Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

適応的ヘルパースレッド実行に基づくマルチコア向け演算/メモリ性能バランシング

今里賢一[†1]　福本尚人[†1]
井上弘士[†1]　村上和彰[†1]

マルチコア・プロセッサでは，複数コアを利用したオンチップ・スレッドレベル並列処理により，高い演算性能を達成できる．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．現在我々は，この問題を解決する新しいマルチコア実行方式として「演算/メモリ性能バランシング」を提案している[1)]．本方式では，従来のマルチコア実行とは異なり，並列実行可能部分において幾つかのコアを用い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．本稿では，文献 1) を改良し，プログラム実行中に演算/メモリ性能バランスを調整可能な（ヘルパースレッド実行コア数を決定可能な）新しい動的最適化実行法を提案する．提案方式の性能を評価した結果，すべてのプロセッサコアで並列プログラムを実行する方式と比較して，最大で 66%の性能向上を得ることができた．

Performance Balancing based on Adaptive Helper-Threads for Multi-Core Executions

Kenichi Imazato,[†1] Naoto Fukumoto,[†1] Koji Inoue[†1]
and Kazuaki Muraka[†1]

Conventional multi-core processors 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 66% speed up in the best case compared with a conventional parallel execution.

1. はじめに

現在，複数のプロセッサコアを 1 チップに搭載するマルチコア・プロセッサ (以降，マルチコアと略す) が主流となっている．複数コアを利用したオンチップ・スレッドレベル並列処理により，高い演算性能を達成できるためである．また，順調に成長を続ける半導体微細化技術を背景に，搭載されるコア数は年々増加傾向にある．例えば，米国インテル社の Nehalem は 8 個，米国 Sun Microsystems 社の Rock は 16 個のコアを集積する[2)3)]．

マルチコアの理論ピーク性能は搭載するコア数に比例して向上する．しかしながら，実際には様々な性能阻害要因により実効性能は必ずしも高くなるとは限らない．その主な理由の一つとして，メモリウォール問題の深刻化が挙げられる．コア数の増加により演算性能は高くなる反面，I/O ピンボトルネックによるオフチップ・メモリバンド幅の制限や主記憶アクセスレイテンシの増大により，プロセッサ－主記憶間の性能差がよりいっそう増加する．

現在我々は，この問題を解決する新しいマルチコア実行方式として「演算/メモリ性能バランシング」を提案している[1)]．本方式では，従来のマルチコア実行とは異なり，並列実行可能部分において幾つかのコアを用い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．そして，メモリ性能がボトルネックとなる場合には，並列プログラム実行用コア数を削減する一方，メモリ性能向上を目的としたヘルパースレッド実行用コア数を増加させる．すなわち，プログラム実行におけるスレッドレベル並列性を敢えて制限し，メモリ性能を改善する．このように，演算性能とメモリ性能のバランシングを行うことでマルチコア性能を向上する．文献 1) では，プログラム実行においてヘルパースレッド実行コア数は固定であり，かつ，その値は既知であると仮定し評価を行った．これに対し，本稿では，プログラム実行中に演算/メモリ性能バランスを調整可能な（ヘルパースレッド実行コア数を決定可能な）新しい動的最適化実行法を提案する．また，Linux カーネルへの実装を行い，ベンチマーク・プログラムを用いた定量的評価を行う．

本稿の構成は以下の通りである．第 2 節でマルチコアの問題点を整理する．第 3 節では提案方式とアーキテクチャ・サポートについて説明し，第 4 節でベンチマークプログラムを用いた定量的評価を行う．第 5 節で関連研究との違いを明確にし，最後に第 5 節でまとめる．

†1 九州大学大学院 システム情報科学府/科学研究院
Graduate school / Faculty of Information Science and Electrical Engineering, Kyushu University

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

2. マルチコア実行における問題点

マルチコアでの並列実行においては，実行コア数（並列プログラムを実行するために使用するコアの数）に比例した性能向上を実現できない場合が多くある．図 1 は実行コア数と性能向上の関係を示している．なお，実験環境の詳細は第 5.2 節を参照されたい．横軸は実行コア数，縦軸は L2 キャッシュサイズ 1MB を有するシングルコアプロセッサ（つまり，実行コア数は 1）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 と仮定した場合，“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の結果を示す．図 1(d) の *Barnes* に関しては，実行コア数に対してほぼ比例した性能向上を得る．しかしながら，図 1(a)(b)(c) に示すように，*Cholesky*，*Ocean*，*Raytrace* といったプログラムでは，期待する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．

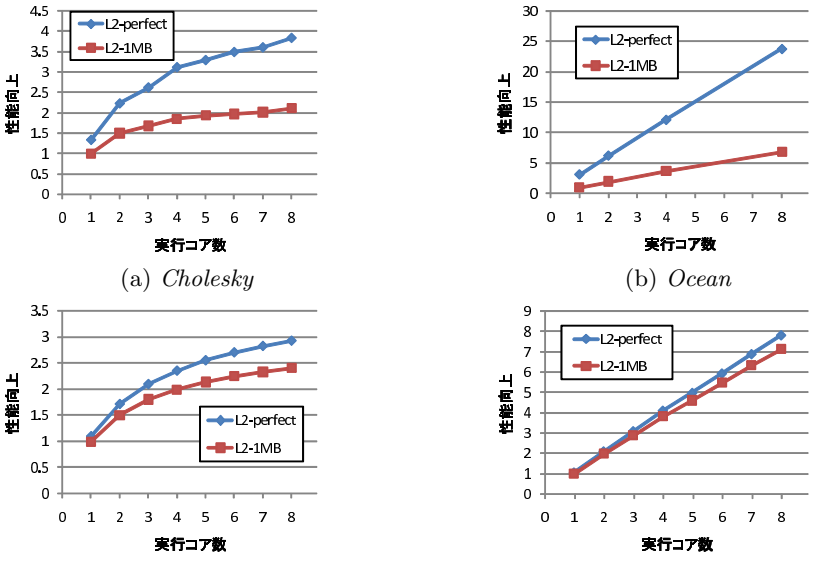
- 低いスレッドレベル並列性：図 1(a)(c) に示す *Cholesky* と *Raytrace* では，理想的なメモリシステムを想定した L2-perfect においても，実行コア数が 4 以上になると十分な性能向上を実現できていない．また，実行コア数の増加に伴い，その性能改善率が徐々に低下している．これは，並列化不可能な処理部分が顕在化したものと考えられる．
- メモリウォール問題：図 1(a)(b) に示す *Cholesky* や *Ocean* において，L2-1MB モデルではコア数の増加に伴う性能向上が低い．これに対し，L2-perfect では極めて高い性能となっている．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．

このような性能向上阻害要因において，本稿では後者（つまり，メモリウォール問題の改善）に焦点を当てる．

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

マルチコア実行において，より高い性能向上を達成するためには，演算性能のみならずメモリ性能の改善が重要となる．そこで我々は，マルチコア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新しいマルチコア実行方式を提案している[1)]．従来の単純な並列処理では，チップに搭載された全てのコアをスレッド実行に活用する．これに対し提案方式では，一部のプロセッサ・コアをメモリ性能向上用に利用する．つまり，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，マルチコア全体の性能向上を目指す．本方式では，使用目的に応じて各コアを以下のように定義する．



(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

- **メインコア**：並列プログラム（メインスレッド）を実行するコア．従来のマルチコア実行においては，全てのコアがメインコアとして動作する．
- **ヘルパーコア**：メモリ性能の向上を目的とし，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．メインコアが将来参照するであろうメモリアドレスを予測し，プリフェッチを発行する．これにより，メインコアでの L2 キャッシュミス率が減少する．

なお，本方式では，共有 L2 キャッシュを搭載したマルチコア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はメインコアから参照可能となる．

図 2 で示すように，演算/メモリ性能バランシングでは，アプリケーション・プログラムが要求するメモリ性能に応じてヘルパーコアの数を変更する．すなわち，より高い演算性

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

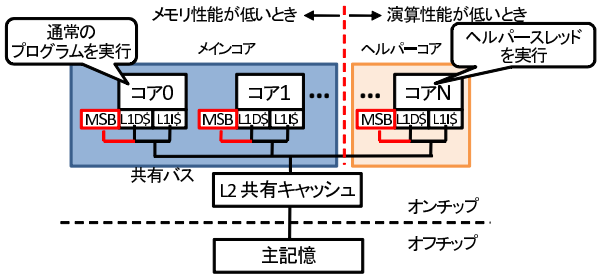


図 2 提案方式の全体図

能が必要な場合はメインコア数を，一方，メモリ性能が必要な場合はヘルパーコア数を増加する．例えば，図 1(a) の *Cholesky* では，実行コア数を 6 から 8 に増加しても得られる性能向上は高々6%程度である．これに対し，演算/メモリ性能バランシングでは，例えば 2 個のコアをヘルパーコアとして使用する（つまり，メインコア数は 8 から 6 に減少する）．ヘルパースレッド実行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ることができれば，マルチコア性能を向上できる．

3.2 アーキテクチャ・サポート

前節で説明したように，ヘルパーコアはメインコアでのプログラム実行をサポートするためにプリフェッチを発行する．すなわち，ヘルパーコアは，ハードウェアプリフェッチャの動きをソフトウェアで模倣することによりプリフェッチを行う．したがって，ヘルパーコアは，メインコアの L2 キャッシュミス情報を取得し，かつ，それに基づき将来の参照アドレスを予測しなければならない．しかしながら，従来のマルチコア構成では他コアが発生した L2 キャッシュミス情報を伝達する手段を備えていない．一方，アドレス予測に関しては，キャッシュミス情報から予測可能である[4]．

そこで我々は，メインコアによるキャッシュミス情報を共有するためのアーキテクチャ・サポートとして図 2 に示す “**Miss Status Buffer(以降，MSB と略す)**” を導入する．に示すように，MSB はキャッシュミス情報を格納する小容量の FIFO バッファであり，各コアに搭載される．MSB の各エントリには，L2 キャッシュミスを発生したメモリ参照アドレス，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は一意に決定される．したがって，各ヘルパーコアの MSB は担当するメインコアのキャッシュミス情報のみを格納する（具体的には，ヘルパースレッド割当て時にキャッシュミス情報取得に関するマスクレジスタの値が設定される）．ヘルパーコアは MSB に格納さ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 ヘルパースレッドの動作

ヘルパースレッドは図 3 に示す疑似コードとして実装される．ここで，2 行目では MSB からキャッシュミス情報を読出している．もし，MSB に有効なエントリが登録されていればそれを読出し，アドレス予測を実施する．一方，MSB が空の場合には，新たなキャッシュミス情報が格納されるまでストールする．3 行目ではキャッシュミス情報を引数として predict 関数を呼出し，メモリアクセスアドレスの予測を行う．predict 関数内では，ハードウェアプリフェッチ手法の動作を模倣する．そして 4 行目では予測されたメモリ参照アドレスに対して，プリフェッチ命令を実行する．

MSB より得たキャッシュミス情報を用いてメインコアのキャッシュミス・アドレスを予測するアルゴリズムとして，本稿ではストライド・プリフェッチ[5] を用いる．ストライド・プリフェッチでは，連続したメモリ参照アドレスの差分 (以降，ストライド値と呼ぶ) が一定であるメモリアクセスパタンを検出し，プリフェッチを行う．現在のキャッシュミス・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発行する．ただし，d はプリフェッチ・ディスタンスと呼ばれる値であり，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．本稿で用いた手法では命令別にキャッシュミス・アドレスのストライド値を計算する．したがって，キャッシュミスを引き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ソフトウェア・テーブルが必要となる．

3.4 動的ヘルパーコア数決定法

本研究が対象とする並列プログラムは，図 4 のような逐次実行部と並列実行部が交互に出現する．ここで，逐次実行部とは 1 スレッドでしか実行できない部分，並列実行部とは任意のスレッド数で実行可能な部分である．提案方式では，並列実行部のそれぞれにおいて適切なヘルパーコア数を決定する．一方，逐次部分に関しては常にヘルパーコア数は 1 とする[⋆1]．

⋆1 現在の実装では，ヘルパーコアはコア単位でキャッシュミス情報の分担を行うためである．なお，逐次実行部分においても複数コアをヘルパーコアとして使用し，それぞれ特性の異なるプリフェッチアルゴリズムを実行する

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

```
1.  while (true) {
2.      miss_info = msb.addr;
3.      pref = predict(miss_info);
4.      prefetch(pref);
5.  }
```

図 3 ヘルパースレッドの擬似コード

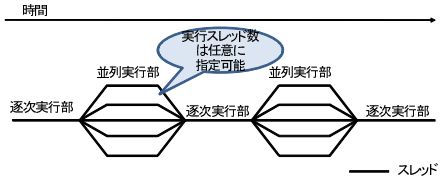


図 4 対象とする並列プログラム

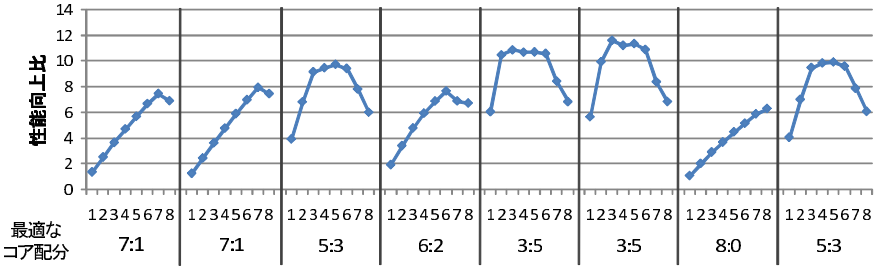


図 5 *UA* の各並列実行部における最適なコア配分 (シングルスレッド実行時の全実行時間に占める割合の多いものから 8 つ)

一般に，メモリ参照の振舞いはプログラム実行の経過と共に変動する．したがって，それに伴いマルチコア性能を最大化する最適なコア配分も変わる．NAS Parallel ベンチマークセットの *UA* に関して，各並列実行部のコア配分と性能向上比の関係を図 5 に示す．ここでは，シングルスレッド実行時に全実行時間に占める割合の大きい並列実行部から 8 個を示している．また，並列実行部は多くの場合で繰返し実行されるため平均値のみを示している．図 5 より，プログラム実行中に最適なコア配分は変わっていることが分かる．したがって，動的なコア配分の変更により高い性能向上が得られると期待できる．

提案方式では，各並列実行部が繰返し実行される場合が多いことに着目し，**各並列実行部毎に最適なコア配分を予測**する．具体的には，各並列実行部の CPI(Cycles Per Instruction) 最小化を目的とした探索を実施する．ここで，CPI はすべてのメインコアの CPI(すべてのメインコアの実行クロックサイクルの合計/すべてのメインコアの命令数の合計) とする．本コア配分決定法では，各並列実行部の最初の数回の実行をサンプリング期間とし，複数のコア配分における CPI を計測する．そして，得られた結果よりコア配分を決める．本方式は，各並列実行部の繰り返し実行において同様のメモリ参照の振舞いが観測される場合には高い精度で適切なヘルパーコア数を決定できる．以下，2 種類のコア配分決定を示す．

- **全探索法**：すべてのコア配分について実行し各コア配分における CPI を測定する．その結果から一番 CPI の低かったコア配分を採用する．
- **ヒルクライミング法**：すべてのコアがメインコアであるコア配分から，ひとつひとつメインコア数を減らしていき (ヘルパーコア数を増やしていき) つつ CPI を測定する．測定した CPI がひとつ前に測定した CPI よりも大きかった場合，サンプリングを終了し，ひとつ前のコア配分を採用する．全探索法よりも最適なコア配分を決定できる精度は落ちると考えられるが，サンプリング回数を短くすることができる．

3.5 Linux カーネルへの実装

プログラム実行中のコア割当てを可能にするため，提案方式を Linux カーネルへ実装した．変更を行ったのは以下の 2 点である．

- **アイドルスレッドをヘルパースレッドに変更**：Linux カーネルでは，実行すべきスレッドが存在しないプロセッサコアではアイドルスレッドが実行される．アイドルスレッドをヘルパースレッドで置換えることにより，常にメインコア数+ヘルパーコア数=搭載コア数とすることができる．これにより，並列プログラムにおいて実行スレッド数が変更されると，それに伴い自動的にヘルパースレッド数が変更される．
- **ヘルパーコア数変更時の MSB の再設定**：ヘルパーコア数が変更された場合，各ヘルパーコアが担当すべきメインコアも同時に変更される．例えば，ヘルパーコア数が 1 の場合，当該ヘルパーコアは全てのメインコアのプリフェッチを担当する．これに対し，ヘルパーコア数が 2 に増加した際には，ヘルパーコア当りの担当すべきメインコアは半分になる．したがって，コア配分を変更する場合には，全てのヘルパーコアに関して，格納すべきキャッシュミス情報を決定するマスクレジスタ値を設定する必要がある．この作業は，コア配分変更後にヘルパーコアの負荷を均等化するために必要となる．

4. 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

ことも考えられるが，それは今後の課題である．

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

表 1 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	バス幅：64B, 動作周波数：CPU と同じ
L2-主記憶間バス	バス幅：16B, 動作周波数：CPU の 1/4
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

う．評価対象モデルは以下の通りである．

- **BASE**：全てのコアで並列プログラムを実行する従来モデル．つまり，全てのコアはメインコアとして動作する．また，ヘルパースレッドを実装していない Linux カーネルを用いて実行する．
- **PB-STATIC-OPT**：全ての並列実行部を同一のコア配分で実行する提案モデル．静的な演算/メモリ・バランシングを行う．ただし，性能が最大となるコア配分は既知と仮定する．具体的には，コア配分に関する全ての組合せにおいて，評価時と同一入力データを用いた事前実行を行う．これにより，最も実行時間の短いコア配分 (以降，最適なコア配分と記す) を求めた．
- **PB-DYN-OPT**：各並列実行部に関して最適なコア配分で実行する提案モデル．動的な演算/メモリ・バランシングを行う．ただし，各並列実行部に関して性能が最大となるコア配分は既知と仮定する．
- **PB-DYN-EXH**：各並列実行部に関して最適なコア配分で実行する提案モデル．動的な演算/メモリ・バランシングを行う．各並列実行部に関するコア配分は全探索法を用いる．
- **PB-DYN-HILL**：各並列実行部に関して最適なコア配分で実行する提案モデル．動的な演算/メモリ・バランシングを行う．各並列実行部に関するコア配分はヒルクライミング法を用いる．

マルチプロセッサシミュレータとしては，ミシガン大学で開発された M5[6] をベースとし，本研究で提案する MSB を実装した．評価において前提とする各種パラメータ値を表 1 に示す．Linux カーネル 2.6.22 に対し，ストライド・プリフェッチに基づくヘルパースレッドを実装した．また，プリフェッチ・ディスタンスは 8，プリフェッチ用テーブルのエントリ数は担当するメインコア当り 1024 とした．

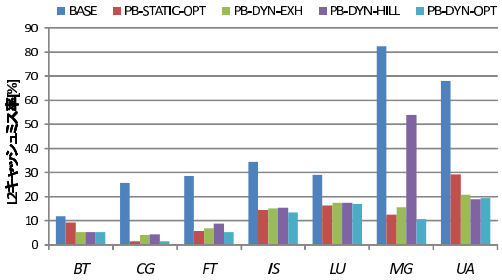


図 6 L2 キャッシュミス率

一方，ベンチマークプログラムには並列計算ベンチマークである NAS Parallel Benchmark 3.3(OpenMP 版)[7] から 7 個を選択した．問題クラスは Class W を用いた．コンパイラは GCC4.2 を用い，最適化オプション-O2 でコンパイルした．ただし，GCC の OpenMP の実装である GOMP を改造し，実行スレッド数が減る場合にはスレッドを消滅させずに次の並列実行部において実行スレッド数が増加する場合に再利用できるようにした．これは，プログラム実行中に頻繁に実行スレッド数を変更することによる性能低下を緩和するためである．

4.2 評価結果

4.2.1 L2 キャッシュミス率

各評価対象モデルにおける L2 キャッシュミス率を図 6 に示す．縦軸は L2 キャッシュミス率，横軸はベンチマーク・プログラムである．また，以下の式で定義されるプリフェッチ・カバレッジならびに精度を図 7 および図 8 に示す．

$$prefetch\_coverage = \frac{HitsHTPFB_{L2}}{Misses_{L2} + HitsHTPFB_{L2}} \quad (1)$$

$$prefetch\_accuracy = \frac{HitsHTPFB_{L2}}{PFMisses_{L2}} \quad (2)$$

ここで，$HitsHTPFB_{L2}$ はヘルパースレッドがプリフェッチしたキャッシュブロックに対

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

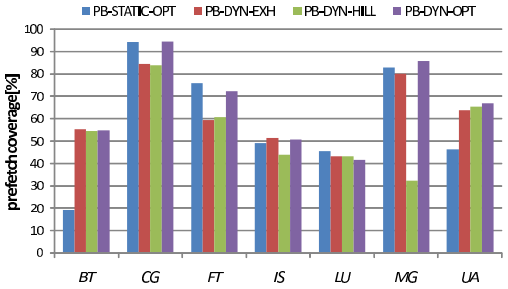


図 7 prefetch coverage

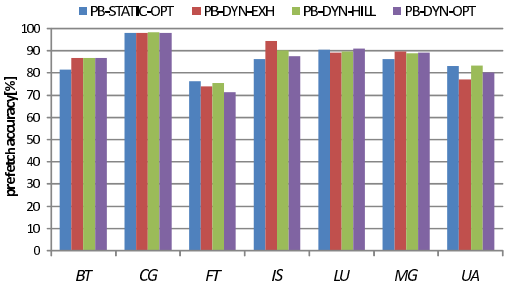


図 8 prefetch accuracy

してメインスレッドからのメモリ参照がヒットした回数，$Misses_{L2}$L2 キャッシュミス回数，$PFMisses_{L2}$ はヘルパースレッドによるプリフェッチ時に当該データを主記憶からロードした回数を表す．

図 6 より，多くのベンチマークに関して，ヘルパーコアを導入することで L2 キャッシュミス率を大幅に削減できていることが分かる．特に，*CG* に関しては，PB-STATIC-OPT, PB-DYN-EXH，PB-DYN-HILL，ならびに，PB-DYN-OPT の全てに関して，高いカバレッジと精度を実現している．これにより，従来実行方式では 25%程度であった L2 キャッシュミス率を 5%以下に削減している．

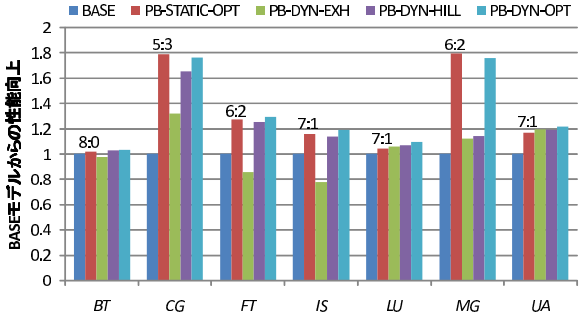


図 9 提案方式による性能向上

4.2.2 マルチコア性能 (PB-STATIC-OPT モデル)

各評価対象モデルに関する性能を図 9 に示す．縦軸は BASE モデルの性能を 1 としたときの相対性能を示している．各バーは左から，BASE モデル，PB-STATIC-OPT モデル，PB-DYN-EXH モデル，PB-DYN-HILL モデル，PB-DYN-OPT モデルを表す．また，PB-STATIC-OPT モデルのバー上に記した数字はコア配分 (メインコア数:ヘルパーコア数) を示す．

本節では，PB-STATIC-OPT モデルに関して議論する．図 9 から分かるように，殆どのベンチマークプログラムにおいて高い性能向上を達成している．*CG*，*MG*，*IS*，*UA* においては，ヘルパースレッドによる L2 キャッシュミス率が大きく改善されたため，この性能向上がメインスレッド数低下による性能低下を大きく上回り性能が良くなっている．特に *CG*, *MG* では約 80% の性能向上となっている．一方，以下 2 つのベンチマークに関しては大きな性能向上を実現できていない．

- *BT*：図 6 で示すように，BASE において L2 キャッシュミス率が低く，ヘルパースレッドを実行する必要がない程度に十分なメモリ性能があるためである．実際，PB-STATIC-OPT モデルでもヘルパーコア数はゼロである[⋆1]．
- *LU*：図 6 より L2 キャッシュミス率は比較的改善することができている．しかしながら，メインコア数の減少による性能低下が顕著に表れたためと考える．

⋆1 ヘルパーコア数がゼロであるに関わらず BASE に対して若干であるが性能が向上している．これは，並列実行部において同期待ちとなったコアにヘルパースレッドが割当てられたためと考える．

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

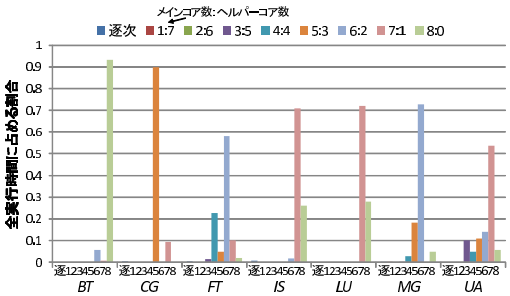


図 10 PB-DYN-OPT における各コア配分の全実行時間に占める割合

4.2.3 マルチコア性能 (PB-DYN-OPT モデル)

本節では，PB-DYN-OPT モデルの性能に関して議論する．図 9 の結果より，*CG* や *MG* に関しては 80%以上の性能向上を実現している．しかしながら，各並列実行部毎に最適なコア配分を設定しているにも関わらず，PB-STATIC-OPT モデルと比較して大きな差は見られない．この理由を解析するため，PB-DYN-OPT モデルにおいて，実際に選択された各コア配分の実行時間がプログラムの全実行時間に占める割合を調査した．その結果を図 10 に示す．各ベンチマークにおいて，横軸に示す値（1〜8）はメインコア数を表す．つまり，「6」に対応するバーは，メインコア数が 6，ヘルパーコア数が 2 の配分で実行された場合に相当する．この図より，ほぼすべてのベンチマークにおいて，ある一つのコア配分が多く選択されていることが分かる．そのため，PB-DYN-OPT モデルの性能は PB-STATIC-OPT モデルの性能とほとんど変わらなかったと考えられる．

また，一部のプログラムでは，PB-DYN-OPT モデルは PB-STATIC-OPT よりも低い性能となっている．この理由は，プログラム実行中に頻繁にヘルパーコア数を切り替えることによる悪影響 (L1 キャッシュのヒット率が低下など) が現れたためと考えられる．したがって，ヘルパーコア数の変更を適切に抑制する必要がある．これと同様の問題は文献[8] においても報告されている．

4.2.4 マルチコア性能 (PB-DYN-EXH/PB-DYN-HILL モデル)

本節では，現実的な実装法である PB-DYN-EXH モデルと PB-DYN-HILL モデルの性能に関して議論する．PB-DYN-HILL モデルでは，*CG* で 66%，*FT* ならびに *UA* で 20%以上の性能向上を達成している．図 11 は，どの程度適切なコア配分で実行されたか (すなわち，コア配分予測精度の効果) を表している．横軸の数字 (1〜8) は適切なコア配分のランキング，縦軸は各ランキングでの実行時間が並列実行部の全実行時間に占める割合を示す．例えば，ある並列実行部においてヘルパーコア数が 2(コア配分は 6:2) とする．この時，当該並列実行部における最適なコア配分が 6:2 であった場合には，その実行時間は横軸の数字 1 の部分に累積される．また，この図において，赤い部分はコア配分決定のためのサンプリング実行時間を，また，青い部分はそれ以外の実行時間を表す．図 11 より，多くのプログラムにおいて，横軸の数字が 1〜2 に対応する実行時間が支配的となっている．これは，ヒルクライミング法により比較的正しく最適なコア配分を予測できていることを意味する．

一方，PB-DYN-EXH モデルでは，*BT*，*FT*，*IS* において BASE モデルより低い性能となっている．この原因を解析するため，図 11 と同様，全探索法によってどの程度適切なコア配分で実行されたかを調査した．その結果を図 12 に示す．この図より，全探索法においてもヒルクライミング法と同程度の精度で適切なコア配分を選択していることが分かる．しかしながら，コア配分の全組合せに関して CPI を計測する必要があるため，サンプリング期間が長くなる (つまり，図 12 における赤いバーが多くなる)．これにより，性能オーバヘッドが発生したものと考える．これに加え，並列実行部の繰返し実行回数が少ないためにはサンプリング期間における性能オーバヘッドが顕著となる．表 2 に，頻繁に実行される上位 6 個の並列実行部の繰返し実行回数を示す．特に *FT* では，並列実行部の実行回数が 8 回を超えるものがなく，全探索を実施する PB-DYN-EXH ではサンプリング期間にプログラム実行が終了してしまう．

次に，PB-DYN-HILL モデルと PB-DYN-OPT モデルの性能差に着目する．*MG* を除く全てのプログラムにおいて，現実モデルである PB-DYN-HILL は理想モデルである PB-DYN-OPT と同程度の性能向上を達成している．一方，*MG* に関しては大きな性能差が存在する．これは，図 11 で示すように，最適なコア配分予測を正しく行えていないためである．*MG* では，並列実行部が繰返し実行される際，各実行での処理量が大幅に変化する．そのため，サンプリング期間にて最適と判断したコア配分の結果を用いても，後続の繰返し実行では異なる配分が最適となる．このような場合においては，本提案手法では適切に対応することは難しい．

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

Vol.2009-ARC-183 No.16
Vol.2009-OS-111 No.16
2009/4/23

表 2 並列実行部の実行回数

プログラム	1	2	3	4	5	6
BT	201(0.307)	201(0.307)	201(0.304)	202(0.077)	201(0.004)	1(0.001)
CG	16(0.952)	1(0.047)	15(0.0001)	15(8e-05)	1(7e-06)	1(5e-06)
FT	8(0.274)	8(0.266)	8(0.208)	6(0.146)	2(0.058)	1(0.026)
IS	11(0.754)	1(0.188)	1(0.028)	1(0.014)	1(0.013)	
LU	301(0.675)	303(0.322)	2(0.001)	1(0.001)	4(0.0003)	1(0.0002)
MG	37(0.502)	35(0.185)	30(0.070)	30(0.0683)	34(0.045)	4(0.039)
UA	1131(0.291)	1010(0.151)	4040(0.104)	1232(0.093)	909(0.081)	1010(0.072)

※ 括弧の中はシングルスレッド実行時に全実行時間に占める割合を示す

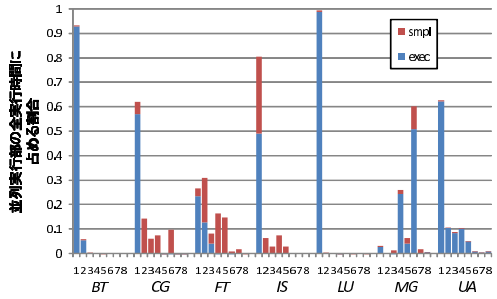


図 11 何番目に最適なコア配分がどれくらい実行されたのか (PB-DYN-HILL)

方式は最大で約 66%の性能向上を得ることができた．

今後は，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，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．なお，本研究は一部，半導体理工学研究センター (STARC) との共同研究による．

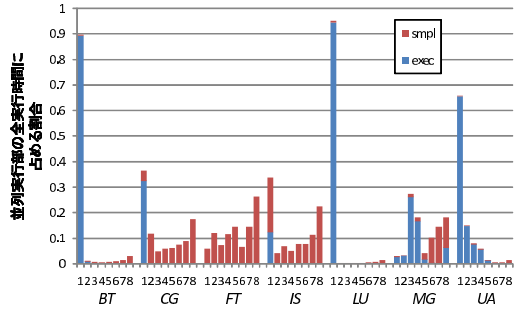


図 12 何番目に最適なコア配分がどれくらい実行されたのか (PB-DYN-EXH)

参 考 文 献

1) 今里, 福本, 井上, 村上：“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”, 第 108 巻 of ICD2008-31, pp. 75–80 (2008).
2) e.StefanRusu: “A 45nm 8-core enterprise xeon processor”, pp. 56–57 (2009).
3) M. Tremblay and S. Chaudhry: “A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparca processor”, pp. 82–83 (2008).
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) 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).
7) H. Jin, M. Frumkin and J. Yan: “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance”, NAS Technical Report NAS-99-011 (1999).
8) e. Matthew Curtis-Maury: “Identifying energy-efficient concurrency levels using machine learning”, International Workshop on Green Computing (GreenCom’07) Austin, TX (2007).