked Systems”, IEEE Micro, **26**, 4, pp. 52–60 (2006).

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta: “The SPLASH-2 programs: characterization and methodological considerations”, Proceedings of the 22nd annual international symposium on Computer architecture, pp. 24–36 (1995).