メモリ階層制御により高性能・低消費電力を実現するプログラムの開発の生産性を高めるシステムソフトウェアに関する研究

城田　祐介

電気通信大学大学院情報システム学研究科

博士（工学）の学位申請論文

2017 年 3 月

メモリ階層制御により高性能・低消費電力を実現するプログラムの開発の生産性を高めるシステムソフトウェアに関する研究

博士論文審査委員会

主査　本多　弘樹　教授

委員　吉永　努　教授

委員　南　泰浩　教授

委員　大須賀　昭彦　教授

委員　三輪　忍　准教授

A Study on System Software Utilizing Memory Hierarchy Control Methods for Increasing Productivity of High-performance and Energy-efficient Programs

Yusuke Shirota

Abstract

With the rapidly growing demands for large-scale data processing, building high-performance and energy-efficient computer systems featuring scalable memory systems is increasingly becoming an important issue. To efficiently process such large-scale data, processors and memories are aggregated with computer clustering technologies utilizing fast interconnects and networks to realize application performance superior to that achievable on a single processor system, and consequently, memory hierarchies get deep and complex. Therefore, memory hierarchy control method for adapting programs to such memory hierarchies is the key for realizing high-performance and energy-efficiency.

In the meantime, new type non-volatile memories, or Storage-class Memories (SCMs), including MRAM (Magnetoresistive Random Access Memory), PCM (Phase-Change Memory) and ReRAM (Resistive Random Access Memory) are emerging. Storage-class memories can reduce access latency by orders of magnitude compared to state-of-the-art non-volatile devices such as SSDs/HDDs, thus have the potential to drastically change existing memory hierarchies and dramatically enhance performance and energy-efficiency of computer systems. Therefore, novel software techniques for non-volatile memories are imperative to fully extract such potential.

However, programming the deep and complex memory hierarchy compels application programmers to be equipped with non-trivial knowledge of both the underlying memory architecture and black-belt programming techniques. To improve software development productivity and to cope with the increasing complexity of the underlying memory hierarchies, designing and developing memory hierarchy aware system software is imperative.

Therefore, the aim of this study is to increase productivity of high-performance and energy-efficient programs by the following four system software methods utilizing automatic memory hierarchy control methods, and abstract application programmers

away from memory hierarchies:

1) Programming system based on memory architecture independent, algorithm description language dedicated to array processing capable of automatically generating memory hierarchy aware parallel programs.

2) Software distributed memory systems for multi-clusters, or aggregate computer clusters, utilizing memory hierarchy aware multi-home cache coherence scheme for exploiting cluster data locality to reduce inter-cluster traffic and hiding inter-cluster latency.

3) SCM-aware low power virtual memory system which aggressively pages out data from DRAM to SCM-based swap device, and minimizes DRAM size to the extent of acceptable performance degradation attributed to swapping overhead, and reduces DRAM background power by powering off unused space.

4) Electronic paper display update scheduler for dynamically localizing memory access and display updates in extremely low power embedded systems with non-volatile memories.

メモリ階層制御により高性能・低消費電力を実現するプログラムの開発の生産性を高めるシステムソフトウェアに関する研究

城田　祐介

概要

近年、アプリケーションの処理するデータの大規模化に伴い、大規模データを効率良くプロセッサに供給することが可能なメモリシステムを持つコンピュータシステムの高性能化および省電力化の重要性が高まっている。単一プロセッサの処理性能には限界があるため、大規模データを処理するためには高速ネットワークで複数のプロセッサを結合しクラスタリングしていくことが必要になり、その過程でコンピュータシステム内の様々なレベルにメモリが複雑に階層化される。大規模データ処理を高速に実行可能なコンピュータシステムを目指すには、メモリの階層構造に対してプログラムを適応させるためのソフトウェアによる制御方式が鍵となる。

一方で、実用化が期待されているMRAM (Magnetoresistive Random Access Memory)やPCM (Phase-Change Memory)あるいはReRAM (Resistive Random Access Memory)などのストレージクラスメモリ(Storage-class Memory)と呼ばれるこれまでと異なるアクセスレイテンシや性能特性を持つ新型高速不揮発メモリによりメモリ階層も大きく変化し、コンピュータシステムを飛躍的に高性能化・低消費電力化できる可能性がある。そのため、巨大メモリ搭載大規模サーバシステムだけでなく超低消費電力が要求される省電力組込みシステムなどの幅広いコンピュータシステムにおいてメモリ階層の変化に適応するソフトウェア技法も必要になる。

しかし、複数のプロセッサのクラスタリングやストレージクラスメモリの実用化によって性能や消費電力の観点で複雑化するメモリ階層を効率良く制御しコンピュータシステムのハードウェア性能を引き出すためのプログラミングには、メモリアーキテクチャの深い知識と高度なプログラミング技術を要求する。このような作業をすべてのアプリケーション開発者に要求するのは現実的ではなく、システムソフトウェアで自動化し解決すべきで

ある。

本研究はこのような背景を踏まえて、メモリ階層制御により高性能化・低消費電力化を実現するプログラムの開発生産性向上の達成をシステムソフトウェアにより実現することが主題である。この主題に対して本研究が狙うのは、つぎに列挙する 4 つのシステムソフトウェアを導入しメモリ階層制御を自動化することでアプリケーションのプログラム開発の容易性を向上させたうえでハードウェアの持つポテンシャルに近い性能を引き出すことである。

1) **高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム:**

高性能サーバなどの単一計算ノードあるいは組込みシステムにおいて、プロセッサの性能を引き出すために必要になるメモリアクセス最適化やマルチスレッド化やベクトル化などの並列プログラミングには、対象メモリアーキテクチャに関する深い理解とそれを活かすプログラミング技法が要求される。しかし、ソフトウェア開発の中でもアルゴリズム開発を主に行っているアプリケーション開発者にとって並列プログラミングは開発の本質ではなく、メモリやプロセッサのアーキテクチャの深い知識を持たなくても対象アーキテクチャを活かした並列プログラムを開発可能なプログラミングシステムが必要である。そこで本研究では、配列処理に特化した配列処理言語を用いてアルゴリズムレベルで記述可能なプログラミングシステムを提案している。配列処理プログラムに明示されるアルゴリズムレベル情報を利用した高位プログラム変換によりアルゴリズムレベル記述から階層メモリに適合する並列プログラムが自動生成でき、開発の生産性を高めることができることを示す。

2) **階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア分散共有メモリシステム:**

単一計算ノードを高速ネットワークで複数接続したハイパフォーマンスコンピューティング向けの計算機クラスタシステムや複数の計算機クラスタシステムを繋げたマルチクラスタシステムなどの階層的な分散メモリを持つ高性能コンピュータシステムにおいては、並列プログラムの作成に分散メモリ型並列プログラミングモデルであるメッセージパッシング方式が多く用いられるが、分散メモリ間の通信を明示的に記述する必要がありプログラミングが容易ではない。この問題を解決する方法として分散メモリ上に仮想的な共有メモリを実現するソフトウェア分散共有メモリ方式があるが、従来型のデータ一貫性制御方式はメモリ階層を意識した設計になっておらずそのままマルチクラスタシステムに適用すると、要素クラスタ間通信遅延等により性能が低下する。そこで本研究では、メモリ階層に適合したデータ一貫性制御方式を提案し、プログラミングしやすさと高性能を同時達成できることを示す。

3) **大容量新型不揮発メモリを活用した階層型主記憶を実現する省電力仮想記憶システム:**

次世代の高性能コンピュータシステムにおけるインメモリ処理で要求されている低消費電力でかつスケーラブルな主記憶を実現するためには、大容量のストレージクラスメモリとDRAMの特性が異なる2つのメモリを組み合わせ階層制御する必要がある。2つのメモリをアプリケーション開発者に陽に見せるとこれらをどのように使い分けるかを開発者に強いることになりプログラミングが複雑になる。そこで本研究では、ストレージクラスメモリ向けにOSの仮想記憶システムを再設計することで大容量なストレージクラスメモリをDRAMと混載し、待機消費電力が小さいストレージクラスメモリの特性を活かしてDRAM上のデータを積極的にストレージクラスメモリに退避して、使用するDRAMサイズを削減するとともに未使用DRAMの電源をオフすることで動作時の消費電力を削減する。仮想記憶システムでDRAMとストレージクラスメモリ間のデータの入れ替え処理であるスワップ処理を効率良くおこなうことで、アプリケーション開発者には大容量な主記憶があるように見せることができるためインメモリデータ処理向けのプログラミングをシンプルにすることが可能になることを示す。

4) **不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ:**

組込み機器特有の課題であるディスプレイを有する組込みシステムの省電力化を実現する。現在のタブレット型端末はDRAMが主記憶として利用されているが、ストレージクラスメモリが実用化されると、待機消費電力が非常に小さい超低消費電力タブレット型端末が実現可能になってくる。ただし主記憶が不揮発でもディスプレイがLCD(液晶ディスプレイ)などの従来型揮発性ディスプレイの場合頻繁なリフレッシュが必要となるため、リフレッシュ処理を実行するディスプレイコントローラが利用する主記憶を電源オフすることができない。そこで本研究では、ディスプレイを有する組込みシステムの省電力化を、不揮発メモリと不揮発ディスプレイである電子ペーパを組み合わせることで実現する。低消費電力を実現するための省電力メモリ制御機能を組み込んだ電子ペーパコントローラ向けデバイスドライバを提案する。不揮発メモリと省電力ディスプレイの省電力性を引き出すためには、DRAMとLCDを利用したこれまでとは異なる複雑なデバイス制御が必要となるが、提案方式により従来型のプログラミング方式を変えずに省電力実行可能なことを示す。

本研究の貢献は、コンピュータシステムにおける本質的な課題である効率的なメモリ階層制御の実現についてその解決方式をメモリ階層の様々なポイントで示した点である。本研究成果は、ストレージクラスメモリの実用化によりメモリ階層が大きく変化しメモリシステムがプロセッサに替わって中心になる次世代コンピュータアーキテクチャとそのシス

テムソフトウェアが取り組むべき課題とその解決方法の方向性を示すものとして意義がある。

目次

1. はじめに.....1

1.1 高性能・省電力コンピュータシステムにおけるメモリ階層制御の課題.....1
1.2 メモリ階層に変化をもたらす新型高速不揮発メモリ.....3
1.3 メモリ階層制御により高性能・低消費電力を実現するプログラムの開発の生産性を高めるシステムソフトウェア.....5
1.4 高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム.....7
1.5 階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア分散共有メモリシステム.....9
1.6 新型高速不揮発メモリを活用した階層型主記憶を省電力制御する仮想記憶システム.11
1.7 新型高速不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ.13
1.8 本論文の構成.....15

2. 高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム.....17

2.1 まえがき.....17
2.2 配列処理言語を利用した並列プログラム開発.....18
2.2.1 配列処理に特化したアルゴリズム記述言語.....18
2.2.2 近傍処理プログラムの記述例.....20
2.2.3 配列処理言語の特徴.....21
2.2.4 配列処理言語の処理系.....21
2.3 ベクトル化向け高位プログラム変換方式.....22
2.3.1 ベクトル化の課題.....22
2.3.2 近傍情報を利用したベクトル化向け高位プログラム変換方式.....24
2.3.3 性能評価.....29
2.4 高位プログラム変換を利用した自動チューニングによる並列Cプログラム生成.....33
2.4.1 自動チューニングによる並列Cプログラム開発とその課題.....33
2.4.2 高位プログラム変換を用いた自動チューニング(1).....34
2.4.3 高位プログラム変換を用いた自動チューニング(2).....35
2.5 関連研究.....39
2.6 まとめと今後の課題.....39

3. 階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア

分散共有メモリシステム.....41

3.1 MIGRATORY ACCESS を効率良く処理する権限委譲プロトコルを組み込んだホームベースソフトウェア分散共有メモリ 41
3.1.1 まえがき 41
3.1.2 従来のホームベースプロトコルの課題 43
3.1.3 MIGRATORY ACCESS を効率良く処理するホームベースソフトウェア分散共有メモリ 45
3.1.4 評価 50
3.1.5 関連研究 58
3.1.6 まとめと今後の課題 59
3.2 マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ 60
3.2.1 まえがき 60
3.2.2 マルチクラスタ上で既存ソフトウェア分散共有メモリ方式を用いる問題点 62
3.2.3 マルチホーム方式の提案 63
3.2.4 マルチホーム方式の実装 65
3.2.5 実機を利用したマルチクラスタシステムシミュレータの実装 68
3.2.6 予備評価 70
3.2.7 関連研究 75
3.2.8 まとめと今後の課題 76

4. 新型高速不揮発メモリを活用した階層型主記憶を実現する省電力仮想記憶システム.....77

4.1 まえがき 77
4.2 新型高速不揮発メモリ（ストレージクラスメモリ） 77
4.3 積極的にストレージクラスメモリへ退避する省電力仮想記憶方式の提案 78
4.4 フルシステムシミュレーションによる評価 80
4.4.1 評価の目的 80
4.4.2 フルシステムシミュレーション評価環境 80
4.4.3 ストレージクラスメモリのアクセスレイテンシが実行時間に及ぼす影響の評価 82
4.4.4 ストレージクラスメモリのアクセス電力が消費電力に及ぼす影響の評価 87
4.5 関連研究 91
4.6 まとめと今後の課題 93

5. 新型高速不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ.....95

5.1 まえがき 95
5.2 電子ペーパ書換え処理の概要 97
5.3 電子ペーパ書換え処理の課題 99
5.3.1 断続的に発行される書換え命令による書換え処理時間の増加 99
5.3.2 コリジョンによる書換え処理時間の増加 100

5.4 電子ペーパ書換え処理の省電力制御方式 101
5.4.1 EPD スケジューラの基本アルゴリズム 101
5.4.2 EPD スケジューラの待ち時間の決定方法 103
5.5 EPD スケジューラの性能評価 105
5.5.1 評価環境 105
5.5.2 EPD スケジューラの省電力効果の評価 105
5.5.3 待ち時間の自動調整手法の評価 109
5.6 省電力書換え制御方式の他デバイスへの適用可能性 110
5.7 まとめと今後の課題 111
6. 結論 113
6.1 研究成果の概要 113
6.2 今後の課題 114
謝辞 119
参考文献 121
関連論文の印刷公表の方法及び時期 133
査読付き参考論文の印刷公表の方法及び時期 135
登録特許の印刷公表の方法及び時期 137
参考口頭発表論文の印刷公表の方法及び時期 139
著者略歴 141

1. はじめに

1.1　高性能・省電力コンピュータシステムにおけるメモリ階層制御の課題

近年、大規模ニューラルネットワークを利用して人工知能(AI)を実現するディープラーニング、各種センサや監視カメラ等により生成される爆発的な量のデータを解析するビッグデータ処理やオンラインリアルタイム処理、SNS 解析や遺伝子解析や創薬等の応用で必要となる超大規模グラフ処理などに代表されるように様々な分野のアプリケーションで処理するデータの大規模化が進んでいる。これに伴い、プロセッサに大規模データを高速に供給することが可能なメモリシステムを持つ高性能コンピュータシステムの要求が高まっている[28,29,40]。そのため、高性能コンピュータシステムの研究の大きなフォーカスは、プロセッサからメモリシステムに移ってきている[128,130]。

データセンターにおける巨大メモリ搭載大規模サーバなどのコンピュータシステムにおいては高性能に加えて低消費電力がもう一つの最重要課題になってきている[16,21-23,25]。データの大規模化が進むと、大規模データを格納するメモリシステムの消費電力がコンピュータシステム全体の消費電力に占める割合が大きくなるため、メモリシステムの省電力化は重要な課題となっている[17-20,24]。巨大メモリ搭載大規模サーバのなかには DRAM で構成される主記憶の消費電力がノードの消費電力の半分に及ぶものも出てきている [65]。データの大規模化は今後も進みこの傾向は強まっていくと予想される[10]。

データの大容量化に伴う低消費電力化は、サーバシステムなどの高性能コンピュータシステムに限らず、高解像度化および高機能化のために高性能化が進むタブレット型端末やイメージセンサから常時入力される大量のデータをリアルタイムに処理する IoT(Internet of Things)機器やウェアラブル型コンピュータのような組込み機器でも重要な課題である。タブレット型端末やウェアラブル型コンピュータでは、搭載するバッテリ容量が製品の大きな差異化要因になる機器の重量にも影響するため、低消費電力化の要求は高まる一方である。このように、幅広い分野のコンピュータシステムにおいて、メモリシステムの高性能化と低消費電力化は今後ますます重要な課題になる。

データセンターにおける大規模データを扱うサーバシステムの構築法を例に、高性能コンピュータにおけるメモリシステムの課題をブレークダウンする。一般に、現在の高性能コンピュータシステムの構成は、大規模データの高速処理を可能にするために、図 1.1 に示すような 3 つのステップにより実現される。

より高性能なデータ処理が可能なコンピュータシステムを目指すための第 1 ステップは、

まずは単一プロセッサとそのメモリシステムの高速性と省電力性を引き出すことである。図 1.1(a)に示すような単一プロセッサシステムにおいては、対象アプリケーションあるいはアルゴリズムが内包する並列性を抽出し、プロセッサが持つ複数のプロセッサコア（マルチコア）や複数データを同時に処理可能な SIMD(Single Instruction Multiple Data)型の命令や GPGPU(General-Purpose computing on Graphics Processing Unit)などのアクセラレータ型並列処理演算器に効率的に写像して処理を高速に実行することが必要となる。また、電力効率の高い並列処理演算器で処理を高速に実行し終えることで、メモリシステムやプロセッサをいち早く低消費電力状態に遷移させることができるため処理に必要な電力量を削減する効果もある[112]。これらのアクセラレータ型並列処理演算器の高速性と省電力性を引き出すためには、単一プロセッサシステムのメモリ階層を効率良く制御して大量のデータをにいかに供給し続けることができるかが鍵であり、SIMD 命令活用の最適化であるベクトル化もメモリシステムからのデータ待ち時間が支配的なプログラムに施しても効果がほとんどなく、メモリ階層にプログラムを適応させてキャッシュのヒット率を向上させるメモリアクセス最適化を同時におこなうのが前提となる[84]。

具体的には、プロセッサの各コアと、プロセッサに接続された DRAM で構成されている主記憶上のデータとの間には、プロセッサ内に多段に階層化されたキャッシュメモリが存在する。プロセッサコアに高速にデータが供給されるようにするためには、キャッシュメモリの複雑な挙動にあわせて効率良くメモリアクセスできるプログラム構造を持つ並列プログラムを記述する必要があるため、メモリ階層のソフトウェアによる制御方式が重要になってくる。この高性能コンピュータシステムを目指すための第 1 ステップについては、マルチコア化や SIMD 命令や GPGPU の採用が進む組込みシステム[81]においても同様である。

第 2 ステップでは、単一プロセッサのメモリシステムでは格納しきれないより大きなデータサイズのアプリケーションを高速に実行するために、図 1.1(b)に示すように複数のメモリやプロセッサを QPI(Quick Path Interconnect)などのキャッシュコヒーレンシを保証する高速インターコネクトで接続することでクラスタリングをおこなう。

第 3 ステップは、図 1.1(c)に示すようにキャッシュコヒーレンシを保証するインターコネクトで接続した複数プロセッサを単一計算ノードとし、複数の単一計算ノードを Infiniband[35]や OmniPath[111]などの高速なネットワークでさらに接続しラックスケールの計算機クラスタシステム、あるいはさらに複数のラックスケールのクラスタシステムを繋げたマルチクラスタシステムを構築することで多くのメモリやプロセッサをクラスタリングする。

しかし、それぞれのプロセッサのキャッシュのコヒーレンシが QPI のようにハードウェアで保証される cc-NUMA(Cache coherent Non-Uniform Memory Access)型の共有メモリシステムに比べ、メモリが物理的に分散している分散メモリシステムである計算機クラスタシステムではプロセッサ間でソフトウェアによる明示的なデータ授受が必要になってくる。

また、高速ネットワークで複数計算ノードを接続したアーキテクチャでは、計算ノード

内のメモリ階層構造に加えて、計算ノード内のローカルメモリと計算ノード外のリモートメモリが混在するメモリ階層化がより進んだ構造になる。分散メモリシステムでは、高速ネットワークによる計算ノード間通信遅延が大きいため、アプリケーションの持つデータアクセスの局所性を活用し階層化されたメモリ間のデータ転送量やデータ転送回数を削減して通信遅延による性能低下を低減するメモリ階層制御を考慮したデータ授受をソフトウェアで実現することが重要になる。

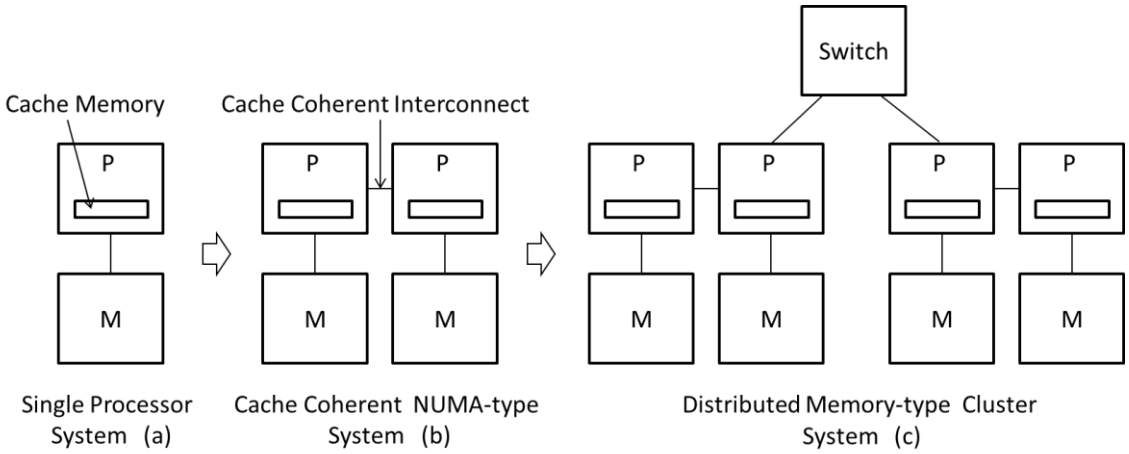

図 1.1.　高性能大規模データ処理のためのクラスタリングにより複雑化するメモリ階層

このように、大規模データを処理するためにはメモリやプロセッサをクラスタリングしていくことが必要になり、その過程でコンピュータシステム内の様々なレベルにメモリが複雑に階層化される。インメモリコンピューティングなどの大規模データ処理を高速に実行可能なコンピュータシステムを目指すための基本的な問題は、この階層化されたメモリシステムをソフトウェアでどのように効率良く制御するかと等価である。

1.2　メモリ階層に変化をもたらす新型高速不揮発メモリ

一方で、IBM 社が提唱した MRAM(Magnetoresistive Random Access Memory)や PCM(Phase Change Random Access Memory)や ReRAM(Resistive Random Access Memory)などのストレージクラスメモリ(Storage-class Memory: SCM)[1]と呼ばれるメモリとストレージの特性を併せ持つ新型不揮発メモリデバイスの実用化が期待されている。ストレージクラスメモリは、不揮発性のメモリであるため待機消費電力が非常に小さく、DRAM に迫る高速な動作が可能で、さらに DRAM より高集積化が可能なため DRAM より大容量化が可能という特徴を持

ち合わせている。MRAM や PCM などのこれまでと異なるレイテンシや性能特性を持つストレージクラスメモリの実用化により、メモリ階層も大きく変わる可能性があり、それによってコンピュータシステムを飛躍的に高性能化・低消費電力化できる可能性がある。

ストレージクラスメモリを活用して大規模データをインメモリデータ処理可能にすることで単一プロセッサシステムの性能を大きく性能向上させることができれば、あとはこれらを複数、前記の大規模データ処理の高速化を目指すための 3 つのステップに基づいて効率良くクラスタリングすればコンピュータシステムのさらなる高性能化・低消費電力化が可能になる。

しかしながら、ストレージクラスメモリは、不揮発性による消費電力の削減と高速性・大容量性による性能向上との両面で期待できる一方で、DRAM よりはアクセスレイテンシが大きく、メモリアクセスする際の動的消費電力も大きいため、単純に DRAM を置き換えればよいということにはならない。単純に置き換えるとシステム性能は低下し、消費電力はかえって高くなる場合もある。

大規模データ処理で要求されるメモリシステムは、図 1.2 のグラフの右上の部分に該当するような大容量で高速なメモリシステムである。そのため、サーバシステムや組込みシステムでも要求されている低消費電力でかつスケーラブルな主記憶を実現するためには、DRAM などの高速なメモリと大容量でかつ低消費電力なストレージクラスメモリを組み合わせる必要がある。この際に、ストレージクラスメモリと DRAM をメモリ階層のなかでどのように組み合わせてどのように使い分けるかというメモリ階層制御が重要になる。

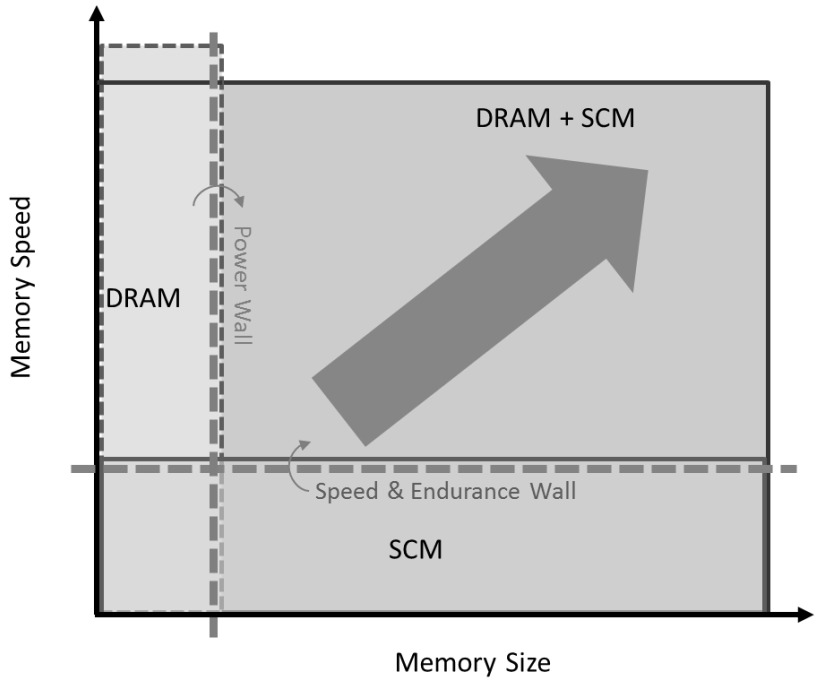

図 1.2.　ストレージクラスメモリと DRAM を組み合わせた大規模データ処理の実現

1.3　メモリ階層制御により高性能・低消費電力を実現するプログラムの開発の生産性を高めるシステムソフトウェア

これまでに述べてきたように、ソフトウェアによるメモリ階層の高効率制御は、アプリケーションの性能に大きく影響を及ぼすため、巨大メモリ搭載大規模サーバシステムや超低消費電力が要求される省電力組込みシステムなどの幅広いコンピュータシステムにおいて鍵となる技術である。しかし、メモリ階層は、大容量化のためのメモリのクラスタリングやストレージクラスメモリの実用化により性能や消費電力の観点で複雑化する。そのため、コンピュータシステムのハードウェア性能を引き出すためのメモリ階層を効率良く階層制御可能なプログラムを作成するプログラミングには、コンピュータアーキテクチャの深い知識と高度なプログラミング技術を要求する。しかし、このような作業をすべてのアプリケーションプログラム開発者に要求するのは現実的ではなく、プログラミング言語と言語処理系、ミドルウェア、OS(Operating System)、デバイスドライバなどのシステムソフトウェア（図 1.3）がアプリケーションプログラム開発者にかわって解決すべきである。

本研究はこのような背景を踏まえて、メモリ階層制御により高性能化・低消費電力化とこれを実現するプログラムの開発の生産性向上の同時達成をシステムソフトウェアで実現することが主題である。

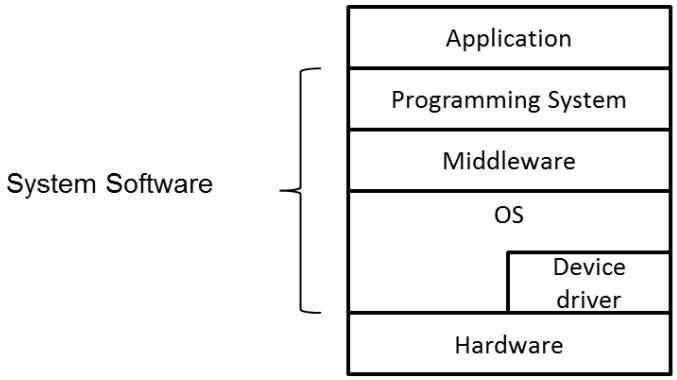

図 1.3.　システムソフトウェア

コンピュータシステムのハードウェアの持つポテンシャル性能が高くても、それを引き出すことができなくては意味がない。そのため、本研究で目指すシステムソフトウェアの性能指標は、得られるアプリケーションの性能(Performance)および省電力性(Energy-efficiency)と、アプリケーションプログラムの開発容易性（Programmability）を両軸に持つ図 1.4 の概念図で表すことができる。図 1.4 の右下の点は、コンピュータア

ーキテクチャの深い知識と高度なプログラミング技術を有する熟練者が C/C++言語や MPI(Message Passing Interface)などの抽象度の低い言語等を使用してメモリ階層制御を明示的にプログラミングし、コンピュータシステムのハードウェアの持つポテンシャル性能を引き出すことができた場合を示している。このように、一般的には、細やかな最適化による性能向上と開発容易性はトレードオフの関係にある。しかし、本研究で狙うのは、階層制御技術を自動化しこれを組み込んだより抽象度の高いプログラミング言語や OS などのシステムソフトウェアを提供することでプログラミングしやすさを向上させたうえでハードウェアの持つポテンシャルに近い性能を引き出して図 1.4 の右上の点に近づけることである。

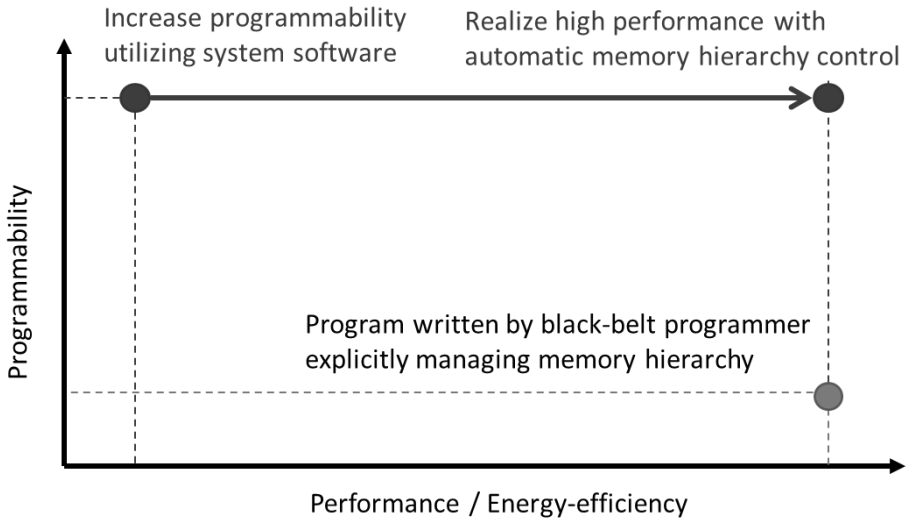

図 1.4.　高性能化・低消費電力化とプログラム開発の高生産性を両立する
システムソフトウェアの性能指標

本研究では、この主題に対して以下の 4 つのアプローチで技術課題の解決を目指す。

1) **高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム**
2) **階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア分散共有メモリシステム**
3) **新型高速不揮発メモリを活用した階層型主記憶を実現する省電力仮想記憶システム**
4) **新型高速不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ**

それぞれについて、1.4 節、1.5 節、1.6 節、1.7 節で概要を説明する。

1.4　高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム

本論文の主題に対する第 1 のアプローチは、高性能サーバなどの単一計算ノードや組込みシステムにおいて、メモリアクセス最適化などによる高速化を実現するプログラムの開発の生産性を高めるプログラミングシステムを実現することである。

[背景・課題]

近年､プロセッサはマルチコア化や SIMD 命令の採用により飛躍的に高速化すると同時に、プロセッサに効率良くデータを供給しプロセッサの高性能を引き出すためにそのメモリアーキテクチャも多様化・複雑化している。そのため、効率良く動作するプログラムの開発には、対象メモリアーキテクチャに関する深い知識が必要となっている。

プロセッサの高性能を引き出すためには、与えられた問題やアルゴリズムが内包する並列性を抽出し、マルチスレッド化やベクトル化する並列プログラミングが必須となる。アプリケーションを並列化することで、メモリバンド幅を十分に活用してデータ転送レートを上げることでプロセッサへデータを効率良く供給しプロセッサの高性能を引き出すことが重要である。ベクトル化もメモリからのデータ待ち時間が支配的なプログラムに対して行っても効果がなくメモリ階層活用を向上させる最適化をおこなうのが大前提となる[84]。

Xeon プロセッサのような階層キャッシュベースのマルチコアプロセッサでは、階層キャッシュの構造を意識してプログラムを最適化することが重要となる。階層キャッシュを効率良く活用するプログラミング技法として、メモリアクセス範囲をブロック幅内で局所化し、キャッシュライン上のデータを再利用するブロック化が知られている[90]。ブロック化を施したプログラムは多重ループ化され複雑化する。ブロック幅はキャッシュの容量にあわせてループを回すように決定する必要があるため、最適なブロック幅を求めるためにはキャッシュに関する深い知識が必要になる。

一方で CELL プロセッサ[85]のような高速なローカルメモリを持ち DMA で主記憶をアクセスするマルチコアプロセッサの場合には、DMA 転送の最適化が重要となる。転送サイズが小さい場合には DMA 転送レートはローカルメモリと主記憶の間の転送レートより大きく低下してしまう一方で、全データ参照領域を転送すると分散したデータ参照領域では領域間の不要なデータも転送してしまいかえって転送時間が増加してしまう。そのため、近接するデータ参照領域のみを選択的に融合する方式[91]が有効だがどのように転送領域をまとめ

て転送レートを上げるかはメモリアーキテクチャに関する深い知識が必要になる。

このように、プロセッサの性能を引き出すためのメモリアクセス最適化やマルチスレッド化やベクトル化などの並列プログラミングは、対象アーキテクチャに関する深い理解とそれを活かすプログラミング技法が要求される。しかし、ソフトウェア開発の中でもアルゴリズム開発を主に行っている開発者にとって並列プログラミングは開発の本質ではなく、メモリやプロセッサのアーキテクチャの深い知識を持たなくても対象アーキテクチャを活かした並列プログラムを開発可能なプログラミングシステムが必要である。

［提案方式］

本研究では、配列処理に特化した配列処理言語を用いたプログラミングシステムを提案する[87, 105, 110]。本プログラミングシステムの狙いを図 1.5 に示す。提案する配列処理言語は、並列処理と相性の良い配列処理に絞り込むことで、データ並列処理に適したアルゴリズムを配列操作用関数を組み合わせた抽象度の高い記述で直観的にプログラミングさせ、並列 C プログラムを配列処理言語の処理系で自動生成する。プログラムの記述をアルゴリズムレベルまで引き上げループレスに記述できるため、対象アーキテクチャの深い知識を持たなくても簡単に記述でき、プログラミングしやすさを実現する。本配列処理言語で記述されたプログラムは、配列処理言語の処理系によって、データ並列処理レベルの高位プログラム変換によって構造が大きく異なるプログラム構造に変換可能である本言語の特長を活かして、対象メモリアーキテクチャや並列化に適したプログラム構造に変換したのち、C プログラムに変換する。さらに、高位プログラム変換と自動チューニングを組み合わせて、これまで匠の技により作成していた対象アーキテクチャにより適合するプログラム構造と性能パラメタを利用した高度にチューニングされた C プログラムを自動生成する方式を提案し、アルゴリズム開発者を対象アーキテクチャの知識を要する複雑なプログラミングから解放する。

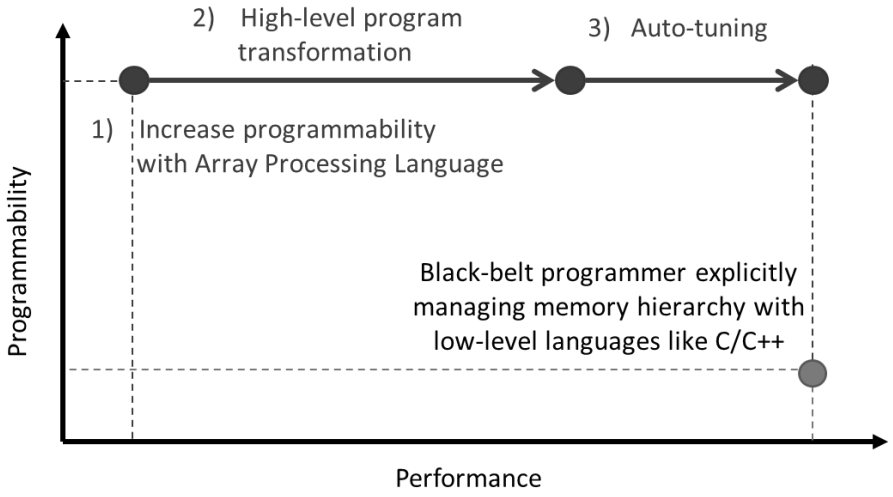

図 1.5.　本研究のプログラミングシステムの狙い

1.5　階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア分散共有メモリシステム

本論文の主題に対する第 2 のアプローチは、複数の高性能サーバから構成されるクラスタシステムや複数のクラスタを繋げたマルチクラスタシステムなどの階層的なメモリ階層を持つ高性能コンピュータシステムにおいて、分散メモリ間のデータ通信最適化による高速化を実現する並列プログラムの開発の生産性を高めるプログラミングシステムである。

[背景・課題]

ハイパフォーマンスコンピューティング向けのクラスタシステムやマルチクラスタシステムでは、高性能が得られるだけでなく、効率良く並列プログラミングできるプログラミング環境が要求される。分散メモリを持つクラスタシステムでは、分散メモリ型並列プログラミングモデルであるメッセージパッシング方式が多く用いられるが、分散メモリ間の通信を明示的に記述する必要がありプログラミングが容易ではない[50-52]。

一方で、分散メモリシステム上に仮想的な共有メモリを実現するソフトウェア分散共有メモリ方式がある [50,53,55,60,62,66]。ソフトウェア分散共有メモリ方式は、共有メモリ型プログラミングモデルを提供することができる。分散メモリ上のデータのノード間の一貫性保持のための通信は暗黙的に行われるためアプリケーション開発者は意識する必要がない。そのため、並列プログラム開発の生産性を大きく向上させることができる。

分散メモリ上に仮想的な共有メモリを構築する場合、ノード間でデータの一貫性を保持するキャッシュコヒーレンシプロトコルが必要になり広く研究されてきた[53,56,57,59,61-63]。ソフトウェア分散共有メモリは、多くの場合 OS が管理するページの単位でデータの一貫性が保持される。一貫性を保持するキャッシュコヒーレンシプロトコルのメモリアーキテクチャは、ライトバックの戻り先であるホームノードがページごとに決められている方式とそうでない方式に分類される。前者の固定のホームノードが存在する方式であるホームベースプロトコルを用いたホームベース型ソフトウェア分散共有メモリシステムが、後者のホームノードを持たずに diff 分散方式を利用するプロトコルを用いたホームレスなシステム[55,56]より総合的に高い性能をもたらすことが明らかになっており[53,57]、前者が決定版となった。

しかし、前者は後者が持つ多くの問題点を解決しているものの、クラスタ内の各ノードに CPU が 1 つであることを前提に設計されていたため、複数 CPU を各ノードで持つ単一クラスタシステムやマルチクラスタシステムなどのメモリ階層が考慮されていなかった。

これまでに、前者の単一クラスタシステムにおいて、複数ノード間で頻繁にライトバッ

クと読み出しが行われるメモリアクセスパタンではホームノードへのライトバックの多くがノード間を介したものになり著しく性能が低下するという問題に対して、ノード内のローカリティを利用可能な権限委譲プロトコルを提案してきた［47, 115, 125］。

後者のマルチクラスタシステムにソフトウェア分散共有メモリの適用範囲を広げる場合、要素クラスタ間通信遅延が大きくなるためソフトウェア分散共有メモリ方式を利用するためには、要素クラスタ内のローカリティを活用するための別の方式が必要になる。

［提案方式］

本論文ではこの問題に対して、ホームノードを多重化し各要素クラスタ内で重複して配置し、同ノードをクラスタキャッシュとして利用することでクラスタのローカリティを利用するマルチホーム方式［113］を提案する。メモリ階層を考慮したマルチホーム方式のキャッシュコヒーレンシプロトコルを組み込むことで、クラスタ間通信の削減やノード数が増えることによるホームノードへのアクセス集中の緩和が可能になる。ネットワークの高速化によりクラスタ内（ラック内）の密結合化が進むにつれクラスタ間（ラック間）の分散メモリのデータの一貫性管理のオーバヘッドは今後ますます顕在化する本質的な課題である。本研究はここに早くから着眼し、マルチクラスタまでソフトウェア分散共有メモリの適用範囲を広げることを目指したものである。

本論文では、マルチホーム方式の先行研究となる権限委譲プロトコル方式を 3.1 節でまとめたうえで、マルチホーム方式について 3.2 節で論じる。

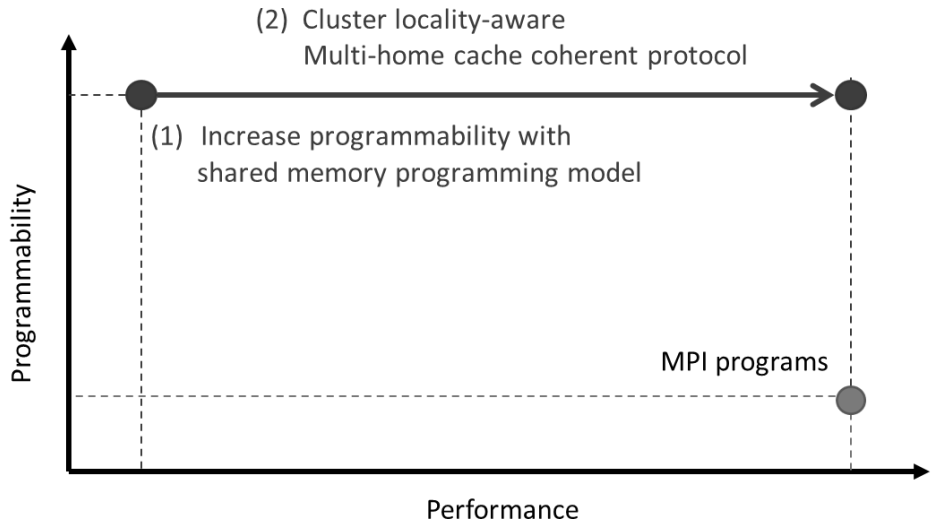

図 1.6.　本研究のソフトウェア分散共有メモリシステムの狙い

1.6　新型高速不揮発メモリを活用した階層型主記憶を省電力制御する仮想記憶システム

本論文の主題に対する第 3 のアプローチは、次世代の高性能コンピュータシステムにおけるインメモリデータ処理のためのストレージクラスメモリと DRAM を組み合わせた高速かつ低消費電力な大容量階層型主記憶とアプリケーションプログラムの開発しやすさを提供する省電力仮想記憶システムである。

[背景・課題]

実用化が期待されているストレージクラスメモリは、不揮発メモリであるため待機消費電力が非常に小さく、DRAM に迫る速度を持ち、さらに DRAM より大容量化が可能である。このように、ストレージクラスメモリは消費電力の削減と性能向上との両面で期待されているが、DRAM よりはアクセスレイテンシが大きくメモリアクセスする際に掛かる動的消費電力も大きいため、単純に DRAM を置き換えることはできない。

そこで、高性能コンピュータシステムで要求されている低消費電力かつスケーラブルな主記憶を実現するためには、高速な DRAM とストレージクラスメモリを組み合わせる必要がある。この際に、2 つのメモリをメモリ階層のなかでどのように組み合わせて利用するかがポイントになる。主記憶の一部をストレージクラスメモリに置き換えるハイブリッド型の主記憶を構成する実現手段も考えられるが、2 つのメモリをどう使い分けるかをアプリケーションプログラム開発者に強いることになりプログラミングが複雑になる。

[提案方式]

次世代データセンターの大規模インメモリデータ処理の実現に向けて、本研究ではストレージクラスメモリ/DRAM 混載メモリシステムの階層制御技法とこれを自動化したストレージクラスメモリ向け仮想記憶の基本方式を提案する。提案方式は、既存の仮想記憶システムを拡張し、そのスワップデバイスとしてストレージクラメモリを利用することで、ストレージクラスメモリと DRAM を混載させる。OS の仮想記憶システムで 2 つのメモリ間のデータの入れ替え処理であるスワップ処理を効率良くおこなうことで、アプリケーションプログラム開発者には大容量の高速主記憶があるように見せることができるためインメモリデータ処理のプログラミングをシンプルにすることを可能にする(図 1.7)。

さらに本研究では、ストレージクラスメモリの高速性と待機消費電力の低さを活かして、DRAM 上のデータを積極的にストレージクラスメモリに退避して使用する DRAM サイズを削減

し、未使用 DRAM の電源をオフすることで動作中のリーク電流を削減するストレージクラスメモリを活用した省電力仮想記憶方式[7]を提案する。スワップデバイスが高速なストレージクラスメモリになると従来 msec オーダの時間が掛かっていた 1 回のページフォルト処理が usec オーダで完了する。ページフォルト処理が高速になると、ある程度のページフォルト回数であれば処理時間に及ぼす影響は小さい。従来はページフォルトが発生して処理が停止しないようできるだけページフォルトを発生させないように大きな DRAM を搭載する必要があったが、スワップデバイスが高速になると、ある程度のページフォルトの増加は容認して積極的に DRAM 上のページをスワップデバイスに追い出して、使用する DRAM の量を減らせる可能性がある。ページフォルト回数が増加して処理時間は若干増えるが、メモリの消費電力を削減できる可能性が出てくる。消費電力を削減できるかどうかは、スワップデバイスと DRAM の間のデータ転送に必要な電力と、DRAM の量を減らすことで削減できる DRAM の静的消費電力（リーク電力）とのトレードオフになる。前者を後者よりも小さくできればメモリの消費電力の削減につながる。上記の関係が満たされるように、使用する DRAM の量を調整して、使わない DRAM は電源を切るかローパワースタンバイ状態にすることで、サーバ稼働中の消費電力を削減する。本研究の貢献は、基本方式の初期評価をフルシステムシミュレーションを用いておこなうことでスワップデバイスとしてストレージクラスメモリを利用した際の高性能化と省電力化の可能性を明らかにすることである。そしてその結果から最終的な大目標に対する課題も明らかにする。

また、スワップデバイスの用途にどのような性能特性を持つストレージクラスメモリが向いているのかを明らかにしていく必要がある。本研究のもう一つの貢献は、各種ストレージクラスメモリの性能特性が本方式の有効性に与える影響を明らかにすることである。

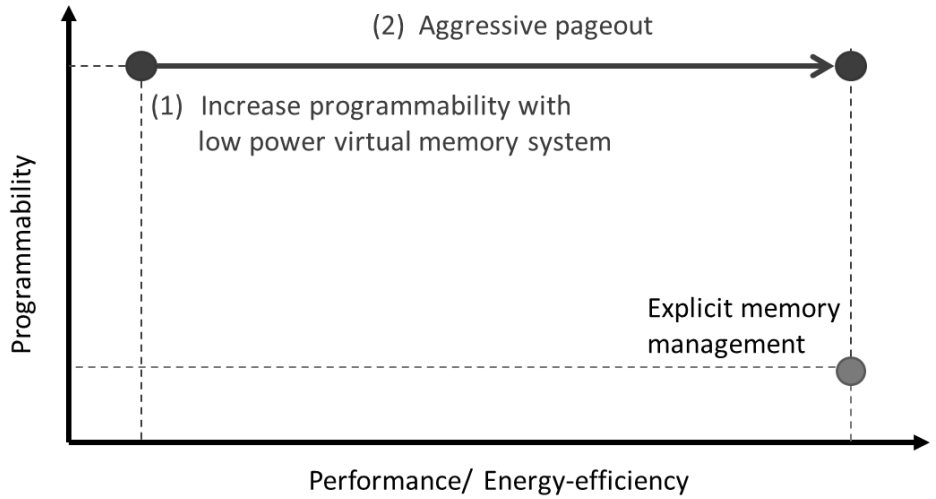

図 1.7.　本研究の省電力仮想記憶システムの狙い

1.7　新型高速不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ

本論文の主題に対する第 4 のアプローチは、組込み機器特有の課題であるディスプレイを有する組込みシステムの省電力化を、不揮発メモリと不揮発ディスプレイを組み合わせることで実現するための省電力制御機能を組み込んだ電子ペーパコントローラ向けデバイスドライバである。

[背景・課題]

近年、タブレット型端末のようなバッテリの制約がある組込みシステムの省電力化が非常に重要になっている。現在のタブレット型端末は DRAM が主記憶として利用されているが、ストレージクラスメモリが実用化されると、待機消費電力が非常に小さい超低消費電力のタブレット型端末が実現可能になってくる。しかし、タブレット型端末の場合、低消費電力の観点ではディスプレイもキーデバイスになってくる。主記憶が不揮発でもディスプレイが LCD(液晶ディスプレイ)などの揮発性ディスプレイの場合頻繁なリフレッシュが必要となるため、リフレッシュするコントローラを内蔵したプロセッサは省電力状態に移行できないからである。LCD 搭載端末では表示装置のバックライトやリフレッシュが占める割合が全体の消費電力のうち高いことが知られている[71]。

一方で、昨今の省電力技術への期待の高まりの背景の下、電源を遮断しても表示を保持できるディスプレイである電子ペーパやリフレッシュレートが 1Hz 程度と非常に低い IGZO[73]や Mirasol[76,79]などの省電力ディスプレイが注目されている。電子ペーパを利用するとシステムのアイドル時の消費電力を大幅に下げられるため、不揮発メモリと組み合わせれば待機消費電力が極めて低い超低消費電力な端末を実現できる可能性がある。

しかし、前述のとおり、不揮発メモリは動的消費電力が大きいため、単純に置き換えるとかえって消費電力が大きくもなり得る。これは不揮発性である電子ペーパでも同様である。電子ペーパの場合、書換え処理は、電源を遮断しても表示を保持できる安定した状態間を切り換えるため、大きな電力が必要となる。さらに、電子ペーパはそのデバイスの特性上、書換え処理時間が LCD と比較すると非常に長いのが特徴である。そのため、不揮発メモリと電子ペーパを組み合わせると電子ペーパを書換え中メモリアクセスし続けるので、消費電力が大きくなってしまう。

また、不揮発メモリと省電力ディスプレイのポテンシャルを引き出すためには、DRAM と LCD を利用したこれまでとは異なる複雑なデバイス制御が必要となり、プログラミング方式を変える必要がある。

［提案方式］

本論文では、高速不揮発メモリと不揮発ディスプレイを搭載したタブレット型端末向けの省電力制御機能を組み込んだディスプレイコントローラのデバイスドライバを提供することで、LCD や DRAM 向けの従来型のアプリケーションプログラム開発方式を変更することなく、省電力化を実現できるプログラムを開発できることを明らかにする(図 1.8)。

実現方法として、デバイスドライバが書換え処理のためのメモリアクセスを階層制御することにより不揮発メモリへのアクセス時間と書換え処理時間を削減する方式を提案する[70,83]。提案方式では、不揮発メモリから直接書換え処理をおこなわずプロセッサの内部メモリへ表示するデータをコピーし、電子ペーパコントローラは内部メモリから表示をおこない、その間不揮発メモリの電源をオフにすることで不揮発メモリの省電力性を引き出す。

現在のプロセッサの内部メモリはまだサイズが小さいため実機で評価できる段階には至っていないが、それが可能になれば、階層制御をおこなったうえでさらに複数の書換え処理をまとめることで書換え処理時間を短縮することができさらなる省電力化が可能になる。これは電子ペーパの書換え処理には時間を要するため、断続的に書換え命令がアプリケーションにより発行されるとその間常に電子ペーパが書換え処理中になってしまうからである。本論文ではこのような前提のもとに後者の書換え処理時間を短縮する方式の評価を実施したものであり、書換え命令を動的に再構成するデバイスドライバを利用することでアプリケーション開発方式を変えることなく低消費電力化を実現できることを電子ペーパディスプレイ搭載端末プロトタイプボードを用いて評価した。

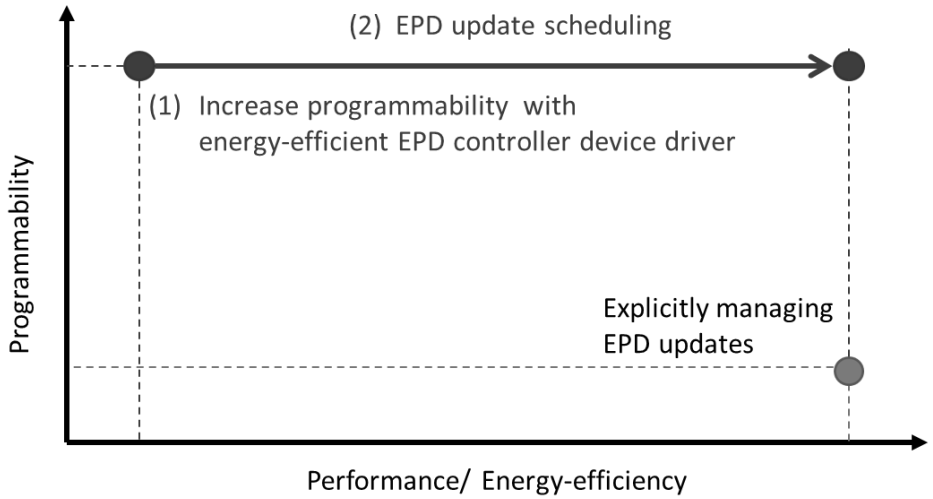

図 1.8.　本研究の電子ペーパ表示制御用デバイスドライバの狙い

1.8　本論文の構成

本論文は、第 2 章でメモリアーキテクチャに適した並列 C プログラムが生成可能な配列処理言語によるプログラミングシステムについて、第 3 章で階層的な分散メモリシステム上に共有メモリ型プログラミングを可能にするプログラミングシステムについて、第 4 章でストレージクラスメモリを活用した階層型主記憶を省電力制御する仮想記憶システムについて、第 5 章で新型高速不揮発メモリ搭載端末のメモリ階層制御型省電力表示制御機能を組み込んだディスプレイ用デバイスドライバについて述べる。第 6 章で本研究の研究成果をまとめるとともに、今後の課題を述べる。

2. 高位プログラム変換により対象メモリアーキテクチャに適合した並列プログラムが生成可能な配列処理言語によるプログラミングシステム

2.1 まえがき

近年、プロセッサはマルチコア化やSIMD型演算アクセラレータ技術の採用により飛躍的に高速化すると同時に、そのメモリアーキテクチャも多様化・複雑化している。そのため、効率良く動作する並列プログラムの開発には、対象メモリアーキテクチャに関する深い知識が必要となっている。また、特定のメモリアーキテクチャを考慮して書かれたプログラムは、再利用性や性能可搬性が低くなってしまうという課題もある。このような背景のもと、プログラムの開発の生産性を向上させるために対象メモリアーキテクチャを抽象化した並列処理言語[86]も登場している。

これらの課題に対して、本研究では、プログラムの記述レベルをアルゴリズムレベルまで上げる並列プログラム開発方式を提案している。プログラムをアルゴリズムレベルで記述することで、特定のメモリアーキテクチャに依存しないプログラムにすることができる。本研究ではアルゴリズム記述に用いる言語として、配列処理に特化した関数型言語を提案する。本言語は、マルチコア上での並列処理を前提としているため、対象を並列処理と相性のよい配列処理に限定している。

本配列処理言語で記述された特定のメモリアーキテクチャに依存しないプログラムは、本配列処理言語の処理系(トランスレータ)によって、各対象メモリアーキテクチャで効率良く動作する C プログラムに変換される。また、プログラム変換時に処理系がメモリアクセス最適化のためにアルゴリズムレベル情報が抽出しやすいように言語を設計している。このため、データ依存解析などの複雑な解析は不要になり、処理系がおこなう解析を軽量化できることから、処理系の開発コストや実行時のオーバヘッドを削減できる。

プログラム変換時に考慮する最適化の一つとして、各アーキテクチャに実装されているSIMD 命令を利用した高速化が効果的である。SIMD 命令を生成する方法として、C++コンパイラの自動ベクトル化機能[84] を利用する方法がある。自動ベクトル化を利用すれば、配列処理言語の処理系でアーキテクチャ毎に SIMD 命令を直接生成する必要がなくなるため、処理系の実装を容易にすることができる。しかし、C コンパイラの自動ベクトル化は、ベクトル化対象のループに十分な並列性がありかつメモリアクセスが連続的であることを前提

としているため、小さい配列同士の配列演算では十分な並列性が抽出できないなど万能ではない。

また、一般にベクトル化は、プロセッサの階層キャッシュメモリを考慮したブロッキングなどのメモリアクセスの最適化をしたうえで行わないと、実行時間がメモリアクセスで律速してしまい、その効果が低減してしまうという課題もある。

そこで本研究では、メモリアクセスを各対象アーキテクチャに適合させる最適化をしたうえで、配列演算の対象となる配列間の間隔などのアルゴリズムレベルでは容易に取得可能な情報を利用して、配列演算間での並列性を抽出して C コンパイラに解析しやすい形で提示するプログラム変換方式を提案する。

以下、2.2 節で本配列処理言語を利用した並列プログラム開発の概要について述べ、2.3 節で本開発方式の主な対象である画像処理や信号処理の近傍処理におけるベクトル化の課題とこれを解決するアルゴリズムレベル情報である近傍情報を用いたベクトル化向け高位プログラム変換方式を説明する。2.4 節では配列処理言語の高位プログラム変換を利用した自動チューニング方式によるメモリアクセスなどのさらなる最適化が可能な並列 C プログラムの自動生成を提案する。2.5 節で関連研究について述べ、2.6 節で本章をまとめる。

2.2　配列処理言語を利用した並列プログラム開発

2.2.1　配列処理に特化したアルゴリズム記述言語

本配列処理言語では、図 2.1 に示すラプラシアンフィルタプログラムのように、配列データの処理を配列演算を組み合わせた抽象度の高い記述でプログラミングできる。本配列処理言語の最大の特長は、C 言語で多重ループで記述していた処理が、2 つの配列切り出し関数 MExtract()(繰返し切り出し関数)と Extract()(単一切り出し関数)、および、引数の配列の要素ごとに関数を適用する高階関数であるマップ関数呼び出し Map()を使うことでループレスに記述することができる点である。

関数 MExtract()は、図 2.2 に示すように、引数の配列 M から部分配列を繰返し切り出し、切り出した部分配列を要素とする配列 N を作成する多重配列切り出し関数である。パラメタ base に最初の部分配列の切り出し開始位置、step に切り出し位置の行方向および列方向のずらし幅、size に切り出す配列の行方向および列方向の個数、esize に切り出す個々の配列のサイズを指定する。これらのパラメタは、配列のインデックスやサイズを表す行と列を示す整数値のペアを ［ と ］で囲んで表記する。

```
% 関数の定義
function img = laplacian(m)
   c = [1   1   1;
        1 -8   1;
        1   1   1];
   p = abs(Sum(m .* c));
end

% トップレベルの関数の定義
function img = lap(M)
   % 入出力変数の型指定
   Type({[0,0], 'uint8'}, M);
   Type({[0,0], 'uint8'}, img);
   img = Map(@laplacian, MExtract(M, [-1, -1], [1, 1], Size(M), [3, 3]));
end
```

図 2.1.　ラプラシアンフィルタプログラム

$$N = \boldsymbol{MExtract}\,(M,\ base,\ step,\ size,\ esize)$$

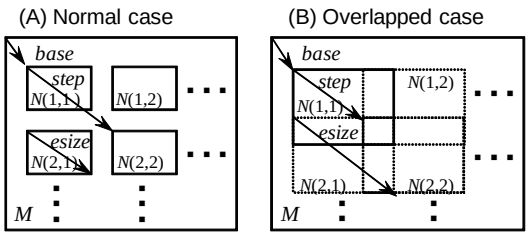

図 2.2.　繰返し切り出し関数 MExtract()

$$\boldsymbol{Map}(@f, \boldsymbol{MExtract}(M, base, step, size, esize)) = \begin{pmatrix} f(N(1,1)) & f(N(1,2)) & \cdots \\ f(N(2,1)) & f(N(2,2)) & \cdots \\ \vdots & \vdots & \end{pmatrix}$$

図 2.3.　MExtract()と高階関数 Map()の組み合わせ

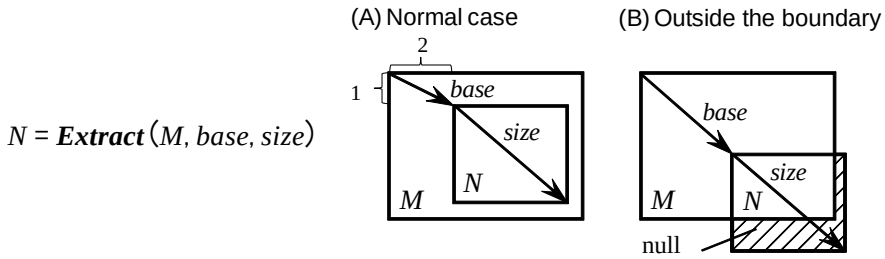

図 2.4.　単一切り出し関数 Extract()

関数 MExtract()で作成した配列を引数にして関数を呼び出す場合、関数名の前に’@’を付けると配列の要素毎に関数が適用される。このように MExtract()と Map()を組み合わせて利用することで、C 言語の多重ループに相当する処理が記述できる。このようにして、性能に影響を及ぼしやすいループを陽に記述させず、対象アーキテクチャに適したループを処理系で生成するのが本配列処理言語の大きな特長である。

関数 Extract()は、既存の配列から部分配列を 1 つ切り出す関数である。切り出し開始位置をパラメタ base に、部分配列のサイズを size に指定する。このように、アルゴリズムレベルの情報である配列のサイズや切り出し間隔を配列関数のパラメタとしてアプリケーションプログラム開発者に明示させることで処理系が最適化に必要な情報が取得しやすい言語設計になっている。

MExtract()、Extract()ともに元の配列をはみ出す部分配列の指定が可能である。はみ出した部分には不定値が入っていると解釈する。これにより、画像処理や信号処理によく現れる配列の境界付近の例外処理の記述を簡略化している。

また、高階関数は Map()以外にも、逐次的な配列処理を記述するための高階関数として Scan()、Reduction()を提供している。

2.2.2　近傍処理プログラムの記述例

本配列処理言語の主な対象は、画像処理や信号処理アルゴリズムにおける近傍処理である。近傍処理は、データ局所性が高いため、マルチコア上での並列処理に向いている。近傍処理では、近傍画素を使った同じ処理を画像の全画素に対して繰返し適用する。このため、これまでに説明した配列処理関数を使ってシンプルに記述することができる。

図 2.1 に、代表的な近傍処理であるラプラシアンフィルタのプログラムを示す。ラプラシアンフィルタは、8-近傍を用いた 2 次微分値を求めるアルゴリズムである。プログラムでは、関数 MExtract()を使って入力画像 M の各画素の周辺の 9 画素からなる 3 行 3 列の

配列からなる配列を作り、その各配列に対して関数 f をマップ関数呼び出しで適用している。f は、切り出した 3 行 3 列の配列 x に 3 行 3 列のフィルタ係数の配列を掛けて、9 つの要素の総和を求める。Sum()は、配列の要素の総和を返す関数である。Sum()と同様に配列からスカラー値やインデックスを計算する関数として、Prod()、Max()、Min()、Maxpos()、Minpos()を提供しており(順に総積、最大値、最小値、最大値の要素のインデックス、最小値の要素のインデックス)、これらを総称してアグリゲーション関数と呼んでいる。また、Size(M) は配列 M のサイズを返す関数である。

2.2.3 配列処理言語の特徴

本配列処理言語のプログラミングスタイルは、入力配列から部分配列を複数切り出し独立に計算する処理をループレスに記述可能とするものである。よって、画像処理や信号処理で頻繁に現れる各要素データに対して近傍要素データから計算する近傍処理 [121]のようなデータ並列性のある画像処理アルゴリズムや信号処理アルゴリズムは記述可能で、かつ、本論文で後述する方法で効率良く動作する C プログラムへ変換可能なため本配列処理言語に向いている。具体的には、画像処理における各種フィルタ処理、エッジ検出処理、動画像の複数フレームを使った動き検出などである。

一方で、本配列処理言語が得意でないアルゴリズムは、逐次的な依存性のあるような処理である。本配列処理言語では逐次的な配列処理を記述するための高階関数として Scan()、Reduction()を提供しているが、そもそも並列化に向かない処理であり最適化は行っていない。また、タスク並列処理などは記述することができない。

このように、本配列処理言語には得意な処理と不得意な処理があるが、配列処理言語で記述されたプログラムは処理系により C プログラムに変換可能なので、本配列処理言語に向くアルゴリズムは本配列処理言語で記述し、それ以外の処理は C プログラムとして記述しリンクして実行する。本配列処理言語と C 言語を使い分けて記述する必要があるが、記述する処理にデータ並列性があるか否かの判断はできるアプリケーション開発者を想定している。

また、言語の仕様を拡張し記述可能範囲を広げることは一長一短があり、記述可能になるアルゴリズムが増える一方で、処理系で最適化が必要なポイントも増加するため処理系の実装コストも高くなってしまう。実装コストのかけ方としては、画像処理や信号処理にターゲットの限定を維持したままで、その範囲で記述可能なアルゴリズムの最適化を推し進めていくことが重要である。

2.2.4 配列処理言語の処理系

本配列処理言語の処理系の構成を説明する。本処理系の入力は、本配列処理言語で記述

されたプログラムである。まず、フロントエンドにより、入力プログラムを中間表現に変換する。つぎに、プログラム変換部により、中間表現に対して最適化をおこなう。そして最後にバックエンドにより、中間表現から、それぞれの対象アーキテクチャ毎に用意するランタイムライブラリを呼び出しながら実行する C プログラムに変換する。生成された C プログラムは、自動ベクトル化機能を有する C コンパイラによって、SIMD 命令を効率良く利用する実行プログラムにコンパイルされる。

プログラム変換部では、C プログラムが自動ベクトル化しやすくなるように 2.3 節で述べる高位プログラム変換を施す。

2.3 ベクトル化向け高位プログラム変換方式

2.3.1 ベクトル化の課題

以下にベクトル化された C プログラムを生成する際の課題を列挙する。

1. **メモリアクセスの最適化が必要**

   一般に、ベクトル化の効果が出るようにするためには、メモリアクセスもあわせて最適化する必要がある。プロセッサがキャッシュを持つ場合には、配列データがキャッシュサイズに収まらない場合、キャッシュミスによりメモリアクセスのレイテンシが大きくなってしまう。その場合、仮にプログラムがベクトル化されていても、その効果はメモリアクセスのレイテンシに隠れて無駄になってしまう。

2. **自動ベクトル化の課題**

   C コンパイラの自動ベクトル化を利用することで、アーキテクチャ毎の SIMD 命令の生成をコンパイラに任せることができる。しかし、自動ベクトル化できる処理は限定的である。一般に、自動ベクトル化の対象となるのは、多重ループの最内ループの処理で、かつ、メモリアクセスが連続である処理が基本となる。さらに SIMD レジスタサイズに対してループ回転数が小さすぎると十分な並列性がないと判定され自動ベクトル化できない。このため、メモリアクセスが連続で、かつ、ループ回転数が大きい最内ループを生成してコンパイラに提示する必要がある。しかし、このようなループを生成しても、ループのデータの依存関係の有無やループ回転数をコンパイラが解析できない場合があるため、コンパイラが自動ベクトル化できるとは限らない。そのため、関連するアルゴリズム情報をプラグマの挿入という形でコンパイラにヒントすることが必要となる。

3. 近傍処理のベクトル化の課題

画像処理における近傍処理では、3x3 画素や 5x5 画素などの範囲にある近傍画素を使った処理を全画素に対して繰返し適用する。この繰返しの単位でベクトル化しようとすると、演算量やデータ転送サイズの観点からも並列性が十分でない。このため、仮に制限の多い自動ベクトル化に頼らずに、SIMD 命令を直接記述しても効率がよいベクトル化ができないという課題がある。よって、配列処理用ライブラリをベクトル化して用意しておくことも解にならない。繰返し単位の配列処理に対して、処理系の基本的なループ生成方式では、対応する 2 重ループを生成する。そのため、近傍処理では、生成される 2 重ループの最内ループのループ回転数が 3 や 5 などとなる。ラプラシアンフィルタの例では、図 2.5 のような C プログラムを生成する。このような 2 重ループに対して SIMD 命令の生成をコンパイラの自動ベクトル化で行おうとすると、最内ループのループ回転数が小さすぎるため自動ベクトル化できないという課題がある。また、2 重ループのループ回転数が小さいため、ループオーバヘッドが顕著化するという課題もあった。

```
for(i=1; i<479; i++){
  for(j=1; j<719; j++){
    tmp = 0;
    T1_p = T1b + 720*(i-1) + (j-1);
    T2_p = T2b;
    for(x=0; x<=2; x++){
      for(y=0; y<=2; y++){ /* trip count too small */
        tmp += *T1_p++ * *T2_p++;
      }
      T1_p += 717;
    }
    tmp = abs(tmp);
    *T0_p++ = (uint8)((tmp > 255)? 255: tmp);
  }
  T0_p += 2;
}
```

図 2.5.　処理系が生成するラプラシアンフィルタの C プログラムの概要

2.3.2　近傍情報を利用したベクトル化向け高位プログラム変換方式

本研究では、配列処理プログラムに明示されている近傍情報を利用した簡単な解析をおこなうことで、効率の良い自動ベクトル化が可能になる C プログラムを生成する高位プログラム変換方式を提案する。

[高位プログラム変換方式の概要]

提案方式の概要を説明する。近傍情報を利用したベクトル化向け高位プログラム変換は 3 段階でおこなう。

最初に、ベクトル化の効果がメモリアクセスのレイテンシで隠蔽されないように、メモリアクセスを最適化する。プロセッサが、本稿の評価で利用する表 2.1 の Intel Core 2 Quad[1] のようにキャッシュを持つ場合、キャッシュの容量を考慮して処理順序を変更するループブロッキングをおこなうことでキャッシュミスを減らすことができる。本並列プログラム開発方式では、対象アーキテクチャにとって最適な処理順序の近似解を求める、レンジスケジューリング方式と呼んでいるメモリアクセスの順序の制御をおこなう。これによりメモリアクセスレイテンシを抑制し、ベクトル化が効きやすい状態を整える。

つぎに、C コンパイラが自動ベクトル化しやすいループを生成するための変換をおこなう。近傍処理の繰返しの単位でベクトル化しようとしても十分な並列性を抽出できない。そこで、本方式では、近傍処理間の並列性を利用できるようにプログラム変換をおこなう。これにより、自動ベクトル化しやすいループ、即ち、メモリアクセスが連続した、ループ回転数が大きい最内ループをつくる。

最後に、生成された最内ループに対して、プラグマを挿入する。C コンパイラにはポインタが指す配列間の依存関係などが解析できない場合がある。プラグマを挿入することで、アルゴリズムレベルの知識をコンパイラにヒントとして与える。

以降、メモリアクセスの最適化、近傍処理間の並列性の抽出、コンパイラヒントの挿入について説明する。

---

[1] Intel, Intel Core 2 Quad, Xeon は、米国およびその他の国における Intel Corporation の商標です。その他本論文に掲載の商品、機能などの名称は、それぞれ各社が商標として使用している場合があります。

表 2.1.　評価環境

項目	スペック
Intel Core 2 Quad	2.66GHz
L1 データキャッシュ	Four 32 KB, 8-way set associative
L2 キャッシュ	Two 4 MB
Linux	2.6.22
C コンパイラ	Intel C++ コンパイラ 10.0 (-ipo -xT -O3)

[メモリアクセスの最適化]

メモリアクセスの最適化をレンジスケジューリング方式を利用しておこなう。レンジスケジューリング方式では、与えられたプログラムのトップレベルの式の結果配列全体をレンジとして、そのレンジを小さい矩形領域のレンジに階層的に区切っていき、区切ったレンジ間の実行順序を決めていく(図 2.6)。

表 2.1 の Intel Core 2 Quad のようなキャッシュを持つマルチコアの場合について説明する。まずは、レンジを複数コアで分割する。最も単純にはレンジをコア数で上下方向に分割してそれぞれのコアが処理する。

このレンジをブロッキングでキャッシュが有効に働く幅のレンジにさらに分割する。レンジの幅の計算には、レンジの処理に必要なデータの集合であるワーキングセットをまず計算する。そして、ワーキングセットの各部分配列が、キャッシュサイズをウェイ数で分割したサブブロックのいずれかに収まる、できるだけ幅の大きなレンジのサイズを選択する。

レンジのワーキングセットは、近傍情報を用いて簡単に解析することができる。図 2.1 のラプラシアンフィルタの例を使って、サイズ[1, c のレンジのワーキングセットを求めてみる。入力変数 M のワーキングセットは、esize+step×[1-1, c-1] と計算できるので、[3, 3+(c-1)]となる。出力変数のワーキングセットは[1, c] と計算できる。これを格納するのに必要なメモリ容量は、行成分と列成分の積に、さらに sizeof(uint8) を掛けた値になるので、それぞれ、3c + 6、c になる。また、後述する、計算過程で必要となる型変換解決用の 32 ビット整数型の配列については、ワーキングセットは[1, c] と計算できる。必要なメモリ容量は、これに sizeof(int32) を掛けた値である 4c になる。以上により、この中で一番大きい、型変換解決用の配列に依ってレンジの幅が決定する。表 2.1 より L1 データキャッシュの 1 ウェイが 4 KB なので、求めるレンジの幅は 4c <= 4 KB を満たす最大の c になるので、c は画像の横幅の 720 と求めることができる。

以降で、最内のレンジの中の処理のベクトル化について述べる。

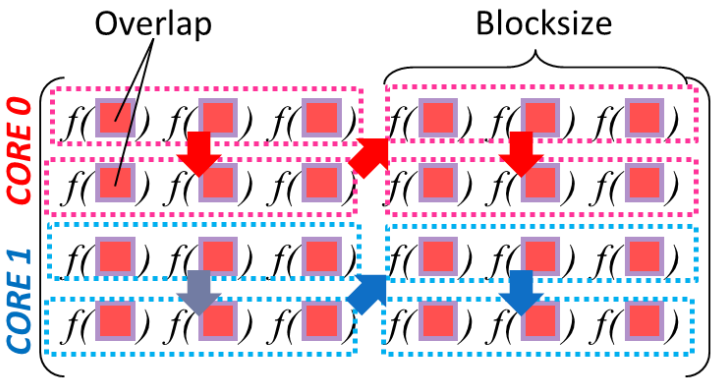

図 2.6.　メモリアクセス最適化

[近傍処理間の並列性の抽出]

近傍処理間の並列性の抽出は、本配列処理言語の最も特徴的なイディオムである、関数 MExtract()と高階関数 Map()の組み合わせで以下のように記述される近傍処理に対して適用する。

Map(@f, **MExtract**(M, base, [1, 1], size, [I, J]));
where　f(m) ← Sum(g(m))

具体的には、MExtract()で[1, 1] 間隔で繰返し切り出すサイズ[I,J]の部分配列 x に対して、エレメントワイズな配列演算 g を適用した後、配列の要素の総和を返すアグリゲーション関数 Sum()を適用し、スカラー型のデータを返す処理である。これは C プログラムでは多重ループに相当する処理で、アプリケーションのホットスポットになる可能性が高く、ベクトル化の効果が大きい。

この処理のベクトル化は、次のように近傍処理間の並列性を利用しておこなう。まず、MExtract()で繰返し切り出すサイズ[I,J]の size 個の部分配列について、そのインデックス[i, j] の要素のみを集めたベクトルデータを i 行 j 列の要素とするサイズ[I, J] の配列を V とする。これらを用いると、近傍処理全体の処理は、Sum(g(V)) と表すことができる。g() をベクトル拡張すると、近傍処理全体の結果であるベクトルデータが返るベクトル演算に変換できる。

本配列処理言語では、MExtract()に明示された近傍情報である step = [1, 1]から近傍処理間の配列の間隔がわかる。このことから、MExtract()で切り出す部分配列の各インデッ

クスの要素の集合は、行方向と列方向に連続している部分配列であることがわかる。部分配列のサイズは size、切り出し位置のずらし幅が[1, 1] なので、V は

MExtract(M, base, [1, 1], [I, J], size)

で切り出せることがわかる。これにより、処理系でこのイディオムを中間表現上で認識すると、次のような変換をおこなう。

Sum(g(MExtract(M, base, [1, 1], [I, J], size)))

V の i 行 j 列の要素配列は、Extract()を使って

Extract(M, base + [i,j], size)

で表現できるのでこれを使って Sum()を展開する。

図 2.1 のラプラシアンフィルタプログラムの例では、中間表現上で、図 2.7 のプログラム相当の中間表現に変換する。

この変換後は、処理系では図 2.8 のような C++プログラムを出力する。最内ループに注目すると、部分配列の各要素を表すポインタ T1 p～T9 p、および、結果を格納する要素を表すポインタ T0 p のメモリアクセスが連続していることがわかる。また、ループ回転数も 718 と十分に大きくなっている。以上のことから、コンパイラの自動ベクトル化が期待できるループが生成できている。

```
abs(Extract(M,[-1,-1], Size(M))
      + Extract(M,[-1, 0], Size(M))
      + Extract(M,[-1, 1], Size(M))
      + Extract(M,[0, -1], Size(M))
      - 8 * M
      + Extract(M,[0, 1],   Size(M))
      + Extract(M,[1,-1],   Size(M))
      + Extract(M,[1, 0],   Size(M))
      + Extract(M,[1, 1],   Size(M))
)
```

図 2.7.　プログラム変換後のラプラシアンフィルタプログラム

```
for (i=1; i<479; i++){
  for (j=1; j<719; j++){
    tmp = abs( (*T1_p) + (*T2_p) + (*T3_p)
          + (*T4_p) + (*T5_p * -8) + (*T6_p)
          + (*T7_p) + (*T8_p) + (*T9_p));
    tmp = (tmp > 255)? 255: tmp;
    *T0_p = (uint8)tmp;
    T1_p++;
    T2_p++;
    ....
    T9_p++;
    T0_p++;
  }
  T1_p += 2;
  T2_p += 2;
  ....
  T9_p += 2;
  T0_p += 2;
}
```

図 2.8.　ラプラシアンフィルタのプログラム変換後の C プログラムの概要

[コンパイラヒントの挿入]

これまで説明していたプログラム変換により、ベクトル化向けの最内ループが生成できた。次は、この最内ループの直前にプラグマを挿入することで、コンパイラにこのループに関するアルゴリズム情報を提示する。ここでは、本稿の評価で利用する Intel C++コンパイラを想定する。

まず#pragma ivdep を挿入する。これにより、最内ループにデータ依存がないことを指示する。これでコンパイラはベクトル化できることが静的にわかる。この挿入ができるのは、本配列処理言語が関数型であるためマップ関数呼び出しの処理間でデータ依存がないことを保証しているからである。

さらに、#pragma loop count を挿入する。これにより、ループ回転数の目安を指示する。ベクトル化向けプログラム変換を施したので、ループ回転数は十分に大きくなっている。しかし、コンパイラにはこれが解析できずベクトル化されない場合があるために必要とな

る。ここでは、#pragma loop count (16) などと SIMD レジスタサイズより大きくすればよい。

最後に、型変換が自動ベクトル化できない制約を解決するための処理をおこなう。画像処理では、入力画像の符合なし 8 ビット整数で表現される画素を入力とし、符合付き 16 ビット整数などで計算をおこない、計算結果を再び出力画像の符合なし 8 ビット整数の画素に戻す型変換が必要となる。この後半の型変換が、想定したコンパイラでは自動ベクトル化できないという制約がある。そこで、計算結果を一時保存する 16 ビット整数型の型変換解決用の配列を用意し、そこへ代入するようにする。これにより、その代入の処理までは自動ベクトル化できない型変換が入らなくなるため、自動ベクトル化が可能となる。型変換解決用の配列から出力画像にデータをコピーする後続の処理のみが自動ベクトル化されないようになる。

2.3.3 性能評価

ベクトル化向けプログラム変換方式の効果を確認するために、性能評価をおこなった。評価には、近傍処理である、ラプラシアンフィルタ(Laplacian)、平滑化フィルタ(Blur)、Prewitt フィルタ(Prewitt) を用いた。平滑化フィルタは、画像を平滑化するために、各画素データに対してその近傍 8 画素との平均値を求める。本配列処理言語では、図 2.9 のように記述することができる。

図 2.10 は、画像のエッジ検出などに使われる Prewitt フィルタの記述例である。Prewitt フィルタは、各画素を中心とする部分配列に対して、畳み込み計算をおこなう処理を、画像全体に対しておこなうプログラムである。具体的には、各画素データとその近傍 8 画素に対して、2 つのフィルタ v、h を適用する。

また、図 2.11 に MExtract2Extract 変換後の Prewitt フィルタプログラムを示す。

なお、本配列処理言語が準拠している MATLAB [101]文法では、配列を[1,2; 3,4]というように要素を”[“, “]”で囲み行の区切りを”;”で記述する。配列の要素間の積、商は、行列積や逆行列の積との混同を避けるため”.*”、”./” と記述し、関数への参照は”@関数名”と記述する。

```
function img = blur(m)
    p = Sum(m) / 9;
end

img = Map(@blur, MExtract(M, [-1, -1], [1, 1], Size(M), [3, 3]));
```

図 2.9.　平滑化フィルタプログラム

```
function img = prewitt(M)
  Type({[1080,1920],'uint8'},M);            % input image type
  Type({[1080,1920],'uint8'});              % output image type
  base = [-1, -1];  step = [1, 1];  size = Size(M);  esize = [3, 3];
  img = Map(@f, MExtract(M, base, step, size, esize));
end

function ret = f(m)
  v = [ 1,   1,   1;
        0,   0,   0;
       -1,  -1,  -1];
  h = [1,   0,   -1;
       1,   0,   -1;
       1,   0,   -1];
  p = abs(Sum(h .* m)) / 2 + abs(Sum(v .* m)) / 2;
  if p > 255
    ret = 255;                  % saturated at largest possible value
  else
    ret = p;
  end
end
```

図 2.10.　MExtract2Extract 変換前の Prewitt フィルタプログラム

```
P = (abs(Extract(M, [-1, -1], Size(M)) + Extract(M, [-1, 0], Size(M)) +
      Extract(M, [-1, 1], Size(M)) - Extract(M, [1, -1], Size(M)) –
      Extract(M, [1, 0], Size(M)) - Extract(M, [1, 1], Size(M))) ./ 2)   +
    (abs(Extract(M, [-1, -1], Size(M)) + Extract(M, [0, -1], Size(M)) +
      Extract(M, [1, -1], Size(M)) - Extract(M, [-1, 1], Size(M)) –
      Extract(M, [0, 1], Size(M)) - Extract(M, [1, 1], Size(M))) ./ 2);
```

図 2.11.　MExtract2Extract 変換後の Prewitt フィルタプログラム

評価環境は、表 2.1 の Core 2 Quad を用いた。また、C コンパイラには、Intel C++ コンパイラ 10.0 を用いた。コンパイルオプションは、-ipo -xT -O3 を用いた。-O3 は、高速なコードを生成する-O2 の最適化に加えて、スカラー置換、ループのアンロール、ループブロッキングなど強力なループの最適化およびメモリアクセスの最適化をおこなう[126]。また、対象のプロセッサを指定する-xT はベクトル化のためのオプションで、SSSE3 命令、SSE3 命令、SSE2 命令、SSE 命令を生成する。

ベクトル化およびマルチコア並列化の効果を図 2.12 で確認する。速度向上は、Extract()への変換を行わない場合の実行時間(図中の MExtract(AutoVec)) を基準としている。Extract()への変換までをおこなった場合を Extract(AutoVec)、同変換後にコンパイラヒントを挿入した場合を Extract + CH(AutoVec)、さらに 4 コアで並列化した場合を 4core で示す。また、比較のために、Intel C++コンパイラのイントリンシックを使ってベクトル化をした場合を Extract(Intrinsics) と示す。

Extract()への変換を行わない場合、ループの最適化やベクトル化のコンパイラオプションを使用しても自動ベクトル化はできなかった。最内ループの回転数が短いため自動ベクトル化がでなかったのためループの展開ができなかったと考えられる。Extract()への変換までをおこなった場合でも自動ベクトル化はできなかったが、ループオーバヘッドが減ったことやフィルタ定数との必要のない演算が減ったことで、それぞれ 3.3 倍、5.8 倍、8.9 倍の速度向上を得ることができた。

同変換後にコンパイラヒントを挿入して自動ベクトル化することで、それぞれさらに 1.4 倍、2.4 倍、2.6 倍の速度向上が得られることが確認できた。自動ベクトル化しない場合は、1 画素単位で処理が行われる。これに対して、自動ベクトル化した場合では、4 画素単位で処理されるようになる。ラプラシアンフィルタの例では、4 バイトのロードが 9 回実行され、それぞれがパックドダブルワードに変換された後、パックドダブルワード同士の演算が行われる、4 並列のベクトル化がなされる。

一方で、イントリンシックを使った場合との比較では、それぞれさらに 1.4 倍、1.3 倍、2.0 倍速度向上できる余地を残していることがわかる。イントリンシックを使ったものでは、パックドワードでの 8 並列のベクトル化を行っている。自動ベクトル化の場合に 4 並列になっているのは、現状の処理系で、プログラムの処理過程で必要な中間値の型を C++のデフォルト型変換ルールに従って素直に決定しているためである。そのため、本来は 16 ビット整数でビット長が足りるところを 32 ビット整数を利用している。また、データ転送は、16 バイトで転送をおこなった方が効率がよい。このため、イントリンシックを使ったものでは、16 画素分に相当する 16 バイト のロードを 9 回おこない、レジスタ上で上位ビットと下位ビットに分けて計算し、結果を合成してから再び 16 バイトでストアしている。このような細やかなベクトル化が、自動ベクトル化では難しいことも性能差に影響している。

最後に、4 コアでの実行では、それぞれ、3.4 倍、3.0 倍、2.5 倍の速度向上が得られることが確認できた。

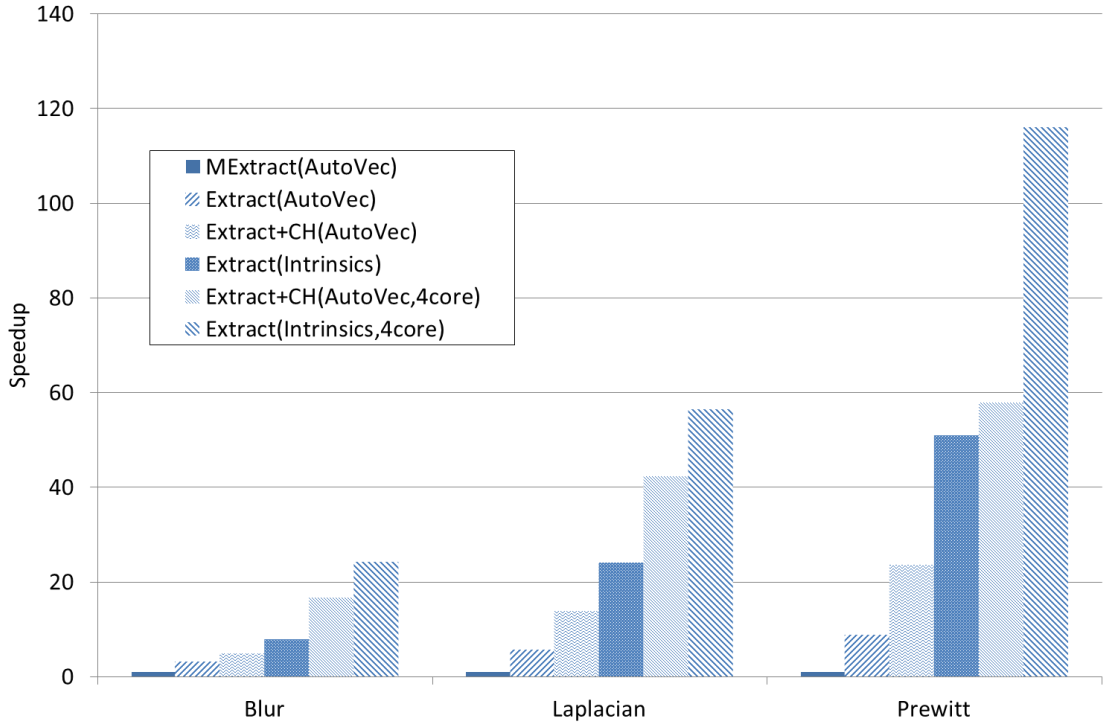

図 2.12. ベクトル化および並列化による速度向上

本配列処理言語のアルゴリズムレベル情報を利用した簡単なプログラム変換と、C コンパイラの自動ベクトル化を組み合わせることで、近傍処理アプリケーションが SIMD 命令を直接記述することなくベクトル化できた。これは、本配列処理言語がアルゴリズムレベル情報が取得しやすくなっているために、複雑なデータ依存解析などが不要になり、処理系で容易に最適化ができるという本開発方式の有効性の一端を示すものであると考えられる。

配列処理言語の処理系と C コンパイラの役割分担という観点では、ベクトル化は C コンパイラに任せることができるようになるのが望ましい。今後 C コンパイラの解析能力やベクトル化機能が強力になりそれが可能になれば、配列処理言語の処理系の開発コストを軽くでき、その分、C 言語のような低レベル言語が入力となる C コンパイラの解析能力では手が届かない高位プログラム変換機能の開発に注力することができる。

その一方で、コンパイラの自動ベクトル化は不連続アクセスに未対応など未だに制約が多い。今後は、さらにベクトル化できる範囲を拡大することも必要であるため、コンパイラの自動ベクトル化の進歩にも期待しつつ、処理系で SIMD 命令の一部生成もおこなっている[110]。その際には、ベクトル化の並列度を上げるためにビット長推論をおこないさらに

効率の良いベクトル化をすることが重要である。

2.4　高位プログラム変換を利用した自動チューニングによる並列Cプログラム生成

本研究の配列処理言語は、これまで説明してきたように、データ並列処理レベルの高位プログラム変換が可能である。本節では、高位プログラム変換に自動チューニングを組み合わせることで、アルゴリズム開発者が記述した直観的なアルゴリズム記述から、対象アーキテクチャに適応するプログラム構造が探索可能な並列 C プログラムを生成する方式の概要を述べる。

2.4.1　自動チューニングによる並列Cプログラム開発とその課題

組込みシステムにおける画像処理や信号処理では、対象メモリアーキテクチャに適応する高速かつ低消費電力な並列 C プログラムを開発する必要がある。既存の開発手法では、まずMATLABやOctave[102]などのプロトタイプ開発ツールで所望のアルゴリズムの開発をおこなう。その後、例えば製品化に向け、組込み機器向けに並列 C プログラムを再実装することが多いが、これはメモリアーキテクチャの深い知識や高度なプログラミング技法を身につけた匠の技に依存するところが大きい。そのため、実装に必要な知識を軽減し、生産性を高める開発手法が望まれる。

対象メモリアーキテクチャに適合する性能パラメタや処理方式を発見する手段として、自動チューニング技術 [94-97,114,129]が注目されている。この技術は、対象アルゴリズムの高速なメモリ階層対応済みのチューニングパラメタ付き並列プログラムを準備し、パラメタ値を変えて試行を繰り返すことで最適値に自動調整する。しかし自動チューニングの適用には、応用ごとにパラメタ化されたプログラムの作成が必要であり、そもそもこの作成が可能なのはメモリアーキテクチャの深い知識と高度なプログラミング技術を有する熟練者に限定されるのが課題になっている。

これらの課題に対し、並列 C プログラムの自動生成が可能で、アルゴリズム開発に広く利用されているMATLABに準拠した配列処理言語を用いた本プログラミングシステムが利用できる。本プログラミングシステムでは、データ並列処理に適したアルゴリズムを、配列操作用関数を組み合わせた抽象度の高い記述で直観的にプログラミングさせ、パラメタ化された並列 C プログラムを処理系で自動生成することができる。パラメタ化された並列 C プログラムさえ生成できれば、あとはパラメタ値を変えて試行を繰り返すことで最適値に

自動調整すればよい。

さらに、データ並列処理レベルの高位プログラム変換によって構造が大きく異なる複数のプログラム構造に変換可能である本配列処理言語の特長を活かして、対象アーキテクチャに合わせた自動チューニングの探索空間拡大を実現した。これにより、メモリアーキテクチャの深い知識と高度なプログラミング技術を有する熟練者しか使えなかった自動チューニングを、抽象度の高いアルゴリズム記述から、つまり多くの開発者から、利用可能にし、アルゴリズム開発者を対象アーキテクチャの知識を要する複雑なプログラミングから解放する。

自動チューニングも含めると、本プログラミングシステムを用いた典型的な開発フローは次のようになる(図 2.13)。メモリ階層に適合するループにおいて重要になるレンジ幅に着目すると、開発の初期段階の画像処理や信号処理アルゴリズムの開発時は、処理系が 2.3.2 のヒューリスティクス活用のレンジスケジューリングで求めた最適に近いレンジ幅でパラメタを調整した並列 C プログラムを最初から生成する。これは、開発中のアルゴリズムの評価をすぐにおこなうことが先決であるためである。そして、このような過程を経て開発された配列処理言語プログラムから製品化などに向けた並列 C プログラムを生成する時は、多少時間が掛かっても可能な限り高性能であることが優先されるので、自動チューニングを適用する。処理系が自動チューニングの入力となるパラメタ化された並列 C コードを生成し、自動チューニングを適用することでパラメタ調整を行う。

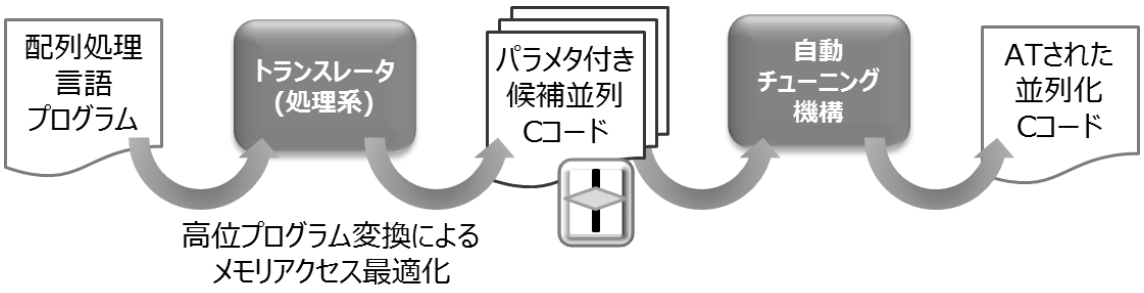

図 2.13. 配列処理言語を利用した自動チューニング方式

2.4.2 高位プログラム変換を用いた自動チューニング(1)

まずは、2.3.2 において処理系のプログラム変換で自動生成されたレンジ幅がパラメタ化された並列 C プログラムを利用した自動チューニングを試行する。これにより、レンジスケジューリング方式によるメモリアクセス最適化の効果を確認する。図 2.14 に、レンジの

幅を変化させた際の実行時間の変化を示す。レンジの幅は、1 から画像の横幅まで変化させている。レンジスケジューリング方式によるレンジの幅の計算値は、ラプラシアンフィルタと平滑化フィルタがそれぞれ 720、Prewitt フィルタは 360 となる。図 2.14 より、よい実行時間となる幅が計算できていることがわかる。今回評価に用いたアプリケーションは、実際にはレンジの幅を画像の横幅としても、ワーキングセットが L1 データキャッシュに収まるほど小さい。よって、実行時間はレンジの幅を画像の横幅とした時に一番よい結果となっている。

レンジスケジューリング方式では、レンジスケジューリングはレンジ幅を控えめに計算している計算結果が最適でないものがあるが、グラフの左側のレンジが小さ過ぎるところは選択されず、性能が下がり切ったところのレンジ幅が選択されているので有効性を確認することができる。

例えば組み込み機器の開発などで最終的に使用する真に高速な並列プログラムを作成するためには、このような自動チューニングが必要になることを示している。

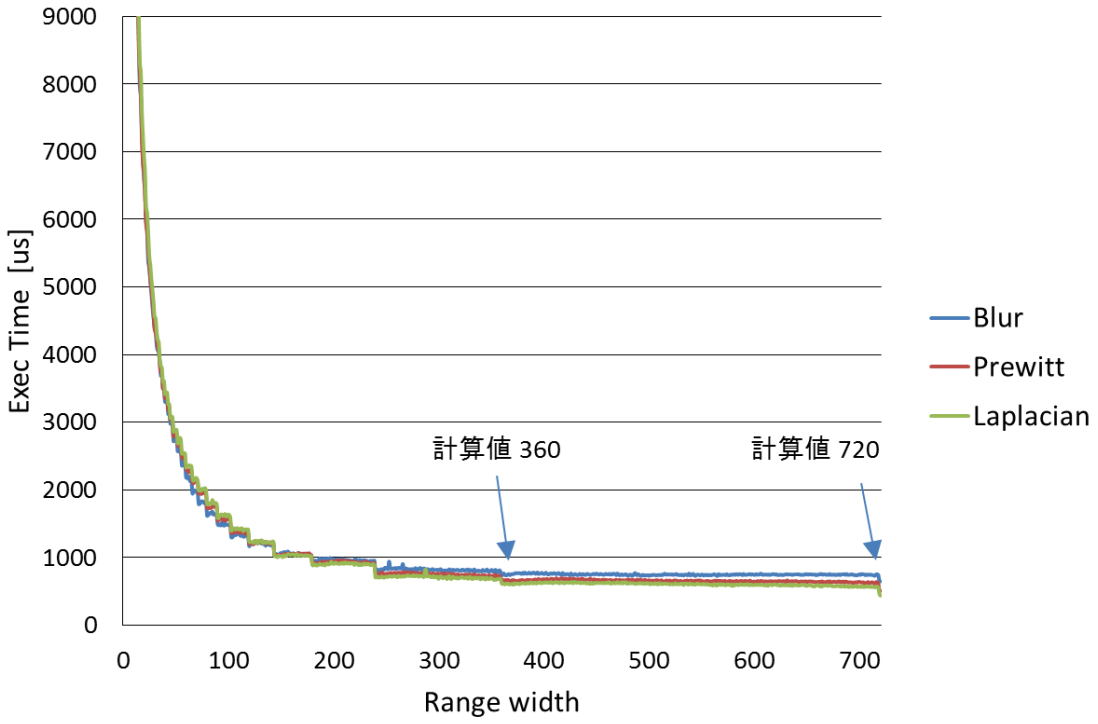

図 2.14. 自動チューニングによるメモリアクセス最適化の効果

2.4.3 高位プログラム変換を用いた自動チューニング(2)

図 2.15 に、信号処理の記述例として、レーダ信号処理の一つである CFAR 処理を本配列

処理言語で記述したプログラムを示す。CFAR 処理とは、クラッタと呼ばれる目標物以外からの不要な反射波を抑圧して、目標物からの反射波を抽出する処理である。

```
function ret = cfar(M)
  Type({[2048], 'single'}, M);
  Type({[2048], 'single'});
  n = 16;

  function r = f(p, near, far)
    d = Sum(near) + Sum(far);
    r = single(10.0) * log10(single(32.0) * p / d);
  end

  ret = Map(@f, M,
      MExtract(M, [-(n + 1)], [1], Size(M), [n]),
      MExtract(M, [2], [1], Size(M), [n]));
end
```

図 2.15.　CFAR 処理プログラム

受信信号 M の注目点に対し、近距離側 near と遠距離側 far のサイズが n の部分配列の総和を求め、注目点の値を総和で割る処理を、受信信号全体に対しておこなう。

図 2.15 のように MExtract()と Map()を組み合わせることで、アルゴリズムを直観的に記述できる。同時に、Map()により各注目点に対する関数 f を並列に実行可能であることが暗黙に記述されるので、処理系で並列性を容易に抽出することができ、データ並列処理レベルでの高位プログラム変換をしやすいという利点がある。

我々の自動チューニング方式の最大の特長は、最適化の調整ポイントを変化させたプログラムを高位プログラム変換で複数生成することで、多様性のある広い探索空間が得られる点である。MExtract()と Map()の組み合わせで記述したアルゴリズムは、高位プログラム変換により構造が大きく異なるパラメタ化された複数の候補 C プログラムに変換される。これらに自動チューニングを適用することで、最適な並列 C プログラムを得ることができる。

様々な局面で高位プログラム変換を可能にするためには、言語に簡潔で高い表現力が要求される。本配列処理言語では、配列の各要素に対して指定された関数を実行する際に、各要素の実行結果を次の要素の実行時の引数に渡すことで逐次的な処理をする高階関数 Scan()や、切り出す領域が元の配列からはみ出た場合の Overlay()関数による例外処理など

を備えており、プログラム変換に利用することができる。

高位プログラム変換の例として、図 2.15 の CFAR アルゴリズムに適用可能なものを 2 つ以下に示す。

(1)　計算量を削減する MExtract2Scan 変換

MExtract2Scan 変換は、計算結果を再利用することで計算量を削減する。同変換を適用した結果を図 2.16 に示す。変換前は部分配列の総和は毎回独立に計算されていたが、変換後は総和を計算する際に、前の注目点の計算結果を基に差分を足し引きして次の注目点を計算する Scan()処理に変換している。近傍処理では、MExtract2Extract で基本的には効率良い並列プログラムが生成できるが、計算量を削減する最適化がより有効性である境界条件が存在する可能性もある。

(2)　ベクトル化を促す MExtract2Extract 変換

MExtract2Extract 変換は、ベクトル化に適したプログラム構造へ変換することで、処理系でベクトル化されたプログラムを生成しやすくする。変換前は MExtract()で切り出す小さい不連続データに対する処理なのでベクトル化には不向きだが、これを Extract()で切り出す連続した大きな部分配列を組み合わせたベクトル化向きの処理に変換する。

i.MX515(ARM Cortex-A8)プロセッサ上で 2 つの高位プログラム変換を比較した結果を図 2.17 に示す。配列サイズ n が小さい場合、ベクトル化の効果で MExtract2Extract 変換版の性能が高い。しかし、n に比例して計算量が増加した場合は、計算結果の再利用の効果で n に依らず計算量が一定である MExtract2Scan 変換版の性能が高いことが分かる。この結果から応用によって異なる最適なプログラム構造を高位プログラム変換で作成する本方式の有効性が確認できる。

MATLAB 上のデータ並列処理レベルでの高位プログラム変換と自動チューニングを組み合わせることで、直観的に記述されたアルゴリズムから対象アーキテクチャの性能を引き出す並列 C プログラムを自動生成できる可能性を示した。

今後は、様々な高位プログラム変換のパタンを蓄積して匠の技の自動化を進めていく。

```
function r = f(p, d)
   r = single(10.0) * log10(single(32.0) * p / d);
end

function r = g(init, near_old, near_new, far_old, far_new)
   r = init - near_old + near_new - far_old + far_new;
end

init =    Sum(Overlay(Extract(M, [-(n + 2)],[n]),0)) + Sum(Extract(M, [1], [n]));
D = Scan(@g, init,
            Overlay(Extract(M, [-(n + 2)], Size(M)),0), Overlay(Extract(M, [-2], Size(M)),0),
            Extract(M, [1], Size(M)),    Extract(M, [1+n], Size(M)));

ret = Map(@f, M, D);
```

図 2. 16.　MExtract2Scan 変換後の CFAR 処理プログラム

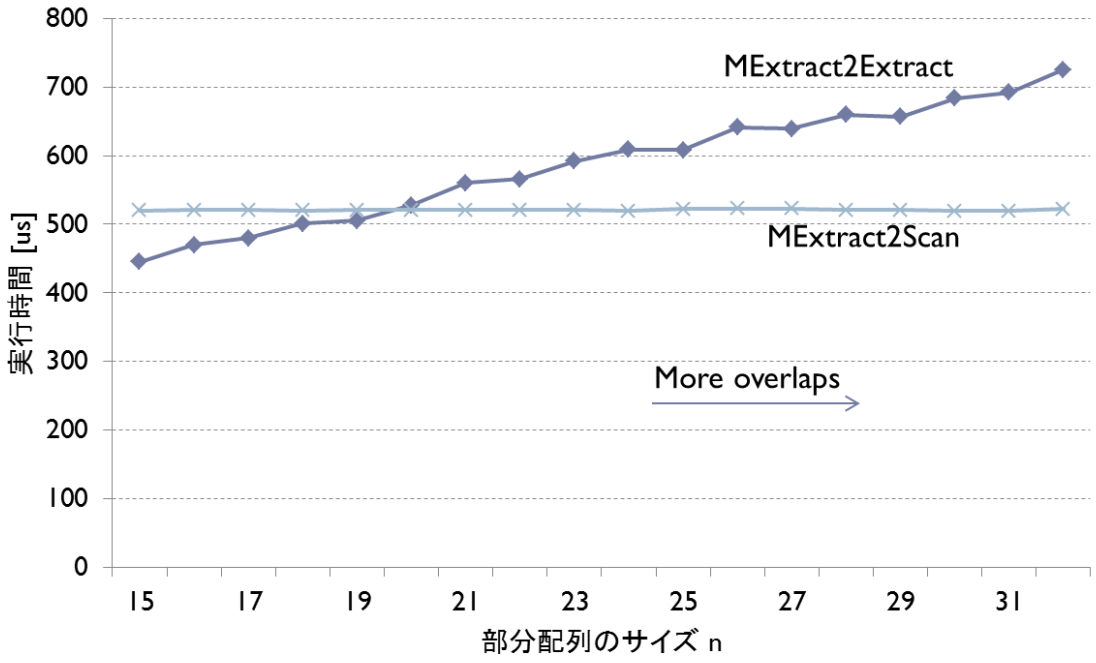

図 2. 17.　2 つの高位プログラム変換の性能比較

2.5　関連研究

X10[12] や Sequoia[15]などのマルチコア言語では、対象アーキテクチャを抽象化してアプリケーションプログラム開発者にみせ、そのうえで問題の分割方法を明示させている。これに対して、本開発方式では、記述レベルをあげることで対象アーキテクチャに依存する処理の記述をアプリケーションプログラム開発者から隠蔽し、対象アーキテクチャへのマッピングの一切を処理系で自動的におこなう。

中西ら[131] は、SIMD 記述の可搬性を保つために、演算レベルで SIMD 向けの共通記述方式を提案している。本処理系でも、より抽象度の高いレベルで共通記述を定義し、これを利用した C++プログラムを生成することも考えられる。

Intel ArBB[106]は、C++で配列処理を簡潔に記述するためのライブラリである。ArBB に比べ、本研究の配列処理言語が、多重配列と Map()の組み合わせにより複雑な配列処理をより簡潔に記述できる。

MATLAB プログラムの C プログラム自動生成ツールである Real-Time Workshop Embedded Coder は、MATLAB から標準的な可読性の高い C プログラムの生成を目的としており、特定のプロセッサ向けの並列化は行っていないが、配列処理言語より多様な MATLAB プログラムからの C プログラム生成が可能であり、互いに補完する関係である。

2.6　まとめと今後の課題

本研究では、提案するプログラミングシステムを用いることで、特定のメモリアーキテクチャに依存しない画像処理や信号処理のプログラムをアルゴリズムレベルで簡潔に記述できることができ、配列処理言語の処理系により対象メモリアーキテクチャに適合した高性能な並列化 C プログラムが自動生成できることを示した。配列処理言語によるプログラミングは、特定のメモリアーキテクチャや並列化を意識する必要がないためアルゴリズム開発者にとって使いやすく、配列処理言語が MATLAB 文法に準拠しているため、習得のコストも小さい。

今後の課題の一つは、GPU や FPGA に代表されるアクセラレータへの対応をおこなうことで、適用可能なプロセッサのバリエーションを増やすことである。アクセラレータは、配列処理の高速化に向いているがプログラミングが難しいため、本研究の手法が効果的に適用できると考えられる。また、本研究では代表的な画像処理や信号処理のプログラムで性能評価をおこなったが、より多様なアプリケーションの配列処理に適用することで、適用可能性の評価や新たな並列化手法の開拓につなげていく。

3. 階層メモリを持つ計算機クラスタ向けの一貫性管理方式を組み込んだ高性能ソフトウェア分散共有メモリシステム

3.1 Migratory Access を効率良く処理する権限委譲プロトコルを組み込んだホームベースソフトウェア分散共有メモリ

3.1.1 まえがき

PC クラスタや SMP-PC クラスタの普及と共に、分散メモリ上に仮想的な共有メモリの構築をユーザレベルのソフトウェアライブラリで実現するページベースソフトウェア分散共有メモリ(SDSM)システム[53,55,56,117]が注目されてきた。オペレーティングシステムの仮想記憶で用いられるページ単位でコヒーレンシを維持するページコヒーレンシプロトコルのメモリアーキテクチャは、ライトバックの戻り先であるホームノードがページごとにある固定ノードに決められている方式とそうではない方式に分類される。近年の研究[53,57]において、前者の固定のホームノードがページごとに存在する方式(以後、ホームベースプロトコル)を用いたホームベース SDSM システム[53,117]が、後者のホームノードを持たずに diff 分散方式 [55]を利用するページコヒーレンシプロトコルを用いたホームレスなシステム[55,56]より総合的に高い性能をもたらすことが明らかになっていて、前者が決定版となっている。

しかし、前者は後者が持つ多くの問題点を解決しているものの、ホームノードを設ける副作用として、同期操作ごとのホームノードへのライトバックが新たなオーバヘッドになる。よって、メモリアクセスパタンに適合したデータ配置(ホームノードの配置)ができるかどうかがパフォーマンスに影響を与える[58]。

なかでも、複数ノード間で頻繁にライトバックと読み出しが行なわれるメモリアクセスパタンでは、データ配置は難しく、ライトバックによって著しくパフォーマンスが低下する。また同メモリアクセスはロックで排他制御されるため、逐次化される。

そのため、このようなメモリアクセスの実行回数が比較的多いアプリケーションはそもそもスケーラビリティが期待できないので、SDSM システムの適用対象外になっている。事

実、これらのアプリケーションに限定すると diff 分散方式を利用したシステムが高い性能を示している[59,53]。また、階層的なネットワーク構成を持つ SMP-PC クラスタではこの傾向が更に顕著化する。なぜならホームノードへのライトバックのほとんどが SMP-PC ノード間を介したもので、SMP-PC ノードのローカリティを利用できないからである。

ロックを用いた排他制御は、ビジネスアプリケーションで頻繁に行われる集計処理[60]、画像処理におけるソフトウェアパイプライン処理、数値計算におけるリダクション演算などにおいて典型的に引き起こされる。また、性能ヘテロなクラスタシステムや、一つのアプリケーションで独占的に使用できないシステムなどにおいて、負荷をバランスさせるためにタスク並列にプログラムを記述することがある。この例では、タスクキューへのアクセスにおいてロックが利用される。

また、ロックを利用するとメモリアクセスが逐次化されるので大規模なクラスタシステムでは逐次部分の実行比率が大きくなってしまう。以上により、ロックで排他制御される共有メモリアクセスを効率良く処理することが、実用的な SDSM システムの実現に向けて重要な課題である。

ロックで排他制御される共有メモリアクセスのうち、排他制御操作ごとに実行されるホームノードへのライトバックを引き起こす可能性のあるものを migratory access [92]と呼ぶことにする。これまでに、複数ノードが migratory access を頻繁に実行するアプリケーションに対してもホームベースソフトウェア分散共有メモリシステムを効率的に適用可能とすることを目的に研究をおこなってきた[125]。

この先行研究では、ホームベースプロトコルを前提に、一連の migratory access を実行する複数ノード間でページを直接更新できる権限を一時的に委譲し巡回させることで同アクセスに伴うライトバックのオーバヘッドを削減する新しいページコヒーレンシプロトコルとして権限委譲プロトコルを提案している。

また、一連の migratory access の実行に先行して複数ノードから発行されるロックリクエストが、システムの処理が追いつかずにキューイングされる状態で同アクセスを自動的に検出し、権限委譲プロトコルで一括処理する機構も提案している。提案方式を既存の SDSM システム JIAJIA version 2.1 [53]に実装し、評価をおこなった。

以下、3.1.2 でこの先行研究が前提とするコンシステンシモデルとホームベースプロトコルについて述べ、その問題点を示す。3.1.3 では権限委譲プロトコルと提案機構を説明する。また、3.1.4 ではベンチマークプログラムによる提案方式の性能評価結果を示す。3.1.5 で関連研究を概観し、3.1.6 で今後の課題を述べる。

3.1.2 従来のホームベースプロトコルの課題

[Scope Consistency と Lock-Based ページコヒーレンシプロトコル]

この先行研究で前提とするコンシステンシモデル Scope Consistency [61]と同コンシステンシモデルを実装するページコヒーレンシプロトコル Lock-Based プロトコル [53](以後、特に断らない場合は、本研究におけるホームベースプロトコルは Lock-Based プロトコルを指すこととする)の概要を説明する。

Scope Consistency では、あるロック変数 i に対するロック操作(Acq(Li))とアンロック操作(Rel(Li))(以後、Scope i)の中で実行された write の値が、後続する Scope i の中で read できることのみを保証する。そのために、Rel(Li)を実行するノードは、対応する Acq(Li)を実行するノードに対し、それ以前に実行された write を次の方法で通知する。

まず、ロックマネージャと呼ぶロック変数ごとに決められたある固定ノードが、ロック構造体(Li)を用いて write の実行情報(もしくはページ無効化情報と呼ぶ)を管理する。Acq(Li)を実行するノードは、ロックリクエストをロックマネージャへ発行する。ロックマネージャでは、受信したロックリクエストを FIFO(以後、ロックリクエストキュー)に蓄えて、 1 リクエストずつこれを処理(以後、サービス)していく。発行したロックリクエストが処理され、ロックマネージャより Li を獲得したノードがページ n(以後、Pn)に対して write を実行すると、n は Li に追加される。Rel(Li)を実行すると、twin と呼ぶ Pn に対して write する直前の Pn のコピーと Pn の差分情報である diff が生成され、これを用いてホームノードへライトバックしアップデートする。ライトバックが完了したことを確認し、Li をロックマネージャに対して解放する。これにより、それ以降に Acq(Li)を実行し Li を獲得するノードは、write が実行された Pn の検出と無効化をおこなうことができ、write された値は Pn のホームノードよりアップデートされたページをフェッチすることで read できる。

[対象とするアクセスパタン]

本研究では、migratory access を議論の対象とする。これまでにいくつかの定義 [58,92] がなされているが、本研究ではホームベース SDSM においてライトバックを引き起こす、オーバヘッドとなる可能性のある共有メモリアクセスを扱う便宜上、migratory access を以下のように再定義する。

migratory access : Scope Consistency において、同一のロック変数に関するロック操作とアンロック操作を用いて排他制御する共有メモリに対する read & write 及び write

[migratory access を実行する場合の問題点]

ホームベース SDSM 上で migratory access を実行する場合の問題点を明確にする。そのために、まず、ホームノードを設ける副作用であるアンロック操作ごとのホームノードへのライトバックのオーバヘッドとホームノードの配置の関係について、diff 分散方式と対比させて議論する。

ホームベースプロトコルが diff 分散方式に対してパフォーマンスが劣る可能性がある 1P-1C (one producer with one consumer) [58]と呼ぶページ単位の共有パタンを用いて説明する。1P-1C の共有パタンにおいて producer と consumer 以外のノードがホームノードである場合、diff は diff 分散方式では producer がローカルに保持するのに対し、ホームベースプロトコルではホームノードへライトバックされるため、これがオーバヘッドとなる。

1P-1C の共有パタンが交互に頻繁に複数ノードで実行される migratory access ではどのノードにホームノードを配置してもホームノードをアップデートするためにホームノード以外のノードはアンロック操作の度に diff をホームノードに転送するので、これが過大なオーバヘッドとなる(図 3.1)。 図中の太線はページの転送、細線は diff の転送を表している。

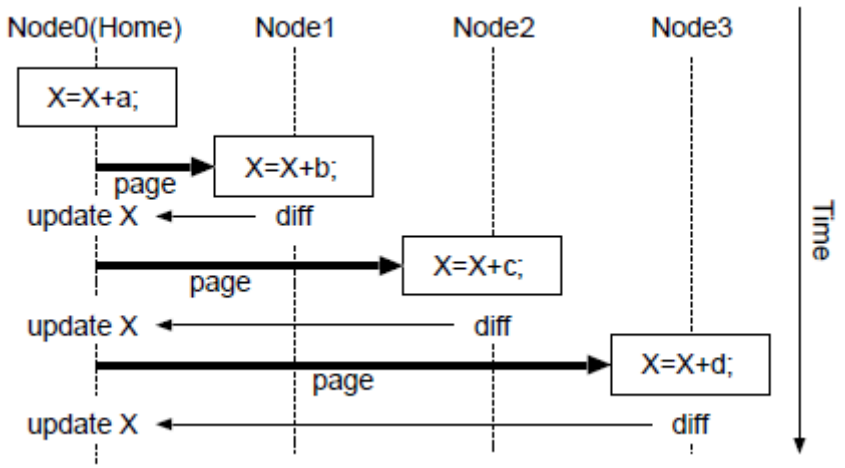

図 3.1.　ホームベースプロトコル

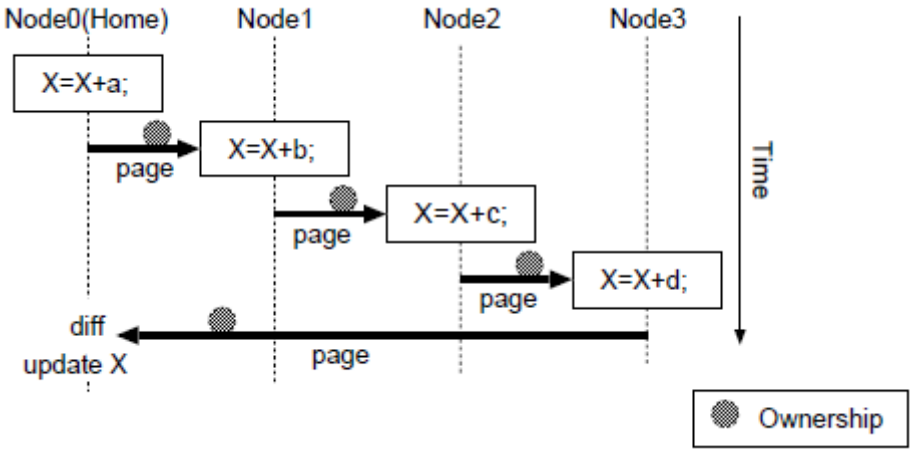

図 3.2.　権限委譲プロトコル

3.1.3　Migratory Access を効率良く処理するホームベースソフトウェア分散共有メモリ

[権限委譲プロトコル]

複数ノードで migratory access が頻繁に実行される場合、アンロック操作ごとのホームノードへのライトバックがオーバヘッドとなり、パフォーマンスが著しく低下する。この一連の migratory access を実行する複数ノード間でページを直接更新できる権限の一時的な委譲を繰り返し、権限とページを巡回させる。そして、一連の migratory access の終了後、ページをホームノードにライトバックする。これにより、従来のホームベースプロトコルで必要とした diff/twin の生成と頻繁にホームノードにライトバックするオーバヘッドの削減が可能となる。

図 3.2 を用いて権限委譲プロトコルを説明する。まず、権限委譲プロトコルのために、新たにページのホームノードの権限(以後、ホーム権限)という概念を導入する。ホーム権限は「直接ページを更新できる(twin/diff を生成しライトバックする必要がない)」、ページごとに存在する、ロック変数に固有の権限で、初期状態ではページの本来のホームノードが所有している。図中の円形がホーム権限を表している。ホームノード以外のノードが

ホーム権限を与えられることで、同ノードはそのページの仮想的なホームノードとみなされる。これにより本来のホームノードが持つ「直接ページに書き込める」利点を持つことになる。

ホーム権限は、migratory access が引き起こすページフォルトのときに、そのページに対して migratory access を実行したノードに委譲される。例では、ノード 1 が共有変数 X に対する read ミスに起因するページフォルトで X を含むページ Pn のホームノード 0 に対してページリクエストをおこなう。図中の円形の移動は、ホームノード 0 がノード 1 に対してクリーンなページの転送(以後、ページサービス)をおこなう時に、ロック変数 i のページ Pn のホーム権限 O[i,Pn] を、一時的にノード 1 に委譲することを示す。

O[i,p_n]を委譲されたノード 1 は、Pn の仮想的なホームノードとみなされる。従って従来のホームベースプロトコルで行っていた twin/diff の生成と、同 diff を用いた本来のホームノード 0 へのライトバックをおこなう必要がなくなる。また、ノード 1 は、O[i,Pn]が一時的に委譲されたことを、ロック構造体 Li に付加してこれを解放する。それによって、つぎにノード 2 が Li を獲得した際に、O[i,Pn]の委譲を検出できる。これにより、1P-1C のページ単位の共有パタンにおいて、producer がホームノードに配置されている関係を常時保ちながら、更にホーム権限を次々に委譲することが可能となる。

最後に Pn に migratory access を実行するノード 3 が複数ノード(ノード 1、ノード 2、ノード 3)で write が実行されたページをホームノード 0 に転送し、O[i,Pn]も同ホームノードに戻す。ホームノード 0 は、O[i,Pn]を委譲している間に Pn に対して行われた変更点を diff として抽出し、これを用いてホームノードをアップデートすることが必要となる。このために、ホームノード 0 は、O[i,p_n]を委譲する直前のページのコピー(Home Twin [i,Pn])を予め作成しておき、同コピーとノード 3 から転送されたページの diff を作成し、これを用いてホームノードのページをアップデートする。このように、一連の migratory access において、ホーム権限を巡回させることでホームノードを介することなく共有変数を更新できるので、一連の権限を委譲されたノード列の最後までライトバックのオーバヘッドを削減することができる。

[権限委譲プロトコルとホームベースプロトコルを動的に切り替える機構]

コンテンションの検出手法:

まず、SDSM システムにおける実行時の特別な状態を利用することで、同システムの処理におけるコンテンションを動的に検出する手法を提案する。この実行時の特別な状態とは、複数ノードから発行されるページリクエストやロックリクエストが、システムの処理が追いつかずに同リクエストを受信したノードでキューイングされている状態である。この状態では複数ノードがリクエストが処理されるまでただ待っているので、コンテンションを検出しキューイングされている複数のリクエストを一括処理できればシステムを高速化す

ることができる。

権限委譲プロトコルの適用方式:

上記のコンテンションの動的検出手法を利用して、権限委譲プロトコルを適用する方式について論じる。

複数ノードでmigratory accessが頻繁に実行されるケースでは、同アクセスの実行に先行して発行されたロックリクエストがロックマネージャのロックリクエストキューに複数到着する。

ロックリクエストが複数キューイングされている場合、キューイングされているリクエストの数と同数のノードはロックがサービスされるのをただ待っているので、権限委譲プロトコルが効果を発揮する。

つまり、オーバヘッドとなり得るmigratory accessの検出は、ロックリクエストキューの状態を検査しロックのコンテンションが高いことを検出することで可能となる。ロックリクエストが一定数以上キューイングされている時に、ホームベースプロトコルから権限委譲プロトコルに動的に切り替え(以後、権限委譲プロトコルを起動する)、これらロックリクエストに関する一連のmigratory accessを一括処理し、同処理終了後に再度切り替える(以後、権限委譲プロトコルを終了する)ことでロックをサービスするスループットをあげる仕組みを提案する。また、権限委譲プロトコルの起動から終了までを「権限委譲の旅」と呼ぶことにする。

ホームベースSDSM上に、権限委譲プロトコルを実装する図3.3の例を使って説明する。例では、ノード0以外のノードが、共有変数X、Yを含むP0と、同Zを含むP1(ホームノードはいずれもノード0)に対して、図中のようなmigratory accessを実行する。
Acq(L0)の実行で発行されるロックリクエストはロックリクエストキューに、(ノード1→ノード2→ノード3)の順番でキューイングされている。この順番を、「権限委譲の旅の順番」と定義する。

例において、権限委譲プロトコルの起動条件を、ロックリクエストが3つ以上キューイングされていることとする。ロックリクエストが3つ未満のとき、ロックマネージャはホームベースプロトコルに従いロックのサービスをおこなう。

ロックマネージャは、ロックのサービスをおこなう時に権限委譲プロトコルの起動条件を評価し、条件が成立した場合、権限委譲プロトコルを起動し、以下の手順が実行される。

ロックマネージャは、ノード1にL0をサービスするときに、旅の順番も通知する。ノード1は、L0を解放するとき、ロックマネージャを経由せずに、通知された旅の順番において次点のノード2に同順番を付加したL0を直接転送する。同様に、ノード3が旅の順番が付加されたL0を受信する。

旅の順番の最後であるノード3は、権限委譲プロトコルを終了させるために、以下の手

順を実行する。ホームノードをアップデートするために、一時的に O[0,P0]を委譲されているノード 2 と O[0,P1]を委譲されている自分自身に権限委譲プロトコルの終了リクエスト(図中の shutdown)を発行する。これにより、ホームノードにホーム権限が戻り、アップデートも正しく行われる。同手順の終了後、L0 をロックマネージャに転送する。

権限委譲の旅の間に発行されロックマネージャに到着するロックリクエストは新規にキューイングされる。また、Scope 0 の外で発行されるページリクエストは、上記の旅の間も本来のホームノードに到着し、同ノードがページをサービスする。

[マルチプルライタの実現]

Overwrite-safe Twin:

False Sharing の問題を解決するためには、同一ページの書き込みを複数ノードに同時に許すマルチプルライタプロトコルの導入は不可欠である。ここでは、権限委譲プロトコルにおいてマルチプルライタプロトコルを実現するために Overwrite-safe Twin というメカニズムを提案する。

図 3.3 の例を使ってこれを説明する。ここではノード 1 に注目する。ノード 1 は、L0 を獲得し、X に write する。L0 を解放後、続けて L1 を獲得し、(L1 のページ無効化情報により P0 がダーティであれば P0 を一度無効化し、P0 を改めてフェッチしてから) Y に write をおこなうと、この後で O[0,P0]が委譲されているノード 1 が P0 をノード 2 にサービスするときに、migratory access の中で先におこなった X への write のみが反映されていなければならないにも関わらず、Y への write まで反映されてしまう。これを防止するために、ノード 2 にサービスする P0 のコピー(以後、Overwrite-safe Twin)は別に保存しておき、ノード 2 から P0 がリクエストされたら、同 Overwrite-safe Twin をサービスする。

Overwrite-safe Twin を作成するタイミング:

上記のケースにおいて、ノード 1 が獲得した L1 のページの無効化情報により P0 が無効化されたり、ノード 1 が P0 に write を行わないのであれば、ノード 2 に P0 をそのままサービスすればよいので、Overwrite-safe Twin を作成する必要がない。Overwrite-safe Twin は必要になった時点で作成する。

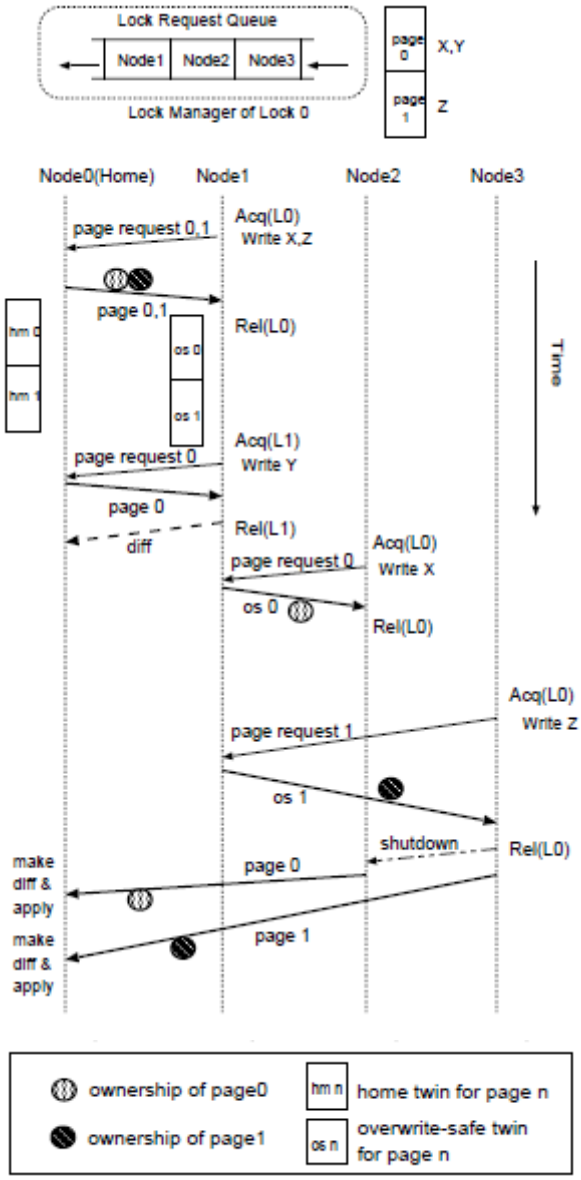

図 3.3.　権限委譲プロトコルにおけるマルチプルライタ

［積極的な権限委譲プロトコル］

ここまでは、ページがリクエストされた時点でページサービスをおこなう権限委譲プロ

トコル(lazy な実装)について議論してきた。migratory access を実行する複数ノードが同一ページを参照する可能性が高い場合には、ロックのサービスと同時にページサービスをおこなうことで更なるオーバヘッドの削減を目指す積極的(以後、eager) な実装が考えられる。　具体的には、権限委譲の旅の順番において次点のノードにロック構造体を転送するときに、 migratory access をおこなった複数のページも合わせて転送する。eager な実装にする利点は、ページごとのページリクエストがページのプリフェッチによりなくなることである。欠点は、予め転送しておいたページを利用しなかったときに、転送量が増えることである。

SMP-PC クラスタを対象とする権限委譲の旅の順番の最適化:

SMP-PC クラスタは階層的なネットワーク構成を持つため、権限委譲の旅の順番を最適化することで SMP-PC ノードのローカリティを効率良く利用することが可能となる。図 3.3 の例で説明する。例では、ロックのリクエストはロックリクエストキューに、(ノード 1→ノード 2→ノード 3)の順番でキューイングされている。ここで SMP-PC クラスタシステムの構成において、ノード 1 とノード 3 が同じ SMP-PC ノードに属し、ノード 2 がこれとは別の SMP-PC ノードに属するとする。この場合、権限委譲の旅において、SMP-PC ノード間を 2 回通過する。旅の順番を(ノード 1→ノード 3→ノード 2)と変更することで、これを 1 回にすることが可能となる。この旅の順番の最適化の正当性を証明する。まず、各ノードからのロックのリクエストがロックマネージャに到着する順番はタイミング依存であること、また、ロックリクエストキューにキューイングされる各ノードからのロックリクエストは高々一つであることから、上記の旅の順番の変更は不都合を生じない。

3.1.4　評価

[評価方法]

既存の SDSM システム JIAJIA version 2.1 [53]をベースに提案方式を実装し、ベンチマークプログラムで性能を評価した。JIAJIA は、Lock-Based プロトコルを用いて Scope Consistency を実装している。

[評価用ベンチマークプログラム]

ベンチマークプログラムには、NAS Parallel Benchmarks(NPB)の IS と単純なサンプルプログラムを用いた。問題サイズは表 3.1 の通りである。

IS では、各計算ステップ終了時に、すべてのノードが一斉に排他的に read&write を実行

する。

サンプルプログラムは、タスク並列処理のアプリケーションにおけるタスクキューに対する非同期的な migratory access を想定している。migratory access の処理効率を検証することが目的である。プログラムは、タスクキューにみたてた共有メモリ領域(4 バイト)への排他的な read&write をすべてのノードで合わせて N 回繰り返す。

表 3.1.　問題サイズ

ベンチマークプログラム	問題サイズ
IS サンプルプログラム	鍵の数 2^26, 鍵の最大値 = 2^14 N = 320

表 3.2.　評価実験環境

	システム A	システム B
# Nodes	16(32CPU)	8(16CPU)
CPU	PentiumIII 866 MHz	PentiumIII 866 MHz
Memory	1 GB	640 MB
Network	100 BASE-TX	Myrinet
OS	Red Hat Linux 7.1	Red Hat Linux 6.2
Compiler	gcc 2.96	gcc 2.91.66

表 3.3.　権限委譲プロトコルの基本コスト

操作	コスト[usec]	
	システム A	システム B
ページフォルト処理(SMP ノード間)	725	311
ページフォルト処理(SMP ノード内)	219	236
各種 twin 作成	33	28
diff 作成	57-175	52-171
diff 適用	1-65	1-72

[評価実験環境]

提案方式に対するネットワーク性能の影響を検証するために、実験には 2 つのプロセッサを持つ 2 種類の SMP-PC クラスタを用いた(表 3.2)。

[評価項目]

単一プロセッサノードで構成される PC クラスタを対象にした評価:

表 3.2 のそれぞれの SMP-PC クラスタの各ノードの 1 つのプロセッサを用いた 2 種類の PC クラスタ上で提案方式の評価を行なった。評価に用いたページサイズは 4K バイト、権限委譲プロトコルの起動条件はロックリクエストキューにリクエストが 2 つ以上キューイングされていることとした(以降の評価においても同条件)。また、SDSM システム JUMP version 1.0 [62]との性能比較もおこなった。JUMP が実装している Migrating-Home プロトコルではページサービス時にホームノードの再配置をおこなう。

SMP-PC クラスタを対象にした評価:

SMP-PC クラスタで提案方式の評価をおこなうために、JIAJIA を SMP-PC クラスタで動作させた。SMP-PC クラスタ上の実行においては、SMP-PC クラスタの各 SMP-PC ノードで 2 つのプロセスを起動させ、SMP-PC ノード内通信にはノード間と同様に UDP を用いている。

SMP-PC クラスタを対象にした権限委譲の旅の順番の最適化手法の評価:

SMP-PC クラスタ上で提案方式における権限委譲の旅の順番の最適化手法の効果の評価をするために、カットオフのある粒子シミュレーションプログラムを利用した。このプログラムは、2 万個の粒子間の相互作用の計算を 60 ステップおこなうものである。プログラムの概要を説明する。カットオフの判定は空間分割法で粒子をセルに割り当てることでおこなう。各計算ステップが終るとバリア同期をとり、セル間を移動した粒子最適なセルに再度割り当てる。バリア同期をとる直前にローカルに計算した粒子の加速度を足し込む migratory access を実行する。セルをサイクリックに各プロセッサに割り当てることでアプリケーション側で負荷分散もしているが、それでもプロセッサごとに計算量は異なる点が IS と異なる。これにより、各ステップで権限委譲の旅の順番が異なる。

[権限委譲プロトコルに関する基本コスト]

権限委譲プロトコルに関する基本的なコストを、対象とするクラスタシステムごとに表 3.3 に示す。ここで、ページフォルト処理のコストは、ページアクセスによるページフォル

トによりホームノードからページをフェッチし、ページアクセスが再開するまでの時間である。

[評価結果]

単一プロセッサノードで構成されるPCクラスタを対象にした評価:

JIAJIA version 2.1、JUMP version 1.0及び提案方式を実装したシステムの3つのシステムについてISの実行時間を測定した結果を図3.4と図3.5に示す。各図にはロック操作のコストも示す。ロック操作のコストとして、Scopeに入る以前のread/writeが終了した時点からScopeの中のread/writeを開始するまでの時間を測定する。提案方式のeagerな実装の場合には、ロック操作時間に積極的に転送されたページの処理が含まれる。図に示すのは、ベンチマークプログラムの実行においてロック操作の実行時間が最も長かったノードのもの(以降の図についても同様)である。

JIAJIAの評価は、version 2.1の新機能であるHome Migration、Write Vector Technique、Adaptive Write Detection [53]を用いたものである。逐次実行では、JIAJIAライブラリはリンクされているが、オーバヘッドはほとんどない(以降の評価についても同様)。ISの逐次実行時間を表3.4に示す。

まずISの評価結果について説明する。 図3.4、図3.5より、提案方式ではmigratory accessを効率良く処理してロック操作時間を短縮し、高い性能を得ている。ISをJIAJIAと提案方式を用いて16ノードで並列実行した場合のページリクエストの実行回数と、ホームノードのアップデートに利用されたdiffの数を表3.5に示す。ページリクエストの実行回数をJIAJIAと比較する。eagerでは積極的に転送されたページを利用できたことで実行回数を削減しているが、lazyではこれが増加してしまっている。これは、ホーム権限が委譲されているノードに対してページの本来のホームノードが同ページとそのホーム権限を取り戻すためにページリクエストをおこなうからである。しかしホームノードをアップデートするdiffの数が大きく削減できているため性能が向上している。

つぎに、サンプルプログラムの実行時間図 3.6 を比較する。サンプルプログラムはmigratory accessのみを抽出したプログラムであり、権限委譲プロトコルの基本性能と考えることができる。

PCクラスタBの8ノードにおいては、eagerではJIAJIAと比較して46.1%、JUMPと比較して76.2%の処理速度の向上が得られた。サンプルプログラムをJIAJIAと提案方式を用いてPCクラスタAの16ノードで並列実行した場合のページリクエストの実行回数とdiffの数を表3.6に示す。ISの場合と同様の傾向がみられることが確認できる。

SMP-PCクラスタを対象にした評価:

SMP-PCクラスタの8ノード(16プロセッサ)においてISを評価した結果を図3.7に示す。

評価結果を図 3.4 の PC クラスタ A の 16 ノードで並列実行した場合と比較する。JIAJIA では､単一プロセッサノードで構成される PC クラスタ A を用いた場合の性能が高い｡しかし、権限委譲プロトコルを用いると SMP-PC クラスタ A を用いた性能が高い。これは、IS ではロックリクエストが複数ノードで規則的な順番で発行され、権限委譲の旅の順番が最初から最適になっているからである。これにより、SMP-PC ノードのローカリティを利用できていることで、性能が向上している。

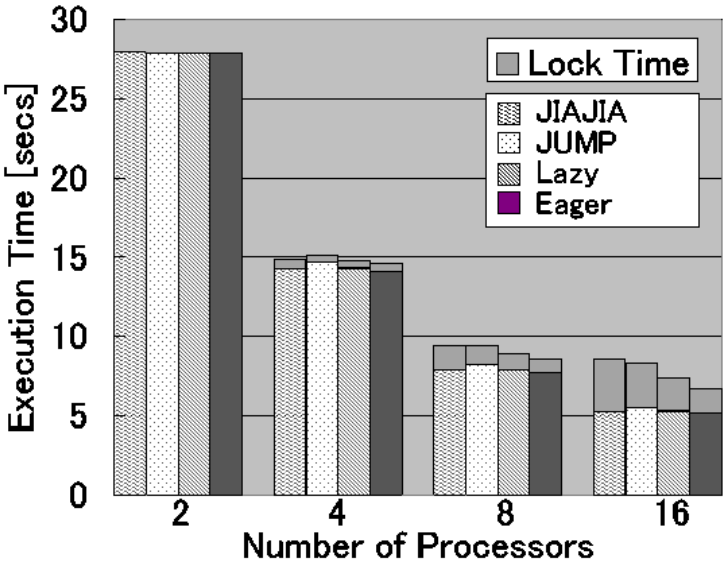

図 3.4.　PC クラスタ A における性能比較(IS)

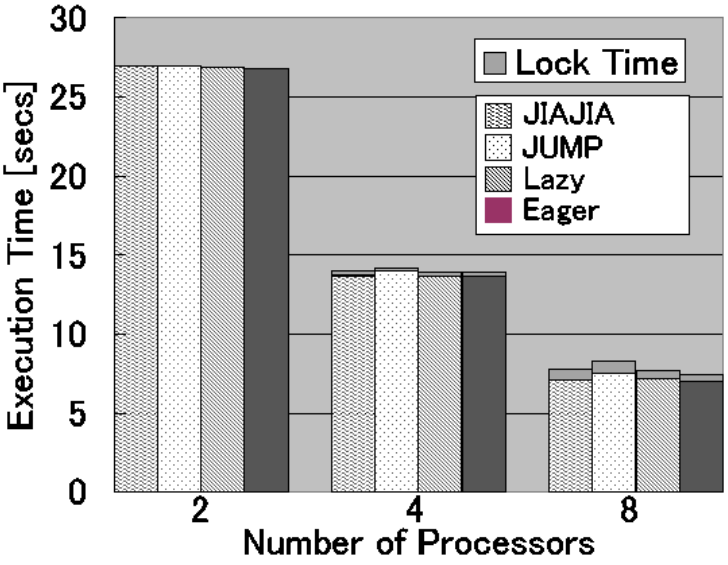

図 3.5.　PC クラスタ B における性能比較(IS)

表 3.4.　逐次実行時間

操作	逐次実行時間［usecs］	
	システム A	システム B
IS	55.2	53.8

表 3.5.　IS の各種イベント実行回数

operation	JIAJIA	lazy	eager
page requests	4800	4960	2720
diff updates	2400	320	320

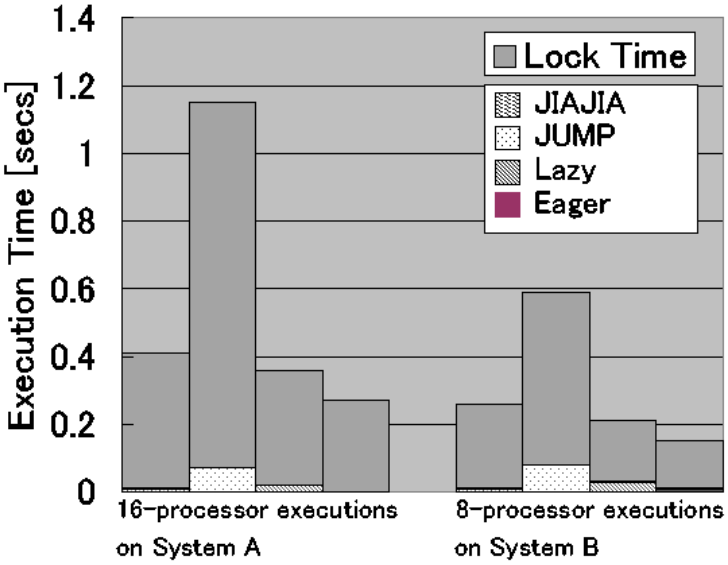

図 3.6.　migratory access の処理時間（サンプルプログラム）

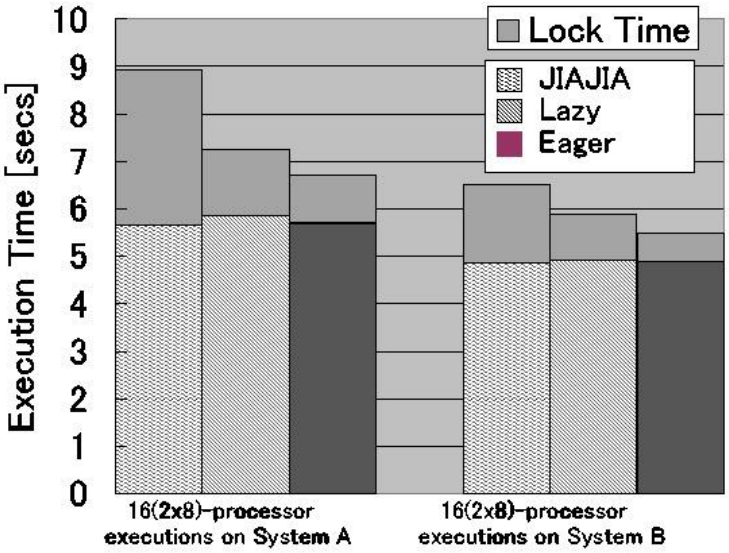

図 3.7.　SMP-PC クラスタにおける性能比較(IS)

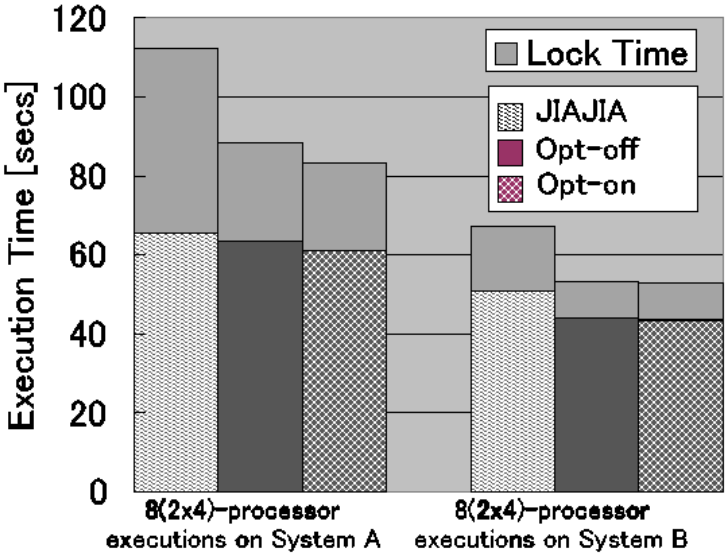

図 3.8.　権限委譲プロトコルの旅の順番の最適化手法の効果

表 3.6.　サンプルプログラムの各種イベント実行回数

operation	JIAJIA	lazy	eager
page requests	315	334	22
diff updates	300	23	23

表 3.7.　逐次実行時間

逐次実行時間［sec］	
システム A	システム B
164.7	214.0

SMP-PC クラスタを対象とした権限委譲プロトコルの旅の最適化の評価:

粒子シミュレーションプログラムの評価結果を示す。逐次実行時間を表 3.7 に示す。2 種類の SMP-PC クラスタの 4 ノード(8 プロセッサ)を用いた実行時間を図 3.8 に示す。 図 3.8 では、JIAJIA と、権限委譲プロトコル eager に関して最適化をおこなった場合とそうでない場合で比較したものである。最適化をおこなうことにより、SMP-PC クラスタ A で 5.65% の性能向上を得た。SMP-PC クラスタ B では、SMP-PC ノード内と SMP-PC ノード間で通信性能に大きな違いがないことから最適化の効果は小さく、性能向上は 0.74% にとどまった。

[提案プロトコルの改良に関する検討]

上述においては、ホーム権限委譲の旅を平等に扱っていた。しかし、中には、権限委譲プロトコルのオーバヘッドが逆に性能を低下させる旅(以後、悪い旅)があり得る。例えば、旅の中で migratory access が実行されない場合、Home Twin をつくることはオーバヘッドとなる。このようなことに対処するために、権限委譲プロトコルを起動するか否かを前回の旅の評価によって決定し、オーバヘッドをなくす方法などが考えられる。このように悪い旅を省略するだけでよいので、オーバヘッドを容易になくすことが可能である。

3.1.5 関連研究

一部のホームベース SDSM システム [53,120]では、ホームノードを動的に再配置することにより diff の転送量を減らすよう工夫している。しかし、再配置はバリア同期時に行われるため、migratory access による問題を解決するものではない。

migratory access の処理の効率化に対して、アクセスパタンに応じてシステムがシングルライタプロトコルとマルチプルライタプロトコルを選択するシステムが提案されている[63]。同方式は、ページ全体を更新するような migratory access のみを対象としており、その適用範囲は狭い。

JUMP で提案されている Migrating-Home プロトコルでは、ページサービス時にホームノードの再配置をおこなうことによりホームノードの再配置を migratory access パタンにまで適合させる点で、権限委譲プロトコルと類似している。しかし、権限委譲プロトコルは実際にホームノードの再配置を行わないでホーム権限のみを委譲する点で異なっている。JUMP では、ホームノードの再配置には、それを全ノードに通知するブロードキャストを伴うため、ホームノードの再配置が多発する migratory access ではこれが新たなコストとなりスケーラビリティが得られない。一方、権限委譲プロトコルでは、ホーム権限の一時的な委譲のみを行い、ホームノードの再配置は行わないので、余計な通信オーバヘッドがない。また、権限委譲プロトコルでは、ホームノードの再配置は行わないので、ホームノー

ドの再配置時に発生する可能性がある配置ミスを起こすこともない。SMP-PC クラスタを対象にした場合、上記オーバヘッドによる問題は更に顕著化すると考えられる。更に、提案方式では一連の migratory access を一括処理することで、SMP-PC クラスタを対象に有効な最適化を行なうことを可能としている。

また、JUMP との比較において Overwrite-safe Twin を作成するオーバヘッドがあるが、Overwrite-safe Twin の作成はクリティカルパスの外で行なわれるので、オーバヘッドは大きくない。

3.1.6 まとめと今後の課題

この先行研究では、一連の migratory access を実行する複数ノード間でページを直接更新できる権限を一時的に委譲し、巡回させることで同アクセスに伴うライトバックのオーバヘッドを削減する新しいページコヒーレンシプロトコルとして権限委譲プロトコルを提案した。また、一連の migratory access の実行に先行して複数ノードから発行されるロックリクエストが、システムの処理が追いつかずにキューイングされる状態で同アクセスを自動的に検出し、権限委譲プロトコルで一括処理する機構の提案もおこなった。

提案方式を既存のソフトウェア分散共有メモリシステム JIAJIA に実装し、ベンチマークプログラムで性能評価した結果、PC クラスタ及び SMP-PC クラスタ上で高い性能を得られることが明らかになった。

この先行研究をおこなった時点では PC クラスタ及び SMP-PC クラスタで評価を行なったが、今後は複数のクラスタを使用するマルチクラスタなどの計算環境が利用可能になると考えられる。マルチクラスタでは、クラスタノード間とクラスタノード内で通信性能が大きく異なる。権限委譲プロトコルが有する性能の高いホームベースプロトコルベースにブロードキャストを要せずに migratory access を効率良く処理できること、及び、権限委譲プロトコルの適用方式により権限委譲の旅の順番の最適化が可能であるという階層性に強い二つの利点をマルチクラスタ上に展開することが今後の最大の課題である。

3.2　マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ

3.2.1　まえがき

高速ネットワークで結合された要素クラスタ群が高レイテンシの LAN で結合されているマルチクラスタや、同一のネットワークを用いながらスイッチを階層的に用いているためにノード間通信で通過するスイッチ数が異なるような大規模クラスタのように、ノード間の相互結合網にハードウェア的な階層性があり、ノード間の通信レイテンシや通信スループットが均一でないようなクラスタをマルチクラスタと呼ぶ。マルチクラスタは、並列処理の共有アーキテクチャになっていくことが今後期待されている。

このために、このように多様なクラスタシステム上で、高性能が得られ、かつ、効率良く利用できる並列プログラミングライブラリの構築が重要となる。これに対して、メッセージパッシング方式が主に利用されることが多いと考えられる。

しかしながら、メッセージパッシング方式を用いた場合、多様な構成のクラスタシステム上で高性能を達成するには、同構成に適したデータ転送をおこなうためのプログラムの明示的な記述を要する。

一方で、クラスタシステムの分散メモリ上に、仮想的な共有メモリの構築を、ユーザレベルのソフトウェアライブラリで実現するページベースソフトウェア分散共有メモリ（ソフトウェア分散共有メモリ: Software Distributed Shared Memory）も、選択肢のひとつとして考えられる。ソフトウェア分散共有メモリを用いると、ソフトウェア分散共有メモリライブラリがクラスタシステム構成に適合するようにデータ転送を制御する機能を有すれば、同構成を透過的に扱える。これにより分散メモリシステム上で共有メモリプログラミングモデルを提供することとあわせて、柔軟な並列プログラミングモデルを構築できる可能性がある。

また、ソフトウェア分散共有メモリの用途はアプリケーション開発者が直接ソフトウェア分散共有メモリプログラムを記述するだけのものではない。例えば、従来は共有メモリシステムで実現されていた OpenMP やマクロデータフロー処理が、ソフトウェア分散共有メモリを実装したクラスタ上でそれぞれ実現されている[116,117]ように、その利用用途も広い。

しかしながら、従来の多くのソフトウェア分散共有メモリ方式は、高々32CPU 程度の小規模クラスタを志向して設計されている。このため、大規模システム上では、パフォーマンスのスケーラビリティが得られない。

例えば、小規模 PC クラスタ上で高性能を達成する既存のソフトウェア分散共有メモリシステムが利用するライトバック型のページ一貫性制御方式をそのまま利用した場合、ノード間の相互結合網の形態ごとに、以下のような問題点がある。

1) Gbps の高速ネットワークで結合された要素クラスタ群が、Mbps の LAN で結合されているキャンパスグリッド規模のマルチクラスタ上では、要素クラスタ間でのページ転送をともなうクラスタ間メモリアクセスがオーバヘッドになる。
2) 同一ネットワークを用いながらスイッチを透過的に用いているため、ノード間通信で通過するスイッチ数が異なるような大規模クラスタ上では、1)の問題点に加えて、小規模 PC クラスタ上で利用した場合でもオーバヘッドとなるホームノードからのページ読み出しのコンテンションが顕著になる。

そこで本稿では、これら問題点を解決し、多様なクラスタシステム上で高性能・高スケーラビリティを達成するソフトウェア分散共有メモリを実現する、マルチホーム方式を提案する。

本稿では、ノード間の相互結合網にハードウェア的な階層性があり、ノード間の通信レイテンシや通信スループットが均一でないようなクラスタをマルチクラスタと呼ぶことになる。つまり、本稿におけるマルチクラスタは、Gbps の高速ネットワークで結合された要素クラスタ群が、Mbps の LAN で結合されているキャンパスグリッド規模のマルチクラスタなど（以後、高レイテンシマルチクラスタ）に限らず、同一ネットワークを用いながらスイッチを階層的に用いているため、ノード通信で通過するスイッチ数が異なるような大規模クラスタなど（以後、低レイテンシマルチクラスタ）も含む。

マルチホーム方式では、マルチクラスタにおける要素クラスタ間での通信量を削減することを目的として、要素クラスタ内での通信の局所性を利用できるように、ホームノードを多重化し、各要素クラスタ内に重複して配置する。

これにより、高レイテンシマルチクラスタでは、(1)の要素クラスタ間での通信量が削減される。また、低レイテンシマルチクラスタでは、ページ読み出しが複数ホームノードに分散されることで、(2)のページ読み出しのコンテンションが緩和され、メモリ読み出し時のレイテンシが削減される。

マルチホーム方式でその性能に大きな影響を与えるのは、(A)多重化されたホームノード間での一貫性の保証方式と(B)ホームノードの多重度（ホームノードの数）の決定方式である。(A)に関しては、各ノードがページに書き込みをおこなった際に、すべてのホームノードに対してライトバックする方法、あるいは、特定のホームノードだけに更新をおこない、別途ホームノード間での一貫性保持をおこなう方法、が考えられる。本研究では、まずは前者で実装をおこなう。(B)に関しては、ホームノードの数を、要素クラスタ数などに固定する方法、あるいは、アプリケーションプログラムに応じて数を最適化する方法、が考えられる。本研究では、前者については実装をおこない、後者についてはページ参照履歴を用いた最適化方法について実装をおこなう。

以下、3.2.2 ではマルチクラスタ上でソフトウェア分散共有メモリを実行する問題点を述べる。3.2.3 ではマルチホーム方式、3.2.4 では同方式の実装方法を説明する。また、3.2.5 では予備評価に用いるマルチクラスタシミュレータの実装方法、及び、ベンチマークプログラムによる性能評価結果を示す。3.2.6 で関連研究を概観し、3.2.7 で今後の課題を述べる。

3.2.2 マルチクラスタ上で既存ソフトウェア分散共有メモリ方式を用いる問題点

[ソフトウェア分散共有メモリ方式の選択]

後術のマルチクラスタとの親和性を考慮し、マルチクラスタを指向したソフトウェア分散共有メモリ方式を設計するうえで、ベースとするソフトウェア分散共有メモリシステムとして、既存の各種システムから JIAJIA[53]を選択する。

まず、JIAJIA のメモリアーキテクチャが、マルチクラスタで前提となる大規模問題を効率良く扱えるように設計されている点があげられる。JIAJIA は、利用するページ一貫性制御方式がライトバック型であるため、総合的に高性能[57]なばかりでなく、使用可能なメモリサイズが 1 ノードの物理メモリサイズに限定されない。

また、多様なマルチクラスタ上で、高性能を達成するためには、何らかのパフォーマンスチューニングをおこなう必要があると考えられる。この際、各種プロファイリング技法を利用することが考えられる。メモリの読み出しに際し、ソフトウェア分散共有メモリ TreadMarks[55]などの diff 分散方式がページ差分情報である diff 単位でおこなうのに対して、ライトバック型の JIAJIA では同読み出しがページ単位でおこなわれる。このため、参照履歴として扱い易く、各種プロファイリング技法を比較的容易に適用できるなど多くの利点を有する。

[マルチクラスタ上での実行に伴うオーバヘッド]

小規模 PC クラスタ向けの既存の JIAJIA や TreadMarks のページ一貫性制御方式は、他の多くの各種ソフトウェア分散共有メモリと同様に、すべてのノードを均一な構成とみる。このため、例えば、TreadMarks を高レイテンシマルチクラスタ上でそのまま用いると、メモリ読み出し時の頻繁なページ転送などの要素クラスタ間での通信が性能ボトルネックになってしまう[118]ことが課題となる。

3.2.3　マルチホーム方式の提案

［クラスタ局所性の利用］

マルチクラスタでは、要素クラスタ内での通信の局所性を利用することで、要素クラスタ間での通信量を削減することが有効であると考えられる。

通信の局所性を利用できるように、更新されたページを各要素クラスタごとのあるノードにキャッシュし、同キャッシュをクラスタキャッシュとして用いる。これにより、本来ならばクラスタ外のホームノードとのページ授受が必要なページフォルト処理を、クラスタ内で解決することができる（図 3.9 および図 3.10）。これにより、高・低レイテンシのマルチクラスタで、メモリ読み出しが高速化される。

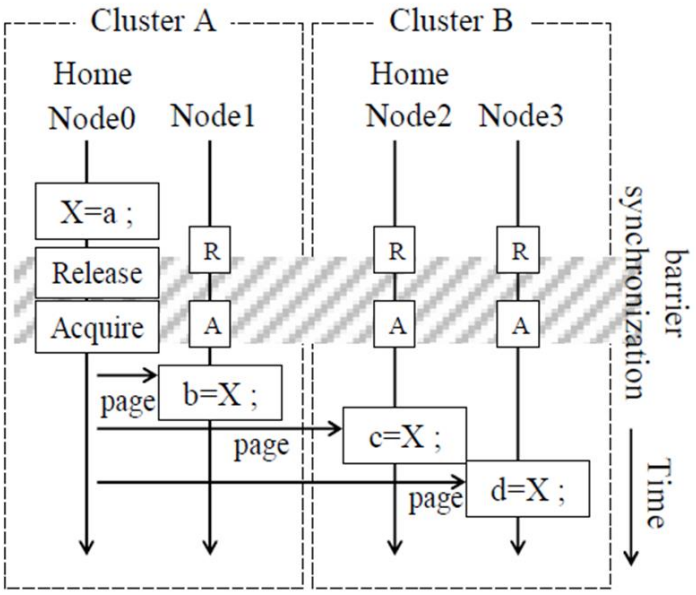

図 3.9.　シングルホーム方式のバリア同期機構

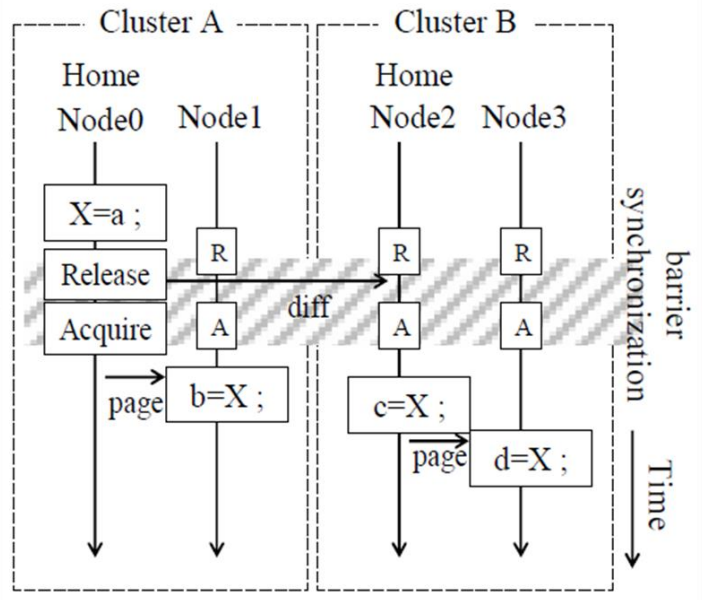

図 3. 10.　マルチホーム方式のバリア同期機構

［ホームノードの多重化と一貫性保証］

本研究では、クラスタキャッシュ方式として、ホームノードを多重化し、各要素クラスタ内に重複して配置するマルチホーム方式を提案する。

マルチホーム方式では、多重化された各要素クラスタに配置されたホームノードを、クラスタキャッシュとして利用する。これに対して、ホームノードが一つの従来方式をマルチホーム方式におけるホームノードの数が 1 の場合（便宜上、シングルホーム方式とも呼ぶ）と位置付けることとする。

つぎに、多重化したホームノード間での一貫性保証方式について論じる。マルチホーム方式において、複数ホームノード間で一貫性をとる方法はいくつか考えられる。

ひとつは、各ノードがページに書き込みをおこなった際に、特定のホームノードに対してライトバックをおこない、別途ホームノード間での一貫性保持をおこなう方法である。これにより、ライトバックを高速化できる可能性がある。

ふたつめは、各ノードが、すべての複数ホームノードにライトバックをおこなう方法である。本稿では、メモリの読み出しの高速化に焦点をあて、後者の方法を用いる。

一方、マルチホーム方式においてのホームノードの数に関しては、アプリケーションプログラムの実行時にホームノードの数が要素クラスタ数などに固定されていると、ホームノード間の一貫性をとるためにすべての複数ホームノードに対してライトバックをおこなう必要がある。このため、ホームノード間の一貫性をとるオーバヘッドが大きくなってし

まうことは明らかである。そこで、3.2.4 では、マルチホーム方式におけるホームノードの多重度を、アプリケーションプログラムに応じて最適化する方法について検討する。

3.2.4　マルチホーム方式の実装

ホームノードの数が固定されているマルチホーム方式の実装方法、ホームノードの多重度をアプリケーションプログラムに応じて実行時に増減させる最適化について説明し、ホームノードの多重度を、マルチクラスタの性能にまで応じて最適化する機構について提案する。

[前提条件と実装の概要]

マルチホーム方式におけるホームノードの数を固定する実装について説明するうえで、前提とする事項について述べる。

まず、本研究で前提とするメモリ一貫性モデルについて説明する。実際に複数のクラスタをみせるようなモデルも考えられるが、本研究では、JIAJIA が用いている Scope Consistency モデル[61]に対して変更はおこなわずにそのまま利用する。Scope Consistency モデルの実装方法には、JIAJIA が利用している、ライトバック型かつ無効化型マルチプルライタ方式[53]を前提とする。これらの詳細については、文献[53]を参照されたい。

ホームノードの多重度を固定する場合、とくに断らない場合、マルチクラスタにおける要素クラスタの数で固定する。

また、ホームノードの分散配置方法については、たとえば、2 つの要素クラスタからなるマルチクラスタの場合、ホームノードの数が片方の要素クラスタに 1 つで、もう片方に 2 つということもあるうる。しかし、本研究では、マルチホーム方式における複数ホームノードの配置が要素クラスタ間で対称となるようにする。このため、実際のグリッド環境などではクラスタがヘテロジニアスである場合が多いが、本研究においては、要素クラスタがホモジニアスであると仮定している。

各ノードが書き込みをおこなった際には、前述のとおり、すべてのホームノードにライトバックをおこなう。ページリクエストの発行は、個々の要素クラスタ内のホームノードに対しておこなう。

[実装方法]

前述のページ一貫性制御方式であるライトバック型かつ無効化型マルチプルタイタ方式をベースにマルチホーム方式を実装する。

図 3.9 と図 3.10 で、バリア同期機構の実装方法について、JIAJIA が利用する従来方式(シングルホーム方式) とマルチホーム方式とで比較しながら説明する。

まず、前者の実装について説明する。シングルホーム方式の実装では、バリア同期操作を発行する際、その前のバリア同期区間においてページに書き込みがおこなわれていた場合、同書き込みがおこなわれる前とおこなった後でのページの差分情報(diff)を用いて唯一のホームノードにライトバックし、同ライトバックの完了確認を待つ。同確認後バリア同期機構の集中スケジューラノードにバリア同期到達通知と書き込みをおこなったページについての書き込み通知をまとめて送信する。

これに対して、マルチホーム方式の実装では、すべてのホームノードに対して diff でライトバックするようにした。また、シングルホーム方式では、自ノードがホームノードにアサインされているページ（以後、ホームページ）への書き込みに際し、diff 作成・送信の必要がなかった。しかし、マルチホーム方式では、ホームノードが多重化されているため、ホームページへの書き込みについても、それ以外のホームノードに対して diff を送信する必要がある。

ロックを用いた排他制御機構の実装方法についても同様に、ライトバックを複数ホームノードに対しておこなう。

[ページ参照履歴を用いたホームノードの多重度の最適化]

ホームノードの多重度を固定する実装では、複数ホームノード間の一貫性をとるために、ライトバックをすべてのホームノードに対しておこなうため、通信オーバヘッドが大きい。

しかし、アプリケーションプログラムの実行時に、ページごとに適宜にホームノードの数を増減させることで、一貫性維持のための通信オーバヘッドを除去することができる。換言すればアプリケーション実行時においては、基本的には、各々のページについてマルチホーム方式におけるホームノードの数を 1 つとし処理を進める。そして、複数ノードによる一斉のページ読み出しなどがおこる部分で、ホームノードの数を適切な数に増やすことが効果的である。

以下に、マルチホーム方式において、ホームノードの数が性能に影響する典型的なページアクセスパタンを 2 つとりあげる。ひとつは、ホームノードが複数あることが有効な例で、もうひとつは、ホームノードが複数あることがオーバヘッドになる例である。

ホームノードの多重化が有効な例: Producer ノードのページ書き込みの完了を、その他 Consumer ノードがバリア同期成立まで待つ。同バリア同期成立後、Producer から同ページを読み出す Producer-Consumer 型メモリアクセスパタン。バリア同期時に、後続する Consumers 数分の読み出しに先行して複数のホームノードへライトバックをおこなう。これにより、同読み出しはクラスタ内のホームノードからおこなえばよいので、要素クラスタ

間通信を削減できる。また、同読み出しを複数ホームノードに分散させることで高速化することができる。

ホームノードの多重化がオーバヘッドになる例: 連続したバリア同期区間において、書き込みと読み出しがホームページに対してのみ実行されるような、データ局所性の高いメモリアクセスパタン。2 区間の間のバリア同期時に、複数ホームノードに対しておこなうライトバックはまったく必要なく、オーバヘッドにしかならない。この場合は、マルチホーム方式におけるホームの数は、1 のまま、処理を継続することが望ましい。

アプリケーションプログラムに応じてホームノードの多重度を最適化する場合、上記のようなホームノードの多重化が効果的となるメモリアクセスパタンを検出する必要がある。検出方法として、アプリケーション開発者によるソフトウェア分散共有メモリライブラリが提供する API を用いたホームノードの数の増減の明示的な指示や、同期操作に後続するメモリアクセスの投機実行によりメモリアクセスパタンの動的な検出も考えられる。あるいは、後述するマルチホーム方式の lazy な実装を用いて、要素クラスタ内にプロキシ的なノードを設けて、要素クラスタ外へのページリクエストを集中管理するなどが考えらえる。

ここでは、ホームノード数の増減を最適におこなった場合に得ることができる最高性能を示すために、プロファイリング技法を用いる。プロファイリング技法にも、実行時におこなう方法と仮実行して得られるページ参照履歴を利用する方法があるが、ここでは、最高性能を示すことが目的であるため、後者による最適化をおこなう。

本研究では、Producer-Consumers 型メモリアクセスパタンに対して最適化をおこなう。このメモリアクセスパタンを検出するために必要な参照履歴と、その取得方法を説明する。本研究でおこなう高速化は、主にページ読み出しを対象とするため、取得する参照履歴はバリア同期区間ごとに読み出されたページの番号、及び、同読み出しをおこなったノードの番号だけである。これで、同一のバリア同期区間に、どの複数ノードが読み出しをおこなったかが分かる。必要な参照履歴を取得するためのコードを JIAJIA に挿入し、これを得る。

本研究では、取得したページ番号と当該バリア同期区間をマルチホーム方式でライトバックをおこなう対象とし、これ以外のページについてはライトバックをおこなわないように JIAJIA にコードを挿入し、ライブラリを再構築する。再構築した JIAJIA を用いてアプリケーションプログラムを再実行し、得られる実行結果をホームノードの多重度の「最適」なものとする。

[マルチクラスタ性能に特化したライブラリを構築する機構の提案]

マルチクラスタ性能は、ノード数やネットワーク性能などのシステムの構成要素で決定する。一般的に、マルチクラスタ性能によって、適したソフトウェア分散共有メモリ方式

は異なると考えられ、ひとつのシステムが複数のソフトウェア分散共有メモリ方式を用意することが必要となっていた。しかし、マルチホーム方式では、ホームノードの多重度の最適化が、アプリケーションプログラムに対してのみでなく、マルチクラスタ性能に対してもおこなえるため、その必要がない。

ここでは、アプリケーションプログラムとクラスタ性能に対してホームノードの多重度を最適化するための機構を提案する。

本機構は、まず、ユーザが、アプリケーションプログラムを実行する前に、様々なアクセスパタンからなるクラスタ性能測定プログラムを実行する。これにより、アクセスパタンごとにクラスタ性能に適合するホームノードの数を算出する。この値をパラメタにして、ソフトウェアライブラリを再構築することで、容易にクラスタ性能に特化したソフトウェアライブラリを構築することが可能となる。

この際、用いるクラスタ性能測定プログラムが重要となる。クラスタ性能測定プログラムは、アプリケーションプログラムでおこりうる主要なメモリアクセスパタンは完備する必要がある。また、最適なホームノードの数を同定するにあたり、高精度を実現させる必要がある。

本研究では、3.2.5 でおこなう評価の一部において、ホームノードの多重度を、要素クラスタの数に等しくした場合と、要素クラスタにつき 2 つにした場合で、評価をおこなう。

3.2.5　実機を利用したマルチクラスタシステムシミュレータの実装

高レイテンシマルチクラスタに関する評価を、シミュレーションによりおこなった。具体的には、高レイテンシマルチクラスタ(MC)を単一の PC クラスタ(SC)上でシミュレートするシステムを構築し、同シミュレータ上で予備評価をおこなう。低レイテンシマルチクラスタに関する評価は、単一の PC クラスタ上でおこなう。

表 3.8 に示すクラスタ構成要素ノードからなる SMP-PC クラスタを、論理的に 2 つの要素 SMP-PC クラスタとみなす疑似マルチクラスタシステム(図 3.11)を実装した。

表 3.8.　評価実験環境

CPU	PentumIII 866MHz(x2)
Memory	1GB
NIC	100BASE-TX
OS	Red Hat Linux 7.1
Compiler	gcc 2.96

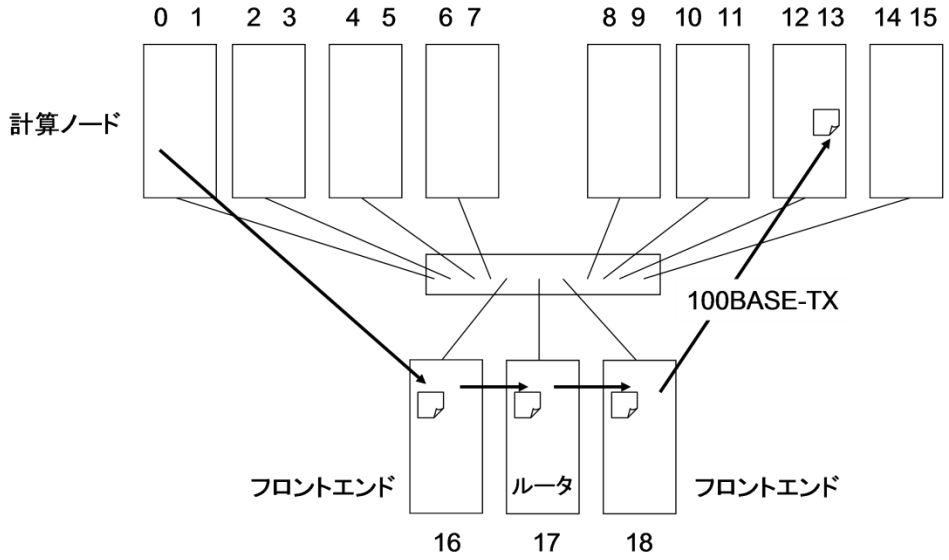

図 3.11.　疑似マルチクラスタシステム

この疑似クラスタシステムは、要素クラスタ間通信を、クラスタ間ルータにみたてた複数台の SMP-PC ノードを経由させることで、クラスタ間通信レイテンシをシミュレートする。図 3.11 において、計算ノード 0 が計算ノード 8～15 にページを送信する場合、ページをまずルータノード 16 に送信する。ルータノード 16 は、送信されてきたページをルータノード 17 にそのまま転送する。同様に、ルータノード 17 はルータノード 18 に転送する。ルータノード 18 は、ページの実際の送信先である計算ノード 8～15 にページを送信する。

高レイテンシマルチクラスタについての評価は、シミュレートするクラスタ間通信レイテンシが、キャンパスグリッド規模程度になるように、クラスタ間ルータノードを 3 つとした。シミュレートされるクラスタ間通信レイテンシを示すために、ページフォルト処理時間コストを測定した結果を表 3.9 に示す。ここで、ページフォルトによりホームノードからページをフェッチし、ページアクセスが再開するまでを測定した時間である。

また、この疑似マルチクラスタの基本性能を検証する目的で、並列化率が非常に高い NAS

Parallel Benchmarks(NPB)の EP を用いた。測定結果を表 3.10 に示す。

表 3.9.　ページフォルト処理時間コスト

ページの取得先	コスト[usec]
SMP ノード内（クラスタ内）	219
SMP ノード間（クラスタ内）	725
クラスタ外	3069

表 3.10.　マルチクラスタの基本性能

システム構成	実行時間[sec]	台数効果
1CPU(逐次)	305.4	-
16CPU の SC	19.1	15.9
16CPU(8+8)の MC	19.2	15.9

3.2.6　予備評価

[予備評価用ベンチマークプログラム]

予備評価用ベンチマークプログラムには、行列積を求めるプログラム(MM)、LU 分解プログラム(LU)、NPB の IS の特性の異なる 3 つを用いた。

問題サイズは、表 3.11 のとおりである。データとホームノードのマッピングは、シングルホーム方式の場合においてメモリ書き込み局所性が最適になるように、アプリケーションプログラム側で人手で設定した。予備評価に用いたページサイズは 4K バイトとした。

表 3.11.　問題サイズ

評価プログラム	問題サイズ
MM	2048x2048
IS	鍵の数=2^26, 鍵の最大値=2^12
LU	1024x1024

[予備評価結果]

アプリケーションごとに、シングルホーム方式(SH)と、ホームノードの数を固定するマルチホーム方式(MH)と 3.2.4 で定義した最適化をおこなった場合について、単一 PC クラスタ(SC)とマルチクラスタ(MC)上で性能評価した結果を表 3.12、表 3.14、表 3.16 に示す。

MM の評価:

表 3.12 に示すとおり、MM では、単一 PC クラスタ及び疑似マルチクラスタ上において、従来方式では 16CPU までのスケーラビリティが得られていない。

一方、マルチホーム方式では、16CPU でも速度向上を示している。

単一 PC クラスタ上でのマルチホーム方式を用いた場合の速度向上の理由として、ホームノードからのページ読み出しのコンテンションが緩和されたためと考えられる。疑似マルチクラスタ上で、マルチホーム方式を用いた場合は、これに加えて、クラスタ間通信レイテンシを削減できた結果、高い性能向上を示している。

特筆すべきは、シングルホーム方式に対してマルチホーム方式で発生してしまうライトバックが、さほどのオーバヘッドになっていない点である。これは、同ライトバックが、シングルホーム方式でおこなわれる scatter 的なライトバックと一括しておこなえるからである。これにより、ホームノードの数を固定するマルチホーム方式で不必要なライトバックが増加しても、性能向上が得られるわけである。

さらに、ホームノードの数を最適にした場合では、期待どおりさらに速度向上した。

また、ホームノードの数を要素クラスタごとに 2 つに増やした場合の性能評価をおこなった。評価結果を表 3.13 に示す。

ホームノードの数を 4 つに増やした場合、逆にパフォーマンスが低下してしまった。最適化をおこなっても、同様の結果となった。これは、増加したホームノードへライトバックが過大になったためである。

IS の評価:

表 3.14 に示すとおり、IS では、疑似マルチクラスタ上でマルチホーム方式を用いた場合

に速度向上を示した。

疑似マルチクラスタ上での速度向上は、MM の場合と同様に、マルチホーム方式にすることで、クラスタ間通信を削減できた結果である。

また、IS では、すべての計算ノードが同一のロック変数を使用し、排他的に交互にページの読み出しと書き込みをおこなう Migratory access 型と、すべての計算ノードがページ読み出しを一斉におこなう Producer-Consumers 型の 2 つのメモリアクセスパタンが内在する。実行時間について、それぞれのメモリアクセスパタンの内訳と、Migratory access 型のメモリアクセスパタンにおいてアクセスされるページの数との関係を表 3.8 に示す。

評価結果より、単一 PC クラスタ及び疑似マルチクラスタ上において、Producer-Consumers 型メモリアクセスパタンでは、マルチホーム方式にすることで速度向上したことがわかる。

IS では Producer が書き込みをおこなうデータがホームノードとして Producer にマッピングされているので、MM のようにマルチホーム方式が引き起こす新たなライトバックが従来方式のライトバックと一括しておこなわれることがない。それにも関わらず、速度向上を達成している。IS では、Producer-Consumers 型メモリアクセスパタンのバリア同期区間において、Producer のデータ生成の実行時間が短い。よって、同区間中に計算をしない Consumers がバリア同期機構の制御用通信をおこなっている間に、Producer はデータ生成ともうひとつのホームノードへライトバックをおこなう。このオーバラップにより、同ライトバックが隠蔽され、速度向上すると考えられる。

一方、migratory access 型メモリアクセスパタンについては、単一 PC クラスタ上ではシングルホーム方式を用いた場合の性能が高いが、疑似マルチクラスタ上ではページ数が増えるとマルチホーム方式の方が性能向上することが明らかになった。

Migratory access 型のメモリアクセスパタンにおいて、ロックとアンロックの中で、例えば表 3.15 のように、32 ページに対する読み出しと書き込みをおこなう場合では、シングルホーム方式では各ノードにおいて 32 回のページの無効化とページの更新が必要となる。従って、シングルホーム方式では、ホームノードが要素クラスタ外にある場合、ページを更新するためのページリクエストとこれに応答するページ転送が 32 回ずつと、diff を用いたライトバックを加えた要素クラスタ間通信が必要となる。

一方、マルチホーム方式では、クラスタ内のホームノードに対してページリクエストを発行すればよいので、クラスタ間での頻繁なページリクエストがなくなる。一方で、複数ホームノード間で一貫性をとるために、diff を用いたクラスタ間ライトバックが多少増加する。しかし、diff は、一括して複数ページ分を送信することができないため、ページ数が増加するとそのオーバヘッドは相対的に小さくなる。

以上により、IS では、単一 PC クラスタ上において、migratory access 型メモリアクセスパタンでホームノードの数を 1 つにして、Producer-Consumers 型メモリアクセスパタンでホームノードの数を 2 つにすることで、速度向上することがわかる。疑似マルチクラスタ上では、Producer-Consumers 型メモリアクセスパタンにはホームノードの数を 2 つにし、

migratory access 型メモリアクセスパタンでは、ページ数に応じてホームノードの数を適切にすることで、最も速度向上する。

LU の評価:

LU では、16CPU で充分な性能が得られなかったため、8CPU で評価をおこなった。表 3.16 に示すとおり、単一 PC クラスタ及び疑似マルチクラスタ上で、大小の差はあれ MH、最適化をおこなった場合では性能低下が生じている。

LU で、ホームノードの数を固定した場合、性能が大幅に低下している。LU では、バリア同期ごとに、一部のページが Producer-Consumers 型メモリアクセスパタンの対象となるがすべての書き込みはホームページに対してのみおこなわれる。このような場合、ライトバックはおこなわれないため、シングルホーム方式が最も効果を発揮する。このため、上記の一部以外のページへの大量の書き込みにより、バリア同期ごとに複数ノードへのライトバックが、本評価で用いた問題サイズでは発生し、性能が著しく低下する。

LU では、上記オーバヘッドを除去した最適な場合についても性能劣化がみられる。この原因として、1)ノード数不足でマルチホーム方式の効果が現れない。2) LU は、IS のようにバリア同期とライトバックがオーバラップしない。3) 上述のとおりホームページ以外への書き込みがないため、MM のように、ホームノードを複数にすることで新たに発生するライトバックがシングルホーム方式のライトバックと一括して転送されることもない。この組み合わせ的要因により、マルチホーム方式による新たなライトバックがシングルホーム方式の場合に対して、新たな逐次処理部分を形成してしまっており、速度低下してしまう。

異なる問題サイズを用いたり、並列度が上がった場合、ページアライメントに合わないデータ配置がおこりうる。このような場合については、シングルホーム方式でもライトバックが引き起こされるため、ホームノードを複数にしたときに新たに発生するライトバックも前者と一括処理することができ、速度向上が期待できる。

表 3.12.　MM(2048x2048)の実行時間[sec]

ノード数	SC			MC		
	SH	MH	最適	SH	MH	最適
1CPU	155.3	-	-	-	-	-
2CPU (1+1)	89.6	89.0	82.2	109.5	103.0	90.3
4CPU (2+2)	61.2	60.1	56.0	80.4	73.0	90.3
8CPU (4+4)	61.2	60.1	56.0	60.2	54.1	47.2
16CPU(8+8)	46.1	32.9	30.7	63.8	45.5	39.2

表 3.13.　ホームノードを 4 つ用いた場合の実行時間[sec]

ホームノード数	SC		MC	
	2	4	2	4
8CPU (4+4)	41.3	42.6	54.1	69.4
16CPU(8+8)	32.9	37.6	45.5	64.1

表 3.14.　IS の実行時間[sec](共有メモリ:4 ページ)

ノード数	SC			MC		
	SH	MH	最適	SH	MH	最適
1CPU	38.1	-	-	-	-	-
2CPU (1+1)	20.0	19.7	19.6	20.2	19.9	19.8
4CPU (2+2)	10.2	10.2	10.1	12.6	12.5	12.6
8CPU (4+4)	5.6	5.6	5.5	7.3	7.1	7.2
16CPU(8+8)	3.9	3.9	3.7	5.8	5.4	5.5

表 3.15.　各アクセスパタンの実行時間[sec]

8CPU(4-4)		SC		MC	
		SH	MH	SH	MH
Migratory	1 ページ	0.1	0.1	0.3	0.4
	32 ページ	3.1	3.9	7.9	7.4
Producer Consumer	1 ページ	0.4	0.3	0.4	0.3
	32 ページ	1.8	1.0	2.3	1.0

表 3.16.　LU の実行時間[sec]

ノード数	SC			MC	
	SH	MH	最適 (ホーム 2)	SH	最適 (ホーム 2)
1CPU	18.3	-	-	-	-
8CPU (4+4)	12.5	99.0	18.5	16.2	20.9

3.2.7　関連研究

文献[118]では、diff 分散方式を採用した TreadMarks を利用して、グリッド型マルチクラスタに向けて同様の評価実験を行っているが、速度向上を示すには至っていない。本研究の提案方式は、クラスタシステムを、いくつかのサブクラスタとみたらよいかという、クラスタ構成法的な観点において、[118]と異なる。このようなアプローチで、JIAJIA では得られなかった速度向上とスケーラビリティを、低速なネットワークでも実現した。

また、本研究では、共通のアプローチでグリッド型マルチクラスタと大規模クラスタを扱っているため、例えば、大規模クラスタ同士を LAN で繋げた大規模マルチクラスタ上で、組み合わせ的に提案方式が適用可能となる点が、提案方式の最大の特徴である。

文献[119]では、グリッド型マルチクラスタ上に向けたアプリケーションレベルでの最適化をアプリケーションごとにおこなっている。本研究のアプローチは、ソフトウェア分散共有メモリシステムで透過的に対応している点で異なる。

ソフトウェア分散共有メモリの研究は 2000 年代後半は下火になっていたが、ここにきて再度注目を集めている[27-29,66-69]。その背景にあるのは、RDMA(Remote Direct Memory Access)と呼ばれる機能を持つ高バンド幅と低レイテンシを実現する Infiniband などが利用可能になったことである。また、ディープラーニングなどの応用が登場していることも大きく影響している。ディープラーニングでは巨大なニューラルネットワークを利用して学習をおこなう。ニューラルネットワークが大きいほど学習の精度が向上することが知られている。また、学習には大量の演算が必要となり分散処理が必要となる[29]。ディープラーニングの研究初期段階の学習処理アプリケーションは MPI を使ってニューラルネットワークモデルのパラメタの更新を明示的に記述していた。しかし、Google[69]や Microsoft[28]やカーネギーメロン大学[67]では、パラメタサーバ方式[69]というソフトウェア分散共有メモリで管理されるようになった。これによりアプリケーションはパラメタ

を単純な読み書きでアクセスできるようになり、一貫性管理や通信はパラメタサーバアーキテクチャで管理される。

3.2.8 まとめと今後の課題

本研究では、マルチクラスタ上でのソフトウェア分散共有メモリ方式として、要素クラスタ間での通信量を削減することを目的として、要素クラスタ内での通信の局所性を利用できるように、各要素クラスタ内に重複してホームを配置するマルチホーム方式を提案し、有効性を検証した。シミュレーションによる予備評価の結果、既存のソフトウェア分散共有メモリ方式に対し最大 38.5%の速度向上を得た。

さらに、ノード間の通信コストが均一な PC クラスタ上でも、ホームアクセスのコンテンションを緩和するという点で、マルチホーム方式が有効となることを確認した。

今後は、マルチホーム方式の lazy な実装をおこなうことも考えらえる。これは、各要素内に、リモートクラスタに対するキャッシュを設ける方法と共通の考え方である。一例を示す。この例では、クラスタ B 内ノード（ノード 2 とノード 3）は、クラスタ外ホームノード（ノード 0）に対するページリクエストを集中管理するプロキシノード(ノード 2)に対して同リクエストを発行する。同一ページに関する複数のリクエストのうち、先頭 n もののみが実際にクラスタ外ホームノードに転送される。取得したページは、プロキシノードにキャッシュされ、これを 2 ノードで利用する。マルチホーム方式との違いは、同期完了時にホームノードの数が決定するマルチホーム方式に対し、必要になった時点でページを取得し、これを再利用する点である。予備評価で用いた LU などの、マルチホーム方式がおこな同期時のライトバックによる更新がその他の処理とオーバラップできないときは、こちらが有効であると考えられる。

本研究の評価においては、各データに対するホームノードの配置が最適であることを前提としてきた。ホームベース方式を用いたソフトウェア分散共有メモリでは、とくに大規模なシステムにおいて、このデータマッピングが大きく性能に影響を及ぼす[57]。ホームノードの配置を最適にするには、ファーストタッチでおこなう方法、ホームノードの動的再配置[62,120]でおこなう方法、ユーザが明示的に指定する方法やこれらの組み合わせが考えられるが、これも今後の重要な課題である。

4. 新型高速不揮発メモリを活用した階層型主記憶を実現する省電力仮想記憶システム

4.1 まえがき

昨今、ビッグデータ解析やディープラーニングやオンラインリアルタイム処理などに代表されるアプリケーションにおいて処理するデータの大規模化に伴い、データセンターなどのサーバシステムで必要とされるインメモリデータ処理用の主記憶のサイズが急速に増加している。インメモリデータ処理では、複数ノードあるいは単一ノードを用いて大規模データをインメモリで処理するための主記憶を大容量化したいため、これまでの DRAM ベースサーバシステムでは主記憶を構成する高速な DRAM を増やし続けてきたが、これが DRAM の集積度、消費電力、そしてコストの観点で限界を迎えている。特に揮発性メモリである DRAM の待機消費電力は大きな問題であり、データセンターの消費電力の 25%以上をメモリが占めているともいわれている[2]。現在は DRAM と SSD/HDD を組み合わせて使うしかないが、DRAM と SSD/HDD 間のデータ転送時間が大きいためこれを低減するためにはアプリケーションプログラムの大幅なリストラクチャリングが必要になり、しかもそれをおこなっても高い性能が得られない[40]。今後、処理データのさらなる大規模化に伴いメモリの占める比率は大きくなっていくと予想され、大規模インメモリデータ処理を実現するコンピュータシステムの大きな課題である。

4.2 新型高速不揮発メモリ（ストレージクラスメモリ）

一方で、MRAM や PCM や ReRAM などのストレージクラスメモリ(Storage-class Memory: SCM)[1]と呼ばれる新型高速不揮発メモリの実用化が期待されている。ストレージクラスメモリは、不揮発メモリであるため待機消費電力が非常に小さく、DRAM に迫る速度を持ち、さらに DRAM より大容量化が可能である。

今後のサーバシステムに要求される、インメモリデータ処理向けの高速・低消費電力でかつスケーラブルな主記憶を実現するためには、このストレージクラスメモリを活用する必要があるが、どのようにサーバに搭載するかが問題である。

ストレージクラスメモリは、消費電力の削減と性能の向上の両面で期待されているが、

裏を返せばDRAMよりはアクセスレイテンシが大きく、メモリアクセスする際の消費電力も大きく、さらに、書換え回数の制約もあるため、単純にDRAMを置き換えて単体利用することはできない場合が多い。

これらを補うためにストレージクラスメモリの容量に対して数割のサイズのDRAMを混載する、大容量のストレージクラスメモリ/DRAM混載メモリシステムを構築することが必要になる。例えば1/8をDRAM、7/8をストレージクラスメモリにするなどである。その際に課題になるのは、「どのように混載させるか」だが考えるべき問題は多い。具体的にはまずは1）DRAMとDRAMより遅いストレージクラスメモリをどのように組み合わせてどのように階層制御してDRAMにより近い性能を出すか、そして、どのようにアプリケーション開発者に見せるかである。主記憶の一部をストレージクラスメモリに置き換えるハイブリッド型の主記憶を構成する実現手段も考えられるが、2つのメモリをどう使い分けるかをアプリケーションプログラム開発者に強いることになりインメモリデータ処理のプログラミングが複雑になる。さらに、2）動的消費電力が高いストレージクラスメモリとDRAMをどのように階層制御して待機電力が低いストレージクラスメモリの省電力性を引き出して低消費電力を実現するか、3）ストレージクラスメモリにはMRAM、PCM、ReRAMなど様々な系統のものがあり[1, 6, 88, 99]、ストレージクラスメモリで想定されているレイテンシ(数百ns～数μs)はレンジが広いが、そのなかで、どのような性能特性を持つストレージクラスメモリが必要なのか、そして、4）どのようにストレージクラスメモリをプロセッサに接続するのか、5）書換え回数の制約をどのように解決するか、などの課題がある。また、6）実ストレージクラスメモリがないため評価環境や評価方法も大きな課題である。さらに7）評価用アプリケーションも課題である。これは例えば現在のインメモリDBはデータの永続化はSSD/HDDに対して行っているので不揮発メインメモリに対して永続化をおこなう真のインメモリ処理にはなっておらず、ストレージクラスメモリが出てきた際にはその性能を引き出すためにインメモリDB自体の大きな変更が必要であり今のインメモリDBを用いて評価をおこなってもストレージクラスメモリの真のポテンシャルを評価することはできないという課題もある。

4.3 積極的にストレージクラスメモリへ退避する省電力仮想記憶方式の提案

この「どのように混載させるか」という課題に対して、本論文では、SCM/DRAM混載メモリシステムの階層制御技法とこれを自動化するストレージクラスメモリ向け仮想記憶の基本方式の提案をおこなう。まず、ストレージクラスメモリの高速性と待機消費電力の低さ

を活かして、DRAM 上のデータを積極的にストレージクラスメモリに退避して使用する DRAM サイズを削減し、未使用 DRAM の電源をオフすることで動作中のリーク電流を削減するストレージクラスメモリを活用した階層制御技法を提案する[7]。

従来は DRAM にデータが乗り切らなくなり SSD/HDD との間でデータ転送が発生し性能が大幅に低下するため、インメモリデータ処理では大きな DRAM を搭載する必要があったが、従来とは異なり、ある程度のデータ転送の増加は容認して積極的に DRAM 上のデータをストレージクラスメモリに退避させて、使用する DRAM の量を減らす階層制御技法である。

この階層制御技法を、アプリケーション開発者にストレージクラスメモリと DRAM の 2 つのメモリを陽に見せたうえでプログラミングさせる方法も考えられるが、書換え回数の制約の課題などの複雑な問題も考慮する必要があり現実的ではない。そこで、本研究では、OS の仮想記憶方式のスワップデバイスにストレージクラスメモリを用いることでストレージクラスメモリと DRAM をシステム内に混載させ、階層制御技法を自動化するストレージクラスメモリ向け仮想記憶基本方式を提案する。

従来型の仮想記憶システムでは、低速なハードディスクや SSD がスワップデバイスとして利用されていた。そのため、スワップ処理に掛かるデータ転送オーバヘッドが大きいため、スワップ処理が発生するとアプリケーションやシステムの性能が大きく低下してしまうため、極力スワップ処理を起こさないようにすることが重要だった。

しかし、ストレージクラスメモリのような高速なメモリをスワップデバイスに利用できるとデータ転送時間を短くできるのでスワップ処理による性能低下が小さいため、積極的にスワップ処理するように仮想記憶方式を改良し、メモリシステムの省電力化を実現する。改良後の省電力仮想記憶方式では、図 4.1 に示すように、高速・大容量なストレージクラスメモリをスワップデバイスとして DRAM と共に主記憶として混載し、待機消費電力が小さいストレージクラスメモリの特性を活かして DRAM 上のデータを積極的にストレージクラスメモリに退避して、使用する DRAM サイズを削減するとともに未使用 DRAM の電源をオフすることで動作時の待機消費電力を削減することが可能になる。

OS の仮想記憶システムを拡張して利用すればアプリケーション開発者には大きな主記憶があるように見せることが可能で、また、今回は評価には至っていないが書換え回数の制約という観点では、仮想記憶であればアプリケーションからの直接アクセスは DRAM に対してのみおこなわれ、かつ、ストレージクラスメモリのどこへスワップアウトするかを容易に OS で制御できるためウェアレベリングもしやすくなる利点もある。

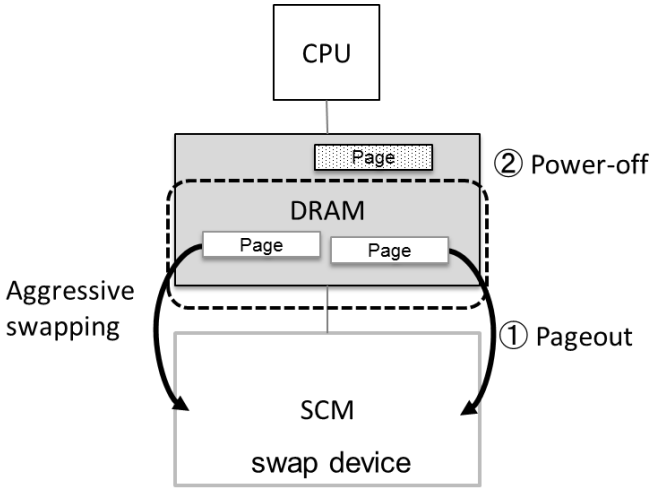

図 4.1.　ストレージクラスメモリ向けの省電力仮想記憶システム

4.4　フルシステムシミュレーションによる評価

4.4.1　評価の目的

ストレージクラスメモリ/DRAM 混載メモリシステムの階層制御技法とこれを自動化するストレージクラスメモリ向け仮想記憶の基本方式がストレージクラスメモリで想定されているレイテンシのレンジで有効か評価した。

評価に利用できる実ストレージクラスメモリがまだないこと、そして、ストレージクラスメモリ性能特性を変化させて評価することが必要なため、それが可能なフルシステムシミュレータで初期評価をおこなった。フルシステムシミュレータを用いると、データセンターで実際に動作しているような大規模な実アプリケーションで評価するのは難しいが、サーバタイプアプリケーションなどを指向し設計されているベンチマークでありコンピュータアーキテクチャの評価に幅広く利用されている PARSEC ベンチマーク集[5]を用いて、ストレージクラスメモリの性能特性やアプリケーション特性により実行時間や消費電力がどのように変化するか、その基本的なパフォーマンスカーブを評価することが狙いである。

4.4.2　フルシステムシミュレーション評価環境

フルシステムシミュレーションは、アプリケーションやオペレーティングシステムを動作させた時のシステム全体の振る舞いを詳細にシミュレーションすることができる。各種

ストレージクラスメリの特性が及ぼす影響の傾向をみるには、これらが柔軟に変更可能なフルシステムシミュレーションが向いている。評価に用いたフルシステムシミュレータはビンガムトン大学で開発された MARSSx86[4]をベースに開発している。省電力仮想記憶方式による省電力化の可能性の評価には、表 4.1 に示すシミュレーションモデルを用いた。

表 4.1.　評価に用いたシミュレーションモデル

CPU	4 out-of-order x86 cores @ 2.9GHz
L1 Cache	32KB L1-I / 32KB L1-D
L2 Cache	256KB
L3 Cache	20MB shared
OS	Linux kernel 2.6.38

評価用プログラムには、共有メモリ型コンピュータシステムの性能評価に広く使われている PARSEC 2.0 ベンチマーク集を用いた。図 4.2 に、PARSEC ベンチマークの各ベンチマークのメモリアクセス量(Byte/Instruction)をフルシステムシミュレータ上で測定した結果を示す。今回の評価には、PARSEC ベンチマークの中でも最もメモリアクセス量が多くそのため DRAM サイズを削減しづらく提案方式に最も不利な条件での評価が可能になる canneal と facesim と、必要なメモリサイズが一番大きくメモリアクセスローカリティも高い提案方式に有利なベンチマークである dedup を利用した。それぞれのベンチマークの特性を表 4.2 に示す（表 4.2 は[8]を基に作成)。評価に用いた PARSEC ベンチマークの入力データセットは、シミュレーション用途では一番大きいサイズの simlarge である。

評価は、ストレージクラスメモリ/DRAM 混載メモリシステムにおいて DRAM 上のデータを積極的にストレージクラスメモリへスワップアウトする方式を模擬するために、SSD や HDD では実用的な処理時間で実行できない量のスワップ処理を引き起こす DRAM サイズまで、DRAM サイズを十分小さく設定し、アプリケーションを実行し評価した。

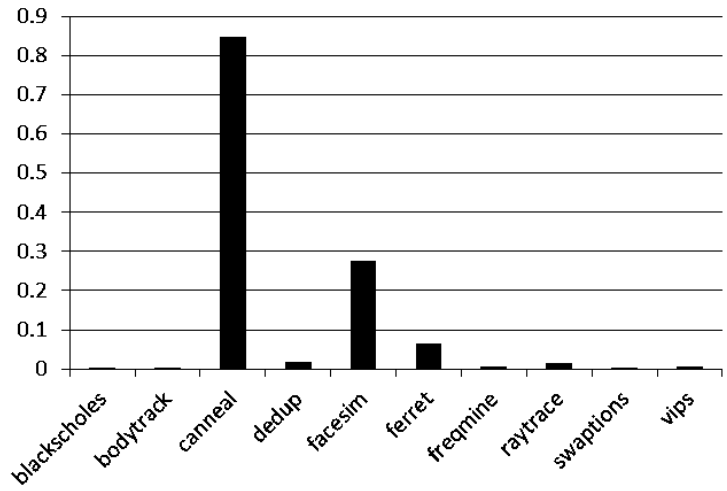

図 4.2.　PARSEC ベンチマークのメモリアクセス量(Byte/Instruction)

表 4.2.　評価に用いた PARSEC ベンチマークの概要

Program	Application Domain	Parallelization Model	Parallelization Granularity
canneal	Engineering	unstructured	fine
dedup	Enterprise Storage	pipeline	medium
facesim	Animation	data-parallel	coarse

4.4.3　ストレージクラスメモリのアクセスレイテンシが実行時間に及ぼす影響の評価

まず、各種ストレージクラスメモリのアクセスレイテンシが実行時間に及ぼす影響について PARSEC ベンチマークを用いて評価した結果を図 4.3～図 4.5 に示す。スワップデバイスのアクセスレイテンシが性能に与える影響のトレンドカーブを見極めるために、ここでは同アクセスレイテンシを幅広く変化させる。具体的には、それぞれの図の横軸に示すように、レイテンシを、SRAM や DRAM 相当のレイテンシから始めて、ストレージクラスメモリで想定されているレイテンシ、SSD や HDD 相当のレイテンシまで変化させている。図の縦軸はベンチマークの実行サイクル数である。

図の一番左側の縦棒だけは特別扱いで、アクセスレイテンシが 0 というわけではなく、十分なサイズの DRAM のみでベンチマークが実行された場合の実行サイクル数を示している。つまり、スワップ処理が発生せず、スワップ処理によるオーバヘッドがないメモリサイズで実行している。即ち、この縦棒と同じ高さで引かれている線より上は、データ転送オー

バヘッドなどのスワップ処理に掛かるオーバヘッドであると考えることができる。

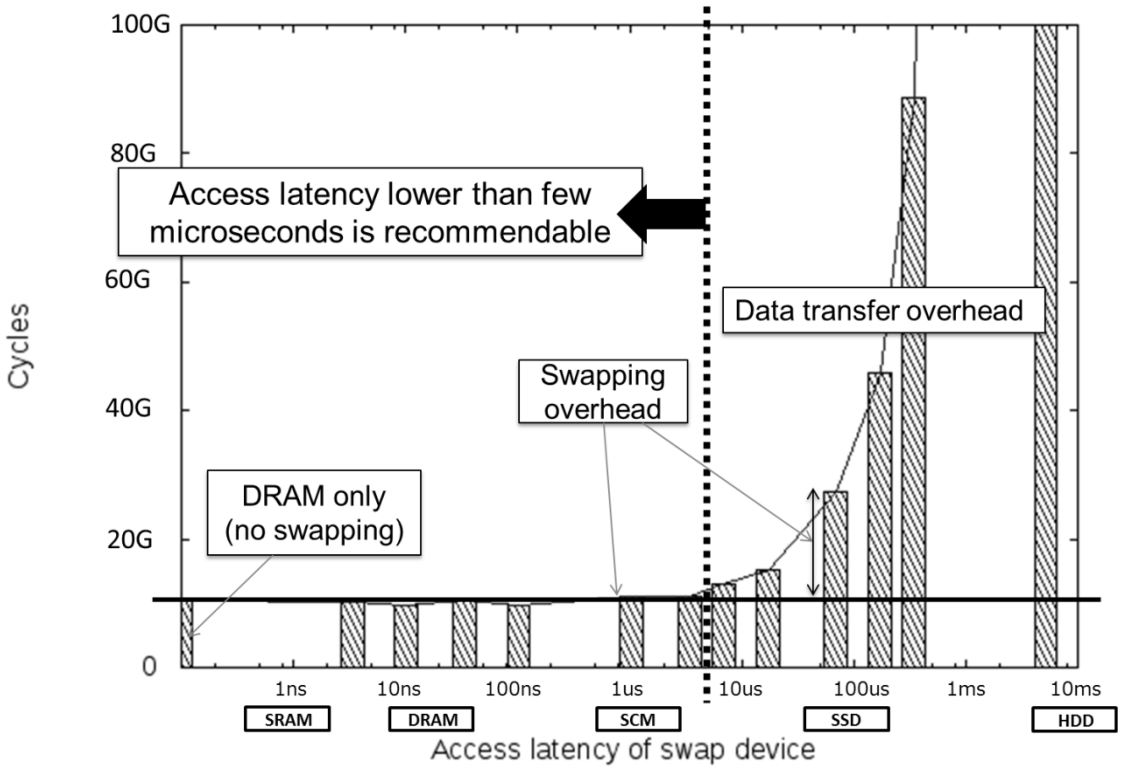

図 4.3.　アクセスレイテンシが実行時間に及ぼす影響（facesim）

まず、PARSEC の中ではメモリアクセスが多く提案方式に不利な条件での評価が可能になる facesim を用いて評価をおこなった。評価は、スワップ処理が発生しないために必要な DRAM サイズに対して、1/5 程度まで DRAM サイズを小さくして実行した。シミュレーション結果を図 4.3 に示す。この図からアクセスレイテンシが SSD や HDD 程度の付近では、スワップ処理のオーバヘッドが非常に大きいことが分かる。DRAM 量よりも必要なメモリ量が大きなアプリケーションを実行するとスワップが発生し始めた途端に処理時間が急激に増大し、実用的な処理時間で実行できないことを示している。

一方で、ストレージクラスメモリで想定されているレイテンシでは SSD や HDD 程度のアクセスレイテンシでは実用的な処理時間で実行できないほどのスワップ処理が発生しても、性能低下はほとんど確認できない程度に抑えられることが確認できた。提案方式に不利な条件である facesim で得られた結果であり他の PARSEC ベンチマークでは提案方式の効果はより大きくなると考えられる。

さらに、数百 ns～数μs という幅広いレンジの中では最も遅い、アクセスレイテンシが

数μ秒の付近でもアプリケーション性能に影響が小さいことが確認でき、例えば PCM 系のようなライトが遅い SCM でも有効であることが確認できた。この結果より、速度は DRAM が牽引するため、ストレージクラスメモリは大容量化しやすくビット単価も低くしやすい数μｓのストレージクラスメモリが向くことが明らかになった。

つぎに、必要なメモリサイズが一番大きくメモリアクセスローカリティも高い提案方式に有利なベンチマークである dedup を用いて評価をおこなった。dedup については、facesim の場合と同程度のスワップ処理を引き起こすように DRAM サイズを 1/25 程度まで小さくして実行した場合の結果を図 4.4 に示す。

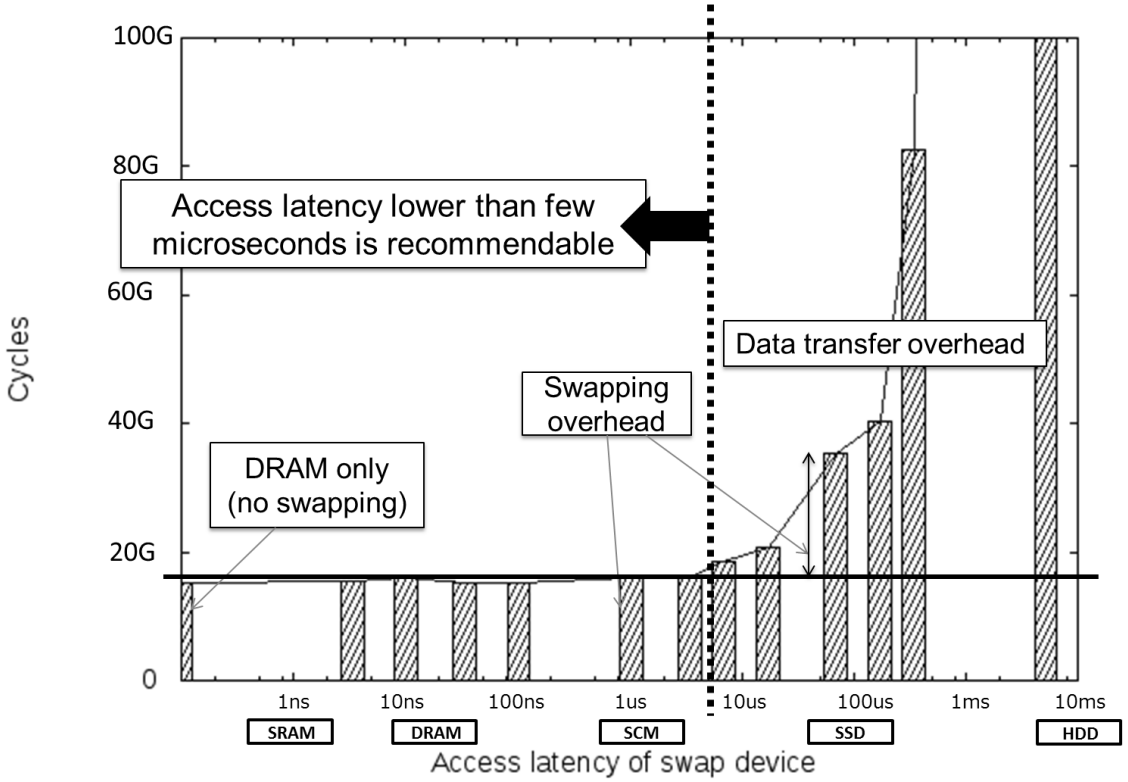

図 4.4.　アクセスレイテンシが実行時間に及ぼす影響（dedup）

この場合についても、facesim の実行結果と同様の傾向を示しており、アクセスレイテンシが SSD や HDD 程度の付近ではスワップ処理のオーバヘッドが非常に大きい一方で、アクセスレイテンシが数μ秒以下の付近までは、SSD や HDD 程度のアクセスレイテンシでは実用的な処理時間で実行できないほどのスワップ処理が発生しても性能低下は抑制されていることが確認できる。

以上の結果から、アクセスレイテンシが数μ秒程度のストレージクラスメモリがスワッ

プデバイスとして利用可能になれば、ある程度のスワップが発生しても処理性能が大きく低下しないことを確認することができた。

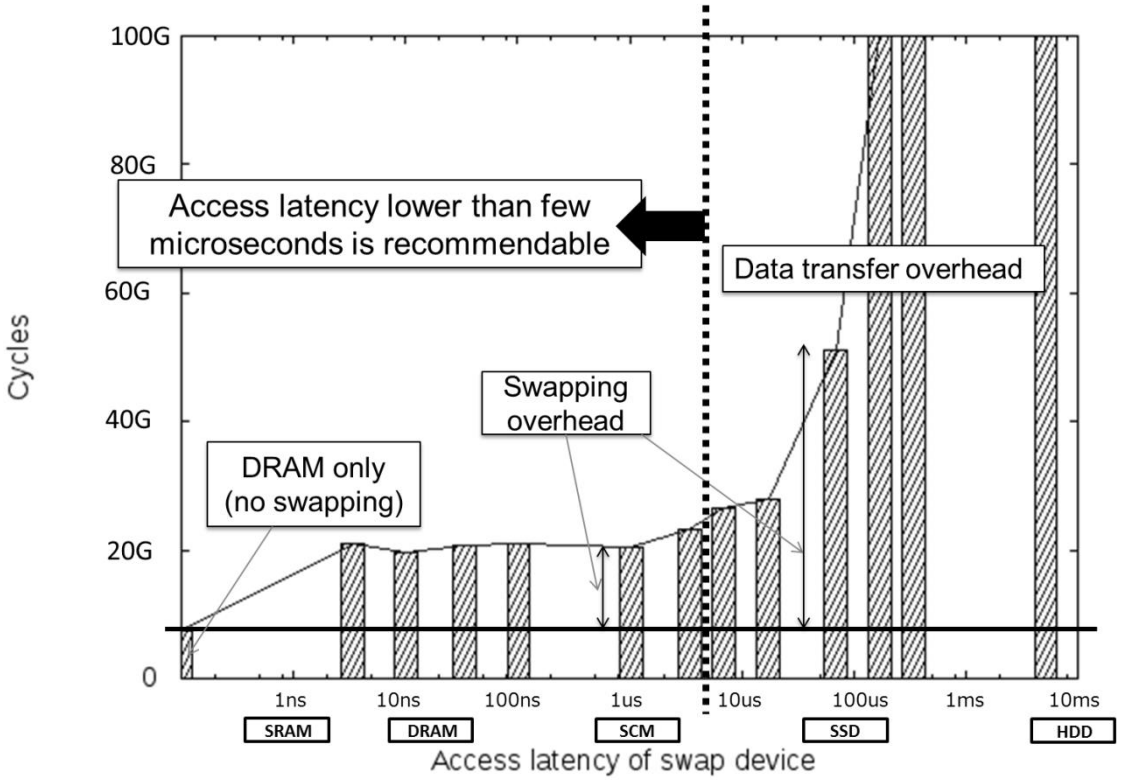

図 4.5.　アクセスレイテンシが実行時間に及ぼす影響（canneal）

最後に、図 4.5 にランダムアクセスが多いベンチマークである canneal のシミュレーション結果を示す。canneal はこれまでの 2 つのベンチマークとは傾向が異なる。具体的には、DRAM サイズを小さくした途端にスワップ処理のオーバヘッドが急激に大きくなり、スワップデバイスのアクセスレイテンシが数 μ 秒程度のところでも、スワップ処理のオーバヘッドが大きく、実行サイクルが DRAM のみで実行した場合の倍以上多くなってしまうことが判明した。これは、canneal の細かい粒度でランダムアクセスしてメモリアクセスの局所性が低いという特性と従来型の仮想記憶方式との相性の悪さが原因である。従来型の仮想記憶方式では、頻繁にランダムアクセスが発生するワークロードでは、極端なケースでは必要なページを 4KB などの単位でページインしたあとで、数バイトのみアクセスし、すぐにページアウトされてしまう。このように、頻繁にランダムアクセスが発生するワークロードでは、スワップ処理が頻発し、単にスワップデバイスを低速な HDD や SSD から高速なスト

レージクラスメモリに置き換えられればよいわけではないことが明らかになった。

これまで、ストレージクラスメモリクラスの速度を持つスワップデバイスが想定されていなかったため顕在化しなかった従来型の仮想記憶方式の課題を本研究の評価により明らかにすることができた。今回の評価結果から、仮想記憶方式の今後の課題の一つとしてはページサイズが挙げられる。これまではラージページ[123]を利用することが重要と考えられてきたが、ストレージクラスメモリの性能を引き出すためには、ストレージクラスメモリが持つバイトアドレッサビリティを活かし、ランダムアクセスパタンではより小さいページサイズを利用できるようにすることも重要と考えられる。

さらに、canneal についてベンチマーク実行時間の内訳を図 4.6 に示す。この図では、実行時間をユーザモードとカーネルモードで分けて表示している。カーネルモードの処理時間は、スワップ処理に掛かる OS オーバヘッドによる時間とデータ転送時間の合計と見ることができる。本評価はフルシステムシミュレーションでおこなっているため、速度が SRAM なみに高速なスワップデバイスという理想的なスワップデバイスを使った場合の評価が可能になることを利用し、データ転送時間の影響を非常に小さくした場合の評価をおこなった。その結果、実行時間がストレージクラスメモリを想定した場合と変わらないことが判明した。これは、ストレージクラスメモリで想定されるレイテンシのスワップデバイスを用いた場合のカーネルモードの処理の大部分は OS オーバヘッドが大部分を占めることを意味している。これはスワップ処理ごとにコンテキストスイッチが発生するためと考えらえる。これはストレージクラスメモリをプロセッサに I/O 接続していることが原因であるため、ストレージクラスメモリを DIMM に搭載しプロセッサと接続することで、ビッグデータ処理などのランダムアクセスが多いアプリケーションのオーバヘッドを大幅に削減できる可能性を示唆する結果が得られた。このように、プロセッサとストレージクラスメモリをどのようなインタフェースで接続するか、そして、それがアプリケーションの性能や消費電力にどのように影響するかは、今後のコンピュータシステムを設計するうえで重要な課題である。本評価では、ストレージクラスメモリを DIMM 化すること、そして、小さいページサイズを利用可能にすることなどが本方式の対象アプリケーションをランダムアクセスが多いものにまで広げていくうえで重要であることを明らかにした。

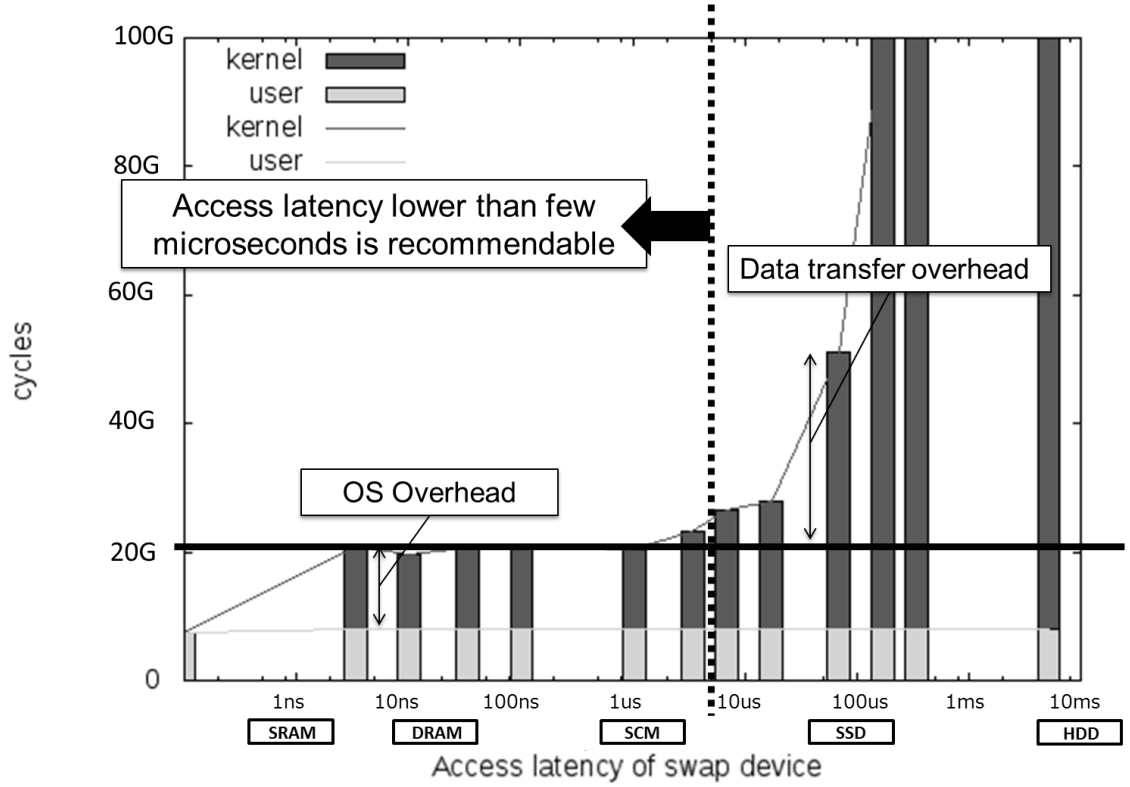

図 4.6.　canneal ベンチマークの実行時間の内訳

4.4.4　ストレージクラスメモリのアクセス電力が消費電力に及ぼす影響の評価

つぎに、各種ストレージクラスメモリのアクセスの消費電力がメモリシステム全体の消費電力に及ぼす影響について同じ PARSEC ベンチマークを用いて評価した。

消費電力量は、省電力化方式の効果を見積もるために本方式に基づく以下のような消費電力モデルを作成し、フルシステムシミュレータにより取得したアプリケーション実行時の統計情報と、各種ストレージクラスメモリのアクセス速度や消費電力等のパラメタを適用して算出している。

```
DRAM_dynamic + DRAM_static + DRAM_dynamic_swap + SCM_dynamic_swap_xN
```

ここで、DRAM_dynamic は、スワップ処理を除くアプリケーションのメモリアクセスに伴う DRAM の動的消費電力量であるため、ストレージクラスメモリのメモリアクセスレイテン

シなどの特性に依らずほぼ一定と考えられる。DRAM_static は、DRAM の静的電力量であり、DRAM サイズを小さくできればそれに応じて小さくできる。一方で、ストレージクラスメモリのレイテンシが大きくなりスワップ処理時間が増加しアプリケーションの実行時間が長くなるとそれに比例して大きくなってしまう。DRAM_dynamic_swap は、スワップ処理に掛かる DRAM の動的消費電力量であり、スワップ処理回数に比例して変化する。なお、DRAM のアクセス電力量は、リードもライトも 10pJ/bit で計算している。SCM_dynamic_swap_xN は、スワップ処理に掛かるストレージクラスメモリの動的消費電力量であり、スワップ処理回数に比例して変化する。この値は、各種ストレージクラスメモリに依って大きくことなるため、3 種類のストレージクラスメモリを想定し、それぞれアクセス電力量が DRAM の 2 倍(x2)、4 倍(x4)、10 倍（x10）とする。

各種ストレージクラスメモリのアクセスの消費電力がメモリシステム全体の消費電力に及ぼす影響について同じ PARSEC ベンチマークを用いて評価した結果を図 4.7～図 4.9 に示す。なお、canneal については、ストレージクラスメモリを用いても実用的な時間で処理できないことが分かっているので消費電力の評価から除外している。

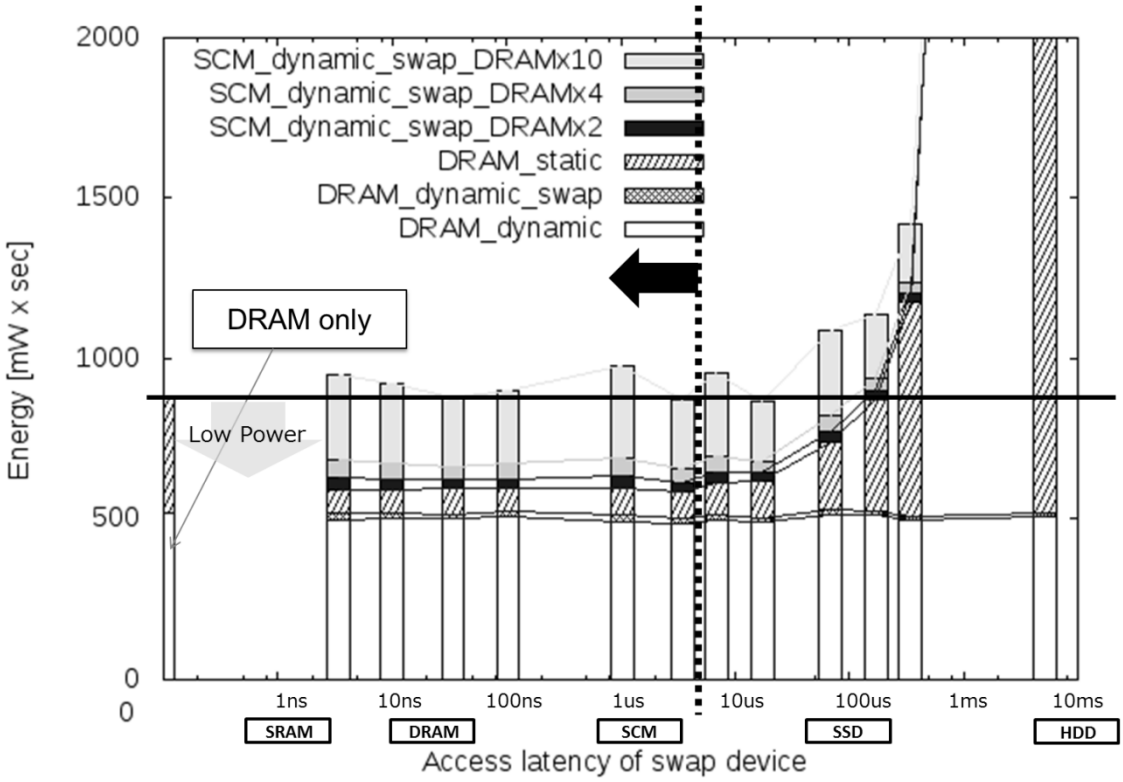

図 4.7.　アクセス電力が消費電力に及ぼす影響（facesim）

ここでは、スワップデバイスのアクセス電力がメモリシステム全体の消費電力に与える影響のトレンドカーブを見極めるために、先ほどと同様にアクセスレイテンシを幅広く変化させる。図の縦軸は消費電力量である。

図の一番左側の縦棒だけは先ほどと同様に特別扱いで、アクセスレイテンシが 0 というわけではなく、十分なサイズの DRAM のみでベンチマークが実行された場合の消費電力を示しており、スワップ処理によるオーバヘッドがない。即ち、この縦棒と同じ高さで引かれている線より下であれば、本方式により省電力化が達成可能であることを示す。

図 4.7 の facesim の場合、4.4.3 の評価結果より、本方式ではストレージクラスメモリのアクセスレイテンシは数μ秒でよいことが分かったのでアクセスレイテンシが 1μ秒で、まずはアクセス電力が小さい(x2)ストレージクラスメモリに注目する(図 4.8)。

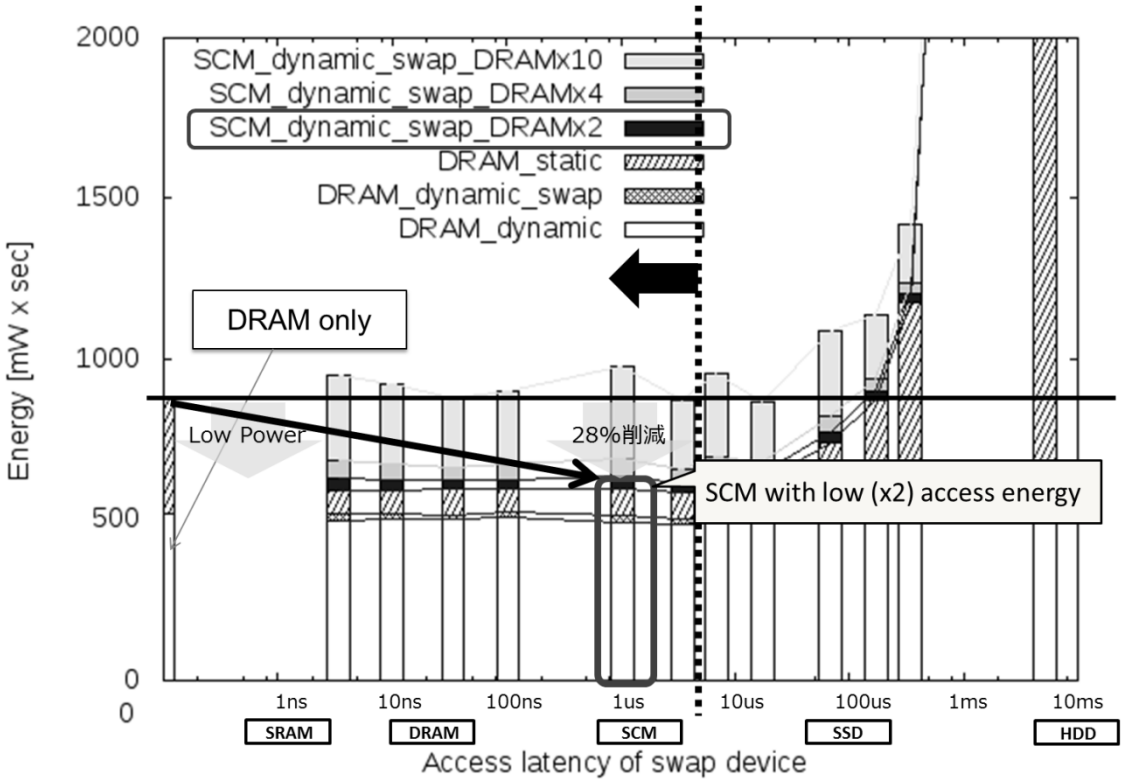

図 4.8.　アクセス電力が小さいストレージクラスを用いた場合の省電力効果（facesim）

図より、DRAM のみで実行した場合と比較すると、DRAM サイズを小さくした効果として、DRAM_static が削減されているのが分かる。一方で、その副作用として増加した SCM_dynamic_swap_DRAMx2 は相対的に小さく、結果として全体で 28%の消費電力量が削減さ

れており、本方式の有効性が確認できた。

つぎに、アクセスレイテンシが 1μ秒で、アクセス電力が大きい (x10)ストレージクラスメモリに注目すると、今度は副作用として増加した SCM_dynamic_swap_DRAMx2 は相対的に大きく、結果として全体での消費電力量は逆に増加してしまった。このように、ストレージクラスメモリのアクセス電力は本方式の有効性に大きく影響することが明らかになった。これは、提案方式は積極的にスワップアウト、つまりライトが発生するため、消費電力の観点では DRAM に対して数倍程度までの小さい動的電力が要求されることが明らかになり、例えば、ライトのアクセス電力が 10 倍程度と高い PCM 系などには向かない可能性があることが分かった。アクセス電力が 2 つの中間の (x4)ストレージクラスメモリではまだ消費電力量は大きく削減できているため、アクセス電力量が DRAM の 4 倍程度までのストレージクラスメモリが望ましい。

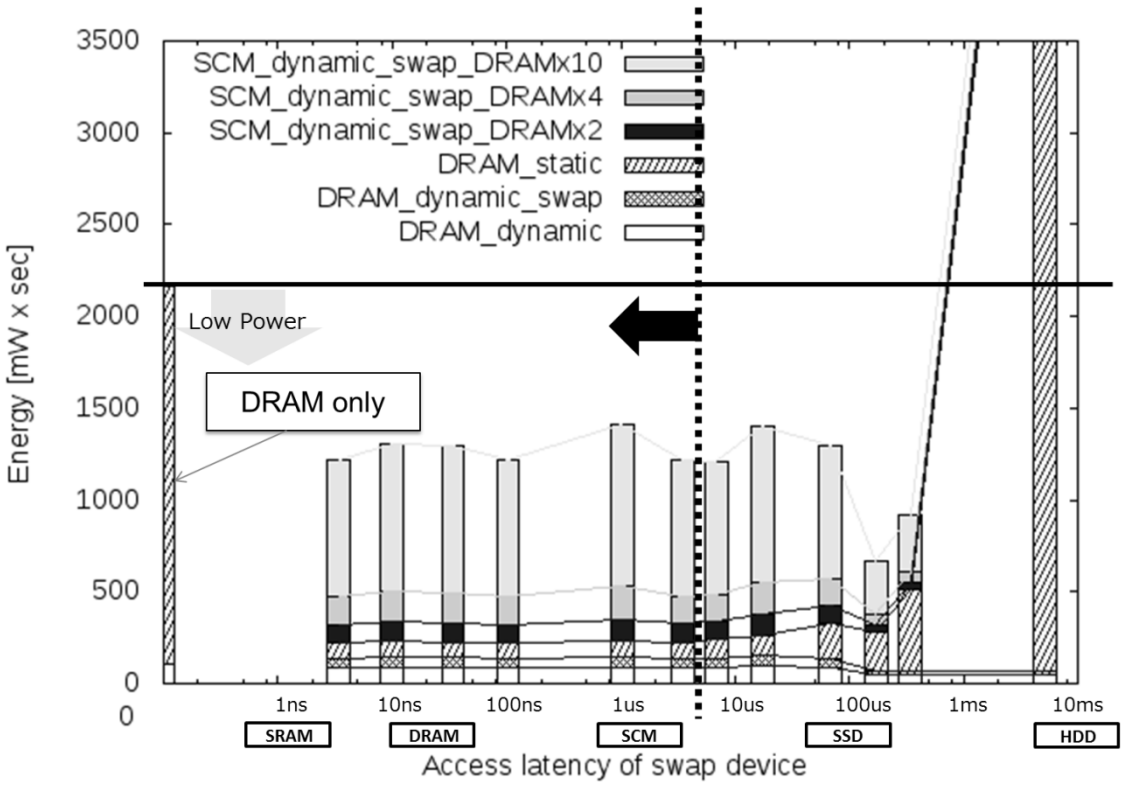

図 4.9.　アクセス電力が消費電力に及ぼす影響 (dedup)

図 4.9 の dedup の場合も同様に、アクセスレイテンシが 1μ秒で、アクセス電力が小さい (x2)ストレージクラスメモリに注目すると、DRAM サイズを小さくした効果として、

DRAM_static が大幅に削減されているのが分かる。一方で、その副作用として増加した SCM_dynamic_swap_DRAMx2 は相対的に小さく、結果として全体で 83%の消費電力量が削減されており、本方式の有効性が確認できた。アクセス電力が大きい (x10)ストレージクラスメモリでも省電力化はできるものの、省電力効果は小さくなる。このように、消費電力の観点では DRAM に対して数倍程度までの小さい動的消費電力が要求されることが明らかになり動的消費電力が低い MRAM 系などや、PCM 系などよりは動的消費電力が低いとされる 2 つのメモリの中間の ReRAM 系などが向く可能性があることが分かった。

また、facesim の場合の DRAM の比率がストレージクラスメモリの 1/5 程度だったのに対して、ローカリティが非常に高い dedup は DRAM の割合をそのさらに数割である 1/25 程度まで小さくしても発生するスワップ回数は facesim と同程度であるため、より積極的なストレージクラスメモリへの追い出しが可能であったり、canneal のようなランダムアクセスの場合は DRAM サイズを小さくしづらいなど、アプリケーションのアクセスパタンに応じて DRAM サイズを動的に変化させ電源オフすることが重要である。しかし、メモリアクセスパタンはスワップレートに集約することができるため、スワップレートを監視する既存研究[127]と組み合わせることで実現できると考えられ、その点については今後の重要な研究課題である。また、今後サーバシステムにストレージクラスメモリと DRAM を混載させる際に、どのような割合で混載させるのかは大きな議論になると考えらるが、1/5 というのはその一つの指標になると考えられる。

データセンターのワークロードは変動するため、動的に DIMM 単位でパワーオフすることが効果的と考えられるが、組込みシステムでも同様に、DRAM のサイズをランクやバンクの単位で動的にリサイズして省電力化できる可能性もあり、さらに言えば、組込みシステム上の DRAM サイズを増やさずによりメモリへの要求が高いアプリケーションを省電力に実行できる可能性もあることを示している。

4.5 関連研究

OS の仮想記憶システムは、アプリケーションプログラムが利用できるメモリを仮想的に大きくする目的で開発・利用されてきた。これによりノードが持つ物理メモリを越えるメモリサイズを要求するアプリケーションを動作させることはできる。しかし、DRAM に乗り切らないページを記憶するためのストレージスワップデバイスは低速な HDD (Hard Disk Drive) であったため、DRAM で構成される主記憶と HDD 間のデータ転送がオーバヘッドとなり、DRAM にすべてのデータが収まる場合と比較して非常に低速になる。そのため、スワップデバイスを主記憶の一部として高性能が要求されるプログラムを動かすのは難しかった。

Myrinet[34]や Infiniband[35]など、大規模クラスタ型並列計算機上で利用されるネット

ワークの高性能化が進むのと同時に、注目されたのが Nswap や Teramem などローカルディスクへのスワップと比較して数桁高速なアプリケーション実行が可能になる遠隔スワップシステムである[30-33,36,37]。遠隔スワップシステムは、ネットワークを経由して大規模クラスタ型並列計算機上の他ノードのメモリをスワップ領域として利用する。これにより、クラスタの各ノードのメモリを集約して巨大な仮想メモリを構築することが可能である。JumboMem[36] は、256 ノードのメモリサーバを利用して 1TB のメモリ空間を実現している。

HDD より 3 桁程度高速な NAND フラッシュ SSD が広く普及しクラスタシステムで利用可能になると、これを DRAM と組み合わせて利用する研究が盛んになる[38-42,44,45]。Saxena らは文献[39]で DRAM とスワップデバイスの容量比率と性能の関係を示しており、ハードディスクの替りに SSD を利用すると、同じ DRAM サイズでアプリケーションを高速化できるメリットと、同じ速度を維持しつつ DRAM サイズを小さくして消費電力やコストを削減できるメリットがあることが分かる。さらに、OS の仮想記憶システムは低速なハードディスクで性能を出すために最適化されており単純にスワップデバイスを SSD に置き換えるだけでは SSD の性能を十分に引き出せないことを指摘している。具体的には、例えば、HDD は大きいシークのレイテンシの影響をできるだけ小さくするため、スワップイン時に隣接する複数ページをプリフェッチする最適化を行っている[46]が、スワップデバイスが SSD くらい高速だと既存の仮想記憶システムのプリフェッチ機能をオフにするだけで性能が 25%も向上すると述べている。それらのハードディスク向けの最適化を SSD 向けに見直す改良を行った仮想記憶システムである FlashVM を開発している[39]。

FusionIO 社主導の OpenNVM プロジェクトで開発された fastswap[42,43]もフラッシュ SSD をスワップデバイスとして最適化したスワップシステムである。しかし、フラッシュ SSD も DRAM に比べれば非常に低速なので、fastswap では、アプリケーションがページのスワップをメモリのアクセスパタンをヒントすることが可能な madvise 機能を利用することで、スワップの発生を極力抑えることをアプリケーション開発者に要求している。

東京工業大学では、消費電力の観点でも優れた SSD を DRAM と組み合わせて利用することで、単一の計算ノード上の DRAM に収まらない容量のグラフの処理の評価を行っている[40]。文献[40]では、グラフ処理のアルゴリズムを改良し、検索中に頻繁にアクセスされ性能に大きな影響を及ぼすグラフデータを DRAM に残してその他の性能に影響が少ないグラフデータを SSD に退避することで、性能の低下を抑えながら DRAM の容量を削減している。アプリケーションを特性にあわせて細やかな改良を施すことで DRAM 使用量は約 1/2 までは削減でき少ない DRAM でより大規模なグラフを実行できる可能性は示しているが、SSD へのアクセスが原因で性能は約 25%低下している。

SSD より高速な新型の高速不揮発メモリであるストレージクラスメモリの登場の期待が高まっており、ストレージクラスメモリをスワップデバイスに利用を提案する論文が少しずつ出始めている[3,7,48,49]。

Park らは、PCM をスワップデバイスとして利用する仮想記憶システムを提案している[3]。

PCM ベースのスワップシステムでは、PCM と DRAM 間のデータ転送が非常に高速でページフォルト処理時間が短いため、スワップのページサイズを小さくして先読みのオプションもオフにして必要なデータを適宜スワップインすることで性能が向上するというシミュレーション結果を得ている。これらは、Saxena らのアイディアを推し進めたものであるともいえる。ストレージクラスメモリを活用したシステムでは文献[3]のように PCM などの特定のストレージクラスメモリを前提としたものが多い。しかし、スワップデバイスの用途にどのようなストレージクラスメモリが向いているのかを明らかにしていく必要がある。本研究の貢献の一つは、各種ストレージクラスメモリの性能特性が本方式の有効性に与える影響をフルシステムシミュレーションにより評価している点である。

4.6 まとめと今後の課題

実用化が期待されている高速・大容量な不揮発メモリであるストレージクラスメモリを主記憶として搭載する次世代のコンピュータシステムの省電力化方式として、SCM/DRAM 混載メモリシステムの階層制御技法とこれを自動化するストレージクラスメモリ向け仮想記憶の基本方式の提案と、基本方式の初期評価をおこなった。具体的には、高速・大容量なストレージクラスメモリを DRAM と共に主記憶として混載し、待機消費電力が小さいストレージクラスメモリの特性を活かして DRAM 上のデータを積極的にストレージクラスメモリに退避して、使用する DRAM サイズを削減するとともに未使用 DRAM の電源をオフすることで動作時の消費電力を削減する方式を検討した。

省電力化方式の効果を見積もるために本方式に基づく消費電力モデルを作成し、フルシステムシミュレータにより取得したアプリケーション実行時の統計情報と、各種ストレージクラスメモリのアクセス速度や消費電力等のパラメタを適用して、消費電力を評価した。アプリケーションのメモリアクセス局所性が極端に低い場合を除き、一般的にストレージクラスメモリで想定されている数百 ns～数μs という幅広いレンジの中では最も遅い数μs のレイテンシでもローカリティの高い facesim と dedup では性能を維持しつつ低消費電力化できることがまず確認でき、ストレージクラスメモリに適した有効な方式であることが分かった。

また、書込みに要する消費電力が小さいストレージクラスメモリがあればその特性が活かせることが分かった。スワップ処理のページングのアルゴリズムを工夫することで更なる省電力化の可能性もあり、今後は更に詳細な評価を進めていく。

本研究の評価は、フルシステムシミュレーションによるものである。フルシステムシミュレーションはスワップデバイスのレイテンシが柔軟に変更できるなどの利点がある一方で、大規模な実アプリケーションによる評価は困難であり、今回はそこに至っていない。

今後は、大規模な実アプリケーションで評価をおこなっていくことが課題である。今後は実サーバを用いてできる評価は実サーバでおこなえるような評価環境を検討し、フルシステムシミュレーションの評価結果と突き合わせながら、評価の精度をあげていくことが課題である。

5. 新型高速不揮発メモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ

5.1 まえがき

近年、携帯型のタブレット端末やウェアラブル型のスマートウオッチなどの情報処理端末において、電力利用の効率化が重要な技術課題となっている。これらの端末においては、全体の消費電力のうち LCD などの表示装置のバックライトやリフレッシュが占める割合が高いことが知られている[71]。それに呼応して、不揮発性メモリをフレームバッファに利用するディスプレイ[72]や、1 Hz 程度の低いリフレッシュレートで静止画を表示可能なIGZO[73]やメモリ液晶などの低消費電力化したディスプレイが続々開発されている。

その中でも、電子ペーパ(EPD: Electronic Paper Display)[74]は、表示内容に変化がない場合には電源を遮断しても表示を保持できる不揮発性のディスプレイとして注目されており、様々な EPD 方式が存在する[75-77]。EPD 技術は進化を続けており、Kindle[78]などの読書に特化した電子書籍端末で広く普及している E Ink[75]の電気泳動方式では画面を白黒反転させるリフレッシュ処理を削減する改良がなされたり、Mirasol[76,79]などのカラーで高速な EPD も開発されはじめている。

また、EPD と LCD を 1 つの端末に持つ[2]Android™ OS を搭載したスマーフォン[80]なども製品化されている。情報処理端末のディスプレイに不揮発性の EPD を採用することで、 電子ペーパ搭載端末でドキュメントワークを行っている最中でもユーザが画面を閲覧している入力待ち状態では、LCD のように定期的にリフレッシュする必要がない。そのため、表示を続けた状態でプロセッサを低消費電力モードに移行させてアイドル時の消費電力を下げることで、不揮発性を活かして情報処理端末の消費電力を大幅に低減することができ、これまでにないような超低消費電力な端末を実現できる可能性が出てきている。

しかし、不揮発性ディスプレイの場合、書換え処理は、電源を遮断しても表示を保持できる安定した状態間を切り換えるため、大きな電力が必要となる。より小さい電力で EPD 搭載端末を駆動可能にしてバッテリ寿命の向上や小さいバッテリで軽量化を実現するためにも、単に既存の LCD を EPD で置き換えるだけでは十分でなく、EPD の書換え処理に掛かる消費電力を最小化する細やかな低消費電力制御が重要である (図 5.1)。

[2] Android は、 Google Inc. の商標です。本論文に掲載の商品、機能等の名称は、それぞれ各社が商標として使用している場合があります。

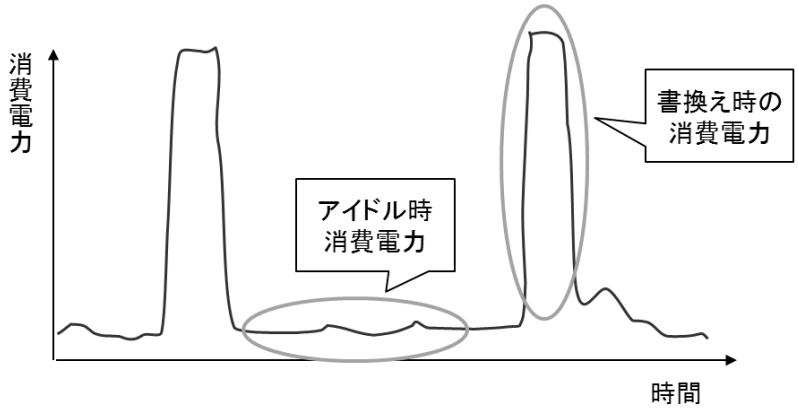

図 5.1.　書換え時の消費電力の削減

一方で、現在のタブレット型端末は DRAM が主記憶として利用されているが、ストレージクラスメモリが実用化されると、ストレージクラスメモリと電子ペーパのような不揮発ディスプレイを組み合わせることで待機消費電力が非常に小さく例えば太陽電池の電力のみで駆動可能な超低消費電力のタブレット型端末の実現可能性も出てくる。しかし、ストレージクラスメモリも動的消費電力が大きいため、DRAM を単純に置き換えるとかえって電力が大きくもなり得る。さらに、電子ペーパは書換え処理時間が LCD と比較すると非常に長いため、ストレージクラスメモリと電子ペーパを組み合わせると電子ペーパを書換え中メモリアクセスし続けるので、消費電力が大きくなってしまう。

本研究では、ストレージクラスメモリと不揮発ディスプレイを搭載したタブレット型端末向けの省電力制御機能を組み込んだディスプレイコントローラのデバイスドライバを提供することで、LCD や DRAM 向けの従来型のアプリケーションプログラム開発方式を変更することなく、省電力化を実現できるプログラムを開発できることを明らかにする。本方式では、図 5.2 に示すようにデバイスドライバが電子ペーパ書換え処理のためのメモリアクセスを階層制御することによりストレージクラスメモリへのメモリアクセス時間と書換え処理時間を削減する[70,83]。提案方式は、2 つの省電力書換え制御方式から構成される。第 1 の省電力書換え制御方式は、図 5.2 に示すように、プロセッサの内部メモリへ表示するデータをコピーし、電子ペーパコントローラは内部メモリから表示をおこない(図 5.2(a))、その間ストレージクラスメモリの電源をオフにすることで不揮発メモリの省電力性を引き出す(図 5.2(b))。現在のプロセッサの内部メモリはまだサイズが小さいため実機で評価できる段階には至っていないが、それが可能になれば、階層制御をおこなったうえでさらに第 2 の省電力書換え制御方式を組み合わせることでさらなる省電力化が可能になる。第 2 の省電力書換え制御方式は、複数の書換え命令を実行時にデバイスドライバレベルで再構成し小さな電力で書換え可能な命令に変換することで書換え処理をまとめて書換

え処理時間とそれに伴うストレージクラスメモリへのメモリアクセス時間を短縮する。これは電子ペーパの書換え処理には時間が掛かるため、断続的に書換え命令がアプリケーションにより発行されるとその間常に電子ペーパが書換え処理中になってしまうからである。2 つの省電力書換え制御方式は個別に適用しても効果があることから、本研究では第 2 の省電力書換え制御方式の評価を実施したものである。

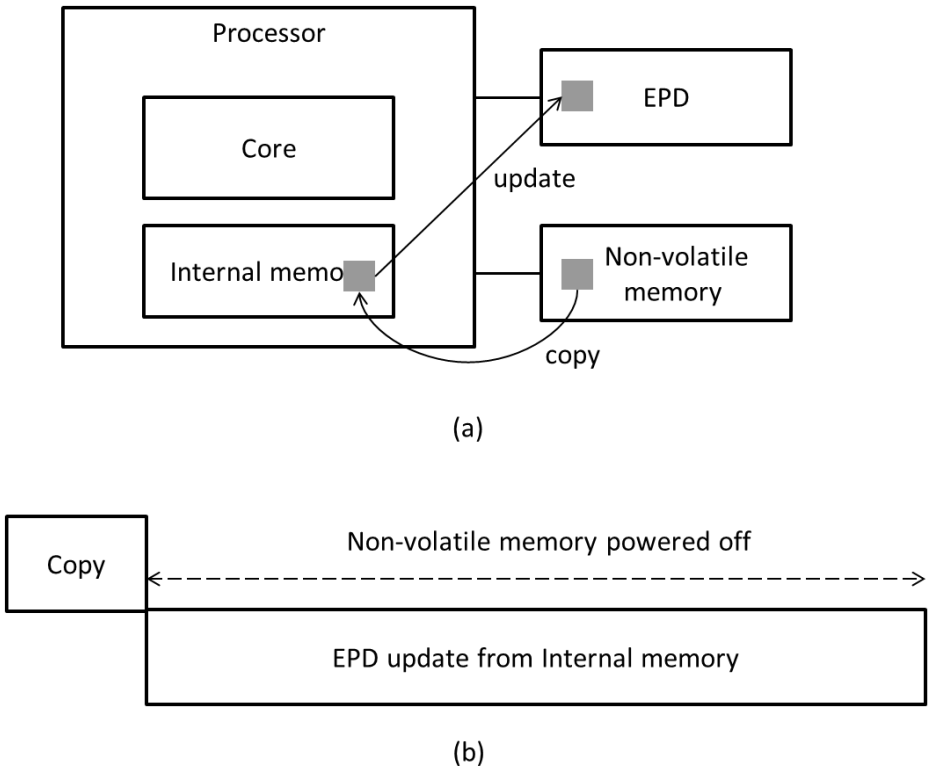

図 5.2.　ストレージクラスメモリ搭載電子ペーパ搭載端末における階層表示制御

5.2　電子ペーパ書換え処理の概要

本節では、EPD 書換え処理の低消費電力制御方式の評価で用いる E Ink EPD 搭載端末評価ボード上での書換え処理シーケンスの概要を説明する。

我々の開発した評価ボード(図 5.3)では、EPD コントローラを内蔵した i.MX508[81]アプリケーションプロセッサを利用している。

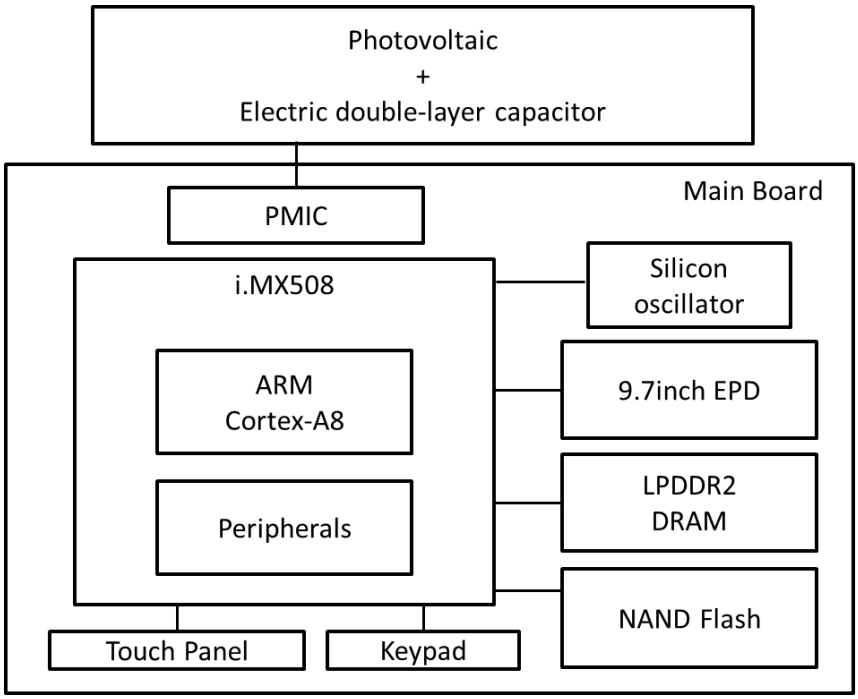

図 5.3.　評価用ボード

まず、i.MX508 上で動作するアプリケーションが、EPD に表示する書換え用データを主記憶上の RGB 形式のフレームバッファに書き込む。LCD と異なり、EPD には定期的なリフレッシュがないので、アプリケーションが明示的に書換え命令を EPD コントローラのデバイスドライバに発行する。

つぎに、デバイスドライバが、アプリケーションから指定された書換え用データをアクセラレータである ePXP(Enhanced Pixel Pipeline)を利用して RGB から EPD で表示可能なグレースケール色空間への変換などの EPD 特有の前処理を実行する。前処理済みデータは、主記憶上に確保された EPD コントローラ専用のワーキングバッファにコピーされ、そこから書換え処理が実行される。

EPD は、現在の LCD と異なり、EPD コントローラにより画面を部分的に書換えられるのが特徴の一つである。i.MX508 プロセッサの EPD コントローラは、16 個の書換えエンジン(LUT)を有しており、矩形の部分領域を並列にかつ非同期的に書換え可能である。

5.3　電子ペーパ書換え処理の課題

5.3.1　断続的に発行される書換え命令による書換え処理時間の増加

タブレット端末などで動作する既存の多くのアプリケーションを EPD 搭載端末上で利用できることが望ましい。しかし、これらは LCD を前提に作成されており、LCD の高速に書換え可能な特性を利用して操作感を高めるために、準備できたものから表示して応答性を高めたり、様々なエフェクトを付加している。そのため、これら LCD 向けアプリケーションがフレームバッファに表示データを書き込む時に単純に EPD に書換え命令を発行するだけの改良では、アプリケーションから書換え命令が断続的に発行されてしまう。

EPD は、デバイスの特性上、書換え処理時間(EPD コントローラが動作中の時間)が LCD と比較すると非常に長いのが特徴である。E Ink EPD には、複数の書換えモードがあり、残像が残るが書換え時間が比較的短い高速書換えモードもあるが、残像を残さずに書換える通常書換えモードでは、我々の評価ボード上で 1 秒程度掛かる。そのため、断続的に書換え命令が到着するとその間、常に EPD が書換え処理中のままになってしまう。

図 5.4 で、画像ビューアからサムネイル表示の際に複数の画像の書換え命令 A、B、C が短期間に集中して発行される例を用いて説明する。EPD コントローラの既存のデバイスドライバは、書換え命令を画像ビューアから受信するとすぐに EPD コントローラに書換え処理開始を指示する。すると、命令 A の書換え処理開始から命令 C の書換え処理終了までの間、EPD だけでなく、EPD コントローラを内蔵している SoC、EPD コントローラのワーキングバッファがある主記憶、EPD に電源供給している EPD 用の PMIC などが常時アクティブになり、消費電力が大きくなってしまう。

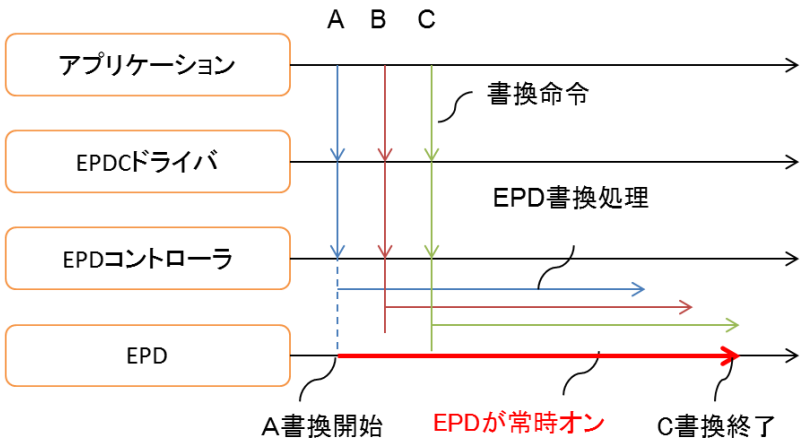

図 5.4.　集中して発行される書換え命令による EPD 書換え処理時間の増加

5.3.2　コリジョンによる書換え処理時間の増加

EPD の各々の書換え処理は非常に時間が掛かるが、複数の書換え命令が連続して発行された場合、EPD コントローラの複数の LUT をうまく利用して並列に書換え処理すれば動きのある表示も見せることもできる。しかし、複数命令間で表示領域が重なる場合、EPD コントローラ内部で書換え処理の衝突（コリジョン）が検出され、先行する書換え処理の完了を待ってから重なった領域の書換え処理を再実行する必要がある。そのため、書換え処理時間が更に長くなり、消費電力のさらなる増大をもたらす。

コリジョンが多発し得る例として、例えば画像ビューアでは、画像がスライドインし表示される処理、画像の拡大・縮小やスクロール処理などがある。ここでは、ドキュメントビューアで、文章の上にページ番号が重ねて表示される例を説明する。例えば、文章の表示領域 D の書換え命令が発行された直後に、ページ番号を重ねて表示する小さい部分領域 E の書換え命令が発行される場合を考える。

図 5.5 は、この際に、後発の領域 E の書換え命令が先行する領域 D の書換え命令とコリジョンする様子を示している。先行する領域 D の書換え処理が完了すると、EPD コントローラからの割り込みが通知されるので、デバイスドライバは、領域 E の再書換えである E’ を指示する。このように、先行する書換え処理が完了するまで後続の書換え処理が待たされるため、書換え処理時間が更に長くなってしまう。

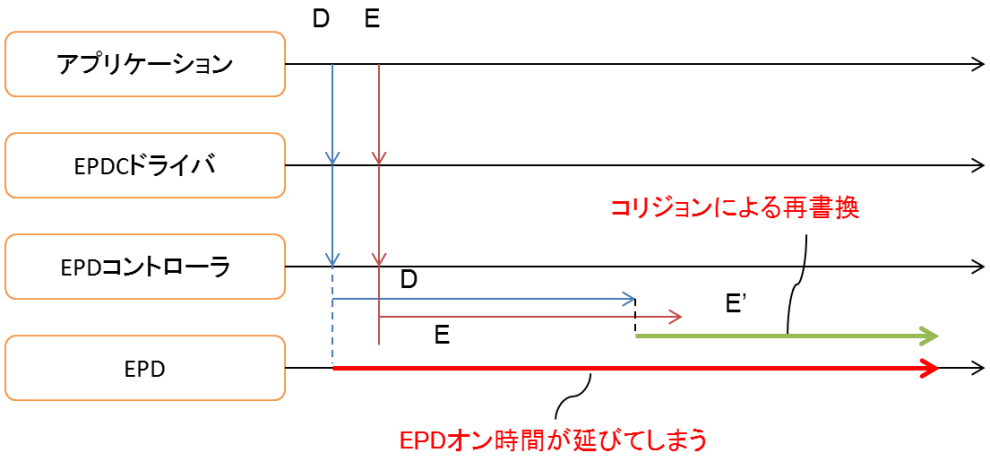

図 5.5.　コリジョン発生による書換え処理時間の増加

5.4　電子ペーパ書換え処理の省電力制御方式

本節では、アプリケーションから断続的に発行される書換え命令から時間的に近接する命令列を検出し、効率良く書換え可能な命令列に再構成する低消費電力制御方式について述べる。

5.4.1　EPD スケジューラの基本アルゴリズム

5.3 節で示した EPD の消費電力を増加させる問題の回避をすべてのアプリケーションに個別に手を入れておこなうのは容易ではない。また、EPD 向けにアプリケーションを新規に作成する場合も、EPD の特性を十分に理解しないといけないのでアプリケーション開発者の負担となる。そこで、本研究では、EPD コントローラのデバイスドライバで、EPD に不向きな書換え処理を EPD 向けに最適化する EPD スケジューラを提案する。EPD スケジューラは、時間的に近接する書換え命令の実行タイミングを揃えたり、コリジョンを回避することで、EPD や EPD コントローラなどの動作時間を短縮し、EPD 搭載端末の超低消費電力化を実現する。

5.3 節で述べた 2 つの課題について、EPD スケジューラでの解決手法を以下で説明する。

まず、1 つ目の課題である集中して発行される複数の書換え命令による動作時間の増加の解決手法について述べる。解決手法では、先行する書換え命令を EPD コントローラにすぐに実行させずに、複数の命令が揃うのを待ってから、書換え開始タイミングを揃えて一気に実行させることで、書換え処理時間の短縮による消費電力の削減を狙う。この際の待ち時間は、例えば数十 ms～100ms 程度と、EPD の書換え時間と比較して十分短くなるようにしてユーザに実行遅延の影響を感じさせないのが望ましい。

図 5.6 は、図 5.4 の書換え処理に本手法を適用した様子を示している。図に示すように、書換え命令 A や B が到着しても、EPD スケジューラではすぐに書換え処理を開始させない。最初の命令 A が到着してから一定時間待ち、その時点でデバイスドライバに到着している命令 C までを近接書換え命令とみなして、空間的・時間的に再構成して EPD コントローラに書換え処理の開始を指示する。空間的な再構成については、例えば、複数の書換え領域が、ある矩形領域で包含できる場合のみ、同矩形領域を新たな書換え領域として EPD コントローラに実行を指示する。仮に空間的に再構成できなくても、時間的には再構成できる。複数の書換え処理の開始時間を揃えて可能な限りオーバラップさせる。これにより、EPD や EPD コントローラや主記憶の動作タイミングを局所化し、動作時間を短くできる。

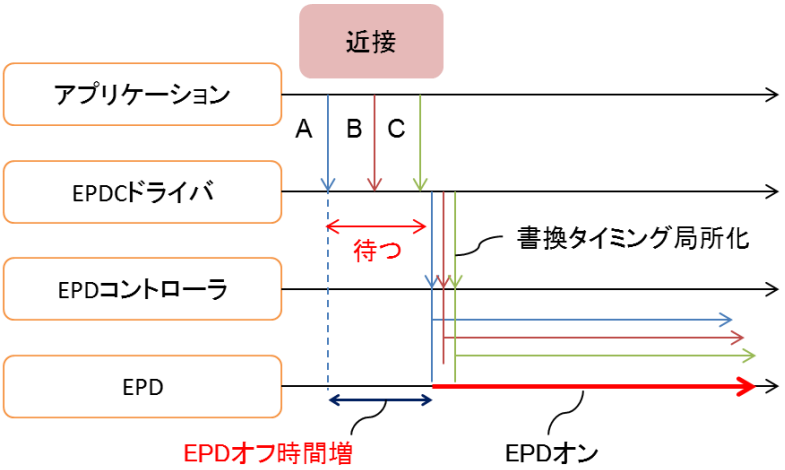

図 5.6.　書換え処理の局所化による低消費電力化

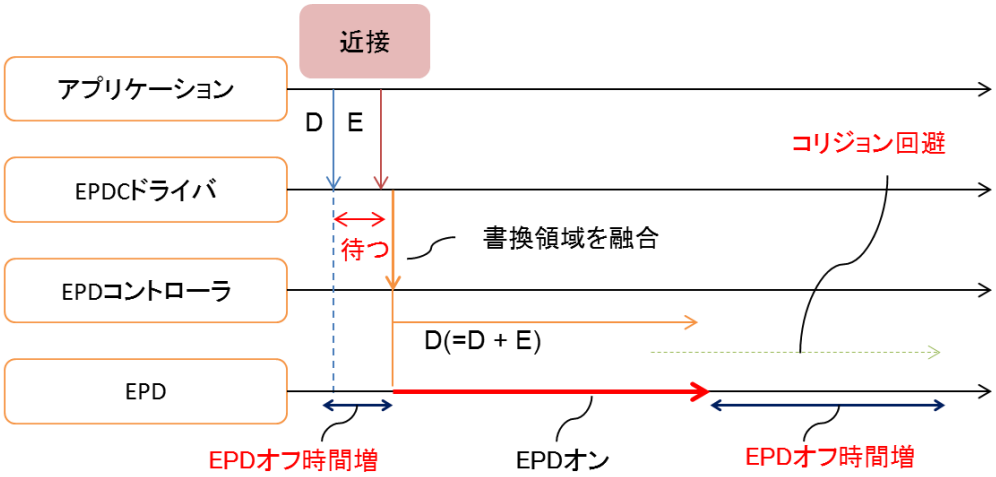

図 5.7.　コリジョン発生回避による低消費電力化

つぎに、2 つ目の課題であるコリジョン発生による書換え処理時間の増加の解決手法を示す。

コリジョンを回避する手法についても、同様に、先行書換え命令の実行を遅延させ、コリジョンを引き起こす複数命令を単一命令に融合してコリジョンを除去することで、書換え処理時間の短縮と消費電力の削減を狙う。

図 5.7 は、図 5.5 の書換え処理に本手法を適用した様子を示している。図に示すように、

最初の書換え命令 D が到着してから一定時間待ち、その時点で溜まっている、コリジョンを引き起こす領域 D と領域 E の書換え命令の単一命令への融合を試みる。この 2 つの領域は、これを包含する領域 D に空間的に融合可能なので、単一の書換え命令に再構成できる。単一の書換え命令は、当然コリジョンしないのでこれを回避でき、その分書換え処理時間が短縮できる。

これらの手法により、EPD 搭載端末のアイドル時間を増やすことができ、不揮発性ディスプレイを採用してアイドル時の消費電力を下げるシステムの省電力性をより引き出すことが可能になる。

5.4.2 EPD スケジューラの待ち時間の決定方法

EPD スケジューラの基本アルゴリズムは、一定時間待つことで時間的に近接する書換え命令を再構成する。しかし、どの程度待てばよいか決める必要があるので、これについてアプリケーションの観点で議論する。

まず、EPD 搭載端末で想定されるアプリケーションには、PDF リーダや画像ビューアのようなページをめくりながら閲覧するものが多い。これらのアプリケーションでは、ページをめくる毎に、似たような時間間隔で複数書換え命令の発行が繰り返されることが多い。

このような規則性のある書換え命令については、待ち時間をオンラインで自動調整する手法が有効である。本研究では、時間的に近接している書換え命令をグルーピングし、グループ内の命令の発行間隔からどの程度待ったらよいか推定する手法を提案する。

図 5.8 の具体例を用いて説明する。図では、ページめくり毎に、2 つの書換え命令が繰り返し発行されることを想定している。最初のページめくりで、最初のグループを形成する近接書換え命令 A、B が順次到着する。最初のグループでは、書換え命令 A が到着する 50ms 後には命令 B が到着するが、EPD スケジューラの待ち時間の初期値が 100ms に設定されていると仮定すると、50ms 無駄に待ってしまう。近接書換え命令 C、 D が形成する 2 つ目のグループでは、命令 C が到着すると、直前のグループに含まれていた命令例の最初の命令の到着時刻から最後の命令の到着時刻までの時間を新しい待ち時間に設定する。この例では、到着間隔が 50ms だったので、これに繰り返し処理でも間隔がぶれることを考慮した 20ms のマージンを加えた 70ms が待ち時間となり、無駄を小さくできる。このように、書換え命令の発行パタンに応じて動的に待ち時間を自動調整することができる。

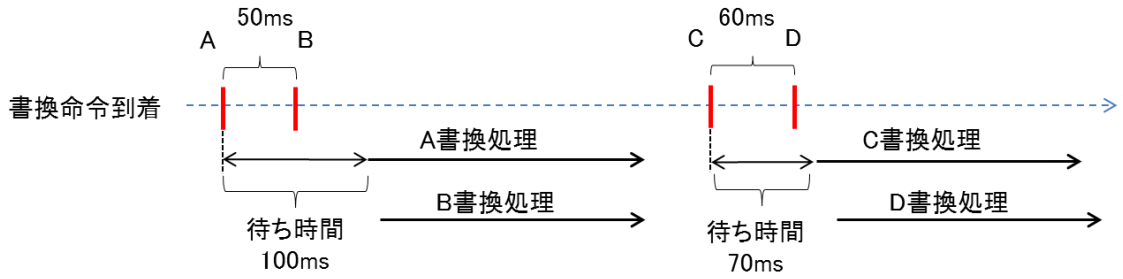

図 5.8.　待ち時間の自動調整手法

一方で、例えばブラウザでは、表示するページに依って、書換え命令の数や発行間隔も異なってくる。このように規則性がない場合には、一連の書換え命令の最後であることをブラウザからヒント情報として書換え命令に明示的に付加できれば、EPD スケジューラでは最後の命令が到着するまで待ってからまとめて書換え処理でき、余計に待って無駄に電力を使ったり EPD に表示されるまでユーザを余計に待たせなくて済む。

また、ブラウザは、データが揃うまでの比較的長い間、書換え処理が断続的に続くことが多く、最後の書換え命令まで待つとユーザエクスペリエンスを低下させてしまう。

そのような場合には、EPD の複数の書換えモードをうまく使えばよい。最初の書換え命令は、すぐに高速モードで書換え処理をおこないユーザを待たせないようにする。そして、データが揃った時点で通常モードで 1 回書換えるようなスケジューリングをすれば途中の書換え処理を削減でき、低消費電力化も同時達成できる。

また、アプリケーションに依っては、EPD スケジューラで待たせず、すぐ書換え処理を開始しないといけない場合もある。その最たるものが、ドキュメントワークでは重要となる手書き処理である[82]。例えば、PDF ファイルのページをめくりながら、途中で手書きのメモを残していくユースケースを想定する。この場合、書換え処理を少しでも待たせてしまうと、ペンの動きに書換え処理が追従せず、使い勝手が大幅に低下してしまう。そこで、例えば、書換え領域のサイズに着目する。手書きアプリケーションでは、応答性を向上させるために、手書きされた部分領域の書換え命令を即座に発行するので、書換え領域のサイズが非常に小さいと考えられる。そのため、領域が小さい書換え命令は待たせないようにすれば、手書きの応答性を低下させずに済む。同様に、待ち時間の自動調整の際にも無視すれば、適切な待ち時間の検出に影響しないようにできる。

5.5　EPD スケジューラの性能評価

EPD スケジューラを、EPD コントローラのデバイスドライバに実装し、性能評価をおこなった。

5.5.1　評価環境

評価は、高性能の 800MHz ARM Cortex-A8 と EPD コントローラを内蔵した SoC である i.MX508 アプリケーションプロセッサと 9.7 インチの E Ink EPD 搭載端末評価ボード上でおこなった。主記憶については、評価に利用可能なストレージクラスメモリがまだないため、LPDDR2 を利用した。OS は、バージョン 2.3.3 の Android™ OS を利用した。

5.5.2　EPD スケジューラの省電力効果の評価

EPD スケジューラの待ち時間をアプリケーションに合わせて設定した場合の低消費電力化効果を検証する。

まず、EPD に不向きな LCD を前提とした書換え処理の例として、*Android™ OS のエフェクトを用いて評価した。

図 5.9 に、*Android™ OS の Home 画面から Menu 画面へ遷移する際の書換え処理のタイミングの例を示す。図では、書換え命令がデバイスドライバに到着するタイミングと、 EPD コントローラで並列に実行される書換え処理（図の矢印）の様子を示している。前述のとおり、エフェクトなどは、LCD で操作感を向上させるためのもので、最終的な表示までじわじわと同じ領域を変化させる。そのため、コリジョンが多発し、それに対応する再書換え処理が発生して EPD の書換え処理時間が長くなってしまっている。この結果から、EPD スケジューラで最後の書換え命令の到着まで掛かった時間だけ待てば、コリジョンを回避できることが分かる。

図 5.10 に、EPD スケジューラを適用した場合を示す。図より、EPD スケジューラで書換え処理を待たせて、1 回の書換え処理に再構成することでコリジョンを回避していることが分かる。その結果、書換え処理時間が 68%削減されている。また、全体の処理時間(最初の書換え命令が到着してから最後の書換え処理が終了するまでの時間)で見ても 28%削減されている。全体の処理時間を削減できている一方で、この例では、ユーザを 1200ms 待たせているのでトレードオフがあり、4.2 節で説明した書換えモードを使い分ける方法で応答性を高めてもよい。EPD 搭載端末が何を重視するべきかは、アプリケーションや電力制約に応じて十分検討されるべきである。

また、前処理や書換え処理を書換え命令が到着したらすぐおこなうことをやめたことで、

アプリケーションが書換え命令を発行しきるまでの時間も 23%ほど短くなっており、これも省電力化に効いていると考えられる。

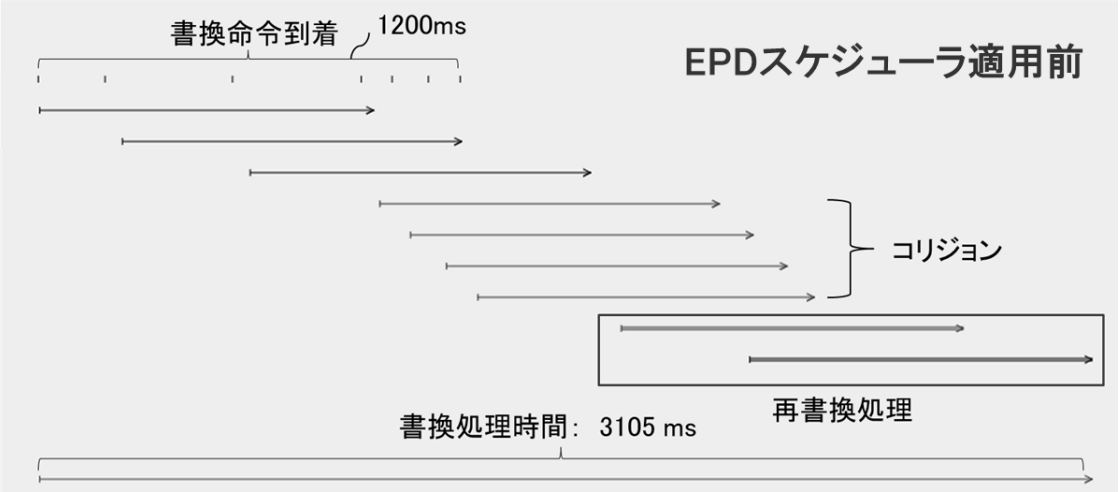

図 5.9.　書換え処理タイミング（LCD 向けエフェクト）

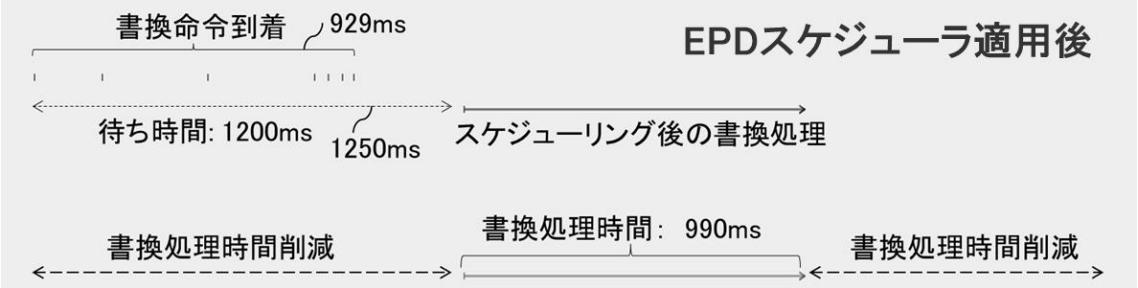

図 5.10.　EPD スケジューラ適用(LCD 向けエフェクト)

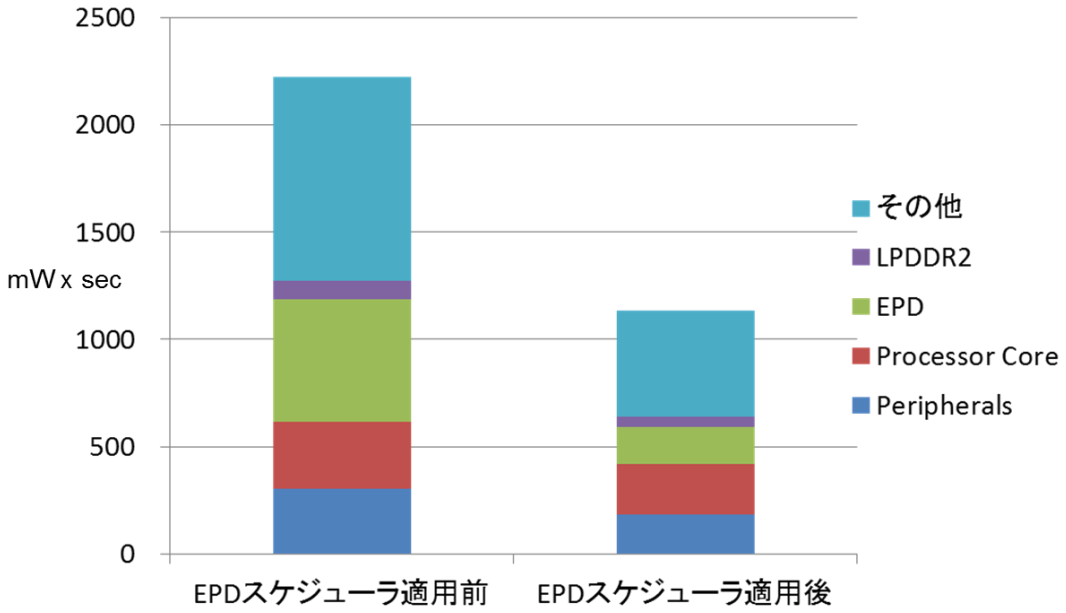

図 5.11.　消費電力量削減効果(LCD 向けエフェクト)

図 5.11 に、消費電力量削減効果を示す。消費電力量の内訳は、LPDDR2、EPD、ARM コア、EPD コントローラを含むペリフェラル、その他となっている。EPD スケジューラによるコリジョン除去により、EPD 搭載端末全体で 49%の低消費電力化が達成できることが分かった。また、書換え処理時間を短縮したことで、消費電力量が大きかった EPD と EPD コントローラが含まれているペリフェラルの消費電力量の削減が顕著であることが分かる。特に、EPD は 70%削減できている。

つぎに、PDF リーダを用いて、省電力効果を評価した。図 5.12 に、PDF リーダでページめくりをした際の書換え処理タイミングの例を示す。評価に用いた PDF リーダでは、書換え命令数も少なく、コリジョンを引き起こす書換え処理もないので、最適化の余地は LCD 向けエフェクトの場合ほど大きくはない。

図 5.13 に、EPD スケジューラを適用した場合を示す。EPD スケジューラで書換え処理を待たせて、1 回の書換え処理に再構成している。最初の書換え命令が到着してから、最後の書換え命令の終了時刻までの時間はほとんど変わらないが、その中で書換え処理をしている時間は 13%短くなっている。

図 5.14 に、消費電力量削減効果を示す。EPD スケジューラを利用することで、複数書換え処理を局所化し、書換え処理時間が削減されたので、EPD の消費電力量削減が 13%となり、全体でも 8%の電力削減となった。最適化の余地が少ない例でも、書換え処理の細かい制御で、うまく電力利用を効率化できることを示唆している。

本評価は主記憶に LPDDR2 を用いているためメモリアクセスの消費電力量の割合が小さくなっているが、ストレージクラスメモリを利用するようになるとこの割合は増加すると考えられる。しかし、EPD スケジューラにより書換え時間を削減することによりメモリアクセスの消費電力量の増加も抑制できることが期待できる。

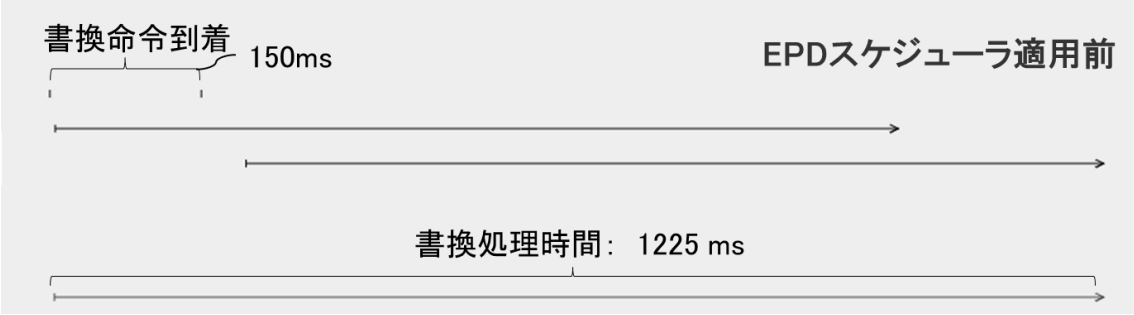

図 5.12.　書換え処理タイミング（PDF リーダ）

図 5.13.　EPD スケジューラ適用（PDF リーダ）

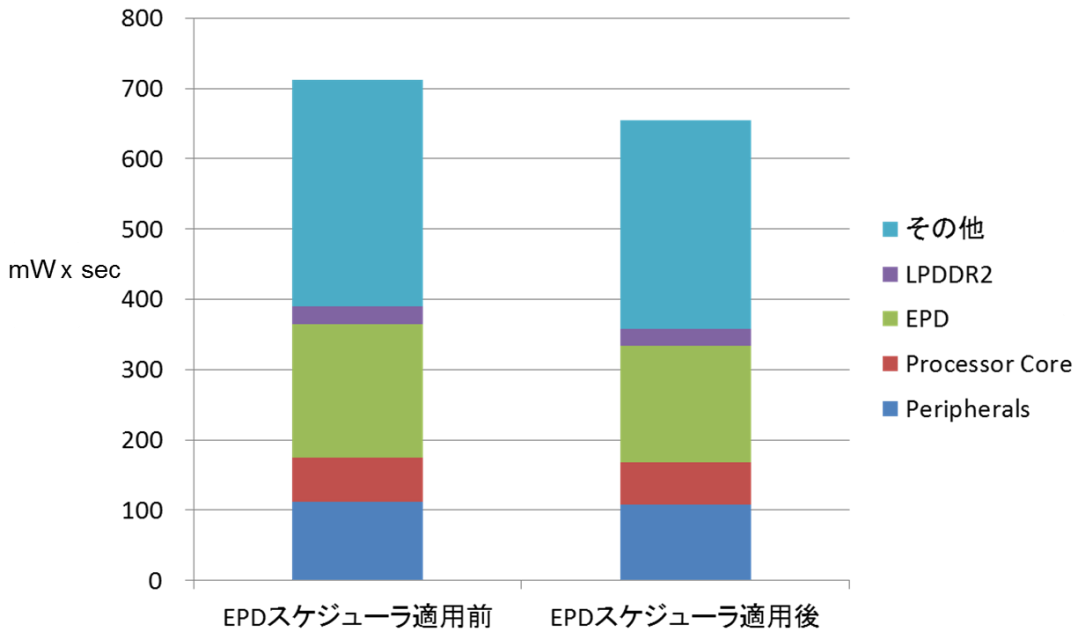

図 5.14.　消費電力量削減効果(PDF リーダ)

5.5.3　待ち時間の自動調整手法の評価

図 5.15 に、5.2 節で使用した PDF リーダでページめくりを繰り返した際の EPD スケジューラの待ち時間の自動調整の効果を示す。

図では、待ち時間の初期値を 150ms に設定している。最初のページめくりを実行すると、2 つの書換え命令が 101ms 間隔で到着するのでこの時は余計に待ってしまう。次のページをめくると、今度は、先程の 101ms にマージンの 20ms を加えた 121ms で待つようになる。この時は、2 つの書換え命令が 104ms 間隔で到着するので待ち時間は小さくなり、待ち時間が自動調整できていることが分かる。ページによってレンダリング時間にばらつきがあるので、マージンの利用は重要である。

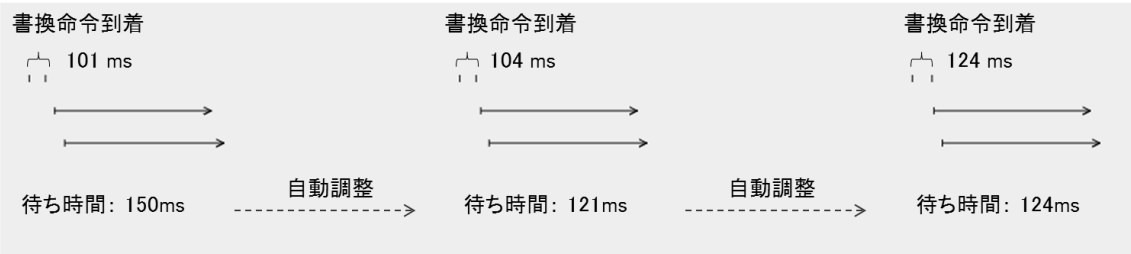

図 5.15.　待ち時間自動調整(PDF リーダ)

5.6　省電力書換え制御方式の他デバイスへの適用可能性

本研究では、ストレージクラスメモリへのメモリアクセス時間を短くするための EPD の省電力書換え制御に関する基本方式を 2 つ提案したが、この基本制御方式は EPD 以外のデバイスでも適用可能である。

まず、第 1 の省電力書換え制御方式である、EPD 書換え処理に伴うメモリアクセスを階層制御して書換え処理時間を削減する方式では、表示データをストレージクラスメモリから内部メモリからコピーし、内部メモリから EPD コントローラが表示をおこない、その間ストレージクラスメモリの電源をオフにすることで不揮発メモリの省電力性を引き出す。この基本制御方式の本質は、ストレージクラスメモリから直接書換え処理をおこなった方が処理は早く完了するところを、内部メモリへコピーするコストを払うがそれと引き換えにストレージクラスメモリの電源をオフできるようにすることで省電力性を引き出すことである。この本質的なアイディアは、他のデバイスでも適用可能であり、さらに、EPD の場合はプロセッサからの出力処理に適用したが入力処理にも適用可能である。

入力処理の具体例として、イメージセンサに適用する方式が考えられる［122］。イメージセンサからの入力画像を通常はストレージクラスメモリにコピーするところを内部メモリへコピーして内部メモリ上で NEON などの SIMD エンジンを使って画像処理することで不揮発メモリへのメモリアクセスを減らすことが可能になる。この方式は、監視カメラやグラス型のウェアラブル端末において、異常やイベントを検出する段階に適用可能である。この段階では、まずは状況に変化があることだけを検出できればよいのでプロセッサの内部メモリに入りきるような解像度の低い画像や画像の変化が起こり得る部分画像を定期的にイメージセンサから入力／検出処理する場合などで広く利用可能である。

他にはネットワーク送受信処理やアクセラレータを利用した処理にも適用できると考えられる［124］。

つぎに、第 2 の省電力書換え制御方式である、複数の書換え処理をまとめて書換え処理時間を削減する方式では、先行する書換え命令を EPD コントローラにすぐに実行させずに複数命令が揃うのを待ち書換え開始タイミングを揃えて一斉に実行することで書換え処理時間を削減する。この基本制御方式の本質は、すぐに処理を開始した方が高速に完了するところを、複数処理のタイミングを揃えて処理時間を短くする。この本質的なアイディアは他のデバイスでも適用可能である。

先の監視カメラやグラス型のウェアラブル端末において異常やイベントを検出するイメージセンサの例では、定期的なイメージセンサから入力／検出処理のタイミングに他の処理を揃えることでプロセッサやストレージクラスメモリの利用時間を短縮することが可能になる［122］。

また、第 2 の省電力書換え制御方式は、EPD 以外の省電力ディスプレイにも適用可能である。昨今、本研究で述べたような省電力ディスプレイを利用してアイドル時の消費電力を下げて低消費電力化するシステムに、将来的に活用できそうなディスプレイが EPD 以外にも続々開発されている。IGZO などの低リフレッシュレートディスプレイでは、静止画表示のために必要な電力は劇的に小さくなっているものの、EPD 搭載端末で想定しているような厳しい電力バジェットで駆動させることを考え始めると、書換え時の消費電力を徹底的に削減する必要がある。その際に、EPD スケジューラで行っているような最適化技術が、幅広く適用できる可能性がある。具体的には、短い間隔で全面を書換える命令が発行された場合、最後の 1 回だけ書換えるように制御すればよい。LCD のように書換え時間が短いディスプレイでは、仮に 3 つの書換え処理をまとめることができれば、書換え処理に掛かる消費電力を約 1/3 にできる可能性がある。このように、本手法は、書換え時間が短くなるとさらに効果が大きくなることが期待できるため、EPD 以外の省電力ディスプレイ向けの低消費電力制御技術の確立も目指していく。

5.7 まとめと今後の課題

ストレージクラスメモリや不揮発性のディスプレイである EPD を採用してアイドル期間の消費電力を削減するシステムにおいては、アクティブ時の低消費電力化も重要な課題となる。EPD 搭載端末のアクティブ時の消費電力の大部分を占める EPD の書換え処理の低消費電力化をおこなった。

本研究では、EPD の書換え時間の長さやコリジョンなどの書換え処理時間が長くなる要因に着目し、これらを回避できるように書換え命令をスケジューリングすることで書換え処理時間を削減して低消費電力化する EPD スケジューラを提案した。EPD スケジューラを実装し、EPD 搭載端末で想定されるワークロードを用いて有効性の評価をおこなった結果、低消費電力化できることが確認できた。まだストレージクラスメモリが利用できないため、評価は DRAM ベースでおこなったが、ストレージクラスメモリは動的消費電力が大きいため、ストレージクラスメモリの場合はメモリ部分の消費電力削減の効果は大きくなると考えられる。

本研究では、EPD コントローラのデバイスドライバに EPD スケジューラを実装し、デバイスドライバ単体でできる最適化をおこなった。今後は、これに加えて、アプリケーションに少しだけ手を加えてヒント情報を様々な形式で通知してもらえるようにする。ヒント情報を利用することで、EPD スケジューラの待ち時間をより適切に調整できるようにする。また、待ち時間調整の自動化の適用範囲拡大など EPD スケジューラの機能拡張等をおこない、アクティブ時のさらなる低消費電力化を目指す。

また、5.6 節で示したように、本研究で提案した、ストレージクラスメモリへのメモリアクセス時間を短くするための EPD の省電力書換え制御に関する基本方式は電子ペーパ以外のデバイスでも適用可能である。今後は、EPD 以外のデバイスとストレージクラスメモリを組み合わせた低消費電力制御技術の確立も目指していく。

6. 結論

6.1 研究成果の概要

本論文では、メモリ階層制御により高性能化・低消費電力化を実現するプログラムの開発生産性向上をシステムソフトウェアにより実現するという主題に対して、自動階層制御技術を組み込んだ 4 つのシステムソフトウェアをメモリ階層の様々なポイントに導入することでプログラム開発の容易性を向上させたうえでハードウェアの持つ性能を引き出すことが可能になることを、高性能クラスタシステムや組込みシステムなどを対象に明らかにした。

第 1 のアプローチである配列処理言語を用いたプログラミングシステムは、高性能サーバなどの単一計算ノードや組込みシステムにおいて、メモリアーキテクチャの深い知識を持たなくても対象メモリアーキテクチャを活かした高性能な並列プログラムを開発可能なプログラミングシステムを実現するという課題に対して研究開発を行った。提案方式により、特定のメモリアーキテクチャに依存しない画像処理や信号処理のプログラムをアルゴリズムレベルで簡潔に記述できることができ、配列処理言語の処理系により対象メモリアーキテクチャに適合した高性能な並列化 C プログラムが自動生成できることを明らかにした。さらに、高位プログラム変換と自動チューニングを組み合わせることで、直観的に記述されたアルゴリズム記述から対象メモリアーキテクチャの性能を引き出す並列 C プログラムを自動生成できることを明らかにした。

第 2 のアプローチである高性能ソフトウェア分散共有メモリシステムは、単一計算ノードを高速ネットワークで複数接続したハイパフォーマンスコンピューティング向けの計算機クラスタシステムをさらに複数繋げたマルチクラスタシステムなどの階層的な分散メモリを持つ高性能コンピュータシステムにおいて、分散メモリ上に仮想的な共有メモリ型プログラミングモデルを提供し、そのうえでメモリ階層に適合可能なデータ一貫性制御方式を実現するという課題に対して研究開発を行った。提案したデータ一貫性管理方式であるマルチホーム方式により、マルチクラスタシステムで課題となるクラスタ間通信遅延に対して、クラスタ毎にホームノードを設けて同ノードをクラスタキャッシュとして利用しクラスタのデータローカリティを利用可能とすることで、マルチクラスタシステムまでソフトウェア分散共有メモリが効率的に適用可能となることを明らかにした。

第 3 のアプローチである省電力仮想記憶システムは、高性能・低消費電力でスケーラブルな主記憶の実現と、大規模インメモリデータ処理のプログラミングをシンプルにすることを実現するという課題に対して研究開発を行った。提案方式により、ストレージクラス

メモリ向けに仮想記憶システムを拡張することで大容量ストレージクラスメモリを DRAM と混載し、待機消費電力が小さいストレージクラスメモリの特性を活かして DRAM 上のデータを積極的にストレージクラスメモリに退避して、使用する DRAM サイズを削減するとともに未使用 DRAM の電源をオフすることで動作時の消費電力を削減できることをフルシステムシミュレーションにより明らかにした。提案方式では仮想記憶システムで DRAM とストレージクラスメモリ間のデータの入れ替え処理であるスワップ処理を効率良くおこなうことで、アプリケーション開発者には単一の大容量高速メモリがあるように見せることができるためプログラミングを非常にシンプルにすることが可能になった。

第 4 のアプローチである不揮発ディスプレイ書換処理省電力スケジューラは、ストレージクラスメモリと不揮発ディスプレイを搭載した組み込みシステムにおいて、従来型のプログラミング方式を変えることなく、ストレージクラスメモリと省電力ディスプレイの省電力性を引き出すことを実現するという課題に対して研究開発を行った。提案方式の省電力メモリ制御機能を組み込んだ電子ペーパコントローラ向けデバイスドライバを利用することで、従来型のプログラミング方式で作成されたプログラムに対してもストレージクラスメモリおよび不揮発ディスプレイの動的消費電力を削減できることを明らかにした。

6.2 今後の課題

本研究は、高性能・省電力コンピューティングに関して、システムソフトウェアによるメモリ階層制御方式を中心課題として、1) メモリ階層の高効率利用を可能にするプログラミング、2) 水平方向のメモリ階層制御、3) 垂直方向のメモリ階層制御、4) 周辺デバイスを考慮したメモリ階層制御の図 6.1 に示す 4 つの技術領域について研究をおこなったものである。

本研究で提案した 4 つのシステムソフトウェアは、上記 4 つ技術領域で解決すべき主要な課題について取り組んだものであり、これらの方式を統合して高性能・省電力コンピューティングシステムの実現を目指していく。

1) メモリ階層の高効率利用を可能にするプログラミング方式は、複雑なメモリ階層制御をユーザから隠蔽して生産性を高めるためには必須である。特に今後ストレージクラスメモリが実用化されると、メモリ階層はこれまで以上に複雑になるため、このようなプログラミング方式の重要度は増す。

この技術領域に対しては、高位プログラム変換により対象メモリアーキテクチャに適応した並列プログラムが生成可能な配列処理言語によるプログラミングシステム（図のメモリ階層に適応したプログラム自動生成方式に対応）を提案した。様々なプログラミング方式が存在するなかで、生産性を究極的に高めるためには、問題を解くアルゴリズム記述と

メモリ階層制御の最適化のための記述を分離できるようにして、アプリケーション開発者にはアルゴリズムだけを記述してもらうようにすることが必要である。その際に、本配列処理言語のように、処理系が高位プログラム変換をおこなうために必要な情報をアルゴリズム記述に暗黙的に書かせることで最適化を実現することが重要である。

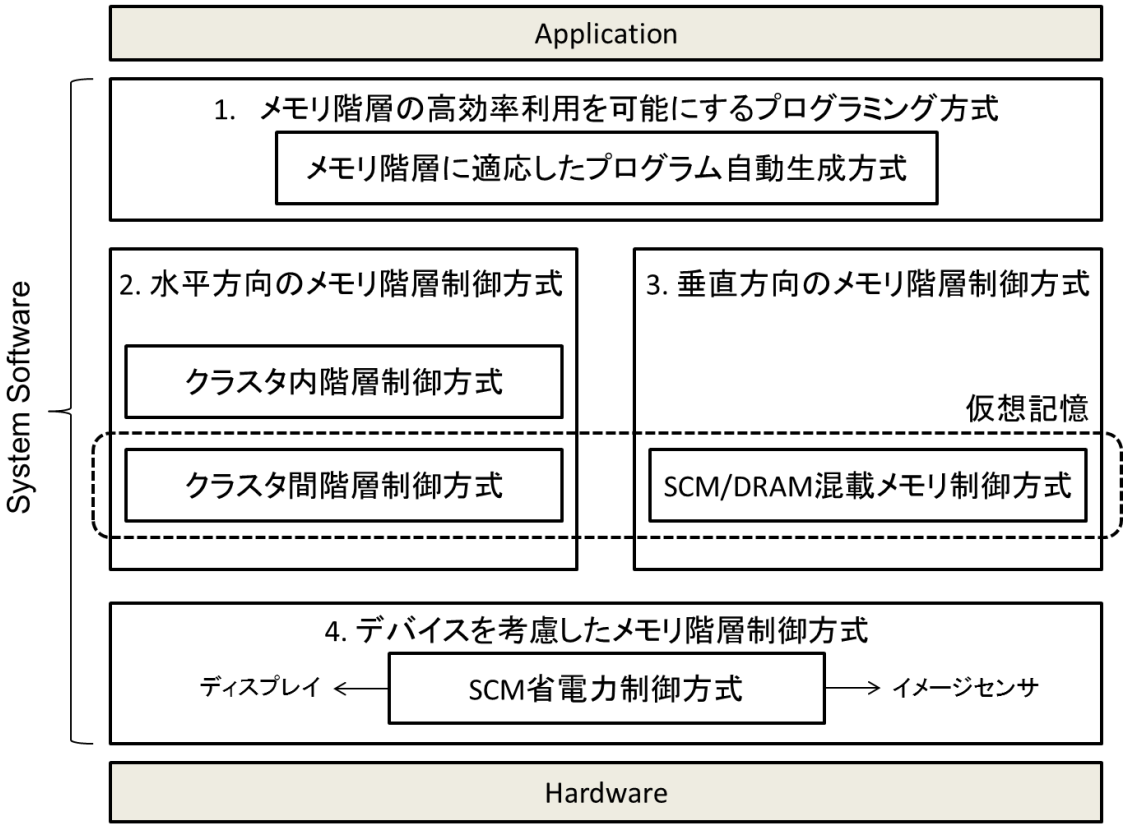

図 6.1.　システムソフトウェアによるメモリ階層制御

つぎに、一般にメモリ階層は水平方向と垂直方向の 2 つを考える必要があるため、2）水平方向のメモリ階層制御、および、3）垂直方向のメモリ階層制御の技術領域は必須である。

前者の 2）水平方向のメモリ階層制御方式については、階層的な分散メモリを持つ計算機クラスタ向けの一貫性管理方式であるマルチホームプロトコルを組み込んだ高性能ソフトウェア分散共有メモリシステム（図のクラスタ間階層制御方式に対応）を提案した。昨今、ストレージクラスメモリなどのメモリとストレージの両側面を持ち合わせた新型の高速不揮発メモリやシリコンフォトニクスなどの高速ネットワークの実用化が近づくなかでメモリセントリックコンピューティング[128, 130]が徐々に注目されつつあり、本研究はこれを

見据えた際に重要になる。メモリセントリックコンピューティングは、まだ研究の初期段階でありそのアーキテクチャの決定版は明らかになっていないが、ラック内のコンピュータアーキテクチャやクラスタ内階層制御方式は大きく変化する。しかし、大規模インメモリデータ処理を実現するためには、複数ラックをさらに接続しメモリやプロセッサを増やす必要があり、ラック間の階層化された分散メモリ上のデータの一貫性管理とラック内のローカリティ活用は最後まで残る本質的な課題であり、本研究はその本質課題に対する解決法や方向性を示したものである。

後者の 3）垂直方向のメモリ階層制御方式については、ストレージクラスメモリを活用した階層型主記憶を実現する高性能・省電力仮想記憶システム（図の SCM/DRAM 混載メモリ制御方式に対応）を提案した。今後のコンピュータシステムでは、種々のストレージクラスメモリの中からどれを選択し、どのように組み合わせて階層制御するかがポイントになる。本研究は、ストレージメモリ活用の大規模データのインメモリデータ処理の要求が高まっている背景をもとに、それを実現するうえで基本となる方式である SCM/DRAM 混載メモリ制御方式に焦点を当てたものである。

最後に、4）周辺デバイスを考慮したメモリ階層制御は、ストレージクラスメモリのメモリアクセスの動的消費電力が高いため今後非常に重要になる。この技術領域に対しては、ストレージクラスメモリ搭載端末の不揮発ディスプレイ書換処理省電力スケジューラ（図のストレージクラスメモリ省電力制御方式に対応）を提案した。

提案した制御方式の本質は、動的消費電力が高いストレージクラスメモリへのアクセス時間を低減しストレージクラスメモリの省電力性を引き出すことで低消費電力化することである。本方式は、周辺デバイスの影響を考慮したストレージクラスメモリ階層制御の基本方式であり、今後は他のデバイスへ適用していくことが重要である。

本論文で提案した 4 つのシステムソフトウェアを有機的につなげていくことが今後の重要な課題である。

まず、1）メモリ階層に適応したプログラム自動生成方式にとって、2）クラスタ間階層制御方式と 3）SCM/DRAM 混載メモリ制御方式は、より複雑なメモリ階層を持つアーキテクチャへのターゲット拡大を容易に実現可能にするものである。具体的には、配列処理言語の処理系で 2）のソフトウェア分散共有メモリ向けのプログラムを生成すればアルゴリズム記述からマルチクラスタ環境を利用可能になり、処理系で 3）を用いたシステムに対してプログラムを生成すればアルゴリズム記述からストレージクラスメモリを活用した大規模主記憶が利用可能になる。このように、2)や 3)と組み合わせることで、処理系の開発コスト削減とターゲットの拡大を同時達成できる。

また、1）メモリ階層に適応したプログラム自動生成方式については、処理系が最適化をおこなうために必要な情報をアルゴリズム記述に暗黙的に書かせることを先に述べたが、この情報をそれぞれのターゲットで必要となる最適化のためのヒント情報として利用できる可能性がある。2）に対しては、例えば、クラスタ間で共有されるページが分かればその

ようなページについてはマルチホーム方式を適用することが有効である可能性がある。3)に対しては、ストレージクラスメモリへ積極的に追い出すページの特定やストレージクラスメモリからプリフェッチするページの特定に利用できる可能性がある。

2) クラスタ間階層制御方式と 3) SCM/DRAM 混載メモリ制御方式は、OS の仮想記憶方式を応用したものである。2) は仮想記憶を水平方向のメモリ階層制御に応用しており、3)は垂直方向のメモリ階層制御に利用している。SCM/DRAM 混載メモリ制御方式については 4 章の評価結果からは、μs オーダのストレージクラスメモリがコンピュータシステム内で利用可能になった時にこれまで隠れていたオーバヘッドが顕在化する課題があることも明らかになっており、ラックスケールでコンピュータアーキテクチャが大きく変化する際に、ストレージクラスメモリの性能を引き出すように仮想記憶を再設計することが今後の最重要課題である。

謝辞

本研究を行うにあたり、多くの方々のご協力とご指導を頂きましたことを感謝致します。

弓場敏嗣先生（電気通信大学名誉教授）ならびに東京工業大学吉瀬謙二准教授には、計算機研究を基本からご指導頂きましたことを心から深謝致します。電気通信大学並列処理学講座在籍時の諸先輩ならびに同窓の皆様に、心から感謝申し上げます。

電気通信大学三輪忍准教授、吉永努教授、大須賀昭彦教授、南泰浩教授には、本論文を纏めるにあたって有益な助言とご指導を賜りましたことを心から感謝申し上げます。

株式会社東芝の木村哲郎氏、吉村礎氏、瀬川淳一氏、白井智氏、樽家昌也氏、松井佑貴夫氏をはじめとする、株式会社東芝の研究開発センターの多くの方々にご協力を頂きました。心から感謝致します。

指導教官の本多弘樹先生に、電気通信大学並列処理学講座在籍時にご指導賜りましたことが、筆者の今日の研究者としての基礎になっております。心から感謝致します。また、思い出深い「学生生活」を与えて頂いたことに、心から感謝致します。

株式会社東芝の金井達徳氏には、株式会社東芝で行った全ての研究で多大なご指導を賜り、社会人大学院生としての研究の機会を与えて頂いたことが学位取得につながったものと思っております。ここに、深く感謝致します。

最後に、すべての面で支えてくれた妻の輝子、ひかり、両親、家族に心から「感謝」致します。

参考文献

[1] R. F. Freitas and W. W. Wilcke, “Storage-class Memory: The Next Storage System Technology,” IBM Journal of Research and Development, Vol. 52, No. 4, pp. 439-447, 2008.

[2] U. Hoelzle and L. Barroso, “The Datacenter as a Computer,” Morgan and Claypool Publishers, 2009.

[3] Y. Park and H. Bahn, “Efficient Management of PCM-based Swap Systems with a Small Page Size,” Journal of Semiconductor Technology and Science, Vol. 15, No. 5, pp. 476-484, 2015.

[4] MARSSx86, http://marss86.org/.

[5] PARSEC, http://parsec.cs.princeton.edu/.

[6] A. Suresh, P. Cicotti, L. Carrington, “Evaluation of Emerging Memory Technologies for HPC, Data Intensive Applications,” In Proceedings of IEEE International Conference on Cluster Computing (CLUSTER), pp. 239-247, 2014.

[7] Y. Shirota, S. Yoshimura, S. Shirai, T. Kanai, “Powering-off DRAM with Aggressive Page-out to Storage-class Memory in Low Power Virtual Memory System,” In Proceedings of IEEE Symposium on Low-Power and High-Speed Chips (COOL Chips XIX), pp. 1-3, 2016.

[8] C. Bienia, S. Kumar, J. P. Singh, K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” Princeton University Technical Report TR-811-08, 2008.

[9] Freescale Semiconductor, Inc. , “i.MX50 Multimedia Applications Processor Reference Manual Rev. 1,” 2011.

[10] D. Zhang, M. Ehsan, M. Ferdman, R. Sion, “DIMMer: A Case for Turning Off DIMMs in Clouds,” In Proceedings of the ACM Symposium on Cloud Computing (SOCC ’14), Article 11, 2014.

[11] D. Callahan, B. L. Chamberlain, H. P. Zima, “The Cascade High Productivity Language,” In Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS’04), pp. 52-60, 2004.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, V. Sarkar, “X10: An Object-oriented Approach to Non-uniform Cluster Computing,” In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA’05), pp. 519-538, 2005.

[13] J. Segawa, Y. Shirota, K. Fujisaki, T. Kimura, T. Kanai, “Aggressive Use of Deep Sleep Mode in Low Power Embedded Systems,” In Proceedings of the IEEE Symposium on Low-Power and High-Speed Chips (COOL Chips XVII), pp. 1-3, 2014.

[14] B. L. Chamberlain, S. Choi, E. C. Lewis, C. Lin, L. Snyder, W. D. Weathersby, “ZPL: A Machine Independent Programming Language for Parallel Computers,” IEEE Transactions Software Engineering, Vol. 26, No. 3, pp. 197-211, 2000.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, P. Hanrahan, “Sequoia: Programming the Memory Hierarchy,” In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC’06), Article 83, 2006.

[16] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, J. L. Hellerstein, “Dynamic Energy-Aware Capacity Provisioning for Cloud Computing Environments,” In Proceedings of the 9th International Conference on Autonomic Computing (ICAC’12), pp. 145-154, 2012.

[17] D. Wu, B. He, X. Tang, J. Xu, M. Guo, “Ramzzz: Rank-aware Dram Power Management with Dynamic Migrations and Demotions,” In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article 32, pp. 1-11, 2012.

[18] H. Huang, K. G. Shin, C. Lefurgy, T. Keller, “Improving Energy Efficiency by Making DRAM Less Randomly Accessed,” In Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED’05), pp. 393-398, 2005.

[19] A. R. Lebeck, X. Fan, H. Zeng, C. Ellis, “Power Aware Page Allocation,” ACM SIGPLAN Notices, Vo. 35, No. 11, pp.105-116, 2000.

[20] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, M. Horowitz, “Towards Energy-proportional Datacenter Memory with Mobile DRAM,” In Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA’12), pp. 37-48, 2012.

[21] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. Katz, “NapSAC: Design and Implementation of a Power-proportional Web Cluster,” ACM SIGCOMM Computer Communication Review, Vol. 41, No. 1, pp.102-108, 2011.

[22] J. Hopper, “Reduce Linux Power Consumption, Part 1: Tuning Results,” https://www.ibm.com/defeloperworks/library/l-cpufreq-1, 2009.

[23] J. Hopper, “Reduce Linux Power Consumption, Part 3: Tuning Results,” https://www.ibm.com/defeloperworks/library/l-cpufreq-3, 2009.

[24] S. Mittal, “A Survey of Architectural Techniques for DRAM Power Management,” International Journal of High Performance Systems Architecture, Vol. 4, No. 2, pp. 110-119, 2012.

[25] L. A. Barroso, “The Price of Performance,” Queue - Multiprocessors, Vol. 3, No. 7, pp. 48-53, 2005.

[26] I. Ghani, N. Niknejad, S. R. Jeong, “Energy Saving in Green Cloud Computing Data Centers: A Review,” Journal of Theoretical and Applied Information Technology, Vol. 74, No. 1, pp. 16-30, 2015.

[27] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, M. Oskin, “Latency-tolerant Software Distributed Shared Memory,” In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC ’15), pp. 291-305, 2015.

[28] T. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman, “Project Adam: Building an Efficient and Scalable Deep Learning Training System,” In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI’14), pp. 571-582, 2014.

[29] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, E. P. Xing, “GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server,” In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys’16), pp. 1-16, 2016.

[30] 山本和典, 石川裕, “テラスケールコンピューティングのための遠隔スワップシステム Teramem,” 情報処理学会論文誌コンピューティングシステム (ACS), Vol. 2, No. 3, pp. 142-152, 2009.

[31] S. Liang, R. Noronha, D. K. Panda, ″Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device,″ In Proceedings of 2005 IEEE International Conference on Cluster Computing, pp. 1-10, 2005.

[32] T. Newhall, S. Finney, K. Ganchev, M. Spiegel, “Nswap: A Network Swapping Module for Linux Clusters,” In Proceedings of Euro-Par 2003 Parallel Processing, pp. 1160-1169.

[33] 後藤正徳, 佐藤充, 中島耕太, 久門耕一, “10Gb Ethernet 上の RDMA を用いた遠隔スワップメモリの実装,” 電子情報通信学会技術研究報告(CPSY), pp. 7-12, 2006.

[34] Myri-10G Overview. http://www.myri.com/Myri-10G/overview/.

[35] Infiniband trade association. http://www.infinibandta.org/.

[36] S. Pakin and G. Johnson, “Performance Analysis of a User-level Memory Server,” In Proceedings of IEEE International Conference on Cluster Computing, pp. 249-258, 2007.

[37] M. D. Flouris and E. P. Markatos, “The Network RamDisk: Using Remote Memory on Heterogeneous NOWs,” Cluster Computing, Vol. 2, No. 4, pp. 281-293, 1999.

[38] 城田祐介, 前田誠司, “性能安定化を実現するデータプレロードに関する一考察,” 情報処理学会研究報告ハイパフォーマンスコンピューティング(HPC), SWoPP, pp. 265-270, 2004.

[39] M. Saxena and M. M. Swift, “FlashVM: Virtual Memory Management on Flash,” In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10), pp. 14-14.

[40] 岩渕圭太, 佐藤仁, 安井雄一郎, 藤澤克樹, 松岡聡,“不揮発メモリを用いた Graph500 ベンチマークの大規模実行に向けた予備評価,” 先進的計算基盤システムシンポジウム(SACSIS)論文集, pp. 130-131, 2013.

[41] K. Liu, X. Zhang, K. Davis, S. Jiang, “Synergistic Coupling of SSD and Hard Disk for QoS-aware Virtual Memory,” In Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 24-33, 2013.

[42] OpenNVM: http://opennvmgithub.io.

[43] N. Talagala, "Creating Flash-Aware Applications", Flash Memory Summit 2013, http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130814_203B_Talagala.pdf, 2013.

[44] A. Badam and V. S. Pai, “SSDAlloc: hybrid SSD/RAM memory management made easy,” In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11), pp. 211-224, 2011.

[45] D. Jung, J. Kim, S. Park, J. Kang, J. Lee, “FASS: A Flash-Aware Swap System,” In Proceedings of International Workshop on Software Support for Portable Storage (IWSSPS), 2005.

[46] H. M. Levy, P. H. Lipman, “Virtual Memory Management in the VAX/VMS Operating System,” Computer, Vol. 15, No. 3, pp. 35-41, 1982.

[47] 城田祐介, 吉瀬謙二, 本多弘樹, 弓場敏嗣, “ホームベースソフトウェア分散共有メモリ上での Migratory Access を効率良く処理する権限委譲プロトコル,” 情報処理学会論文誌 ハイパフォーマンスコンピューティングシステム, Vol. 44, No. 6, pp. 103-113, 2003.

[48] K. Zhong, X. Zhu, T. Wang, D. Zhang, X. Luo, D. Liu, W. Liu, E. H.-M. Sha, “DR. Swap: Energy-efficient Paging for Smartphones,” In Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED '14), pp. 81-86.

[49] H. Kawata and S. Oikawa, ″Experimental Design of High Performance Non Volatile Main Memory Swapping using DRAM,″ In Proceedings of IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1-6, 2015.

[50] K. Li, “IVY: A Shared Virtual Memory System for Parallel Computing,” In Proceedings of the 1988 International Conference on Parallel Processing, pp. 94-101, 1988.

[51] 森眞一郎, 富田眞治, ″計算機クラスタ: 並列計算機アーキテクトからみた計算機クラスタ,″ 情報処理, Vol. 39, No. 11, 1998.

[52] 平木敬, 丹羽純平, 松本尚, “計算機クラスタ:分散共有メモリに基づく計算機クラスタ,” 情報処理, Vol. 39, No. 11, 1998.

[53] W. Hu, W. Shi, Z. Tang, “JIAJIA: A Software DSM System based on a New Cache Coherence Protocol,” In Proceedings of the 7th International Conference on High-Performance Computing and Networking, pp. 463-472, 1999.

[54] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, Y. Ishikawa, “SCASH: Software DSM using High Performance Network on Commodity Hardware and Software,” In Proceedings of Eighth Workshop on Scalable Shared-memory Multiprocessors, pp. 26-27, 1999.

[55] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, “TreadMarks: Shared Memory Computing on Networks of Workstations,” Computer, Vol. 29, No. 2, pp. 18-28, 1996.

[56] P. Kelher, ″The Relative Importance of Concurrent Writers and Weak Consistency Models,” Univ. of Maryland Institute for Advanced Computer Studies Report, 1995.

[57] Y. Zhou, L. Iftode, K. Li, “Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,” In Proceedings of the

Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96). pp. 75-88, 1996.

[58] L. Iftode, J. P. Singh, K. Li, "Understanding Application Performance on Shared Virtual Memory Systems," In Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96). pp. 122-133, 1996.

[59] A. L. Cox, E. Lara, Y. C. Hu, W. Zwaenepoel, "A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory," In Proceedings of 5th International Symposium on High-Performance Computer Architecture, pp. 279-283, 1999.

[60] H. Hirayama, H. Honda, T. Yuba, "Scalable Data Mining with Log Based Consistency DSM for High Performance Distributed Computing," In Proceedings of Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000), pp. 143-150, 2000.

[61] L. Iftode, J. P. Singh, K. Li, "Scope Consistency: a Bridge between Release Consistency and Entry Consistency," In Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '96). pp. 277-287.

[62] B. W.-L. Cheung, C.-L. Wang, K. Hwang, "A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations," In Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 821-827, 1999.

[63] C. Amza, A. L. Cox, S. Dwarkadas, L. Jin, K. Rajamani, W. Zwaenepoel, "Adaptive Protocols for Software Distributed Shared Memory," In Proceedings of the IEEE, Vol. 87, No. 3, pp. 467-475, 1999.

[64] 丹羽純平, "広域ソフトウェア分散共有メモリ機構を支援する最適化手法," 情報処理学会論文誌コンピューティングシステム (ACS), Vol. 45, No. SIG11(ACS7), pp. 36-49, 2004.

[65] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, J. B. Carter, "Architecting for Power Management: The IBM POWER7 approach," In Proceedings of The International Symposium on High-Performance Computer Architecture (HPCA), pp. 1-11, 2010.

[66] J. K. Bennett, J. B. Carter, W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence," In Proceedings of the second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP '90), pp. 168-176.

[67] A. Dragojevi, D. Narayanan, M. Castro, O. Hodson, “FaRM: Fast Remote Memory,” Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pp. 401-414, 2014.

[68] C. Mitchell, Y. Geng, J. Li, “Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store,” In Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13). pp.103-114.

[69] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng, “Large Scale Distributed Deep Networks,” In Proceedings of Advances in Neural Information Processing 25 (NIPS 2012), 2012.

[70] Y. Shirota, S. Yoshimura, T. Kanai, “Electronic Paper Display Update Scheduler for Extremely Low Power Non-volatile Embedded Systems,” In Proceedings of IEEE Symposium on Low-Power and High-Speed Chips (COOL CHIPS XVIII), pp. 1-3, 2015.

[71] A. K. Bhowmik and R. J. Brennan, “System-Level Display Power Reduction Technologies for Portable Computing and Communications Devices,” In Proceedings of IEEE International Conference on Portable Information Devices, 2007.

[72] K. Han, A. Min, N. Jeganathan, P. Diefenbaugh, “A Hybrid Display Frame Buffer Architecture for Energy Efficient Display Subsystems,” In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), pp. 347-352, 2013.

[73] 松尾拓哉, “IGZO 技術,” シャープ技報, 第 104 号, 2012.

[74] 日本画像学会, “電子ペーパ,” 東京電機大学出版局, 2008.

[75] E Ink: http://www.eink.com/

[76] W. Cummings, “mirasol® -- Revolutionary Color Display Technology for Diverse Mobile Applications,” In Proceedings of International Symposium on Electronic Paper (ISEP), pp. 12-16, 2012.

[77] J. D. Watts, “Presenting a 10.7”Flexible Colour Electronic Paper Display Fabricated Using a Qualified Manufacturing Process,” In Proceedings of International Symposium on Electronic Paper (ISEP), pp. 17-21, 2012.

[78] Kindle: https://kindle.amazon.com/.

[79] Toq: https://toq.qualcomm.com/.

[80] YOTAPHONE, http://yotaphone.com/.

[81] Freescale Semiconductor, Inc., “i.MX50 Multimedia Applications Processor Reference Manual (Chapter 24 Electrophoretic Display Controller) Rev.1,” 2011.

[82] S. Nebashi: “High Resolution E-Paper System Platform,” In Proceedings of International Symposium on Electronic Paper (ISEP), pp. 27-30, 2012.

[83] Y. Shirota, T. Kanai, T. Kimura, H. Toyama, K. Fujisaki, J. Segawa, M. Tarui, S. Shirai, H. Haruki, A. Shibata “Control device, data processing device, controller, method of controlling thereof and computer-readable medium,” U.S. Patent No. 9304578, 2016.

[84] Bik A. J. C. Bik, ″The Software Vectorization Handbook,″ Intel Press, 2004.

[85] D. Pham, S. Asano, M. Bolliger, M. N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, “The Design and Implementation of a First Generation CELL Processor,” Proceedings of International Solid State Circuits Conference, pp. 134-135, 2005.

[86] X10: http://x10.sourceforge.net/.

[87] Y. Shirota, J. Segawa, Y. Matsui, T. Kanai, “Autovectorization-Friendly Program Transformation in the Array Processing Language,” In Proceedings of Parallel and Distributed Computing and Networks (PDCN 2009), pp. 157-162, 2009.

[88] Numonyx, “Phase Change Memory,” http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[89] 成瀬彰, 住元真司, 久門耕一, “Xeon プロセッサ向け Linpack ベンチマーク最適化手法とその評価,” 情報処理学会論文誌コンピューティングシステム(ACS), Vol. 45, No. 11, pp. 62-70, 2004.

[90] 片桐孝洋, “スパコンプログラミング入門: 並列処理と MPI の学習,” 東京大学出版会, 2013.

[91] 城田祐介, 橘内和也, 松崎秀則, 前田誠司, “Cell プロセッサにおける DMA 転送の融合方式の提案,” 情報処理学会研究報告ハイパフォーマンスコンピューティン (HPC), pp. 293-298, 2006.

[92] W. Weber and A. Gupta, “Analysis of Cache Invalidation Patterns in Multiprocessors,” In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), pp. 243-256, 1989.

[93] D. Callahan, B. L. Chamberlain, H. P. Zima, "The Cascade High Productivity Language," In Proceedings of 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), 2004.

[94] J. Bilmes, K. Asanovic, C. W. Chin, J. Demmel. "Optimizing Matrix Multiply using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology," In Proceedings of International Conference on Supercomputing, pp. 340-347, 1997.

[95] R. C. Whaley, A. Petitet, J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," Parallel Computing, Vol. 27, No. 1-2, pp. 3-35, 2001.

[96] M. Frigo, "A Fast Fourier Transform Compiler," In Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 169-180, 1999.

[97] M. Puschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, "SPIRAL: Code generation for DSP transforms," In Proceedings of the IEEE Special Issue on Program Generation, Optimization, and Adaptation, Vol. 93, No. 2, pp. 232-275, 2005.

[98] T. Katagiri, K. Kise, H. Honda, T.Yuba, "ABCLibScript: A Directive to Support Specification of an Auto-tuning Facility for Numerical Software," Parallel Computing, Vol. 32, No. 1, pp. 92-112, 2006.

[99] Coughlin Associates, "Future Memories and Today's Opportunities," http://www.snia.org/sites/default/files/NVM/2016/presentations/JimHandy-TomCoughlin_Future_Memories_Today-5.pdf.

[100] XcalableMP: http://www.xcalablemp.org/.

[101] MATLAB: http://www.mathworks.com/products/matlab/.

[102] Octave: http://www.gnu.org/software/octave/.

[103] F. Franchetti, Y. Voronenko, M. Puschel, "A Rewriting System for the Vectorization of Signal Transforms," In Proceedings of High Performance Computing for Computational Science (VECPAR), pp. 363-377, 2006.

[104] R. C. Gonzalez, R. E. Woods, "Digital Image Processing," Prentice-Hall, 2002.

[105] Y. Shirota, J. Segawa, M. Tarui, T. Kanai, "Autotuning in an Array Processing Language using High-level Program Transformations," In Proceedings of International Conference on Computational Science (ICCS 2011), pp. 2126-2135, 2011.

[106] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, D. Zhang, “Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language,” In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11), pp. 224-235, 2011.

[107] M. D. McCool, “Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform,” In Proceedings of GSPx Multicore Applications Conference, 2006.

[108] K. E. Iverson, “A Programming Language,” In Proceedings of Spring Joint Computer Conference, 1962.

[109] Real-Time Workshop Embedded Coder: http://www.mathworks.com/products/rtwembedded/.

[110] 瀬川淳一, 城田祐介, 樽家昌也, 金井達徳, “コード生成容易な MATLAB 上のデータ並列 DSL によるプログラミングシステム,” 情報処理学会論文誌 コンピューティングシステム(ACS), Vol. 4, No. 4, pp.135-145, 2011.

[111] Intel Omni-path Architecture: http://www.intel.co.jp/content/www/jp/ja/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[112] H. Inoue, “How SIMD Width Affects Energy Efficiency: A Case Study on Sorting,” In Proceedings of IEEE International Symposium on Low-Power and High-Speed Chips (COOL Chips XIX), pp. 1-3, 2016.

[113] 城田祐介, 吉川克也, 本多弘樹, 弓場敏嗣, “マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ,” 先進的計算基盤システムシンポジウム(SACSIS)論文集, pp. 315-322, 2003.

[114] 片桐孝洋, “ソフトウェア自動チューニング: 数値計算ソフトウェアへの適用とその可能性,” 慧文社, 2004.

[115] 城田祐介, 吉瀬謙二, 本多弘樹, 弓場敏嗣, “Migratory Access を対象とするホームベース分散共有メモリ,” 並列処理シンポジウム(JSPP)論文集, pp. 119-126, 2002.

[116] 田邊浩志, 本多弘樹, 弓場敏嗣, “ソフトウェア分散共有メモリを用いたマクロデータフロー処理,” 情報処理学会論文誌コンピューティングシステム(ACS), Vol. 46, No. 4, pp. 56-68, 2005.

[117] 佐藤三久, 原田浩, 長谷川篤史, 石川裕, ″Cluster-enabled OpenMP: ソフトウェア分散共有メモリシステム SCASH 上の OpenMP コンパイラ,″ 情報処理学会論文誌ハイパフ

ォーマンスコンピューティングシステム, Vol. 42, No. SIG9 (HPS 3), pp. 158-169, 2001.

[118] L. Arantes, P. Sens, B. Folliot, “The Impact of Caching in a Loosely-coupled Clustered Software DSM System,” In Proceedings of IEEE International Conference on Cluster Computing (Cluster 2000), p. 27-34, 2000.

[119] A. Plaat, H. E. Bal, R. F. H. Hofman, “Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-layer Interconnects,” In Proceedings of Future Generation Computer Systems, Vol. 17, No. 6, pp. 769-782, 1999.

[120] 原田浩, 石川裕, 堀敦史, 手塚宏史, 住元真司, 高橋俊行, “ソフトウェア分散共有メモリSCASHにおけるページ管理ノードの動的再配置機構の実装と評価,” 情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC) , pp. 89-94, 1999.

[121] 田村秀行, “コンピュータ画像処理,”オーム社, 2002.

[122] 城田祐介, 金井達徳, 瀬川淳一, 岐津俊樹, 武田瑛, “情報処理装置および画像入力装置,”特許公開番号 2016-062148, 2014.

[123] S. Qiu and A. L. N. Reddy, “Exploiting superpages in a nonvolatile memory file system,” In Proceedings of IEEE 28th Symposium on Mass Storage System and Technologies(MSST), pp. 1-5, 2012.

[124] 城田祐介, 金井達徳, 木村哲郎, 外山春彦, 藤崎浩一, 瀬川淳一, 樽家昌也, 白井智, 春木洋美, 柴田章博, “制御装置、情報処理装置、制御方法およびプログラム,” 登録番号 5787852 号, 2015.

[125] 城田祐介, “ホームベースソフトウェア分散共有メモリ上で渡り書き込みを効率よく処理する権限委譲プロトコル,” 修士論文, 電気通信大学大学院情報システム学研究科, 2003.

[126] インテルコンパイラ v10 最適化クイック・リファレンス・ガイド: https://jp.xlsoft.com/documents/intel/compiler/qr_guide_jp_10.pdf.

[127] S. Miwa, and H Honda, “Memory Hotplug for Energy Savings of HPC Systems,” International Conference for High Performance Computing, Networking, Storage and Analysis(SC15, poster), 2015.

[128] K. Keeton, “The Machine: An Architecture for Memory-centric Computing,” In Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), p. 1, 2015.

[129] K. Naono, K. Teranishi, J. Cavazos, R. Suda, “Software Automatic Tuning,” Springer, 2010.

[130] Intel Rack Scale Design: http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[131] 中西悠, 渡邊啓正, 平澤将一, 本多弘樹, “コードの性能可搬性を提供する SIMD 向け共通記述方式,”情報処理学会論文誌コンピューティングシステム (ACS), Vol. 48, No. SIG13 (ACS19), pp. 95-105, 2007.

関連論文の印刷公表の方法及び時期

(1) 全著者名： Yusuke Shirota, Junichi Segawa, Yukio Matsui, Tatsunori Kanai

論文題目：「Autovectorization-Friendly Program Transformation in the Array Processing Language」

印刷公表の方法及び時期： In Proceedings of Parallel and Distributed Computing and Networks (PDCN 2009) pp. 157-162, 2009年2月.

(本学位論文の第2章の内容)

(2) 全著者名： Yusuke Shirota, Junichi Segawa, Masaya Tarui, Tatsunori Kanai

論文題目：「Autotuning in an Array Processing Language using High-level Program Transformations」

印刷公表の方法及び時期： In Proceedings of International Conference on Computational Science(ICCS 2011) pp. 2126-2135, 2011年6月.

(本学位論文の第2章の内容)

(3) 全著者名： 瀬川 淳一, 城田 祐介, 樽家 昌也, 金井 達徳

論文題目：「コード生成容易なMATLAB上のデータ並列DSLによるプログラミングシステム」

印刷公表の方法及び時期： 情報処理学会論文誌 コンピューティングシステム(ACS), Vol. 4, No. 4, pp. 135-145, 2011年10月.

(本学位論文の第2章の内容)

(4) 全著者名： 城田 祐介, 吉瀬 謙二, 本多 弘樹, 弓場 敏嗣

論文題目：「ホームベースソフトウェア分散共有メモリ上でのMigratory Accessを効率良く処理する権限委譲プロトコル」

印刷公表の方法及び時期：情報処理学会論文誌 ハイパフォーマンスコンピューティングシステム, Vol. 44, No. 6, pp. 103-113, 2003年1月.

（本学位論文の第3章の内容)

(5) 全著者名： Yusuke Shirota, Shiyo Yoshimura, Satoshi Shirai, Tatsunori Kanai

論文題目：「Powering-off DRAM with Aggressive Page-out to Storage-class Memory in Low Power Virtual Memory System」

印刷公表の方法及び時期：In Proceedings of IEEE Symposium on Low-Power and High-Speed Chips (COOL CHIPS IXI), pp. 1-3, 2016年4月.

(本学位論文の第 4 章の内容)

(6) 全著者名： Yusuke Shirota, Shiyo Yoshimura, Tatsunori Kanai

論文題目：「Electronic Paper Display Update Scheduler for Extremely Low Power Non-volatile Embedded Systems」

印刷公表の方法及び時期：In Proceedings of IEEE Symposium on Low-Power and High-Speed Chips (COOL CHIPS XVIII), pp. 1-3, 2015 年 4 月.

(本学位論文の第 5 章の内容)

査読付き参考論文の印刷公表の方法及び時期

(1) 全著者名： 城田 祐介, 吉川 克也, 本多 弘樹, 弓場 敏嗣

論文題目：「マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ」

印刷公表の方法及び時期： 先進的計算基盤システムシンポジウム(SACSIS)論文集, pp. 315-322, 2003 年 5 月.

(2) 全著者名： 城田 祐介, 吉瀬 謙二, 本多 弘樹, 弓場 敏嗣

論文題目：「Migratory Access を対象とするホームベース分散共有メモリ」

印刷公表の方法及び時期： 並列処理シンポジウム(JSPP)論文集, pp. 119-126, 2002 年 5 月.

登録特許の印刷公表の方法及び時期

（※登録特許のみ掲載。年月は登録年月。< >内は特許の内容）

(1) 全発明者名： 城田 祐介, 金井 達徳, 木村 哲郎, 外山 春彦, 藤崎 浩一, 瀬川 淳一, 樽家 昌也, 白井 智, 春木 洋美, 柴田 章博

発明名称：「制御装置、情報処理装置、制御方法およびプログラム」

時期： 登録番号 5787852 号, 2015 年 8 月.

<不揮発メモリ活用の電子ペーパ省電力制御方式>

(2) 全発明者名： 城田 祐介, 金井 達徳, 木村 哲郎, 外山 春彦, 藤崎 浩一, 瀬川 淳一, 樽家 昌也, 白井 智, 春木 洋美, 柴田 章博

発明名称：「制御装置、制御方法およびプログラム」

時期：登録番号 5755789 号, 2015 年 6 月.

<不揮発メモリ活用の電子ペーパ省電力制御方式>

(3) 全発明者名：城田 祐介, 木村 哲郎, 金井 達徳, 外山 春彦, 藤崎 浩一, 瀬川 淳一, 樽家 昌也, 白井 智, 春木 洋美, 柴田 章博

発明名称：「制御装置および情報処理装置」

時期： 登録番号 5714169 号, 2015 年 3 月.

<キャッシュ省電力化方式>

(4) 全発明者名： 城田 祐介, 木村 哲郎, 金井 達徳, 外山 春彦, 藤崎 浩一, 瀬川 淳一, 樽家 昌也, 白井 智, 春木 洋美, 柴田 章博

発明名称：「制御システム、制御方法およびプログラム」

時期： 登録番号 5674611 号, 2015 年 1 月.

<キャッシュ省電力化方式>

(5) 全発明者名： 城田 祐介, 鳥井 修

発明名称：「ソフトウェア変換プログラム、および、計算機システム」

時期： 登録番号 5017410 号, 2012 年 6 月.

(6)全発明者名： Yusuke Shirota, Tatsunori Kanai, Satoshi Shirai, Tetsuro Kimura,Koichi Fujisaki, Junichi Segawa, Masaya Tarui, Akihiro Shibata, Shiyo Yoshimura, Hiroyoshi Haruki

発明名称：「Control device, display device, control method and program product」

時期： Patent number 9417769, 2016年8月.

〈不揮発メモリ活用の電子ペーパ省電力制御方式〉

(7) 全発明者名： Yusuke Shirota, Tatsunori Kanai, Tetsuro Kimura, Haruhiko Toyama,Koichi Fujisaki, Junichi Segawa, Masaya Tarui, Satoshi Shirai, Hiroyoshi Haruki, Akihiro Shibata

発明名称：「Control device, data processing device, controller, method of controlling thereof and computer-readable medium」

時期： Patent number 9304578, 2016年4月.

〈不揮発メモリ活用の電子ペーパ省電力制御〉

(8) 全発明者名： Yusuke Shirota, Tetsuro Kimura, Tatsunori Kanai, Haruhiko Toyama, Koichi Fujisaki, Junichi Segawa, Masaya Tarui, Satoshi Shirai, Hiroyoshi Haruki, Akihiro Shibata

発明名称：「Control system, control method, and computer program product」

時期： Patent number 9110667, 2015年8月.

〈キャッシュ省電力化方式〉

参考口頭発表論文の印刷公表の方法及び時期

(1) 全著者名： 城田 祐介, 前田 誠司

論文題目：「性能安定化を実現するデータプレロードに関する一考察」

印刷公表の方法及び時期：情報処理学会研究報告ハイパフォーマンスコンピューティング(HPC), SWoPP, pp. 265-270, 2004 年 7 月.

(2) 全著者名： 城田 祐介, 橘内 和也, 松崎 秀則, 前田 誠司

論文題目：「Cell プロセッサにおける DMA 転送の融合方式の提案」

印刷公表の方法及び時期： 情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC) , SWoPP, pp. 293-298, 2006 年 7 月.

(3) 全著者名： 城田 祐介, 瀬川 淳一, 金井 達徳

論文題目： 「配列処理言語における SIMD 化向けプログラム変換」

印刷公表の方法及び時期： 情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC) , SWoPP, pp. 193-198, 2008 年 7 月.

(4) 全著者名： 城田 祐介, 瀬川 淳一, 樽家 昌也, 金井 達徳

論文題目： 「高位プログラム変換を利用した自動チューニングによる並列 C コード生成」

印刷公表の方法及び時期： 日本応用数理学会年度会(JSIAM), pp.31-32 2011 年 9 月.

(5) 全著者名： 城田 祐介, 吉村 礎, 瀬川 淳一, 金井 達徳

論文題目： 「書換処理スケジューラによる電子ペーパ搭載端末の省電力化手法」

印刷公表の方法及び時期： 情報処理学会研究報告組込みシステム (EMB) , ETNET2014, pp. 1-6, 2014 年 3 月.

著者略歴

城田　祐介　（しろた　ゆうすけ）

1997 年 4 月　電気通信大学電気通信学部情報工学科入学

2001 年 3 月　電気通信大学電気通信学部情報工学科卒業

2001 年 4 月　電気通信大学大学院情報システム学研究科情報ネットワーク学専攻博士前期課程入学

2003 年 3 月　電気通信大学大学院情報システム学研究科情報ネットワーク学専攻博士前期課程修了

2003 年 4 月　株式会社東芝入社

2015 年 10 月　現職のまま電気通信大学大学院情報システム学研究科情報システム基盤学専攻博士後期課程入学

2017 年 3 月　電気通信大学大学院情報システム学研究科情報システム基盤学専攻博士後期課程修了予定