ヘルパースレッドを用いた
マルチスレッディングプロセッサのための
高速化技術研究

１７３００２３

平成１７年度～平成１８年度科学研究費補助金

（基盤研究（Ｂ））研究成果報告書

平成１９年５月

研究代表者　山名早人

早稲田大学理工学術院教授

<はしがき>

近年，マルチコア CPU が相次いで登場している．本研究では，こうしたマルチコア CPU 上でのマルチスレッディングを用いた高速化技術について研究を行った．

2005 年度は，本分野でのサーベイ，アルゴリズム検討，アプリケーション側からの検討を行った．アルゴリズム検討では，分岐予測の面からパイプライン中の空きスロットを削減する方法を検討した．さらに，近年のオンチップマルチプロセッサでは，L2 を共有するものが増加しており，L2 の効率的な制御方法についても検討を行った．具体的には，L2 内のどこに必要となるデータを配置するかというデータ配置最適化技術を提案し，SPECint95, SPECint2000 で平均 17%の IPC 向上を確認した.アプリケーションレベルからの検討では，今後その高速化が重要となってくると考えられる検索アプリケーションの動作特徴等を調査した．

2006 年度は，前年度の検討に基づき，ターゲットをディスクアクセスの最適化に特化し，オンチップマルチプロセッサ上でのキャッシュ最適化について研究を進めた．まず，DISK の先読みにヘルパースレッドを用いる例では，先読みスレッドで事前にデータの読み込み込む手法を提案し，gzip が最大で 39.2%性能向上することを確認した．次に，DISK キャッシュ自体をネットワーク接続された他の PC 上に置き，ヘルパースレッドによりディスクキャッシュを制御する高速化手法を提案した．DBT-3 ベンチマークテストによる評価では，最大 3.08 倍の性能向上を確認した．さらに，実アプリケーションとして，シェルスクリプト実行の高速化を実現した．シェルスクリプトは，旧来，小さなアプリケーションで多用されるものであるが，近年の CPU 高速化に伴い，業務処理に多用されるようになってきている．実際に大手小売店が利用している．本研究では，このようなシェルスクリプトの高速化を目指し，シェルスクリプトの自動並列化プログラムを提案した．そして，これらをヘルパースレッドで実行することにより，シェルスクリプト実行を 1.4～1.8 倍高速化することができた．なお，本研究成果は，USP 研究所（http://www.usp-lab.com/）において製品化を目指している．

研究組織

研究代表者：　山名早人（早稲田大学理工学術院教授）
研究分担者：　斎藤文子（早稲田大学理工学術院助手）

交付決定額（配分額）　　（金額単位：円）

	直接経費	間接経費	合計
平成１７年度	６，９００，０００	０	６，９００，０００
平成１８年度	７，９００，０００	０	７，９００，０００
総計	１４，８００，０００	０	１４，８００，０００

研究発表

（1）学会誌等

著者名	論文標題			
山名早人	検索エンジンから分析エンジンへ			
雑誌名		巻・号	発行年	ページ
人工知能学会誌		20・4	2005	471-478

（2）口頭発表

著者名	論文標題			
斎藤史子，山名早人	スレッドレベル投機的実行に関する考察			
雑誌名		巻・号	発行年	ページ
情報処理学会 第4回情報科学技術フォーラム論文集			2005	C-13

著者名	論文標題			
蛭田智則，増田渓介，山名早人	配線遅延を考慮したキャッシュメモリ高速化手法			
雑誌名		巻・号	発行年	ページ
情報処理学会研究会報告		2006・20	2006	19-24

著者名	論文標題			
上田高徳，平手勇宇，山名早人	ネットワーク上のマシンをディスクキャッシュに利用した場合の性能評価			
雑誌名		巻・号	発行年	ページ
電子情報通信学会 第18回データ工学ワークショップ論文集			2007	E7-9

著者名	論文標題			
杉田秀，深山辰徳，蛭田智則， 當仲寛哲，山名早人	マルチコアプロセッサ上におけるシェルスクリプト高速化手法			
雑誌名		巻・号	発行年	ページ
情報処理学会研究会報告		2007・17	2007	73-78

著者名	論文標題			
深山辰徳，杉田秀，蛭田智則， 山名早人	先読みスレッドを用いたDiskアクセスの高速化			
雑誌名		巻・号	発行年	ページ
情報処理学会研究会報告		2007・17	2007	233-238

（３）出版物

著　者　名	出　版　社				
山名早人監訳	オライリー・ジャパン				
書　名	発行年				総ページ数
Google Hacks 第2版― プロが使うテクニック&ツール100選	2	0	0	5	478

研究成果による工業所有権の出願・取得状況

・該当無

1．スレッドレベル投機的実行に関するサーベイ

1.1 はじめに

近年の命令レベル並列性の向上に伴い，さらなる並列性を引き出すためにスレッドレベル並列性が注目されている[2]．本章では，その中でもスレッドレベルの投機的実行に着目しサーベイを実施した．スレッドレベル投機的実行には，load 命令を対象とした提案[3, 5, 8]と分岐命令を対象とした提案[1, 4]がある．

load 命令を対象とした投機的実行と分岐命令を対象とした投機的実行は，同時に適用することが可能である． load 命令を対象とした投機的実行は，delinquent load 命令（キャッシュ・ミスしやすい load 命令）と依存関係のある命令列をヘルパースレッドとして先行実行することで，キャッシュミスを削減する．load 命令を対象とした手法は，ハードウェアを追加する実現方法[3, 5]とソフトウェアに変更を加える実現方法[8]が提案されている．

分岐命令を対象とした投機的実行は，難予測分岐を対象に分岐判定と依存関係のある命令列をヘルパースレッドとして先行実行することで，予測精度を改善する．分岐命令を対象とした手法は，ハードウェアを追加する実現方法[4]とハードウェアとソフトウェア両方に変更を加える実現方法[1]が提案されている．近年，メモリアクセスによる遅延が深刻化し，キャッシュ・ミス・ペナルティが増大している．また，load 命令を対象としたスレッドレベル投機的実行の方が盛んに行われている．これらのことから，本サーベイでは，load 命令を対象としたスレッドレベル投機的実行を中心にとり扱い，load 命令を対象とした代表的な研究とその動向についてまとめる．

1.2 スレッドレベル投機的実行に関する研究

load 命令を対象とした Simultaneous Multithreading (SMT) 環境におけるスレッドレベル投機的実行は，ハードウェアによる実現方法[3, 5]とソフトウェアによる実現方法[8]に大別できる．表 1 に，本サーベイで紹介する手法の比較を示す．

表 1：Speculative Computation

手法	Dynamic SP	DDMT	software 実装
スレッド生成	動的	静的	静的
load 命令遅延解決法	プリフェッチ	レジスタ値再利用	プリフェッチ
変更箇所	追加 HW (DLIT, SIT)	追加 HW (DDTC, IT)	ISA の拡張

1.2.1 ハードウェアによる実現

ハードウェアによる実現方法として，Tullsen らの提案する Dynamic Speculative Precomputation (Dynamic SP) [5] と Roth らの提案する Data-Driven Multithreading (DDMT) [3] が挙げられる．

1) Dynamic SP [5]

Tullsen らの提案する Dynamic SP (Dynamic Speculative Precomputation) では，ハードウェアが動的にスレッドを生成する点に特徴がある．

Dynamic SP の実現には，Delinquent Load Identification Table (DLIT)，Slice Information Table (SIT)，Retired Instruction Buffer (RIB) が必要になる．DLIT は，プログラムの実行中に delinquent load 命令 (PC: Program Counter) を記録する．RIB は，trace を記録し，DLIT によって判断された delinquent load 命令を対象とするヘルパースレッドに含まれる命令列を生成する．SIT は，実行可能なヘルパースレッドの候補 (trigger 命令 1 の PC) を登録し，trigger 命令が実行される場合に helper スレッドを起動する．ただし，Dynamic SP には，ヘルパースレッドがメインスレッドからしか生成されないという制限がある．

2) Data-Driven Multithreading [3]

Roth らの提案する DDMT (Data-Driven Multithreading) [3] は，ヘルパースレッドによる先行実行結果を格納したレジスタを利用する点に特徴がある．

DDMT は，Data-Driven Thread Cache (DDTC) と Integration Table (IT) を追加した SMT プロセッサである． DDTC は，ヘルパースレッドを構成する命令のための命令キャッシュである．ヘルパースレッドの命令キャッシュ (DDTC) をメインスレッドの命令キャッシュと別に準備することによって，メインスレッドの命令キャッシュは，ヘルパースレッド実行による影響を受けずに済む．IT は，ヘルパースレッドが ROB (Reorder Buffer) に登録されないため，ヘルパースレッドが利用中，もしくは，実行結果を保持するレジスタを解放しないように管理する．実行されるヘルパースレッドは，PTHSEL と呼ばれる条件式の集合 [7] によって，load 命令の遅延に応じて静的決定される．

近年，DDMT は，CMP (Chip Multiprocessor) を前提とした，性能向上だけではなく消費電力に注目した研究 [10] も行われている．

1.2.2 ソフトウェアによる実現

ソフトウェアによる実現方法 [8] では，プロファイルから得られるキャッシュミス情報やコンパイラの分析によって delinquent load 命令を特定し，ヘルパースレッドを生成する．スレッドを区別するために，ヘルパースレッド起動命令の追加など ISA (Instruction Set Architecture) に変更を加える必要がある．Software による実現では，helper スレッドがさらに helper スレッドを生成することもある．

1.3 スレッドレベル投機的実行の実機評価

1.2 で紹介した文献 [3, 5, 8, 10] では，シミュレーションによる評価しかされていない．

2004年頃からhelperスレッドを実機に実装した評価も行われるようになってきている．ここでは，hyperthreadingが実装されたintelのプロセッサへの適用についてまとめる．

1.3.1 Pentium 4[9]

Kimらは，2.66GHz Pentium4にヘルパースレッドを実装した．Kimらの評価では，SPECint2000 mcfにおいて，ヘルパースレッドは約50%のキャッシュミスに適用されたにもかかわらず，2.7%しか性能が向上しなかった．同期オーバヘッドの大きいWindows APIを利用せずに，スレッド同期処理用のハードウェア[12]を導入することで，8.5%性能向上する見込みがあることがわかっている．

1.3.2 Xeon[11]

我々は，2.4GHZ Xeonにおいてヘルパースレッドをこれまでに実装した．我々の評価では，SPECint2000 181.mcf(ref入力)において，SPECint2000 mcfは，Windows APIを使用したにもかかわらず，3.26%の性能向上を達成できた．

1.4 同期オーバヘッド削減の課題

SMT環境でのヘルパースレッド実装の際にはスレッド間の同期オーバヘッドが深刻な課題となる．データ依存を起因とするスレッド間の同期オーバヘッドを削減するだけでも，SPECint95, SPECint2000に含まれるプログラムで，平均31.76%処理性能が向上するという報告[6]がある．また，イタレーションごとの同期回数を削減するだけで，27.1%性能向上するという報告[11]もある．このように，同期オーバヘッドの与える影響は大きい．SMT環境でのヘルパースレッドによる性能向上を実現するには，同期オーバヘッド削減に関する問題を解決することが不可避であると考えられる．

1.5 まとめ

本章では，SMT環境におけるスレッドレベル投機的実行に関する文献調査を行った．既存のSMTプロセッサ(Intel Pentium4, Xeon)による評価の結果，スレッド間の同期オーバヘッドによる影響を受け，キャッシュミス削減の割に処理性能が向上しないことが判った．スレッド間のデータ通信や同期オーバヘッドを解決することで，30%以上の性能向上が期待できると考えられる．

一方で，スレッド制御オーバヘッドが大きいことも分かっており，ヘルパースレッドを用いた高速化を達成するには，ターゲットとなるプログラムを絞り込んだ最適化が必須であると考えられる．

参考文献

[1] Chappell, R. S., J. Stack, et al.: “Simultaneous Subordinate Multithreading

(SSMT)", Proc. of 26th ISCA (1999)

[2] Oplinger, J. T., D. L. Heine, M. S. Lam: "In Search of Speculative Thread-Level Parallelism", Proc. of 8th PACT (1999)

[3] Roth, A. and G. S. Sohi: "Speculative Data Driven Multithreading", Proc. of 7th HPCA (2001)

[4] Zilles, C. and G. Sohi: "Execution-based Prediction using Speculative Slices", Proc. of 28th ISCA (2001)

[5] Collins, J. D., D. M. Tullsen, et al.: "Dynamic Speculative Precomputaion", Proc. of MICRO-34 (2001)

[6] Steffan, J. G., C. B. Colohan, el al.: "Improving Value Communication for Thread-Level Speculation", Proc. of 6th HPCA (2002)

[7] Roth, A. and G. S. Sohi: "A Quantitative Framework for Automated Pre-Execution Thread Selection", Proc. of MICRO-35 (2002)

[8] Kim, D. and D. Yaung: "A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code", ACM Trans. on Computer Systems, Vol. 22, No. 3 (2004)

[9] Kim, D., S.- W. Liao, et al.: "Physical Experimentation with Prefetching Helper-Threaded Processors", Proc. of 2nd CGO (2004)

[10] Renau, J., K. Strauss, et al.: "Thread-Level Speculation on a CMP can be Energy Efficient", Proc. of 19th ISCA (2005)

[11] 本田，斎藤，山名："ハイパースレッディング環境における投機的スレッド間の同期手法"，情処研報(2004-ARC-161) (2005)

[12] Tullsen, T. M., J. L. Lo, et al.: "Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor", Proc. of 5th HPCA(1999)

2．配線遅延を考慮したキャッシュメモリ高速化手法

2.1 はじめに

近年，マイクロプロセッサにおいて著しい速度向上が成し遂げられてきた．その一方で，メモリ速度は，マイクロプロセッサほど向上しておらず，主記憶へのアクセスによる遅延がシステム全体のボトルネックとなっている[1]．そのため，従来のプロセッサでは，高速・小容量のキャッシュメモリをマイクロプロセッサとメインメモリの間に設け，メモリの一部をコピーしておくことによってこのボトルネックを緩和している．しかし，プロセスの微細化に伴うキャッシュメモリ量の増大により，キャッシュメモリにおける配線遅延が増大し，キャッシュメモリのアクセスレイテンシも増大すると考えられる．したがって，今後は，キャッシュメモリアクセスによる遅延が，システムのボトルネックになる可能性がある．

そこで，本章では，ヘルパースレッドを用いた高速化を実現するにあたり、必要不可欠と考えられるハードウェア上の問題を解決すべく，配線遅延を考慮したキャッシュメモリ高速化手法について提案する．提案手法はデバイスレベルではなく，プロセッサアーキテクチャレベルで配線遅延の緩和を試みている．

提案手法では，バンク化されたキャッシュにおいて，頻繁にアクセスされるラインを，キャッシュコントローラからの距離が近いバンクに移動し，配線遅延によるアクセスの遅延を削減する．具体的には，LRU 置換法を用い，キャッシュヒットしたラインのデータを，キャッシュコントローラからの距離が近いバンクに移動する．しかし，従来のキャッシュモデルでは，キャッシュ内でのデータの移動は不可能であるため，本手法ではキャッシュモデルとして D-NUCA[2] を用いる．D-NUCA は，提案手法と同様にプロセッサアーキテクチャレベルで配線遅延の緩和を試みた手法であり，スイッチを用いて，バンク間のデータを移動する．D-NUCA では，データの移動は，隣接バンクであるのに対し，本手法ではデータの配置が LRU 順になるようにデータを移動する．

提案手法の有効性を示すため，サイクル駆動のプロセッサシミュレータである SimpleScalar 3.0d を用いて本手法を実装し，SPEC95 CINT，SPEC2000 CINT を用いて評価を行った．

2.2 配線遅延とプロセス

配線遅延とは，配線抵抗や配線間容量によって生じる電気信号の遅延である．配線遅延 T は，式(1)で表され，配線抵抗 R と配線容量 C の積 RC に比例する．

$$T = 0.69R * C \quad \cdots \quad (1)$$

単位長の配線抵抗 R は，式(2)で表され，低効率に比例し，配線の幅・高さに反比例する．

$$R = \rho_{wire} \frac{1}{W_{wire} * H_{wire}} \quad \cdots \quad (2)$$

ここで，ρ_{wire} は配線の抵抗率を表す．また，W_{wire}・H_{wire} は配線の幅・高さを表す．配線の幅・高さはプロセスの微細化(ここでのプロセスとは，シリコンウェハから半導体チップを製造する過程を意味する)と同じ割合で減少していくと仮定できる．したがって，単位長の配線抵抗 R はプロセスの微細化の 2 乗に反比例して増加する．配線容量 C は，プロセスの微細化に直接影響を受けないが，配線間距離が縮小するため，影響を受ける配線数が増加するため，配線間容量は増加していくと考えられている[3]．

配線遅延の増加は，データ転送レイテンシの増加，クロック同期の困難化等，LSI に対して大きな影響を与える．したがって，プロセスの微細化が進んだ現在では，配線遅延を十分に考慮する必要があると考えられる．

2.3 キャッシュメモリ

2.3.1 プロセスとメモリ量

プロセスの微細化に伴い，マイクロプロセッサに搭載されるキャッシュメモリ量は増大し続けている(表 1)．表 1 は，キャッシュ設計ツールである Cacti[5]を用いて，将来の 2 次キャッシュへのアクセスレイテンシの増加を予想したものである．プロセッサの動作周波数はプロセスが次の段階へと微細化される毎に 2 倍になると仮定し，キャッシュ容量は，面積がほぼ同じになるように設定している．なお，近年になってプロセッサの周波数の伸びは緩やかになっているが，Intel 社は 65nm プロセスにおいて 9GHz で動作する演算器の開発に成功している[6]．

表 1　プロセスとアクセスレイテンシ

年	プロセス	周波数	L2キャッシュメモリ量	アクセスレイテンシ
2001	130nm	1.5GHz	1MB	4
2004	90nm	3.0GHz	2MB	9
2007	65nm	6.0GHz	4MB	20
2010	45nm	12.0GHz	8MB	64
2013	32nm	24.0GHz	16MB	104

近年のマイクロプロセッサのキャッシュメモリ量は 2MB 程度であるが，次世代のプロセッサでは 4MB を超えると推測される．しかし，表 1 に示すように，キャッシュメモリ量が増大するにつれ，配線遅延によりキャッシュメモリへのアクセスレイテンシも増大する．

特に 2010 年以降に着目すると，キャッシュメモリへのアクセスアクセスレイテンシは 45nm プロセスでは 50 サイクル，32nm プロセスでは，100 サイクルを超える．

2.2 節で述べたように，単位長の配線抵抗は，プロセスの 2 乗に反比例して増加するため，仮に，プロセッサにおいてキャッシュが占める面積は同じであっても，配線抵抗が増加し，キャッシュメモリへのアクセスレイテンシが増大する．表 1 では，デバイス技術の進歩を考慮しておらず，今後，動作周波数がどこまで伸びるかは不透明であるため，アクセスレイテンシが表 1 の通りに増加するとは限らないが，十分に考慮する必要がある．

今後は，プロセスの微細化により，キャッシュメモリアクセスにおける遅延もシステムの速度を低下させる要因になると考えられる．したがって，キャッシュメモリアクセスがシステムのボトルネックとならないよう，キャッシュメモリアクセスによる遅延を緩和していく必要がある．そこで，以下では，デバイスレベルではなく，プロセッサアーキテクチャレベルから，配線遅延によるキャッシュメモリアクセスレイテンシ増加の緩和を試みる．

2.3.2 関連研究

従来のキャッシュモデルでは，複数のパイプラインから同時にキャッシュアクセスがあった場合，ポート数による制限による待ちが発生し，パフォーマンスの低下の要因となっている．また，タグ検索回路，データの出力回路は，最も時間がかかるラインアクセスにタイミングを合わせている．したがって，キャッシュコントローラからの距離が短く，配線遅延の影響が小さいラインへのアクセスも，キャッシュコントローラからの距離が遠く，配線遅延の影響が大きいラインへのアクセスにタイミングを合わせる必要がある．そのため，プロセスの微細化が進むつれ，キャッシュメモリアクセスによる遅延も増大する．

この問題を解決する手法として，キャッシュのバンク化が挙げられる (図 1b)

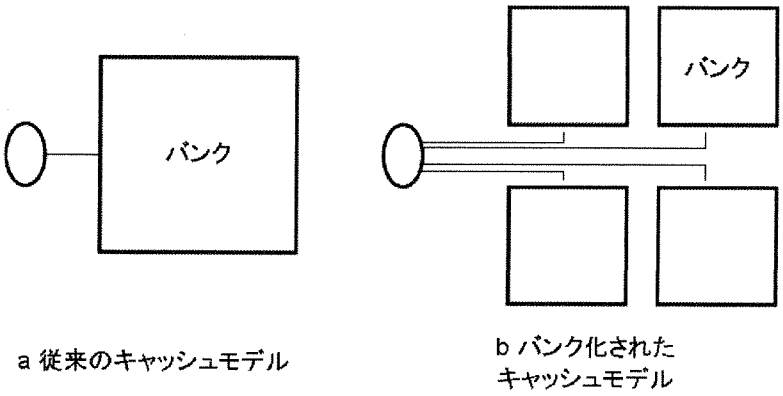

図 1　キャッシュモデル

この手法は，IBM 社の Power5 プロセッサに用いられている．バンク化されたキャッシュ

では，複数のパイプラインから同時にキャッシュアクセスがあった場合でも，同じバンクに対するアクセスでなければ，競合が発生することはない．また，回路が各バンクで閉じたものとなり，ラインへのアクセスの際，他のラインへのアクセスにタイミングを合わせる必要がない．そのため，キャッシュメモリアクセスによる遅延を緩和することができる．しかし，バンク数が多くなると，キャッシュコントローラから各バンクまでの配線数が多くなるため，キャッシュメモリ面積が増加するという問題がある．

別のアプローチとして非同期式キャッシュがある[7]．非同期式キャッシュでは，メモリ領域へのアクセスは，クロックと同期せず，非同期に行われる．従来の同期式のキャッシュでは 8 サイクルであったアクセスレイテンシが，非同期式キャッシュでは 5 サイクルに緩和される．また，クロック分配回路やラッチが必要なくなるため，消費電力削減の面でも有効である．

上の 2 つの手法は，従来のアーキテクチャを基に改良を行ったものであるが，新たなキャッシュアーキテクチャを導入してアクセスレイテンシを緩和する手法として D-NUCA (Dynamic Non Uniform Cache Architecture) が挙げられる[2]．D-NUCA はバンク間のデータ移動が可能なキャッシュモデルであり，2002 年に kim らによって提案された．D-NUCA は，バンク間でのデータの移動を可能にすることで，キャッシュへのアクセスレイテンシを削減する．D-NUCA の構成を図 2 に示す．D-NUCA は，複数の小バンクで構成され，各バンクはスイッチに接続されている．スイッチは，2D メッシュ上に配置され，各スイッチは point-to-point で接続される．そのため，配線数が増加し，キャッシュメモリ面積が増大するのを防ぐことができる．また，バンク，スイッチにはバッファが設けられている．さらに，各バンクは，複数のサブバンクによって構成され，各サブバンクに対しては，同時アクセスが可能である．

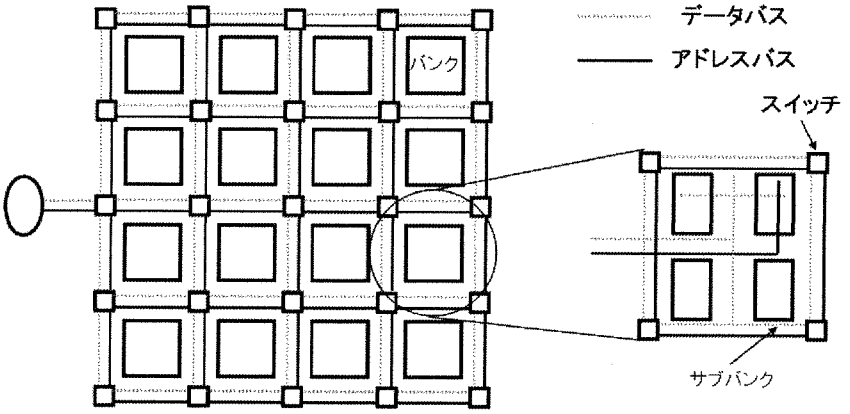

図 2　D-NUCA

従来のキャッシュモデルを改良する手法は，プロセスの微細化によるレイテンシの増加を防ぐことはできない．それに対し，D-NUCA では，バンクの大きさを調整することで，微細化によるレイテンシの増加を緩和することが可能である．しかし，データの移動が隣接バンク間に限られるため，空間的局所性を活用できない．

2.4 提案手法

本節では，提案手法について説明する．2.3 節で述べたように，キャッシュのバンク化は配線遅延の緩和に有効である．提案手法は，バンク化されたキャッシュをさらに有効に活用する．バンク化されたキャッシュにおいて，頻繁にアクセスされるラインが，キャッシュコントローラからの距離が近いバンクに存在すれば，配線遅延によるアクセスの遅延をさらに削減できると考えられる．同様の手法として D-NUCA が挙げられるが，D-NUCA では，データの移動は，隣接バンクであるのに対し，本手法ではデータの配置が LRU 順になるようにデータを移動する．LRU 順に移動することで，同じラインに連続してアクセスする場合のアクセスレイテンシを最小にする．

提案手法ではバンク間でのデータ移動が必要であるため，キャッシュモデルとして D-NUCA を用いる．提案手法では，図 3 に示すように，各 set の各 way を，キャッシュコントローラから way 順になるよう，各バンクに割り当てる．そして，LRU 置換法を用い，キャッシュヒットしたラインを，キャッシュコントローラからの距離が近いバンクに移動する．way1 に存在したラインは way2 に移動する．以後，この動作をラインヒットした way まで連続して行う．D-NUCA に比べ，移動のコストが増加するが，提案手法では，空間的局所性があるプログラムにおいて，アクセスレイテンシの小さいバンクに連続してアクセスすることが可能になるため，アクセスレイテンシを削減できる．

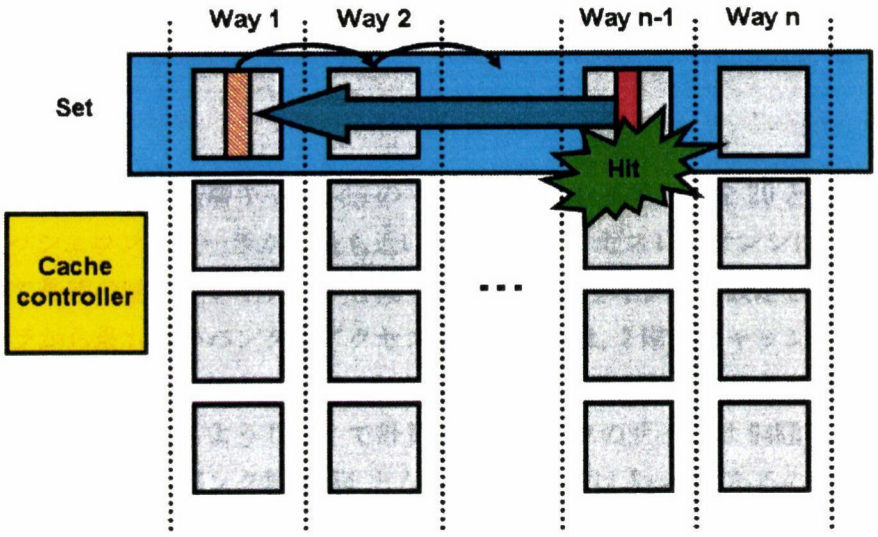

図 3　提案手法の動作例

2.5. 評価

2.5.1 実験環境

提案手法による性能向上を検証するため，SimpleScalar ver3.0d[8] の sim-outorder を用いて提案手法を実装し，IPC の向上率を求めた．sim-outorder はアウトオブオーダー実行可能なサイクル駆動のプロセッサシミュレータであり，提案手法実装による性能への影響を計測することが可能である．本研究では，Intel 社の Pentium M プロセッサアーキテクチャを基に，sim-outorder のパラメータを設定した（表 2）．

ベンチマークプログラムとして，SPEC95 CINT から 6 プログラム，SPEC2000 CINT から 6 プログラム採用した．また，今回の実験は，65nm プロセス(周波数 6GHz，キャッシュ容量を 4MB)のプロセッサとしている．

表 2　SimpleScalar パラメータ諸源

L1 命令キャッシュ	方式	ダイレクトマップ
	容量	16KB
	ラインサイズ	32B
	レイテンシ	1サイクル
L1データキャッシュ	方式	4way
	容量	16KB
	ラインサイズ	32B
	レイテンシ	1サイクル
L2統合キャッシュ	方式	8way
	容量	4MB
	ラインサイズ	64B

アクセスレイテンシ

実験では，表 1 より 2 階層キャッシュの平均アクセスレイテンシを 20 とした．提案手法では，キャッシュコントローラから最も近いバンクのアクセスレイテンシは，2 階層キャッシュにおいて最速の場合のレイテンシと同じ値とし，10 サイクルに設定した．キャッシュコントローラから最も遠いバンクのアクセスレイテンシは，2 階層キャッシュにおいて最遅の場合のレイテンシと同じ値とし，30 サイクルに設定した．さらに，各バンクのアクセスレイテンシが 1 以下になるよう Cacti で計算し，各バンクのサイズは 64KB とした．したがって，提案手法におけるバンク数は 64 となる．ウェイ数は 8way であるので，各バンクは 8*8 の 2D メッシュ型の配置となる．また，最短のバンクと最遠のバンクのレイテンシの差は 20 であり，通過するスイッチ数の差は 10 であるので，各バンク間のレイテンシは 2 と

なる．したがって，アクセスするまでに通過するスイッチ数を a (1<=a<=11)，データ移動にかかるオーバヘッドを b とすると，レイテンシ T は式 (3) で表される．

$$T = 10 + 2*(a-1) + b \quad \cdots \quad (3)$$

オーバヘッド b は，移動した way 数分だけ新たにバンクに書き込みが発生することから，各バンクのアクセスレイテンシ 1 と way 数の積となる．したがって，先頭の way1 にアクセスする場合，データ移動がないのでオーバヘッドは 0，way8 にアクセスする場合，7 回のデータ移動が発生するのでオーバヘッドは 7 となる．

2.5.2 評価結果

評価結果を表 3 に示す．表 3 の理論値とは，提案手法において，最もアクセスレイテンシの小さなバンクに必要なデータが存在すると仮定した場合の値である．

表 3　評価結果

	compress	go	ijpeg	li	m88ksim	perl	gap	gcc	mcf	parser	perlbmk	vpr
2階層キャッシュ	1.39	0.44	1.76	1.15	0.69	0.51	0.58	0.56	0.20	0.93	0.50	0.43
提案手法	1.46	0.54	1.78	1.36	0.97	0.60	0.56	0.59	0.20	1.02	0.63	0.70
理論値	1.50	0.69	1.94	1.39	1.06	0.76	0.75	0.86	0.22	1.15	0.79	0.72
L2キャッシュミス	0.017	0.010	0.005	0.020	0.010	0.004	0.038	0.002	0.050	0.006	0.000	0.000
向上率	1.05	1.23	1.01	1.18	1.40	1.18	0.97	1.04	0.99	1.10	1.24	1.62

実験から，提案手法を用いることで平均 1.17 倍の IPC 向上率が得られることがわかった．なお，移動のオーバヘッドを考慮にしない場合の IPC 向上率は 1.22 倍である．特に，m88ksim，vpr では，高い IPC 向上率が得られた．これらのプログラムは，L2 キャッシュミス率が低く，また空間的局所性が高いため，提案手法が有効に働いたと考えられる．

また，gap，mcf では IPC の低下が見られた．空間的局所性を活用できず，L2 キャッシュのミス率が高いため，メモリアクセス時のバンク間のデータの移動がボトルネックになったと考えられる．

いくつのかのプログラムでは，IPC の低下が見られたが，IPC の平均低下率は 5%未満であり，プログラム全体の平均向上率が 17%であることを考慮すると，提案手法はキャッシュメモリのアクセスレイテンシの緩和に有効であると考えられる．

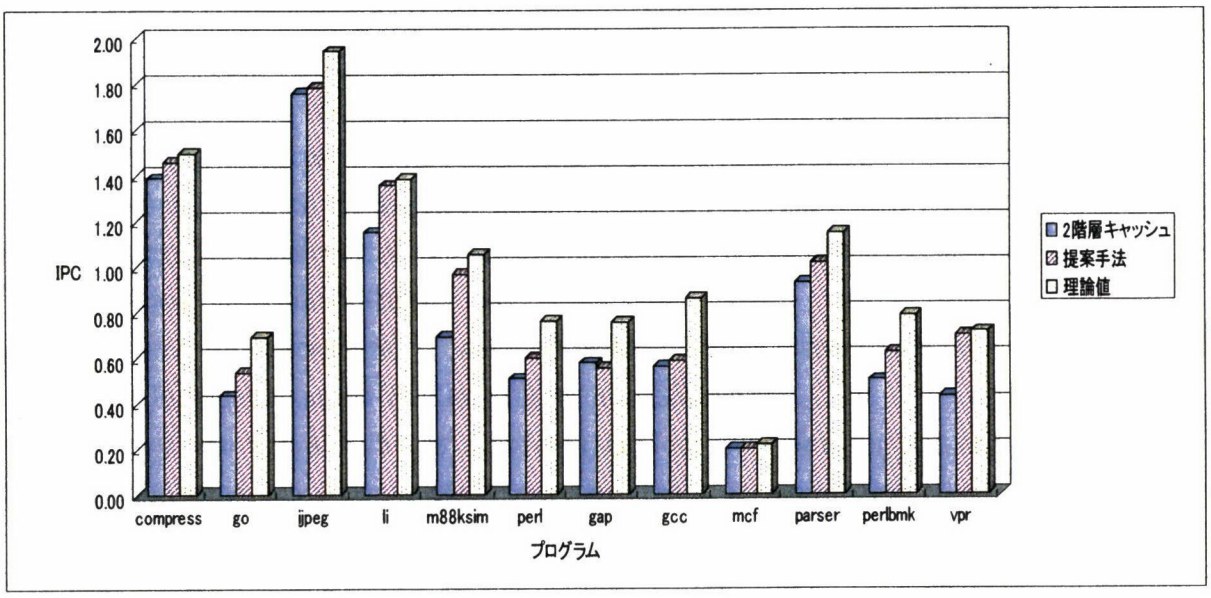

図4　IPC 向上率

2.6 おわりに

本章では，プロセッサアーキテクチャレベルでキャッシュメモリの配線遅延を緩和する手法を提案した．提案手法を SimpleScalar3.0d を用いて実装し，SPEC95 CINT，SPEC2000 CINT を用いて評価した結果，平均の 1.17 倍の IPC 向上率が得られた．しかし，キャッシュミス率の高いプログラムにおいて，IPC の低下が見られた．メモリアクセス時のデータ移動に関して検討する必要がある．

提案手法は，頻繁にデータの移動が発生するため，消費電力が高くなると考えられる．そのため，今後は消費電力の観点から提案手法の改良を行っていく必要があると考えられる．提案手法では，ラインがキャッシュコントローラから LRU 順に並んでいるため，way 適応型のキャッシュや，キャッシュサイズを動的に変更する手法が有効に利用できる可能性がある．したがって，これらの手法との連携を今後の課題とする．

参考文献

[1] Hennessy, J. L., and Patterson, D. A.: Computer Architecture: A Quantitative Approach, 3rd Edtion, Morgan Kaufmann Publishers (2002)

[2] Changkyu kim, Doung Burger and Stephan W. Keckler: An Adaptive, Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches, Proc. of the 10th Int. Con. on Architectural Support for Programming Languages and Operating Systems (2002)

[3] 山本一郎，レイアウト設計 1，SoC 設計技術 LSI 設計編 (B コース)，半導体理工学研究センター (2003)

[4] Linda. W and R. Mike: ITRS Overview, ITRS(International Technology Roadmap for Semicondoctors) Public Home Page, http://public.itrs.net, 2004

[5] CACTI, Western Research Laboratory - Compaq, http://research.compaq.com/wrl/people/jouppi/CACTI.html

[6] S. Wijeratne, N. Siddaiah, S. Mathew, M. Anders, R. Krishnamurthy, J. Anderson, S. Hwang, M. Ernest and M. Nardin: A 9GHz 65nm Intel Pentium□4 Processor Integer Execution Core, ISSCC(2006)

[7] Naffziger, S. Stackhouse, B. and Grutkowski, T.: The Implementation of a 2-core Multi-Threaded Itanium-Family Processor, ISSCC(2005)

[8] D.Burger and T.M.Austin:The SimpleScalar Tool Set, Version 2.0, University of Wisconsin-Madison Computer Sciences Department Technical Report no. 1342(1997)

3．ネットワーク上のマシンをディスクキャッシュとして利用した高速化手法

3.1 はじめに

ハードディスク(以下，単にディスク) は構造上の理由から，半導体メモリと比較してアクセス速度が低速である．特にランダムアクセス時はヘッドのシーク動作が必要となるため，著しくアクセス速度が低下する．これらの理由から，ディスクアクセスをチューニングすることでシステムの性能向上を実現できる．しかし，ディスクアクセスのチューニング手法は，ハードウェアとアプリケーションの組み合わせによって異なる．個々の環境に合わせてチューニングを行うことは，最適なチューニングが可能である反面，労力と費用を要する．より簡便に既存のシステムにおいて効果が得られる手法が望ましい．

そこで本章では，OS が備える機能であるディスクキャッシュに注目し，ヘルパースレッドを用いて高速化する仕組みを考えた．リスト 1 はファイルからデータを読み込む際のプログラム例である．

```
int fd = open("name", O_RDONLY);
char buf[4096];
int ret = read(fd, buf, sizeof(buf));
```

リスト 1　read システムコール

ここで，read は POSIX 準拠の OS システムコールであり，char 型の配列 buf に最大 sizeof(buf)=4096 バイト読み込むように指示している．read の返り値は実際に読み込まれたデータサイズ(byte) である．

通常，OS は read システムコールにより読み込まれたデータを空きメモリ内にキャッシュしているため，再度同じファイルを読み込む時にキャッシュから高速に読み込むことが可能である．このディスクキャッシュの効果により，特別なチューニングをせずともファイルにアクセスするアプリケーションの高速化が期待できる．しかし，OS はディスクキャッシュの領域に空きメモリ領域を利用するため，メモリの搭載量が少ないマシンの場合や，メモリを大量に利用するアプリケーションが動作している場合，ディスクキャッシュ領域を充分に確保できない．この場合，ディスクアクセスが発生し，ディスクキャッシュにヒットした場合よりもシステム性能が低下する．

そこで本研究では，ネットワークに接続されたリモートマシンの物理メモリを用いてディスクキャッシュ領域を拡張する手法（ヘルパースレッドを用いて実現）を提案する．ネットワークとして現在広く利用されている Gigabit Ethernet の帯域は，CPU とメモリ間の帯域はもちろん，ローカルハードディスクのインターフェース帯域よりも劣る．しかし，リモートマシンのメモリは半導体メモリであるため，ランダムアクセス時はローカルディ

スクに対してアクセスするよりもリモートマシンのメモリにアクセスした方が高速という特徴がある．この特徴を生かして，ランダムアクセスされるファイルをリモートマシンのメモリにキャッシュすることで，ディスクアクセスの高速化を図ることが可能である．

本研究では，ローカルディスクのキャッシュの拡張としてリモートマシンのメモリを用いる．これまでも，リモートマシンのメモリ資源を有効に利用する研究はなされてきた．しかし，リモートマシンのメモリをローカルディスクキャッシュに利用する手法は実用化されていない．本研究は，実用化を目指して実装を行ったプロトタイプの報告である．本研究ではLinux カーネル2.6.15 を修正することで提案手法を実現した．

3.2 関連研究と関連製品

これまで，ネットワークに接続されたリモートマシンのメモリを有効に利用する研究が行われてきた[1]～[5]．リモートマシンのメモリの利用モデルを提案した Remote Memory [1] を筆頭に，ネットワーク経由で物理メモリ上のデータを共有する分散共有メモリ[5]といった研究が1990 年代を中心に行われた．また近年では，クラスタ環境においてスワップ先に他のノードを利用するといった研究[3] も行われている．

リモートマシンのメモリを利用する際の問題はネットワークの帯域である．ネットワーク経由でのリモートマシンのメモリへのアクセスは，ローカル物理メモリにアクセスするよりも遅い．そのため，ローカル物理メモリの代替としてリモートマシンのメモリを利用する場合，性能低下をいかに抑えるかが研究の主要課題となる．

これに対して，ローカルディスクのインタフェースはローカルメモリバスほど高速ではなく，リモートマシンのメモリをローカルディスクの代替に利用することで，場合によってはシステムの高速化を図ることができる．例えば，ディスクはランダムアクセスにおいて低速であるため，Memory Servers [2] の実験のように，リモートマシンのメモリをスワップに利用することでスワップ発生時のシステム高速化が可能である．また Nswap [3] のように，クラスタ環境において他ノードの空き物理メモリをスワップ領域に利用すれば，スワップ発生時のクラスタシステム高速化が可能である．

これらスワップ高速化の既存研究は，リモートマシンのメモリをディスクの代替として利用することで性能向上を図れることを示している．ただし，スワップが発生するのはメモリ不足という緊急的場面であり，スワップの高速化は直ちにシステムの性能向上には繋がらない．これに対して本研究の手法では，スワップ領域ではなく，ディスクキャッシュ領域を拡張するためにリモートマシンのメモリを利用する．ディスクキャッシュ領域の拡張であれば，スワップ発生時という限られた状況でなくとも，ファイルにアクセスするアプリケーションの高速化が期待できる．

ディスクキャッシュ領域をリモートマシンのメモリではなく，PC に付加されたデバイス上にとり高速化する製品は既に存在している．具体的には，Microsoft [6] が提唱する ReadyBoost [7]と ReadyDrive [7]，そして Intel [8] が提唱する Robson [9] といった製

品である.

ReadyBoost は Windows Vista から新しく搭載された機能であり，USB メモリをディスクキャッシュ拡張に利用する．普及している USB メモリを簡単に利用できる点が長所である．突然 USB メモリが外されても，データの損失がないように配慮されている．また，セキュリティ対策のため，USB メモリに書き込まれるデータは暗号化される．

ReadyDrive はディスクドライブ本体にフラッシュメモリを搭載し，ディスクとフラッシュメモリをハイブリッドに使用する手法である．ReadyDrive を利用するには対応ディスクドライブを用いる必要があるため，すでに動作しているシステムに適用するにはドライブの交換が必要となる．また，OS と連携してフラッシュメモリの利用を制御するため，Windows Vista における利用が前提となっている．

Intel が提唱する Robson は，PCI Express スロットにハードディスク専用のキャッシュカードを装着する方式である．2007 年以降に発売予定の Intel のチップセットを搭載したマザーボードで利用できる．PCI Express の帯域は高速であるが，Robson で利用されるのは不揮発性メモリであり，動作速度は搭載される不揮発性メモリの速度により制限される．

こうした従来研究・現行製品がある中で，本研究では，リモートマシンのメモリをローカルディスクのキャッシュとして利用した場合の性能評価をするべく実装を行った．これまでも，たとえば[2] において，ローカルディスクのキャッシュとしてリモートマシンのメモリを利用できるという記述はあったものの，実現はされていない．

本研究は，現在の PC 環境での実用化の可能性を探るべく作成したプロトタイプの報告である．過去のリモートマシンのメモリを利用する研究[1]～[5] は，研究時期の関係で 100Mbps 以下のネットワーク環境において行われており，本研究の意義は 1Gbps におけるデータを取得した点にもあると考える．環境の変更を余儀なくされる現行製品に対して，本研究の実装では，既存環境の Linux カーネルを入れ替えれば導入ができるという利点がある．なお，制御実装はヘルパースレッドを用いて実現した．

3.3 提案手法

3.3.1 概要

図 1 に提案手法の概念図を示す．本提案手法はネットワーク上のマシンの物理メモリをディスクキャッシュとして利用する．以下，ディスクキャッシュとして物理メモリを提供するネットワーク上のマシンを DC(Disk Cache) サーバと略す．また，DC サーバをディスクキャッシュとして利用するマシンを DC クライアントと略す．

ネットワークに接続されたリモートマシンの物理メモリをディスクキャッシュに利用する概念自体は，ハードウェアや OS に制限されるものではない．しかし，本研究では提案手法を実装する OS に Linux を選択しており，本研究の手法は Linux に依存する部分がある．

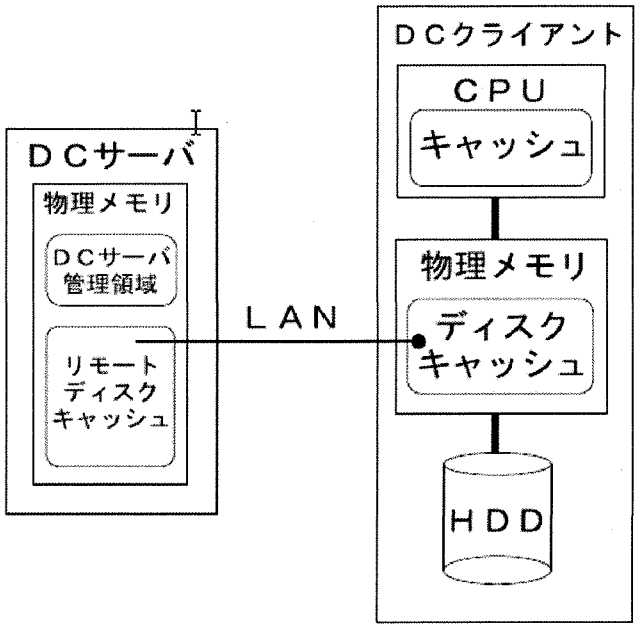

図 1　提案手法の概念図

提案手法の特徴は以下の通りである.

- Linux2.6.15 が動作していれば，カーネルの置き換えで導入が可能.
- DC サーバはユーザプログラムとして実装可能.
- 過去の PC 資産を DC サーバとして再利用が可能.
- ネットワーク接続が失われてもデータ損失無し.
- Linux の先読み機構と協調したキャッシュ動作.
- Direct I/O に対しては機能しない.

3.3.3 必要環境

提案手法の実装は Linux カーネル 2.6.15 に行った．DC クライアントへの導入は Linux カーネルの入れ替えのみで良く，既存のアプリケーションは再コンパイル・再インストールの必要がない．DC サーバのプログラムはユーザプログラムとして実装可能なため，DC クライアントと同じネットワークに接続できれば DC サーバの OS 種別は問わない．すなわち，DC サーバをディスクレスで稼動させることも可能である．また，DC サーバに過去の PC 資産を活用することも可能である.

3.3.4 帯域の問題

本手法で最大の問題となるのが帯域である．表 1 は各種インターフェース速度を示した

ものである．1000Base-T の理論値は 1Gbps (≒ 125Mbytes/sec) であるが，実効転送速度は 1Gbps よりも遅い．このため，DC サーバからデータを読み込むよりも，直接ローカルディスクから読み込んだ方が高速な場合がある．しかし，本提案手法においてディスクキャッシュとして利用するリモートマシンのメモリは半導体メモリである．半導体メモリはランダムアクセスでも高速という特徴がある．そのため，ランダムアクセスされるファイルを優先的に DC サーバにキャッシュするといったキャッシュアルゴリズムの実装により性能の向上を実現できる．

表 1　各種インターフェース速度

USB 1.1 (Full Speed)	12Mbps (1.5MB/sec)
USB 2.0 (High Speed)	480Mbps (60MB/sec)
1000Base-T	1Gbps (125MB/sec)
Serial ATA 1.0	150MB/sec
Ultra160 SCSI	160MB/sec
Ultra320 SCSI	320MB/sec

3.3.5 耐障害性とセキュリティ

本手法はネットワーク経由でデータを転送するため，耐障害性とセキュリティが問題となる．耐障害性を確保するため，本研究における実装では，write back キャッシュとはしていない．すなわち，DC サーバ上のキャッシュデータは，DC クライアントのディスク上か，DC クライアントのディスクキャッシュ上に必ず存在する．そのため，障害により DC サーバと DC クライアント間の接続が失われたとしてもデータ損失は発生しない．write back キャッシュとすることで，ランダム書き込みの性能を向上できる可能性がある．write back キャッシュの検討は今後の課題とする．

セキュリティ対策のためには，ReadyBoost のようにキャッシュデータの暗号化を行う必要がある．しかし，現在の実装では暗号化に対応していないため，DC サーバ用にプライベートネットワークを構築して運用することが望ましい．通信の暗号化も今後の検討課題とする．

3.3.6 Linux カーネル

本節では Linux カーネルの機能のうち，本研究に関係する部分について概要を説明する．ここではカーネルバージョン 2.6.15 に基づいて説明を行う．

1) システムコール

リスト 1 で示した通り，ファイルを読む際には open でファイルを開いたあと read を利用する．ファイルに書き込む際は write を利用する．これら open, read/write はシステムコールである．システムコールは，呼び出したプロセスの延長上で動作する．そのため，

シーケンシャルリードを行うプロセスが複数並行して動作していた場合，ディスクにおいてはランダムアクセスが発生する．このようなケースは今後到来するマルチコア時代において多く発生すると考えられ，より一層のディスク高速化技術が求められる．本提案手法は，このようなマルチタスクに起因するランダムアクセスの高速化にも対応している．

2) ページキャッシュ

Linux カーネルにおいて物理メモリ上のディスクキャッシュはページキャッシュと呼ばれる．read の場合，ページキャッシュにデータがあれば，ページキャッシュからユーザ空間にデータがコピーされる．ページキャッシュにデータが無ければ，ディスクからページキャッシュにデータを読み込んだあと，ページキャッシュからユーザ空間にコピーされる．

write の場合，すぐにはディスクへの反映は行わず，まずページキャッシュ上にデータが保持される．ディスクへの反映は一定時間後か，アプリケーションからリクエストがあった場合，もしくは，ディスクに未反映のページキャッシュ容量が閾値よりも多くなった時に行われる．

通常，ページキャッシュには空き物理メモリが最大限に利用され，空きメモリが不足した際に開放される．開放の際，write されたページキャッシュのうちディスクへの反映が必要なものは，ディスクに反映が行われる．

これらページキャッシュの機能により，同じファイルが繰り返しアクセスされる場合，ユーザ空間へ高速にデータを転送できる．一方で，カーネルによるページキャッシュ管理のオーバヘッドが入るという欠点もある．例えば read でディスクから読み込まれる場合，ディスクからページキャッシュへ，そしてページキャッシュからユーザ空間へ転送するため，最悪 2 回のコピーが発生する．この余分なコピーはファイルを１度しか読まない場合には明らかに無駄である．また，大容量のファイルを読み込むとページキャッシュが溢れ，それまでキャッシュされていたデータが破棄されてしまうという問題もある．これらの問題を回避するため，Direct I/O と呼ばれるページキャッシュを経由しないアクセス手段が用意されている．Direct I/O を用いれば，ディスク上のファイルにページキャッシュを介さずアクセスすることが可能になる．Direct I/O を用いることで，アプリケーションが自身に適したキャッシュ機構を実装し，ディスクアクセスを高度にチューニングすることが可能となる．

本研究における実装では，Direct I/O によるアクセスに関してはキャッシュを行わない機構とした．そのため，Direct I/O を行うアプリケーションに対しては高速化の効果が得られない．

Direct I/O 時も強制的に本手法を機能させる実装を行うことは不可能ではないが，オーバヘッドを挿入することになり，性能を低下させる可能性がある．

3.3.7 提案手法におけるキャッシュ管理

Linux は read/write のリクエストがあったファイルがページキャッシュ上にあるかをソフトウェアで管理している．同様に提案手法でも，DC サーバがどのキャッシュを保持しているかについて，ソフトウェアにより管理する必要がある．

実装では，Linux のページキャッシュの管理単位と同じく，DC サーバ上のキャッシュの管理単位を 4KB とした．これは ext3 ファイルシステムのブロックサイズに等しい．この場合，キャッシュは i-node とファイルオフセット値で管理が可能である．Linux2.6.15 の実装ではファイルオフセット値は 4KB ごとに 1 つの値が取られる．

DC サーバにキャッシュされているデータの管理には，ファイルオフセット値をノードとする木構造を利用する．本研究における実装では，Linux のページキャッシュ管理でも利用されている radix tree を用いた．

さて，この管理領域(radix tree) の配置場所について以下の２通りが考えられ，それぞれに特有の問題がある．

a. 管理領域を DC サーバ上に確保する

read/write の際に，キャッシュが存在するか DC クライアントから DC サーバに問い合わせる必要がある．あるいは，DC サーバにキャッシュが存在するものとして投機的にキャッシュの取得を試みることになる．しかし，DC サーバにキャッシュが存在しなかった場合，ネットワークパケットの往復時間により大幅に性能が低下するという問題がある．

b. 管理領域を DC クライアント上に確保する

a. の欠点は無くなるが，ローカルディスクキャッシュ領域やカーネルメモリ空間が管理領域により圧迫されるという問題がある．

DC クライアントが 32bit マシンの場合，現在の実装では DC サーバのキャッシュ 1GB を管理するのにおよそ 8MB のメモリが必要である．本研究における実装では，a. の DC クライアントに管理領域を保持する方式としている．なお，DC サーバへの接続時に DC サーバの管理に必要なだけのメモリ空間を予約する実装としている．

3.3.8 キャッシュアルゴリズム

本提案手法のキャッシュアルゴリズムを実装する際は，少なくとも以下の３点を考慮する必要がある．

(1) どのデータを DC サーバにキャッシュするか．
(2) DC サーバにデータがある時，DC サーバから読み込むか，ローカルディスクから読み込むか．

(3)　キャッシュ領域が不足したとき，どのキャッシュデータを破棄するか.

(1)(2) に関しては，3.3.4 で述べた帯域の制限により，本手法がシーケンシャルアクセス時に動作すると性能低下を引き起こすという問題を考慮する必要がある.

まず(1) については，ランダムアクセスされるファイルを優先的にキャッシュする必要がある．不必要に DC サーバへキャッシュを行うと，ディスクからの読み込みと DC サーバへの送信の 2 倍の I/O が発生し，システム性能が低下する．また同様に，(2) に関してもシーケンシャルリード時にはローカルディスクから読み込み，ランダムリード時には DC サーバから読み込む必要がある．通常のキャッシュシステムではキャッシュから読み込む方が高速であるため，(2) は本手法特有の考慮事項であるといえる．例えば OS のディスクキャッシュ機構では，ディスクから読むよりもローカル物理メモリから読み込んだ方が高速である．そのため，OS の場合はシーケンシャルリードされるファイルデータもローカル物理メモリキャッシュすることで性能向上を図れる.

本研究における実装では(1)(2) を解決するために，Linux の先読み機構と協調する動作とした．Linux はシーケンシャルアクセスと判定した場合，アプリケーションが要求したファイルオフセットよりも先まで読み込み命令を発行する．そこで，本研究における実装では，以下に示す戦略をとった．なお，n は DC サーバよりもローカルディスクから読んだ方が高速であると期待できるブロック数であり，DC 動作閾値と呼ぶ．DC 動作閾値はシステムに依存する変数であり，本提案手法の効果を最大化する n を利用する必要がある.

・ n ブロック以上の先読み
　　必ずディスクから読み込む.
・ n ブロック未満の先読み or 単一ブロック読み込み
　- DC サーバにキャッシュされている場合
　　DC サーバから読み込む.
　- DC サーバにキャッシュされていない場合
　　ディスクから読み込み，Linux のディスクキャッシュにキャッシュし，DC サーバにもキャッシュする.
・ 書き込み
　- DC サーバにキャッシュされている場合
　　Linux のディスクキャッシュにキャッシュし，DC サーバのキャッシュを破棄する.
　- DC サーバにキャッシュされていない場合
　　Linux のディスクキャッシュのみにキャッシュをする.

DC 動作閾値 n として，固定値を利用する方法，DC 動作閾値を動的に決定する方法，そ

して，ユーザがチューニングを行い決定する方法が考えられる．ユーザがチューニングを行わずとも最高の性能向上が得られるという目標のためには，DC 動作閾値を動的に決定するべきである．しかし，現在の実装では動的な DC 動作閾値の決定に対応していない．そこで，本研究における実験では DC 動作閾値として固定値 32 を用いた．これは，Linux のデフォルトでは先読みの最大値が 128KB であり，ブロックサイズ 4KB 換算で 32 ブロックが先読みの最大値だからである．すなわち，先読みの最大値がデフォルトの環境では，最大限に先読みが働いている場合，必ずローカルディスクから読み込むことになる．一方，書き込み時は DC サーバのキャッシュデータを破棄する実装となっている．

以上の実装から，DC サーバにデータがキャッシュされるのは，32 ブロック未満の読み込みが発生した時のみである．なお，ハードディスク上のデータ配置はファイルシステムに依存するという問題がある．そのため，ファイルオフセット値が連続していても，ランダムアクセスとなる可能性がある．この点を解決するためにはファイルシステムと協調した実装が必要であるが，本研究においてはファイルオフセット値のみでの実装としている．

（3）に関しては LRU を用いた．ここでの LRU とは，DC サーバから DC クライアントへのキャッシュ転送回数に基づく LRU である．つまり，最近 DC サーバから DC クライアントへ転送されていないキャッシュから順に破棄される．

3.3.9 プロトコル

本研究における実装では，データの正当性確保のため DC サーバと DC クライアント間のコネクションに TCP を利用した．UDP，その他のプロトコルを利用した場合の性能比較等は今後の課題とする．

DC サーバと DC クライアント間の通信は，キャッシュデータの送受信と，DC クライアントから DC サーバへのキャッシュ破棄指示である．キャッシュの破棄指示は，write された場合に発行される．

3.4 評価

3.4.1 実験環境

1 台の DC サーバと 1 台の DC クライアントに限定して実験を行った．環境は，業務レベルの環境Ａ（表 2）と個人レベルの環境Ｂ（表 3）との２種類である．以下のいずれの実験も「手法無効」の場合は公式の Linux カーネル 2.6.15 上で実験し，「手法有効」の場合は修正した Linux カーネル 2.6.15 上で実験した．

3.4.2 ディスクアクセスに対する評価

本節ではシーケンシャルとランダムなディスクアクセスに対する評価を行う．

1) シーケンシャルリード

実験環境 B におけるシーケンシャルリードの実験結果を表 4 と図 2 に示す. 図 2 は本手法による性能向上比を示したものである.

この実験では, 実験環境 B において 64MB から 2GB までのファイル全領域に対し 2 回続けてシーケンシャルリードを行い, 1 回目と 2 回目の実行時間を比較した. 1 回目でキャッシュされ, 2 回目でキャッシュヒットすれば 2 回目の性能は 1 回目と比較して向上する. 2 回目でも性能向上がない場合は, キャッシュが溢れていることを示す.

本研究では, シーケンシャルリード時は動作しない実装としているため, 提案手法による性能向上は得られない. すなわち, 表 4 の 2 回目における性能向上はローカルディスクキャッシュによるものである. 1 回目と 2 回目の結果を比較すると, 提案手法が無効の場合は 512MB, 有効の場合は 256MB でローカルディスクキャッシュが溢れていることが分かる. 提案手法が有効の場合に, 無効の場合よりも少ない容量でローカルディスクキャッシュの溢れが起こったのは, 3. 7 で述べたとおり, DC サーバの管理に必要なメモリをローカルに確保するため, ローカルディスクキャッシュ領域が圧迫されたためである.

表 2 実験環境 A (業務レベル)

	DC クライアント (Dell PowerEdge 2850)	DC サーバ (Dell Precision 490)
CPU	Intel 64bit Xeon 3.2GHz (HT 有効) × 2	Intel Xeon 5110 × 2
Memory	DDR2 400 8GB (PAE36 使用)	DDR2 533 16GB (DC サーバ用に 12GB)
Disk	Ultra SCSI 320 15000rpm RAID5 (PERC 4e/Di Standard FW 521S DRAM:256MB)	SAS 15000rpm NoRAID
FS	ext3	ext3
OS	Linux 2.6.15 (x86 smp 公式版 or 修正版) (Fedora Core 5)	Linux 2.6.9-5.EL (x86_64 smp) (Red Hat Enterprise Linux WS Release 4)
Link	1000Base-T	1000Base-T
HUB	Dell PowerConnect 2716	

表 3 実験環境 B (個人レベル)

	DC クライアント	DC サーバ
CPU	AMD Geode NX 1500	Intel Pentium4 2.8CGHz
Memory	DDR333 512MB	DDR400 2GB (DC サーバ用に 1.2GB)
Disk	SATA 7200rpm NoRAID	SATA 7200rpm NoRAID
FS	ext3	ext3
OS	Linux 2.6.15 (x86 公式版 or 修正版) (Fedora Core 5)	Linux 2.6.15 (x86 smp) (Fedora Core 5)
Link	1000Base-T	1000Base-T
Hub	BUFFALO LSW-GT-5NS	

表 4　シーケンシャルリード 提案手法の効果 (実験環境 B)

アクセス容量	所要時間 [sec]			
	提案手法無効		提案手法有効	
	1 回目	2 回目	1 回目	2 回目
64MB	1.41	0.24	1.41	0.25
128MB	2.72	0.61	2.75	0.73
256MB	5.38	2.03	5.78	5.33
512MB	10.55	10.53	10.60	10.57
1GB	21.31	21.37	21.30	21.31
2GB	42.60	42.69	42.61	42.71

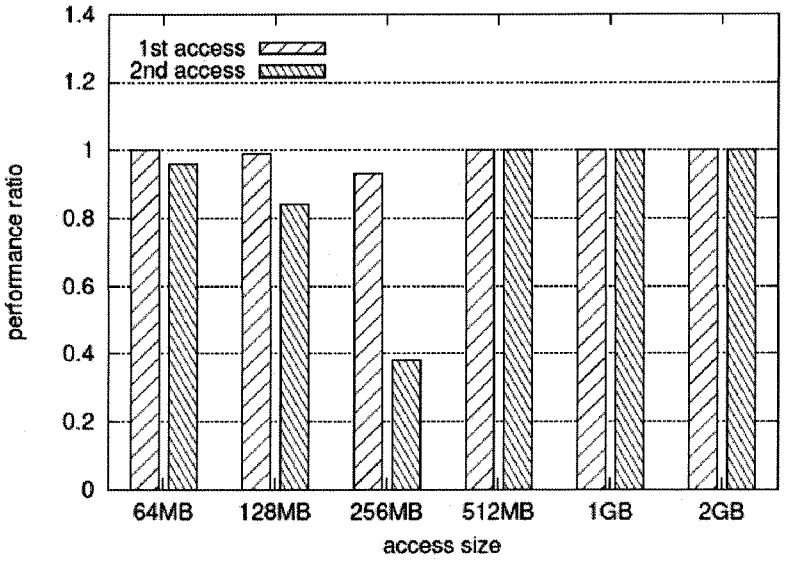

図 2　本手法によるシーケンシャルリード性能向上比 (実験環境 B)
(手法無効時を 1.00 とした)

2) ランダムリード

ランダムリードの実験結果を表 5 と表 6 に示す. 図 3 は, 本手法による実験環境 B における性能向上比を示したものである. この実験では, 実験環境 A・B において表 5 と表 6 の「アクセス容量」の大きさに等しいファイルに対し 2 回続けてランダムリードを行い, 1 回目と 2 回目の実行時間を比較した. 2 回目ではキャッシュヒットすれば性能が向上する. 各回の実験は, 4KB の倍数でファイル位置を選択し, 選択した位置から 4KB の読み込みを行う. これを繰り返し, 読み込んだ合計量が「アクセス容量」に等しくなるまで続ける. 1 回目と 2 回目では同じ乱数列を用いており, アクセスパターンは等しい. また, 乱数を用いたため, ある領域が 2 回以上読み込まれる可能性もある. なお, 4KB は ext3 ファイルシステムのブロックサイズと等しい.

表 5　ランダムリード 提案手法の効果 (実験環境 A)

アクセス容量	所用時間 [sec]			
	提案手法無効		提案手法有効	
	1 回目	2 回目	1 回目	2 回目
4GB	1630.65	2.87	1543.30	4.38
8GB	5898.31	5.60	3626.48	131.49
16GB	12255.99	6280.92	9500.07	946.26

表 6　ランダムリード 提案手法の効果 (実験環境 B)

アクセス容量	所用時間 [sec]			
	提案手法無効		提案手法有効	
	1 回目	2 回目	1 回目	2 回目
64MB	40.01	0.30	43.01	0.29
128MB	92.99	0.58	99.76	0.59
256MB	197.05	3.23	209.62	1.78
512MB	538.95	454.27	441.54	46.92
1GB	1447.30	1413.55	945.30	136.76
2GB	3451.03	3430.72	2390.32	1999.29

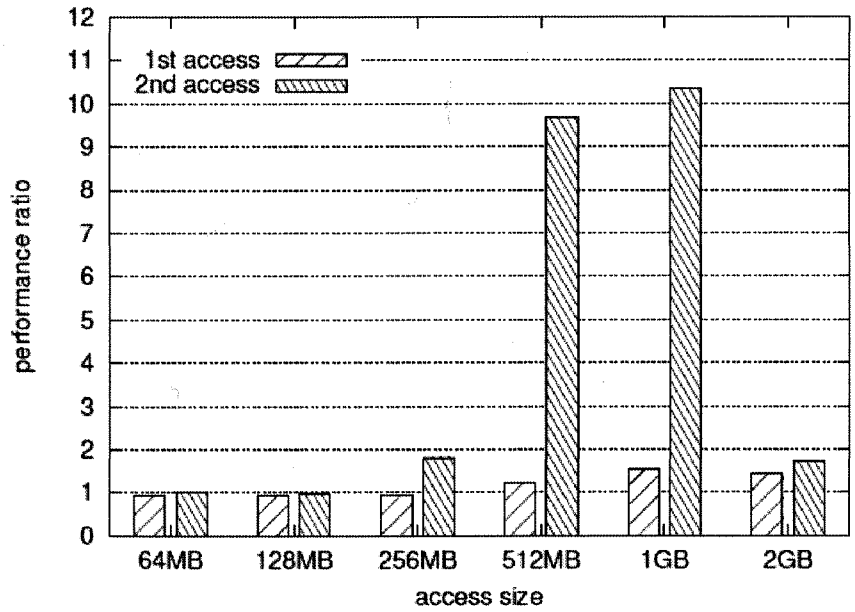

図 3　本手法によるランダムリード性能向上比 (実験環境 B)
(手法無効時を 1.00 とした)

表 6 のとおり，ランダムアクセス時は本手法により性能が向上する．アクセス量 1GB の 2 回目を手法無効時と有効時で比較すると，本手法により 10.3 倍の性能向上が得られていることが分かる．手法無効時のアクセス量 512MB 以上で 2 回目の性能が急激に低下したのは，ローカルディスクキャッシュが溢れたためである．ローカルディスクキャッシュが溢れるアクセス量 512MB 以上の場合は，乱数の重複のため，提案手法を用いると 1 回目でも高速化されている．なお，提案手法有効時にアクセス量 2GB で急激に性能が低下しているのは，DC サーバのキャッシュ(1.2GB) が溢れたためである．

3.4.3 データベースベンチマークテストによる評価

本研究ではデータベースに対してもベンチマークを行った．ベンチマークでは，TPC-H [10] を参考に作成され，フリーで利用が可能な DBT-3 [11] を用いた．測定対象のデータベースは PostgreSQL [12] とした．組み合わせは，いずれも執筆時点で最新の DBT-3 1.9 と PostgreSQL 8.2.0 とした(表 7)．ベンチマークにあたっては OSS iPedia [13] で配布されているパッチ(dbt3-1.9-20060329.patch.gz) を適用した．

DBT-3 はロードテスト，パワーテスト，スループットテストの 3 種類のベンチマークを行う．ロードテストでは，データをデータベースに投入し，インデックスの作成を行う．ロードテストの結果は所要時間で示され，所要時間が短いほど高性能といえる．パワーテストでは，接続数 1 でデータベースへの問い合わせとデータの追加・削除を行う．スループットテストでは，複数接続でパワーテストと同様のテストが行われる．パワーテストとスループットテストの結果は 1 時間毎のトランザクション数×スケールファクタで示され，値が大きいほど高性能といえる．

DBT-3 が行うベンチマークの負荷レベルはスケールファクタで表される．スケールファクタはデータベースへ投入されるデータソースファイルの容量(GB) である．本実験では，実験環境 A においてスケールファクタ 1 から 8，実験環境 B ではスケールファクタ 1 で測定を行った．また，手法有効時と無効時で公平な状態になるよう DBT-3 の乱数シードは同じ値を使用している．

考察

環境 A の実験結果を表 8 と図 4 に示す．キャッシュの転送量を表 9 に示す．図 4 は，本手法による性能向上比を示したものである．データベースでは書き込み処理や，シーケンシャルアクセスとランダムアクセスが複合的に発生するため，3.4.2 のような単純なベンチマークよりも性能向上が難しくなる．提案手法を有効にした場合，ロードテストでは性能低下が発生した．ロードテストではテキストファイルからデータを読み込み，データベースへ投入する．テキストファイルからの読み込みはシーケンシャルリードであり，本手法の効果が得られない．また，インデックス作成における書き込み処理は，読み込み時のみキャッシュを行う本手法の実装により高速化がされない．パワーテストでは提案手法

表 7　データベース測定内容

データベースソフト	PostgreSQL
バージョン	8.2.0
ベンチマークソフト	DBT-3 1.9
スケールファクタ	1~8
ストリーム数	2

表 8　DBT-3 による PostgreSQL 測定結果 (実験環境 A)

スケールファクタ	ロードテスト [sec]		パワーテスト		スループットテスト	
	提案手法の有効／無効					
	無効	有効	無効	有効	無効	有効
1	273	283	713.89	666.2	556.22	553.45
2	606	543	611.37	552.02	523.17	525.34
4	1169	1299	279.39	257.75	224.06	261.22
6	2231	2390	153.2	182.94	53.96	127.21
8	2590	3221	108.97	101.47	34.15	90.91

を有効にすると性能が最高 1.19 倍になった一方で，最低の場合では 0.90 倍になった．スループットテストでは，スケールファクタ 1 の時に 0.5%低下したが，スケールファクタ 8 の時には 2.66 倍の性能向上を得られた．スループットテストにおいてはデータベースへの接続数が増えるため，ランダムアクセスが増加し，本手法の効果が現れたからと考えられる．

環境 B の結果 (表 10) では，スループットテストにおいて， 3.08 倍の性能向上が得られた．これは，本研究におけるデータベースに対するベンチマークでの最高性能向上率であった．

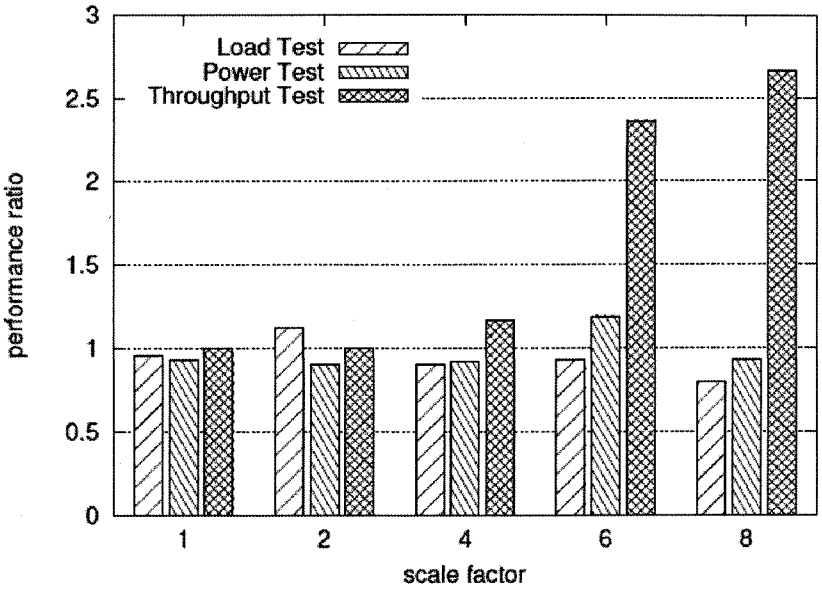

図 4　本手法による DBT-3 性能向上比 (実験環境 A)
(手法無効時を 1.00 とした)

表 9　表 8 の実験におけるキャッシュ転送量 (実験環境 A)

スケールファクタ	キャッシュ転送量 [GB]		
	To DC サーバ	From DC サーバ	破棄キャッシュ容量
1	0.01	0.00	0.00
2	0.29	0.06	0.04
4	8.36	0.40	1.67
6	17.22	93.02	6.43
8	22.25	193.82	10.25

表 10　DBT-3 による PostgreSQL 測定結果 (実験環境 B)
(スケールファクタ：1)

	ロードテスト [sec]	パワーテスト	スループットテスト
手法無効	1114	52.71	7.98
手法有効	1147	59.72	24.54
性能向上	0.97	1.13	3.08

3.5 おわりに

本研究では，ネットワークに接続されたリモートマシンのメモリを利用してディスクキャッシュの領域を拡張する手法を提案し，性能を評価した．なお，実現にあたってはマルチスレッディングの技術を用いている．PostgreSQL に対する DBT-3 によるベンチマークでは，環境 B におけるスループットテストで最大 3.08 倍の性能向上を達成した一方で，環境 A におけるロードテストでは最大 24%の性能低下があった．

本実験では，1 台の DC サーバと 1 台の DC クライアントを接続している．また，DC サーバのメモリ量は DC クライアントよりも大きい．このような環境を用意できるなら，DC サーバを運用に回した方が良い．本手法が想定するのは，既存のシステムに余剰資産のマシンを DC サーバとして接続することでキャッシュ容量を確保するという利用法である．最大容量のメモリを搭載している場合や，システム構成を変更するのが困難な場合でも，性能向上の最終手段としての利用が可能であると考える．

参考文献

[1] D. Comer and J. Griffioen, " A New Design for Distributed Systems: The Remote Memory Model, " In Proceedings of the USENIX Summer 1990 Technical Conference, pp. 127-136, June, 1990.

[2] Liviu Iftode, Kai Li, Karin Petersen, " Memory Servers for Multicomputers, " In Proceedings of the IEEE Spring COMPCON93, pp. 538-547, February 1993.

[3] Tia Newhall, Sean Finney, Kuzman Ganchev, Michael Spiegel, " Nswap: A Network Swapping Module for Linux Clusters, " In Proceedings of Euro-Par International Conference on Parallel and Distributed Computing, vol. 2790, August 2003.

[4] Kai Li, Karin Petersen, "Evaluation of Memory System Extensions." In Proceedings of the 18th annual International Symposium on Computer Architecture, pp. 84-93, 1991.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks ofWorkstations," In IEEE Computer, vol. 29, no. 2, pp. 18-28, February 1996.

[6] Microsoft, http://www.microsoft.com/.

[7] Windows PC Accelerators, http://www.microsoft.com/whdc/system/sysperf/perfaccel.mspx.

[8] Intel, http://www.intel.com/.

[9] Michael Trainor, " Overcoming Disk Drive Access Bottlenecks with Intel Robson Technology, " Technology@Intel Magazine, vol. 4, no. 9, December 2006.

[10] Transaction Processing Performance Council, http://www.tpc.org/.

[11] OSDL Database Test 3, http://www.osdl.org/lab activities/kernel testing/osdl

database test suite/osdl dbt-3/.
[12] PostgreSQL: The world's most advanced open source database, http://www.postgresql.org/.
[13] OSS iPedia, http://ossipedia.ipa.go.jp/.

4．先読みスレッドを用いたDiskアクセスの高速化

4.1 はじめに

主記憶装置とHDDとでは，アクセス速度に大きなギャップがある．具体的には，表1の実験環境Bにおいて，主記憶装置上にある1GBのデータにアクセスする際にかかる時間は約6秒であるのに対して，HDD上にある1GBのデータにアクセスする際にかかる時間は約23秒であり，4～10倍の差がある[1] [2]．

アプリケーションの実行に際して，HDDへのアクセスはしばしばボトルネックになるとして，様々な工夫や研究が行われてきた．その一つは，Disk Readahead [3]であり，シーケンシャルアクセスされているデータに対して，要求されたデータよりも多くのデータを読み込み，次回のアクセスに備えてディスクキャッシュに載せるというものである．現在すでにLinuxには，この機能が実装されている．シーケンシャルアクセス以外の手法としては，Informed Prefetching[1][2]がある．これは，アプリケーションレベルで事前にヒントを与え，次にどのデータが要求されるかを予測し，その結果を元にPrefetchを行う手法である．シーケンシャルでは無いものに対しても効果があるとされている．

しかし，従来手法のDisk Readaheadは，IO待ち時間を軽減することができるものの，依然として繰り返しディスクキャッシュミスが発生している．さらにInformed Prefetchingは，次に要求されるデータの予測が失敗した場合，Prefetchの失敗を意味し，ディスクアクセスの高速化が実現できない．そこで，本研究では，近年一般的に普及してきたHyperThreding対応ＣＰＵ，及びマルチコアCPUの構造に着目し，別スレッドでディスクの先読みを行い，ディスクキャッシュミスを軽減する手法を提案する．具体的には，メインスレッドでアプリケーションを実行していると同時に，別スレッドで次のＨＤＤアクセスを実行する．その結果，提案手法を適用することによって，メインスレッドは，HDDに記録されているデータにアクセスしているにも関らず，あたかもメモリ上のデータにアクセスしているような性能を得ることができる．

4.2 関連研究

HDDへアクセスする際に発生する遅延時間をできる限り減らそうと長年研究が行われてきた．大きく分けるとハードウェア面で性能を向上させる手法[2] [4]と，ソフトウェア面で性能を向上させる手法[1] [3]に分けることができる．ハードウェア面で性能を向上させるものには，Disk Striping[2] [4]により読み取り速度を向上させるものがある．ソフトウェア面で性能を向上させるものは，Disk Cacheを効果的に使うことにより，遅延時間を減らす研究である．

Disk Cacheに着目した研究はさらに分類することができ，Prefetch, Readahead, Cache Controlのどれかに属するものとして考えることができる．Prefetchはプログラムにヒントを与え，次に必要になりそうなデータを事前に予測して取得するという手法である．主

にランダムアクセスの多いプログラムに対して効果が高いが予測が外れると効果が無い．提案手法は，次の Disk アクセスを特定できるため，確実に効果がある．Readahead はシーケンシャルにデータにアクセスする際に効果のある手法で，要求のあったデータの次のいくつか先のブロックまでまとめて読み込むという手法である．Readahead は，すでに Linux に組み込まれている．Cache Control は，主記憶上にあるディスクキャッシュを扱うためのアルゴリズムに関する研究であり，他の Prefetch や Readahead の手法と合わせて利用することができる．

4.3 提案手法

I/O によるプログラム実行の遅延を軽減するために，先読みスレッドを用いて事前にファイルの読み込みを行う手法を提案する．事前に先読みスレッドがデータの読み込みを行うことで，メインスレッドがデータを取得する際には，ディスクキャッシュ上から目的のデータを取得することが可能になり，I/O による遅延を軽減することができる．

4.3.1 対象プログラム

本手法が対象とするのは，データに対してシーケンシャルにアクセスを行い，かつデータを読み込みつつ逐次演算処理を行うようなプログラムである．その時演算にかかる時間と，I/O にかかる時間の関係は，式(1)の状態にあることが望ましい．逆に式(2)の状態にある時，本手法は有効で無いばかりか性能の低下を招くこともある．式(1)の状態にある時，本来 I/O にかかる時間を演算処理と I/O 処理を並列に行うことにより，演算時間の中に埋め込むことが可能である．

対象プログラムとして I/O 時間と演算時間の関係が式(1)の状態にあるプログラムとしてしまうことで対象となるプログラムは限定されてしまう．しかし，式(1)の状態にあるプログラムには圧縮系のプログラムやマルチメディア系のプログラムが含まれるため有意義であると考えられる．

$$I/O\text{時間} \leq \text{演算時間} \quad (1)$$

$$I/O\text{時間} >> \text{演算時間} \quad (2)$$

4.3.2 対象アーキテクチャ

本手法は，Hyper-Threading 対応の CPU，複数のコアを持った CPU，メモリが共有されているマルチプロセッサ上で性能向上を得ることができる．4.3.5 で述べるように，本手法では先読みスレッドの制御に多数の演算処理を伴うため，メインスレッドと先読みスレッドが同時に実行できる環境でより高い性能向上を得ることができる．

4.3.3 提案手法のモデル

提案手法のモデルは，次に示すとおりである．図1にあるようにメインスレッド上で，読み込むファイルが確定した時点で先読みスレッドを起動する．先読みスレッドは，指定されたファイルを先行して読み込み続ける．先読みスレッドによって読み込まれたデータはディスクキャッシュに乗ることになる．メインスレッドがデータを必要とする時には，ディスクキャッシュ上にデータが存在する．その結果，HDDから読み込む場合に比べて短時間でデータの取得が可能になる．

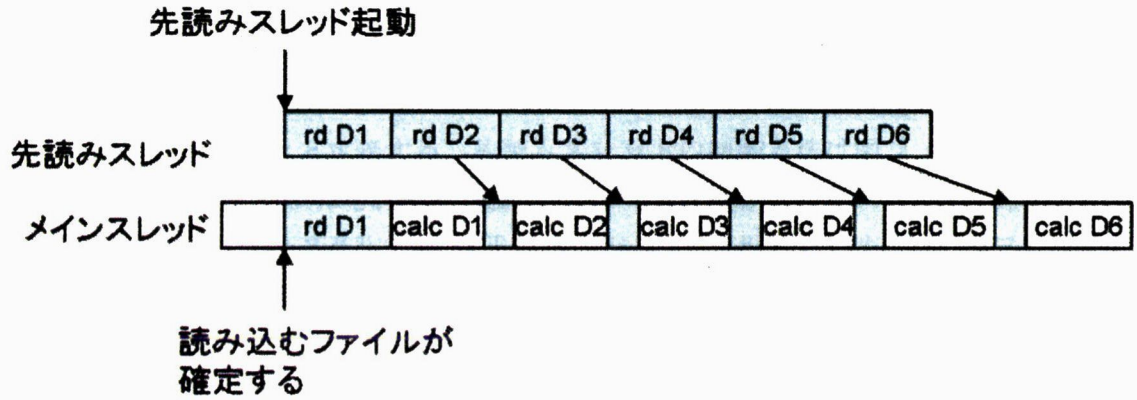

図1 提案手法のモデル

4.3.4 ディスクキャッシュの汚染

コンピュータ上のメモリリソースは限られているため，闇雲に先読みスレッドが先読みをしてしまうと，ディスクキャッシュの汚染が発生し逆に何倍も遅くなることがある．この汚染を防ぐために，提案手法ではメインスレッドと先読みスレッドの距離を定義し，距離の値によって先読みを制御する方策をとる ．ここで言う距離とは，ある時点でのメインスレッドのファイルディスクリプタが指す位置と先読みスレッドのファイルディスクリプタが指す位置の差である．この距離が空きメモリ領域以上に離れてしまうと，先読みスレッドが先行して読み込んだデータにより，メインスレッドがこれから必要とするデータがディスクキャッシュから追い出されてしまう．結果としてメインスレッドと先読みスレッドの双方が，ディスクキャッシュ上にデータを発見することができなくなるため，二重にHDDへのアクセスが発生してしまう．こうなると，それぞれのスレッドが新たにI/O命令を発行する度にディスクヘッドが移動することになる．ディスクヘッドが移動する度にシークタイムと回転待ち時間の遅延が発生してしまう．この状態は，メインスレッドと先読みスレッドとの距離が離れてしまう場合だけでなく，他の実行中のプロセスがメモリリソースを使う場合にも発生しうる．

4.3.5 先読みスレッドの制御

4.3.4 で述べたように，先読みスレッドは闇雲に先行すればいいわけでは無い．メインスレッドと先読みスレッドの距離が一定以上開かないように制御する必要がある．この制限距離が，小さすぎると思ったような効果を得ることができない．

メインスレッドと先読みスレッドに，それぞれ現在何バイトのデータを読み込んだかを変数として持たせる．データ読み込み後に，読み込んだバイト数を変数に加算する．先読みスレッドは，事前に指定される一定量のデータを読み込んだ段階で変数に格納された値を元に，メインスレッドとの距離を計算する．距離が予め指定された閾値を超えてしまっている場合には，閾値を下回るまで usleep を実行する．

ここでは，メインスレッドと先読みスレッドの距離を計算するまでに読み込むデータ量を ss_distance と定義し，先読みスレッドがメインスレッドよりも先行することを許される限界の距離を limit と定義する．先読みスレッドの制御方法は図 2 のようになる．prefetch_pos, main_pos はそれぞれ先読みスレッドとメインスレッドの現在位置を表す．IFD は入力ファイルディスクリプタ，BUF は入力バッファ，BUFSIZE は入力バッファのサイズをそれぞれ表している．

```
while (keep_reading)  {
  ss_distance =  16MB
  while (ss_distance > 0 )
    size = read (
      IFD,
      BUF,
      BUFSIZE) ;

    prefetch_pos + =  size;

    ss_distance − =  size;
  }
  while  (
    prefetch_pos - main_pos > limit
    ) {
    usleep ( 1 ) ;
  }
}
```

図 2 先読みスレッドの制御方法

ss_distance が小さすぎると頻繁に先読みスレッドとメインスレッドの距離が計算されてしまい，Hyper-Threading 環境下では性能の低下につながってしまう．ss_distance が制限距離に比べて大きすぎると，ディスクキャッシュの汚染に繋がる可能性がある．

制限距離が小さすぎると，先読みスレッドは頻繁に止まってしまい性能向上を得ることができない．制限距離が空きメモリ領域よりも大きいと，ディスクキャッシュ汚染に繋が

ってしまう可能性がある.

4.4 評価実験

提案手法の有効性を検証するため，本提案手法を適用させる前の gzip と適用後の gzip の 2 種類の実行時間の比較を行った．ここで，本手法を適用する前の gzip を標準の gzip，本手法を適用後の gzip を高速 gzip と呼ぶ．今回本手法を適用した gzip のバージョンは 1.3.5 である.

4.4.1 実験環境

実験の環境は表 1 に示す通りである.

表 1　実験環境

	実験環境 A	実験環境 B
CPU	Pentium Extreme Edition 3.2GHz	Pentium4 3.4GHz
Memory	1GB	2GB
OS	Fedora Core 5 Kernel	Windows XP Professional
HDD	SAMSUNG HD160JJ	Hitachi Deskstar 7k250
容量	160GB	160GB
プラッターサイズ	80GB	80GB
キャッシュ	8MB	8MB
回転数	7200RPM	7200RPM
内部転送速度	845Mbits/sec	757Mbits/sec
平均シークタイム	8.9ms	8.5ms
最小シークタイム	0.8ms	1.1ms
最大シークタイム	18ms	15.1ms

4.2　実験内容

まず，標準の gzip を用いて 8 つのファイルを圧縮し，圧縮処理にかかった時間を計測した.次に高速 gzip を用いて 8 つのファイルを圧縮し,圧縮処理にかかった時間を計測した.実行時間の計測方法は，圧縮処理を行っている箇所の前後で，gettimeofday() 関数を用いて測定した．プログラム実験の対象のファイルの容量は表 2 に示す通りである.

表２　実験に用いたファイルサイズ

	実験環境 A	実験環境 B
File No. 1	0.61GB	0.61GB
File No. 2	1.00GB	1.00GB
File No. 3	1.61GB	1.61GB
File No. 4	2.22GB	2.22GB
File No. 5	2.83GB	2.84GB
File No. 6	3.44GB	3.45GB
File No. 7	4.05GB	4.07GB
File No. 8	4.66GB	4.68GB

4.4.3 実験結果

図 3，4 は実験環境 A における標準 gzip と高速 gzip の実行時間の比較である．図 5，6 は実験環境 B における標準 gzip と高速 gzip の実行時間の比較である．それぞれ gzip の圧縮レベルは 3 を指定している．図 3 のグラフは，先読みの距離の制限値を左から順に 4KB，8KB，16KB，・・・，1MB まで変化させた時の結果になっている．図 4 のグラフは，先読みの距離の制限値を左から順に 2MB，4MB，8MB，・・・512MB まで変化させた時の結果になっている．図 5，6 も同様に先読みの距離の制限値を変化させたものの結果となっている．図 3～6 の折れ線グラフで表示されているものが，標準 gzip の実行時間を表している．いずれのグラフも縦軸が時間 (単位は秒) を表し，横軸はファイル番号に対応している．

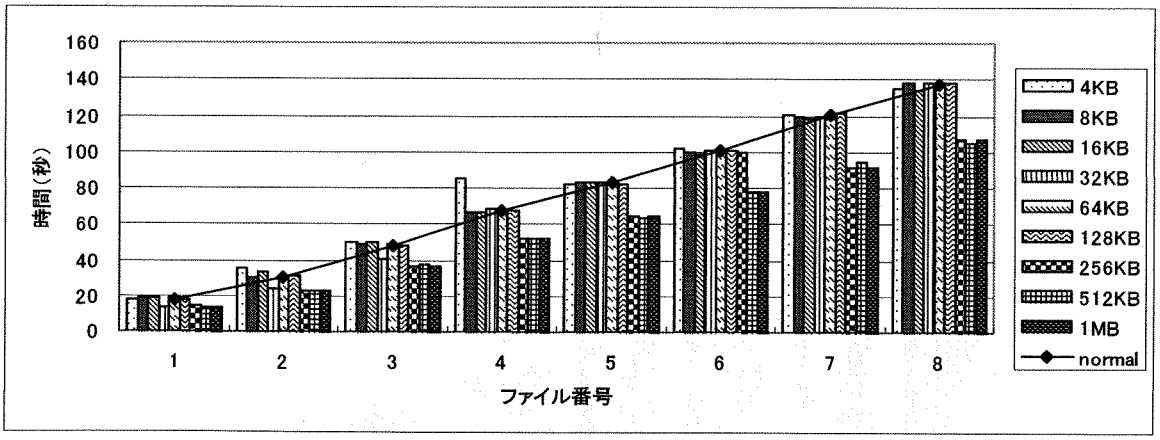

図 3 gzip 実行時間の比較 A-1

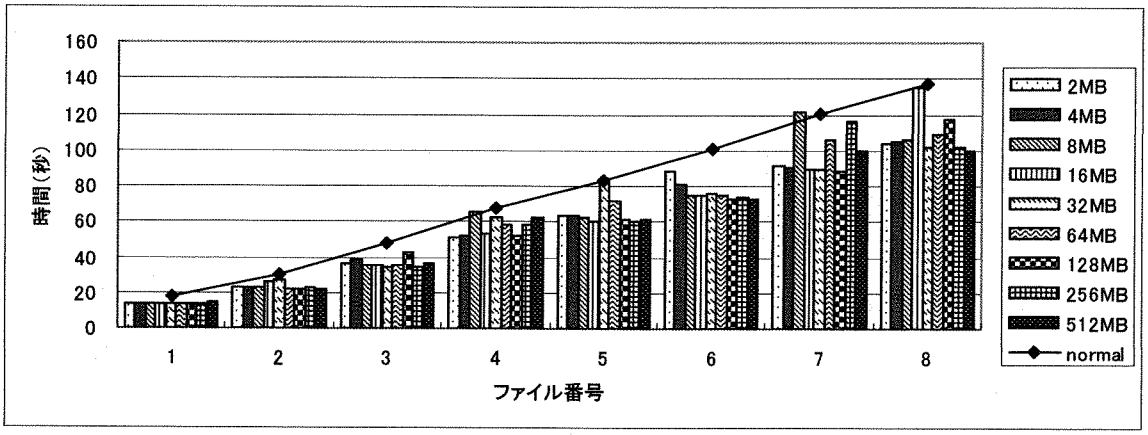

図 4 gzip 実行時間の比較 A-2

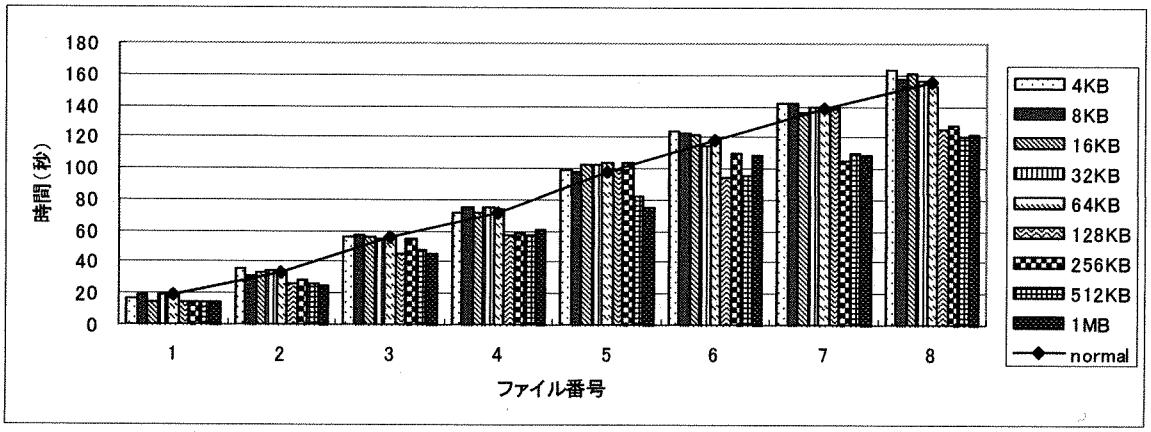

図 5 gzip 実行時間の比較 B-1

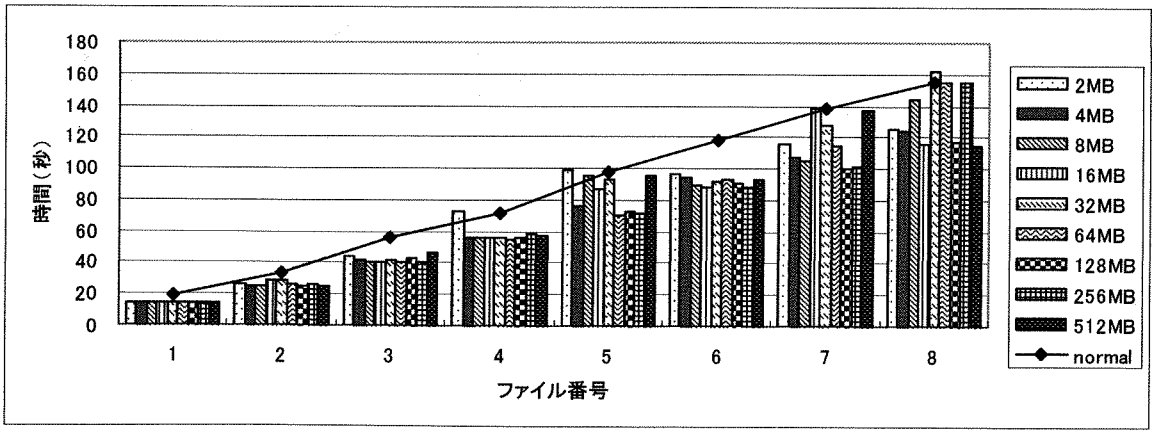

図 6 gzip 実行時間の比較 B-2

表 3 高速 gzip の性能向上率

File No.	標準 gzip A	A	A の向上率	標準 gzip B	B	B の向上率
1	17.31	13.03	32.9%	19.16	14.08	36.1%
2	29.65	21.54	37.6%	32.90	24.80	32.6%
3	47.66	34.55	37.9%	56.25	40.42	39.2%
4	67.11	50.56	32.7%	71.16	55.40	28.5%
5	82.80	60.30	37.3%	97.57	70.20	39.0%
6	101.04	72.79	38.8%	118.11	88.15	34.0%
7	120.04	88.65	35.4%	138.29	99.87	38.5%
8	136.99	100.02	37.0%	154.81	114.52	35.2%

実験環境 A では，先読み距離の制限値が大きくなるに従い，全体的に標準 gzip よりも短い時間で処理を終えていることが分かる．グラフより 256KB 以降に設定すると安定して性能向上を得られることが分かる．4KB～128KB のものに関しては，先読みする距離が短いため十分な性能向上が得られていないことが分かる．実験環境 A はメモリが 1GB あるので，図 4 より先読みの距離の制限値を 512MB に増やしてもディスクキャッシュの汚染が発生していないことが分かる．

実験環境 B でも同様に，標準 gzip よりも短い時間で処理を終えているのが分かる．図 5 より先読みの距離の制限値が 4KB～64KB のものに関しては，実験環境 A と同様に先読みする距離が短いため十分な性能向上が得られていないことが分かる．実験環境 B はメモリが 2GB あるので，図 6 より先読みの距離の制限値を 512MB に増やしてもディスクキャッシュの汚染が発生していないことが分かる．

表 3 に標準 gzip と高速 gzip の実行時間を示す．File No. は表 2 のファイル 1～8 に対応している．標準 gzip は標準 gzip で 4 回圧縮処理を行った際にかかった時間の最小値である．A と B はそれぞれ実験環境 A，B において高速 gzip で圧縮処理を行った際にかかった時間の最小値である．向上率はそれぞれ，標準 gzip の時間を高速 gzip の時間で割り 1 を引いた値になっている．正の向上率は，gzip の実行速度がそれだけ上昇したことを示している．逆に負の向上率は，gzip の実行速度がそれだけ低下していることを示している．

4.5 おわりに

従来から disk read ahead という考えはある．今回の実験環境 A には disk read ahead が実装されている．しかし，この disk read ahead はアプリケーションが現在読み込んでいる位置よりも，常に一定容量読み込むというものでは無い．近年 CPU のクロック数が頭打ちになり，コアが増える傾向にある．既存のプログラムでコアが増えることにより即座に性能が向上するものは少ない．今回は，余剰な計算資源を有効活用することにより，既存のプログラムを簡単に高速化する手法として先読みスレッドを提案した．本手法を gzip に実装することにより圧縮レベル 3 において，最大で 39.2%の性能向上を得ることができ，実験環境 A では平均して 36.2%の性能向上，実験環境 B では平均して 35.3%の性能向上を得

られることを確認した.

参考文献

[1] Randy H. Katz, G. A. Gibson, D. A. Patterson, "Disk System Architectures for High Performance Computing," Proceedings of the IEEE, Volume 77 (12), December 1989, pp. 1842-1858.

[2] David A. Patterson, Garth A. Gibson, Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD), Chicago IL, June 1988, pp. 109-116.

[3] Garth A. Gibson, R. Hugo Patterson, M. Satyanarayanan, "Disk Reads with DRAM Latency," Third Workshop on Workstation Operating Systems, Key Biscayne, FL, April, 1992, pp. 126-131.

[4] Garth A. Gibson, Redundant Disk Arrays: Reliable, Parallel Secondary Storage, Ph. D. Dissertation, University of California, Technical Report UCB/CSD 90/613, March 1991. To be published by MIT Press.

5．マルチコアプロセッサ上におけるシェルスクリプト高速化手法

5.1 はじめに

プロセッサの誕生以来，プロセッサ性能の向上は，主にクロック周波数の向上やパイプラインの細段化によってなされてきた．つまり，単位時間当たりの処理量，IPC (Instructions per clock)の向上を目指して，プロセッサ性能を向上させてきた．しかしIPCの向上のために，製造プロセスの微細化によるリーク電流の増加やトランジスタ数の増加により，消費電力や対コスト比の問題が無視できないものとなってきた．単純にプロセッサ性能を上げることができなくなってきたのである．

そのため，近年複数のプロセッサコアを１つのチップ上に集積するマルチコアプロセッサや，複数のスレッドを同時に処理するマルチスレッドの技術が注目されている．1999年，IBM社がデュアルコアプロセッサを搭載したIBMのPower4が発表されたことをきっかけに，その後様々なマルチコア対応のプロセッサおよびSMT（Simultaneous Multi-Threading）対応のプロセッサが発表された．Sun Microsystems社からは，UltraSPARC IV, UltraSPARC T1, Intel社からは，Pentium D, Coreなどが発表された．

しかし，並列性を考慮していないプログラムや通常のコンパイラを用いても，マルチコアプロセッサやマルチスレッドを有効活用することはできない．通常，マルチコアプロセッサやマルチスレッドの恩恵を受けるためには，並列化コンパイラによるプログラムの自動並列化[1, 2, 3, 4, 5, 6]，あるいはOpenMPやMPIなどを使用した並列化プログラミングが必要である．そのため，プログラムの並列化を実現するための研究が多数なされている．

一方，コマンドやシェルの組み込みコマンドなどを組み合わせてプログラミングを行うシェルスクリプトは，業務用バッチ処理などで多用されており，その高速化が望まれている．シェルスクリプト内の各コマンドはプロセスとして動作するため，パイプライン処理[7]を利用することで，並列化が容易に行える特性がある．すなわちシェルスクリプト実行時には，各コマンドがコアやスレッドへ適切に割り当てられることで，容易に処理性能の向上が図れる可能性がある．

そこで，本研究ではシェルスクリプト自体が持つ並列性に着目し，マルチコアプロセッサ・SMT (Simultaneous Multi-threading)環境における，シェルスクリプトの高速化を実現する手法を提案する．提案手法では，シェルスクリプト全体の実行効率を高めるために，スクリプトの細分化を行い，その細分化されたファイルをマルチコアプロセッサ上でマルチスレッディングの技術を用いて並列実行させることで，シェルスクリプトの高速化を実現する．本研究では，提案手法を適用したシェルスクリプトを，様々なマルチコアプロセッサ・SMT環境のマシン上で実行し，その処理効率を測定することで，マルチコアプロセッサ上でのシェルスクリプト実行の有効性を示す．

5.2 マルチコアプロセッサと SMT

本節では，近年注目されているマルチコアプロセッサと SMT 技術について触れる．

5.2.1 マルチコアプロセッサ

マルチコアプロセッサ（チップマルチプロセッサとも呼ぶ）とは，１チップ上に複数のプロセッサコアを搭載した CPU のことである．原理は従来のマルチプロセッサと同様，複数のコアで処理を分担する仕組みである．マルチコアプロセッサでは，内部のコア同士がキャッシュを共有しているものと共有していないものがあり，用途によっては性能を大きく左右することがある．例えば，共有キャッシュの場合，コア同士でまったく違うデータを参照していると，互いにキャッシュのデータを追い出してしまうことになり，性能が大きく低下する．しかし，互いに同じデータを使用すれば，メモリではなくキャッシュからデータを取り出すことができ，性能は大きく向上する．非共有キャッシュの場合は，逆の動作をすることになる．したがって，アプリケーションの使い方を考えて，キャッシュ共有／非共有のプロセッサを選ぶ必要がある．

5.2.2　SMT

SMT (Simultaneous Multi-Threading) とは，単一のプロセッサを複数のプロセッサに見せることで，マルチスレッドに対応した OS において，複数のスレッドを同時に扱うことができるようにする技術である．SMT に対応しているプロセッサを使用すると，OS からそのプロセッサは複数のプロセッサとして認識される．

5.3 シェルスクリプトと並列化

本節では，本研究の対象プログラムとなるシェルスクリプトについて説明する．また，シェルスクリプト自身が持つ並列性に着目し，シェルスクリプトの並列化についても言及する．

5.3.1 シェルスクリプト

シェルスクリプトとは，UNIX 等のコマンドやシェルの組み込みコマンドなど（以下「コマンド」と表記）を組み合わせてプログラミングを行い，実行できるようにしたものである．

シェルでは，プログラムの制御やファイルの操作などを簡単に行うことができ，その組み合わせによっては様々な動作を行うプログラムを書くことができる．用途としては，定期的にまとめて一括の処理を行う“バッチ処理”のプログラムを書く際に使用されることが多い．例えば，コンピュータ起動時の環境設定，企業における売上データや受注データの集計処理などに使われる．

5.3.2 パイプライン処理

シェルにおけるパイプライン処理とは，あるコマンドＡの処理結果をコマンドＢの入力として処理することである．シェルの実行時，左から右，あるいは上から下へ処理結果が流れることからパイプラインと呼ばれている．シェルでパイプライン処理を行うには，バーティカルバー（|）を用いる．具体的な書き方は，

COMMAND A | COMMAND B

のように，コマンドとコマンドの間にバーティカルバーを挿入することである．パイプライン処理における，コマンドＡからコマンドＢへの処理結果の受け渡しの様子を図 1 に示す．

コマンドＡに対応するプロセス＃１とコマンドＢに対応するプロセス＃２はほぼ同時に起動する．そして，各プロセスは次に説明する標準入出力命令等で待ちが発生しない限り，同時に実行される．プロセス＃１は標準出力命令を読み込むと，write()命令によりカーネル内のバッファに対して書き込みを行う．また，プロセス＃２は標準入力命令を読み込むと，read()命令によりカーネル内のバッファから読み込みを行う．その際，プロセス＃１による書き込みとプロセス＃２による読み込みは，同期的に行われる．したがって，プロセス＃１の書き込みがバッファ容量（通常数ＫＢ，ただしシステムにより異なる）を超えると，プロセス＃１は中断される．逆に，バッファに残るデータが０になると，プロセス＃２は中断される．

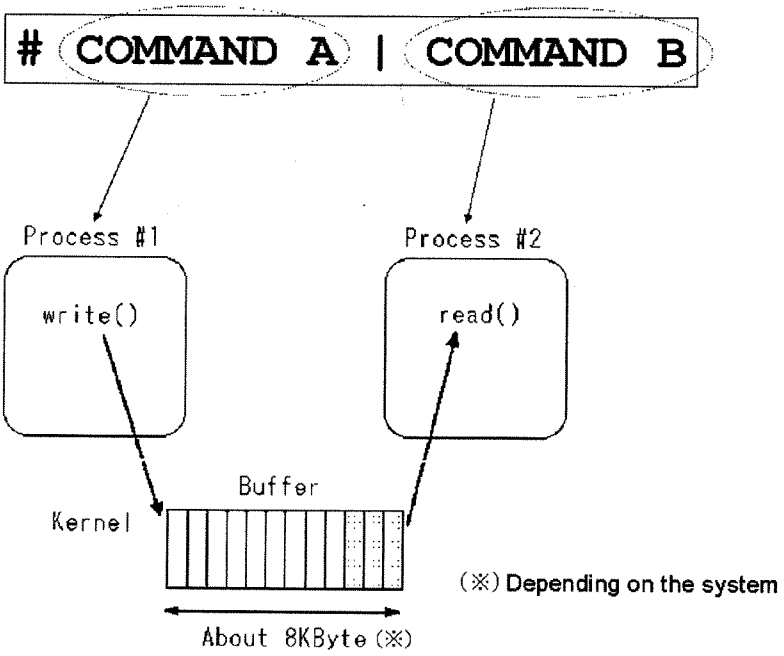

図１　パイプライン処理

マルチコア・SMT プロセッサ上において，プロセス＃１とプロセス＃２は各コアまたはスレッドに割り当てられ，ほぼ同時に実行することができる．そのため，パイプライン処理が張り巡らされたプログラムを，マルチコア・SMT プロセッサ上で動作させることで，コアやスレッドの有効活用をすることができる．

5.4 提案手法：シェルスクリプトの自動並列化

本節では，シェルスクリプトの実行効率を高めるための手法について説明する．

前節で述べたように，シェルスクリプト中にパイプライン処理が書かれている場合，マルチコア・SMT プロセッサ上においてはその処理は並列実行される．そのため，C 言語や JAVA のプログラムのように，シングルコアからの移植の際にプログラマが並列化プログラミングを意識する必要はない．しかし，並列実行される部分はパイプライン処理，あるいはバックグラウンド処理のある部分のみとなるため，プログラムによっては数%～数十%のスピードアップしか見込めない場合もある．

そこで，本研究ではスクリプトを複数のファイルに分割し，分割ファイルを同時実行することで，全体の処理効率を高めるための手法を提案し，その手法を実現するプログラムを開発した．

5.4.1 提案手法イメージ

提案手法の全体のイメージを図 2 に示す．提案手法では，対象となるシェルスクリプトを読み込み，一定のルールに従いスクリプトファイルの分割を行う．ファイル分割後，分割ファイル同士の依存関係を考慮しながら，分割ファイルの実行順序を決定する．その際，相互に依存関係がない分割ファイル同士は，パイプライン処理を利用した同時実行ができるようにする．ユーザは，最終的に出力されるシェルスクリプトを実行するだけで，元のスクリプトを実行するよりも短時間で効率よく，スクリプトの結果を得ることができる．

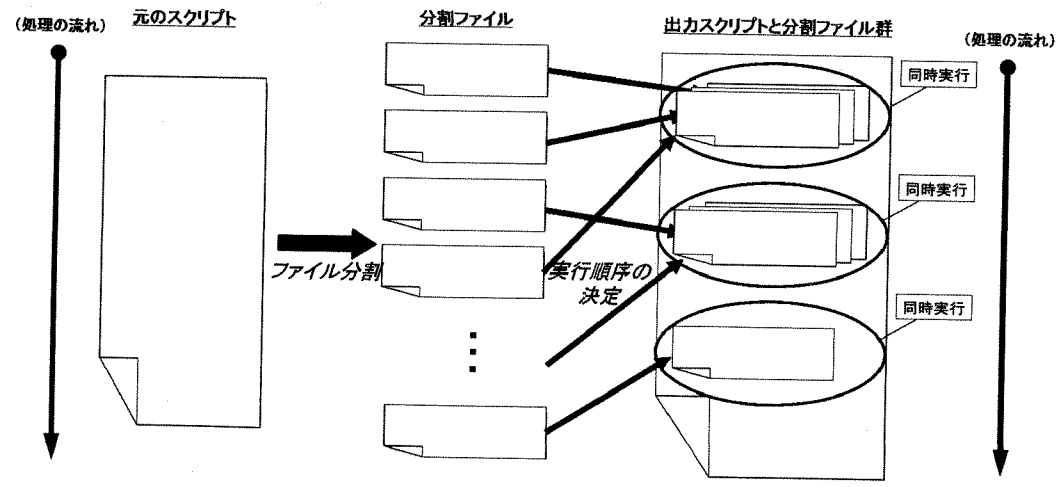

図２　提案手法のイメージ

5.4.2 提案手法詳細

提案手法の全体の流れを図3に示す．はじめに，対象となるシェルスクリプトを指定し，ファイルの読み込みを開始する．ファイルの読み込みは1文字ずつ行い，空白や改行，その他指定文字（# や > など）を区切りにして文字列を取得し，以下の処理を行う．

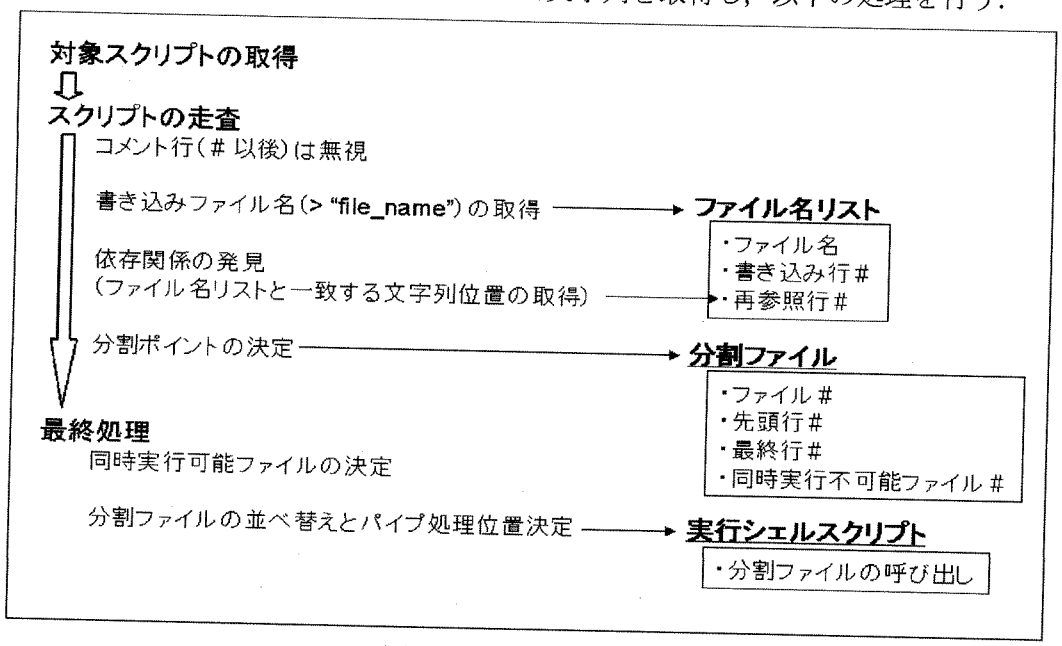

図3　提案手法の流れ

1. リダイレクション（>, >>）による書き込みの発生するファイルの名前と行番号を取得し，ファイル名リストに加える．
2. ファイル名リストと一致する文字列を検索し，ファイルの再参照による依存関係を調べる．
3. 【分割ルール】に従いファイルの分割ポイントを決定し，ファイルを分割する．その際，同時に実行が不可能なファイルの番号も記憶しておく．
4. 分割ファイルの同時実行不可能ファイル番号を参照し，同時実行可能ファイルを選び出し，分割ファイルの実行順序を決定する．
5. 変換後の実行シェルスクリプトとして，分割ファイルの実行順序を記した主スクリプトと，元のファイルを分割した分割スクリプトを生成する．

【分割ルール】

- 基本的には，ファイルへの書き込みが行われたところまでをひとまとまりとし，その行を分割ポイントとする．
- ただし，例外として次の制御文中でファイルの書き込みが行われても分割を行わず，その制御文の終端行を分割ポイントとする．

1. if 文（if ～ fi 部分）
2. for 文（for ～ done 部分）
3. while 文（while ～ done 部分）
4. ヒアドキュメント書き込み中（<< "name" ～ "name" 部分）

5.5 評価実験

本節では，複数のマルチコアプロセッサ・SMT (Simultaneous Multi-Threading) マシン上での，シェルスクリプト実行結果を示す．実験の目的は，マルチコア・SMT プロセッサ上でのシェルスクリプト実行の有用性，および提案手法である「シェルスクリプトの自動並列化」の有効性を示すことである．

5.5.1 実験環境

本実験で用いたマシンの構成を表 1 に示す．

表 1　マシン構成

名称	SPARC T1	SPARC IV+	power5	Xeon 共有	Xeon 非共有	Opteron
マシン名	Sun Fire T2000	Sun Fire V490	IBM p5	Dell Precision 490	Dell Precision 490	PowerEdge SC1435
CPU	UltraSPARC T1 1.2GHz	UltraSPARC IV+ 1.5GHz	Power5 1.65GHz	Xeon 5110 1.6GHz	Xeon 5060 3.2GHz	Opteron 1.8GHz
実 CPU 数	1	2	1	2	2	2
論理 CPU 数	32	4	4	4	8	4
キャッシュ共有	あり	あり	あり	あり	なし	なし
メモリ量	16GB	16GB	2GB	16GB	16GB	4GB
HDD	SAS 10000rpm 73GB× 2	FC 10000rpm 146GB× 2	Ultra320 SCSI 10000rpm 73GB	Serial ATA 7200rpm 80GB	Serial ATA 7200rpm 80GB	SAS 15000rpm 73GB
パイプサイズ	5KB	5KB	32KB	4KB	4KB	8KB
OS	Solaris 10	Solaris10	AIX 5L	Red Hat Enterprise linux 4 WS	Red Hat Enterprise linux 4 WS	Red Hat Enterprise linux 4 WS

5.5.2 評価実験

実験は，業務処理プログラムを対象に行った．なお，実行時間の測定には bash の time コマンドを使用した．特に断りがない限り，実行時間は REAL の示す値とする．

【time コマンド】
・REAL：コマンド実行の実時間
・USER：ユーザ CPU 使用時間（コア総計）
・SYSTEM：システム CPU 使用時間（コア総計）

業務処理プログラム実験の目的は，実際の企業データとバッチ処理プログラムを用いたとき，マルチコア・SMT の恩恵をどの程度受けられるのかを調べることである．

業務処理プログラムは，データの解凍やテキスト変換などの基幹処理および帳票の作成をするプログラムである．プログラムはすべてシェルスクリプトで書かれている．業務処理プログラムは，大きく分けて次の３つのパートに分けられており，(1)，(2)，(3) の順番で実行される．

(1)　汎用データのテキスト形式変換および分割・ソート
(DATAMASTER，111 行)
ホストコンピュータで作成された汎用データを解凍し，EUC テキストファイルに変換する．その後，販売情報，在庫情報，仕入情報，初回仕入情報，物流センター間振替情報，各店舗間振替情報，差益情報，逆伝票情報に分解し，各生成ファイルをソートする．

(2)　商品勘定実績部門別帳票作成（KANJO，1086 行）
各生成ファイルを集計，編集して，販売チャネル毎の各項目の予算比，昨年比を求め，エクセル形式の一帳票 (勘定実績表) にまとめる．

(3)　商品勘定実績部門別詳細帳票作成（KANJO.DEPA，3324 行）
上記の勘定実績表（KANJO）の各商品部門版を作成する．

今回の実験では，株式会社良品計画の実データを使用した．元データのファイルサイズは約１５０ＭＢ，解凍後に扱う総データは約４ＧＢである．また，最終的に出力される帳票データは約２５０ＫＢとなる．シングルコア，マルチコア・SMT（自動並列化適用前），マルチコア・SMT（自動並列化適用後）における，スクリプトの処理の実行時間を図 4 に示す．自動並列化プログラムに通すことで，各スクリプトは以下の数のファイルに分割された．

- DATAMASTER：27 ファイル
- KANJO：80 ファイル
- KANJO.DEPA：188 ファイル

これらの分割ファイルを，相互の依存関係を侵さないように同時実行する．同時実行ファイルは最高で DATAMASTER プログラムが 18，KANJO プログラムが 24，KANJO.DEPA プログラムが 70 ファイルになる．

実験の結果，通常のシェルスクリプトをマルチコア・SMT プロセッサ上で動作させるだけで，並列効果を得ることができた．また，自動並列化プログラムを使用すると，さらに並列効果を得ることができることが分かった．

DATAMASTER プログラムでは，もともと処理の重い部分がパイプライン処理で記述されているため，自動並列化プログラム適用前でも，平均使用論理 CPU 数が 1.5～3.1 と並列効果が高い．また，自動並列化プログラム適用後は，平均使用論理 CPU 数 1.5～3.5 となり，Power5 を除いて実行時間ベースで 1.2～1.5 倍のスピードアップが図れた．なお，Power5 は途中でメモリエラーが生じ，自動並列化プログラムの恩恵を受けることができなかった．これは，Power5 のみメモリ容量が 2GB だったため，十分なメモリを確保できなかったことに起因する．

KANJO プログラム，KANJO.DEPA プログラムでは，自動並列化プログラム適用前は，平均使用論理 CPU 数が 1～1.3 と，ほとんど並列効果が現れなかった．しかし，自動並列化プログラム使用後は平均使用論理 CPU 数が 2～3.7 にまでなり，実行時間ベースで 1.8～3.0 倍のスピードアップが図れた．

マシン間で比較すると，論理 CPU 数の多い SPARC T1 と Xeon NetBurst マシンにおいて，KANJO プログラムの平均使用 CPU 数がそれぞれ 3.6，3.1，KANJO.DEPA プログラムそれぞれ 3.7，3.1 となり，高い並列効果が得られた．

業務処理プログラム全体の実行効率を見ると，シェルスクリプトの自動並列化により，シングルコアに比べて，SPARC T1 では 3.7 倍，SPARC IV+では 2.6 倍，Power5 では 1.8 倍，Xeon Core では 2.7 倍，Xeon NetBurst では 3.2 倍，Opteron では 2.6 倍のスピードアップが図れた．また，シェルスクリプトの自動並列化適用前と比べても，SPARC T1 では 1.8 倍，SPARC IV+では 1.6 倍，Power5 では 1.4 倍，Xeon Core では 1.7 倍，Xeon NetBurst では 1.8 倍，Opteron では 1.6 倍のスピードアップが図れた．

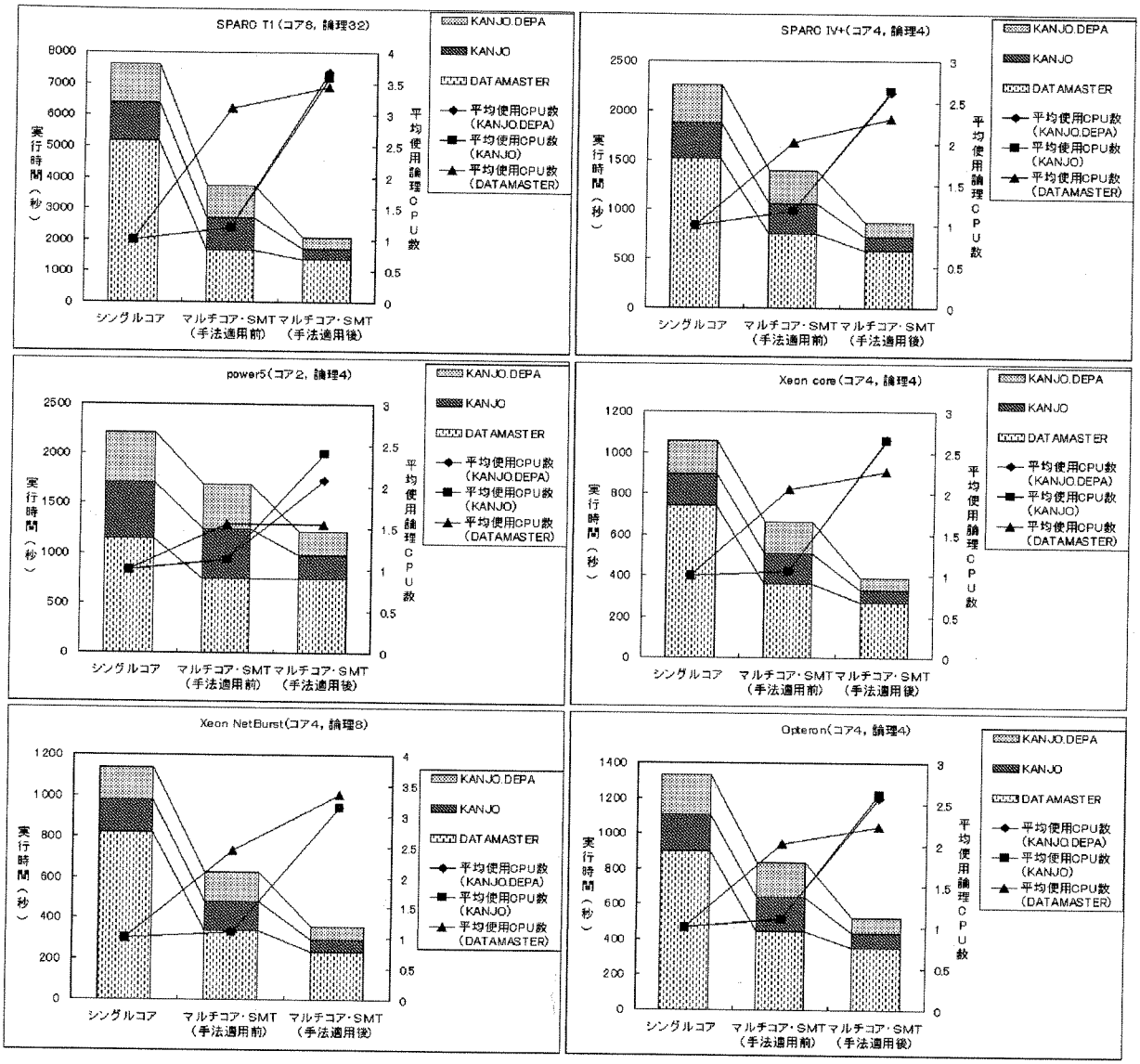

図４　業務処理プログラム実行結果

5.6 関連研究

本研究は，シェルスクリプトを対象にした自動並列化の技術である．プログラムの自動並列化コンパイラについては，現在までに様々な研究がなされている．

自動並列化コンパイラの歴史は，１９７０年代に始まった．プログラムの実行時間の多くがループ処理であるという観点から，自動並列化コンパイラの研究は，長らくループ処理の並列化を中心に行われた[1, 2, 3]．しかし，ループ並列化の研究は，３０年にわたる研究によりほぼ成熟期に達しており，今後の大幅な性能向上は難しいと言われている[4]．そこで，近年自動並列化コンパイラの研究では，ループ並列化に加えて新たな並列化手法を組み込んだ研究がなされている．

[4, 5, 6]は，マクロデータフロー処理の技術を並列化コンパイラに導入している．マクロ

データフロー処理とは，単一プログラム中のループ，サブルーチン，ブロック間の並列性を利用した，従来の並列処理よりも粗粒度の（マクロ）並列処理である．[4]では，RB (Repetition Block)，SB (Subroutine Block)，BPA (Block of Pseudo Assignment Statements)の３種類のマクロタスクを生成する．RBは最外殻ループ，SBはサブルーチン，BPAは並列性を考慮して融合または分割されたブロックである．これらの生成されたマクロタスク間の依存関係を解析し，並列性を考慮しながら実行コードを生成する．

また，[4]はメモリ利用の最適化技術を導入している．マルチコア・SMTプロセッサでは，メモリアクセスのレイテンシをカバーするためにキャッシュの有効利用が必要である．プログラムの解析時，データがキャッシュに収まるようにループとデータを分割する．また，一度キャッシュに載せたデータを複数のループで使用できるように，実行順序を調整する．

5.7 おわりに

本研究では，シェルスクリプト自身が持つ並列性に着目し，マルチコア・SMTプロセッサ環境において，シェルスクリプトの高速化を実現する手法，シェルスクリプトの自動並列化プログラムを提案した．その結果，マルチコア・SMTプロセッサ上での業務処理プログラム実行において，提案手法適用前に比べて，1.4～1.8倍のスピードアップが確認できた．

参考文献

[1] Z. Li, P. Yew, and C. Zhu: An Efficient Data Dependence Analysis for Parallelizing Compilers, In IEEE Trans. of Parallel and Distributed Systems, Vol. 1, No. 1, pp. 26-34, 1990.

[2] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S-W. Liao, C-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy: SUIF: an infrastructure for research on parallelizing and optimizing compilers, ACM SIGPLAN Notices, pp31-37, 1994.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu: Advanced Program Restructuring for High-Performance Computers with Polaris, IEEE Computer, pp. 78-82, 1996.

[4] H. Kasahara, M. Obata, K. Ishizaka, K. Kimura, H. Kaminaga, H. Nakano, K. Nagasawa, A. Murai, H. Itagaki, J. Shirako: Multigrain automatic parallelization in Japanese Millennium Project IT21. Advanced Parallelizing Compiler, In IEEE International Conf. on Parallel Computing in Electrical Engineering, pp. 105-111, 2002.

[5] H. Tanabe, H. Honda and T. Yuba: Macro-Dataflow using Software Distributed Shared Memory, IEEE International Conf. on Cluster Computing, 2005.

[6] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in SUIF," In ACM Trans. on Programming Languages and Systems, Vol. 27, No. 4, pp. 662-731, 2005

[7] Cory Isaacson, "Software Pipelines - An Overview," on White Paper, Rogue Wave Software.

６．おわりに

本研究では，近年のマルチコア CPU の効果的な利用を行う手法として，ヘルパースレッドを用いた各種効率化手法について研究を進めた．１年目でターゲットを絞り，２年目では，ディスクアクセスの高速化をターゲットに研究を進めた．

まず，DISK の先読みにヘルパースレッドを用いる例では，先読みスレッドで事前にデータの読み込み込む手法をヘルパースレッドを用いて提案し，gzip が最大で 39.2%性能向上することを確認した．次に，DISK キャッシュ自体をネットワーク接続された他の PC 上に置き，ヘルパースレッドによりディスクキャッシュを制御する高速化手法を提案した．DBT-3 ベンチマークテストによる評価では，最大 3.08 倍の性能向上を確認した．さらに，実アプリケーションとして，シェルスクリプト実行の高速化を実現した．シェルスクリプトは，旧来，小さなアプリケーションで多用されるものであるが，近年の CPU 高速化に伴い，業務処理に多用されるようになってきている．実際に大手小売店が利用している．本研究では，このようなシェルスクリプトの高速化を目指し，シェルスクリプトの自動並列化プログラムを提案した．そして，これらをヘルパースレッドで実行することにより，シェルスクリプト実行を 1.4～1.8 倍高速化することができた．

なお，本研究成果は，USP 研究所（http://www.usp-lab.com/）において製品化を目指して検討を進めている．