マルチコアプロセッサにおける Cプログラムの自動並列化と低消費電力化に関する研究

Studies on Automatic Parallelization and Low Power Optimization of C Programs on Multicore Processors

2011年2月

間 瀬 正 啓

マルチコアプロセッサにおける Cプログラムの自動並列化と低消費電力化に関する研究

Studies on Automatic Parallelization and Low Power Optimization of C Programs on Multicore Processors

2011年2月

早稲田大学大学院基幹理工学研究科
情報理工学専攻
先端プロセッサ構成研究

間 瀬 正 啓

目 次

第1章 序論 **15**

1.1 本研究の目的と研究背景 16

1.2 本論文の構成 23

第2章 OSCAR 自動並列化コンパイラ **27**

2.1 はじめに 28

2.2 マルチグレイン並列処理 30

2.2.1 マクロタスク生成 30

2.2.2 粗粒度タスク並列性抽出 30

2.2.3 データローカライゼーション 32

2.2.4 プロセッサグループへのマクロタスク割り当て 33

2.3 OSCAR コンパイラの構成と OSCAR API 36

2.3.1 OSCAR マルチコアアーキテクチャ 37

2.3.2 OSCAR API 指示文 37

2.3.3 OSCAR API を用いたコンパイルフロー 39

2.4 関連研究 40

2.5 まとめ 41

第3章 ポインタ解析 **43**

3.1 はじめに 44

3.2 基準となるポインタ解析 47
3.2.1 Points-to 解析 47
3.2.2 Flow-Sensitive ポインタ解析 48
3.2.3 Context-Sensitive ポインタ解析 50
3.2.4 Heap-Sensitive ポインタ解析 50
3.2.5 Field-Sensitive ポインタ解析 51
3.3 Element-Sensitive ポインタ解析 51
3.3.1 *element alias* 属性 52
3.3.2 伝達関数 52
3.3.3 交わり (meet) 演算子 $\wedge$ 54
3.3.4 インタープロシージャ解析 55
3.3.5 Cycle-Sensitive 不使用時の Element-Sensitive ポインタ解析の適用例 55
3.4 Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用 56
3.4.1 Element-Sensitive ポインタ解析と Aging の併用例 57
3.5 Element-Sensitive ポインタ解析の自動並列化への利用 59
3.5.1 イタレーション空間における依存解析 60
3.5.2 イタレーション空間における依存解析の例 60
3.6 マルチコアシステム上での性能評価 61
3.6.1 自動並列化を適用する際のコンパイル手順 62
3.6.2 評価対象プログラム 62
3.6.3 評価環境 63
3.6.4 自動並列化による処理性能 63
3.6.5 OSCAR コンパイラによる自動並列化結果 65
3.7 関連研究 67

3.8 まとめ 70

第 4 章 Parallelizable C **71**

4.1 はじめに 72

4.2 Parallelizable C の概念 73

4.2.1 C 言語のコーディングガイドライン 74

4.2.2 対象のコンパイラの解析精度 74

4.3 Parallelizable C のコーディングルール 75

4.3.1 C 言語記述におけるルール 75

4.3.2 外部関数に関するヒント情報 78

4.4 Parallelizable C へのプログラム書き換え 79

4.4.1 書き換え項目とコード変更量 79

4.4.2 コード書き換え内容 80

4.5 OSCAR 自動並列化コンパイラの概要 80

4.6 マルチコアシステム上での性能評価 81

4.6.1 評価方法 82

4.6.2 評価対象プログラム 83

4.6.3 評価環境 83

4.6.4 自動並列化による処理性能 84

4.6.5 OSCAR コンパイラによる自動並列化結果 87

4.7 関連研究 91

4.8 まとめ 94

第 5 章 低消費電力化制御 **95**

5.1 はじめに 96

5.2 マルチグレイン並列処理 97

5.2.1 マクロタスク生成 . . . 97
5.2.2 マクロタスクグラフ生成 . . . 98
5.2.3 マクロタスクスケジューリング . . . 98
5.3 OSCAR コンパイラによる低消費電力化手法 . . . 98
5.3.1 コンパイル時における低消費電力化手法 . . . 99
5.3.2 実行時負荷不均衡に対する低消費電力化手法 . . . 101
5.3.3 デッドライン時刻までの待機電力最小化 . . . 104
5.4 OSCAR API における電力制御用指示文 . . . 106
5.4.1 電力制御およびリアルタイム処理向け指示文 . . . 107
5.5 性能評価 . . . 107
5.5.1 情報家電用マルチコア RP2 . . . 108
5.5.2 消費電力評価条件 . . . 110
5.5.3 最速実行モードの評価 . . . 110
5.5.4 リアルタイム制約モードの評価 . . . 112
5.6 関連研究 . . . 116
5.7 まとめ . . . 118

第 6 章 ソフトウェアコヒーレンシ制御 119

6.1 はじめに . . . 120
6.2 ソフトウェアコヒーレンシ制御 . . . 122
6.2.1 ノンコヒーレントキャッシュアーキテクチャ . . . 122
6.2.2 ソフトウェアによるキャッシュコヒーレントプロトコル . . . 124
6.3 コンパイラによるソフトウェアコヒーレンシ制御手法 . . . 126
6.3.1 階層的粗粒度タスク並列処理 . . . 126
6.3.2 並列処理階層の決定 . . . 126
6.3.3 変数配置のアラインメント . . . 127

6.3.4 フォルスシェアリングの解析 128
6.3.5 フォルスシェアリングの回避 130
6.3.6 キャッシュのセルフインバリデートおよびライトバック操作の挿入 133
6.4 OSCAR API におけるキャッシュ制御用指示文 133
6.4.1 ノンコヒーレントキャッシュ向け指示文 134
6.5 性能評価 135
6.5.1 情報家電用マルチコア RP2 135
6.5.2 評価条件 136
6.5.3 コンパイラの実装 137
6.5.4 並列処理性能 137
6.5.5 ソフトウェアコヒーレンシ制御による影響 139
6.6 関連研究 141
6.7 まとめ 143

第 7 章 結論 **145**

7.1 本研究により得られた成果 146
7.2 今後の課題 148

参考文献 **150**

謝辞 **167**

著者研究業績 **169**

図 目 次

2.1 階層的マクロタスク定義 31
2.2 マクロフローグラフとマクロタスクグラフ 32
2.3 データローカライゼーションにおけるループ整合分割 34
2.4 階層的コード生成イメージ 36
2.5 OSCAR マルチコアアーキテクチャ 38
2.6 OSCAR コンパイラを用いたコンパイルフロー 39

3.1 従来のポインタ解析では識別できないプログラム例 45
3.2 プログラムと Points-to 集合の例 48
3.3 *element alias* 属性の束 (lattice) 52
3.4 gen における *element alias* 属性計算アルゴリズム 54
3.5 Cycle-Sensitive を利用しない Element-Sensitive ポインタ解析の適用例 57
3.6 Element-Sensitive ポインタ解析と Aging の併用例 59
3.7 ポインタ解析を利用したイタレーション空間における依存解析結果 61
3.8 IBM p5 550Q における OSCAR コンパイラによる自動並列化結果 . 65

4.1 lbm におけるループ中でのポインタの更新の除去 81
4.2 hmmer におけるバッファの使い回しの除去 82
4.3 IBM p5 550Q における OSCAR コンパイラによる自動並列化結果 . 86
4.4 Intel Core i7 920 における OSCAR コンパイラによる自動並列化結果 86

4.5 ルネサステクノロジ／日立製作所／早稲田大学 RP2 における OSCAR コンパイラによる自動並列化結果 87
4.6 hmmer における主要処理ループ構造 90
4.7 hmmer における主要処理ループのデータローカライゼーション . . 91

5.1 マクロタスクのスケジューリング結果の例 102
5.2 周波数・電圧制御の結果の例 103
5.3 負荷不均衡のある並列ループに対する低消費電力化手法の適用例 . . 104
5.4 リアルタイム制約モードの適用例 106
5.5 本手法で用いた OSCAR API 107
5.6 RP2 の構成図 108
5.7 最速実行モードにおける実行時間 113
5.8 最速実行モードにおける消費エネルギー 113
5.9 art の電力波形 114
5.10 リアルタイム制約モードでの平均電力 115
5.11 AAC エンコーダにおける電力波形 116
5.12 MPEG2 デコーダにおける電力波形 116

6.1 ノンコヒーレントキャッシュアーキテクチャ 123
6.2 ソフトウェアによるキャッシュコヒーレントプロトコル 125
6.3 階層的粗粒度タスク並列処理のイメージ 127
6.4 分割後の部分ループに関するフォルスシェアリングの検出 129
6.5 配列のパディングによるフォルスシェアリングの回避 131
6.6 キャッシュライン境界に整合したループ分割とノンキャッシャブルバッファを用いた通信の生成 132
6.7 キャッシュのセルフインバリデート及びライトバック操作指示の挿入134

6.8　情報家電用マルチコア RP2 のブロック図 136
6.9　RP2 における並列処理性能 138
6.10 ソフトウェアコヒーレンシ制御における性能への影響要因 141

表目次

3.1 評価環境 63

4.1 Parallelizable C への書き換え項目とコード変更量 79
4.2 評価環境 84

5.1 RP2 の電力状態 109

第1章

序論

1.1　本研究の目的と研究背景

組込み機器から PC，スーパーコンピュータに至るあらゆる情報機器において 1 チップ上に複数のプロセッサコアを集積したマルチコアプロセッサの普及が進んでおり，チップ内に集積するコア数の増加による高性能・低消費電力化が期待されている．半導体集積度の向上は続いているが，動作周波数向上の限界，命令レベル並列性の限界，プロセッサとメモリの速度差の乖離等の要因によりシングルプロセッサ性能の向上は頭打ちとなっている．特に，プロセッサの高周波数動作による発熱や電力消費は大きな問題となっており，消費電力を抑えつつ高性能を実現可能なマルチコアプロセッサの普及が進んでいる．

一方で，マルチコアプロセッサを含む，マルチプロセッサシステムを用いて高性能・低消費電力を実現するためには，並列化を含む高度な最適化技術を駆使したソフトウェア開発が必須となる．この並列ソフトウェア開発では，ループレベルの並列性のみならずプログラム中の複数レベルの並列性を利用した並列処理，キャッシュやローカルメモリを有効活用するためのデータローカリティ最適化，および低消費電力化のための周波数・電圧制御やクロック・電源制御が必要となる．しかしながら，並列処理の高度な専門知識を用いたプログラミングは，一般のアプリケーションプログラム開発者には難易度が高く，長期間の性能チューニングを要する．そのため，並列ソフトウェア開発の生産性向上が求められており，それを全自動で実現可能な自動並列化コンパイラの利用が求められている．

従来よりプログラム全域からの適切な粒度の並列性抽出，プロセッサ近傍の高速キャッシュメモリあるいはローカルメモリの最適利用を実現するための自動並列化コンパイラの研究・開発が行われてきた [Wol96, EHP98, HAA+96]．これらの研究・開発の大部分はプログラム中のループ部分の並列化を対象としたものであり，現在までに様々なループ並列性解析手法やリストラクチャリング手法が開発されている．マルチプロセッサシステムの更なる実効性能向上のためには，従来のルー

プ並列性に加え，ループ間やサブルーチン間といった粗粒度タスク並列性や，基本ブロック内での命令・ステートメント間の近細粒度並列性などの複数レベルの並列性を利用することが必須である．OSCAR コンパイラ [本多 90, KHM+92, 笠原 03] では粗粒度タスク並列処理とループ並列処理および近細粒度並列処理を組み合わせたマルチグレイン並列処理を実現しており，さらにデータローカリティ最適化 [吉田 94, 石坂 02] や低消費電力化 [白子 06] を実現している．

自動並列化に関する研究は FORTRAN 言語の科学技術計算を主な対象として成果を上げてきた．近年のマルチコアプロセッサの普及に伴い，ハイパフォーマンスコンピューティングのみならず組込み分野等でも自動並列化の需要が高まっており，それに伴い組込み分野で標準的に利用されている C 言語を対象とした自動並列化が強く求められている．しかしながら，C 言語でプログラムを記述する際にはポインタを多用した柔軟な記述が可能であり，自動並列化を含めたコンパイラによる最適化が困難となっている．C プログラムに対して自動並列化を適用するには，コンパイラによりポインタ変数がどのメモリ領域を指し示しているかを解析するポインタ解析が必須となる．ポインタ解析の実装においては解析精度と解析効率を両立したポインタ解析手法の実現が問題となっており [Hin01]，自動並列化に有効な解析精度を効率的に得ることができるポインタ解析手法が必要となる．また，ポインタ解析を利用しても C プログラムのあらゆる記述に対する完璧な解析は事実上不可能なため，ユーザプログラマによるプログラム記述方法による対処，すなわち自動並列化コンパイラを有効利用するためのプログラミングモデルが必要となる．そして，近年のプロセッサの低消費電力化への要求から，プロセッサハードウェアが備える周波数・電圧制御やクロック・電源遮断等の低消費電力機能をソフトウェアにより適切に制御することによる低消費電力化が求められている．さらに，ハードウェアによるコヒーレントキャッシュのみならず，ソフトウェア制御のコアローカルメモリ (スクラッチパッドメモリ) や、ソフトウェア

でコヒーレンシを制御するノンコヒーレントキャッシュ等，さまざまなメモリ構成のアーキテクチャへの最適化が求められており，特にソフトウェアによるコヒーレンシ制御が重要な課題となっている．

ポインタ解析に関しては多くの研究がなされており，その精度と効率について多くの議論がなされている [Hin01]．現在では高い精度の解析を効率的に行う手法が確立されつつあり，これらのポインタ解析手法はコンパイラによる定数伝搬や部分冗長性の除去等の一般的な最適化，プログラムのバグ検出ツールへの応用等で大きな成果を上げている．しかしながら，科学技術計算やメディア処理アプリケーションのアルゴリズムは潜在的に高い並列性を持っていながら，従来のポインタ解析技術では並列性の自動抽出にはしばしば不十分なことがある．例えば，アルゴリズム上は多次元配列として扱うことが可能なデータ構造を，ポインタへのポインタとメモリ動的確保を行うループの記述により実装する場合がある．このようなデータ構造に対して従来のポインタ解析の適用を考えた場合，配列の各要素の情報が配列全体で単一の情報に縮退されてしまうため，コンパイラによる依存解析ができず，自動並列化が阻害されてしまう．そこで，本研究では科学技術計算やメディア処理アプリケーション等の自動並列化が特に有効と考えられるアプリケーションプログラムで頻出するプログラム記述に対して特に有効なポインタ解析手法を提案する．Flow-Sensitive[EGH94, WL95, Kah08, HL09], Context-Sensitive[EGH94, WL95, WL04, NKH04, LLA07, Kah08, HL09], Heap-Sensitive[NKH04, LLA07], Field-Sensitive[PKH04, LLA07] ポインタ解析を基準となるポインタ解析精度として採用し，さらにこれらに加えて，従来あまり議論されていなかったヒープを確保するループのイタレーションごとの精度を持たせる Cycle-Sensitivity[RRH08], 配列の要素ごとに精度を持たせる Element-Sensitivity[WFPS02, 間瀬 10] について焦点を当てる．本論文では新たに Element-Sensitive ポインタ解析を提案し，C プログラムで頻出するパターンの自動並列化を実現する．

高い精度のポインタ解析を効率的に行うことでコンパイラによる自動並列化を利用することで，ユーザプログラマの並列ソフトウェア開発の負担を大きく低減できると考えられる．しかしながら，C 言語のあらゆる記述に対してポインタの指し先を静的に特定するのは事実上不可能となっており，自動並列化コンパイラのようなツールを有効活用するためには，用途に応じてコーディングガイドラインにより記述の自由度を制限することが必要となる．MISRA-C[MIS04] は主に車載ソフトウェア向けの高信頼性，高安全性，高移植性を目的としたコーディングガイドラインであり，産業界で広く普及しておりコーディングガイドラインを遵守しているかどうかを確認するチェッカツールも市販されている．マルチプロセッサ向けの自動並列化においては，IMEC によるマルチプロセッサ向けのコーディングガイドラインである Clean C[Cle] が提案されている．また，Hwu ら [HRU+07] により assertion 等によるヒント情報を用いて自動並列化コンパイラを有効利用する暗黙的な並列プログラミングモデルが提案されており，逐次プログラミングのガイドラインと補完的に利用可能と考えられる．しかしながら，このようなガイドラインを利用した場合の有用性については，実際に自動並列化フレームワークを用いた評価は行われていない．そこで，本論文ではコーディングガイドラインとして Parallelizable C を提案するとともに，実際にプログラムの書き換えを行い，様々なマルチコアプロセッサにおいて自動並列化結果の性能評価を行うことにより有用性を示す．

並列処理による高い処理性能のみならず，コンピュータシステムの低消費電力化が求められており，特に組込み機器において低消費電力化への要求が強く，ハイパフォーマンスコンピューティングにおいてもシステム全体の消費電力増加が大きな問題となっている．そのため，処理性能の向上に加え，増加する消費電力をいかに抑えるかが大きな課題となっており，本論文ではコンパイラによる低消費電力化手法の提案と，その実マルチコアにおける評価を行う．その際に，コン

パイラの手法と併せて，コード生成の際に利用可能な並列処理 API を併せて提案することで，様々なマルチコアプロセッサ上での低消費電力化制御を可能とする．低消費電力化に関して，従来から様々な手法が提案されている．例えばキャッシュミス回数測定用カウンタや命令キューなどのハードウェアサポートにより実行時にプログラム中の各フェーズにおける負荷を判断し，不要なリソースを停止する手法 [ABD+03] や，計算資源の各部分に対して実行時の負荷に応じた周波数・電圧制御（FV 制御）を行う手法 [WJMC04] や，ループイタレーションレベルのプロセッサ間の負荷不均衡に対して，実行時のハードウェアサポートにより電力制御を行う Thrifty Barrier[LMH04] などがある．しかしながら，これらの実行時の情報を利用した手法では，プログラム全域にわたるグローバルな消費電力の最適化は難しく，局所的な最適化にとどまってしまう．また，実行時電力制御のための処理やハードウェア追加が必要となるため遅延が大きいという欠点がある．このような手法に対して，コンパイル時の静的な情報に基づく低消費電力化手法では，プログラムの解析による詳細な情報を利用することができるため，よりきめ細やかな電力制御が可能となり，プログラム全域にわたる消費電力の最適化が実現できる．コンパイラ制御によるシングルプロセッサ向けの低消費電力化手法 [HK03] が提案されている．しかしながら，この手法はシングルプロセッサのみを対象としており，マルチコアプロセッサ向けの並列処理と電力制御との両立を実現できていない．マルチコアプロセッサ向けの OSCAR コンパイラではマルチグレイン並列処理による並列性抽出に加え，プログラムの各部分に対するコアごとの適切な動作周波数/電圧および電源遮断のタイミングを決定し，必要な処理性能を維持しつつ電力を低減させている [白子 06]．本研究では，この OSCAR コンパイラによる低消費電力化機能を実際にマルチコアプロセッサ上で適用するための API を策定し，さらに低消費電力化機能を拡張して，負荷不均衡がある場合や，リアルタイム処理においても適用可能とする．

現在主流の 4 から 8 コア程度のマルチコアでは主記憶共有型のマルチプロセッサシステム (SMP) が一般的であるが、そのキャッシュコヒーレンシ制御機構のハードウェアは，プロセッサコア数の増加に伴い，その実装が回路規模的にも消費電力的にも困難となることが知られている [CSG99]．そのため，今後コア数が 32 コアから 64 コア以上のメニーコアとなっていくと，共有メモリ空間とコア毎のコヒーレンシ制御機能を持たないキャッシュ上でも，ソフトウェアによりコヒーレンシ制御を行うことができれば，コスト及び電力消費を抑えつつ効率的な並列処理を実現できると期待されている．特にハードウェアコストや消費電力およびリアルタイム制約に厳しい組込み系マルチコアでは，4 コアの富士通 FR1000[SKK+05]，8 コアの東芝 Venezia[NTF+08] 等において，すでにノンコヒーレントキャッシュアーキテクチャが採用されている．また，ルネサステクノロジ／日立製作所／早稲田大学の RP2[IHY+08] は 8 コア集積しているが，ハードウェアではチップ価格を抑えるためにコヒーレンシ機構は 4 コアまで対応とし，5 コア以上のコア数ではノンコヒーレントキャッシュとして利用する設計となっている．高性能計算分野においても，イリノイ大の Rigel[KJJ+09]，Intel の Single-Chip Cloud Computer (SCC)[HDH+10] は，それぞれのコアがキャッシュを持つが，チップ全体でのコヒーレンシ制御はソフトウェアにより行うことが想定されている．ソフトウェアによりコヒーレンシ制御を行う手法としては OS や Runtime によるアプローチとコンパイラやプログラミングモデルによるアプローチがある [CSG99]．OS や Runtime によりソフトウェアによりコヒーレントな仮想共有メモリ空間を実現する Shared Virtual Memory[LH89] や Treadmarks[ACD+96] が提案されている．これらの手法は一般的なプロセッサハードウェアで実現可能であり SMP 向けのソフトウェアがそのまま動作するが，コヒーレンシ維持のオーバーヘッドが大きい．Rigel[KJJ+09] では Bulk Synchronous モデルを元にしたタスクベースのプログラミングモデルを提案し，そのタスクモデルを前提としたソフトウェアコヒーレンシ制御を提案し

ている [KJL+09]．しかしながら，Rigel ではバリア同期ごとのインターバル内でのコア間通信はサポートしておらず，また false sharing の対処には，ワードごとの dirty bit を持つ特別なハードウェアを想定しているため，この手法を汎用のマルチコアハードウェアでそのまま適用することはできない．本研究では，一般的なキャッシュハードウェアにおいて実現可能な，コンパイラによるソフトウェアコヒーレンシ制御手法を提案し，ハードウェアによるコヒーレンシ制御と同等以上の性能が自動で得られることを示す．その際に，コンパイラの手法と併せて，コード生成の際に利用可能な並列処理 API を併せて提案することで，様々なマルチコアプロセッサ上でソフトウェアコヒーレンシ制御を実現する．

以上のような背景を踏まえて，本論文では以下を研究の目的とする．

- C プログラムの自動並列化のためのポインタ解析の精度向上を提案し，頻出するプログラム記述のパターンについてポインタ解析による所望の解析精度を効率的に実現し，自動並列化による性能向上が得られることを示す．
- C プログラムの自動並列化のためのプログラム記述法を提案し，実際にプログラムの書き換えを行い，自動並列化コンパイラを利用することで，様々なマルチコアプロセッサ上で自動並列化による性能向上が得られることを示す．
- 実行時の負荷不均衡が存在する場合やリアルタイム処理時にも適用可能な，コンパイラによる低消費電力化制御手法を提案し，マルチコアプロセッサ向けの並列処理 API を提案して利用することで，実際にマルチコアプロセッサ上での消費電力が削減されることを示す．
- 一般的なキャッシュハードウェアを用いて自動で高性能が実現できるコンパイラによるソフトウェアコヒーレンシ制御手法を提案し，マルチコアプロセッサ向けの並列処理 API を提案して利用することで，実際にマルチコアプロセッサ上でソフトウェアコヒーレンシ制御により高い並列処理性能が得られ

ることを示す．

1.2　本論文の構成

本論文の構成は以下の通りである．

第2章「OSCAR 自動並列化コンパイラ」では，本研究で開発した OSCAR 自動並列化コンパイラの機能と構成ならびにコンパイラの解析や最適化の処理フローについて述べる．OSCAR コンパイラは逐次 C または Fortran プログラムを入力して並列処理用の指示文を含む並列 C または Fortran プログラムを自動生成する．OSCAR コンパイラは複数レベルの並列性を利用したマルチグレイン自動並列化を行い，プログラムの階層 (関数やループネスト等) ごとに並列性の抽出を行い，それぞれの階層の持つ並列性に応じて，ベーシックブロック，ループ，サブルーチン等の粗粒度タスクをプロセッサにスケジューリングしていく．さらに，プロセッサ近傍のキャッシュやローカルメモリを有効利用するためのデータローカリティ最適化を適用する．並列コード生成には OpenMP の持つ４つの指示文に加えて，電力制御，メモリ配置，データ転送，グループバリア同期，時間管理をサポートする OSCAR API の指示文を利用する．この OSCAR コンパイラにより自動生成された OpenMP および OSCAR API を含む並列プログラムを各マルチコア向けのコード生成コンパイラを用いてさらにコンパイルすることで実行バイナリを生成する．これにより，様々なマルチコアアーキテクチャ上で自動並列化および低消費電力化を実現できる．本論文で提案するプログラムの解析や最適化手法は，この OSCAR コンパイラに実装されている．

第3章「ポインタ解析」では，C プログラムを自動並列化する際に必要となるポインタ解析の解析精度向上について提案する．並列処理が求められるアプリケーションプログラムで頻出するデータ構造として，ポインタのポインタにより実現さ

れる擬似的な多次元配列があるが，従来のポインタ解析では解析できず自動並列化ができなかった．このようなデータ構造を解析して自動並列化を適用するために，ポインタの配列の各要素が指しているメモリ領域のエイリアス関係を解析する Element-Sensitive ポインタ解析を新たに提案する．この Element-Sensitive ポインタ解析は，従来のポインタ解析アルゴリズムの解析情報に 1 ビットのデータ構造を追加するだけの簡易な拡張であり，軽量な実装により頻出のデータ構造を用いたプログラムを自動並列化するのに必要な解析精度を得ることができる．従来のポインタ解析では自動並列化が不可能であった SPEC2000 equake および第 4 章で述べる Parallelizable C への修正を行った SPEC2006 hmmer について，Element-Sensitive ポインタ解析を利用することで自動並列化が可能となった．自動並列化結果について 8 コア搭載の IBM p5 550Q サーバにおいて性能評価を行ったところ，8 コア使用時に逐次実行時と比べてそれぞれ equake で 4.96 倍および hmmer で 6.06 倍の性能向上が得られた．

第 4 章「Parallelizable C」では，コンパイラによる自動並列化が可能なプログラム記述法として Parallelizable C を提案する．コンパイラの解析精度，特にポインタ解析精度を考慮して，ポインタ演算やポインタのキャストを制限するようなプログラム記述のガイドラインを設定することで，C プログラム自動並列化支援の枠組みを構築する．実際にこのガイドラインに沿ってプログラムの書き換えを行ったところ，市販コンパイラでは全く並列化できなかった SPEC2000 art, equake は書き換えが不要であり，SPEC2006 lbm で 1.8%，hmmer で 0.03%の書き換えにより自動並列化可能となった．Parallelizable C で記述した 6 つの逐次プログラムに対して OSCAR コンパイラによる自動並列化を適用し，マルチコアシステム上での処理性能の評価を行った．その結果，逐次実行時と比較して，2 コア集積のマルチコアである IBM Power5+ を 4 基搭載した 8 コア構成のサーバである IBM p5 550Q において平均 5.54 倍，4 コア集積のマルチコアである Intel Core i7 920 プ

ロセッサを搭載したPCにおいて平均2.43倍，SH-4Aコアベースの情報家電用マルチコアRP2の4コアを使用したSMP実行モードにおいて平均2.78倍の性能向上が得られた．

第5章「低消費電力化制御」では，現在のプロセッサで問題となっている消費電力削減のために，最速実行時とリアルタイム実行時におけるコンパイラによる低消費電力化手法を提案する．併せて，各社のマルチコア上で低消費電力化を利用するためのOSCAR APIの提案を行い，自動並列化コンパイラで逐次プログラムからOSCAR APIで記述された並列プログラムを自動生成することで，各社マルチコア上でコンパイラによる自動並列化と低消費電力化が利用可能となった．NEDO ”リアルタイム情報家電用マルチコア ”プロジェクトにおいて，新たに開発した8コア集積の情報家電用マルチコアRP2において，マルチコアにおけるコンパイラによる消費電力制御を世界で初めて実現し，4コアを用いた場合の最速実行モードにおいて消費エネルギーがSPEC2000のartで13.06%，equakeで3.99%，メディア処理のAACエンコーダで3.84%，MPEG2デコーダで9.01%削減された．また，8コアを用いた場合のリアルタイム制約モードにおいて，平均電力がAACエンコーダで87.9%，MPEG2デコーダで76.0%削減された．

第6章「ソフトウェアコヒーレンシ制御」では，今後のメニーコアプロセッサにおける重要技術となる，コンパイラによるソフトウェアコヒーレンシ制御手法を提案する．コンパイラによる自動並列化と併せて，OSCAR APIを用いてキャッシュライトバック，キャッシュセルフインバリデート、および変数配置のアラインメントを指示することで，ソフトウェアによる自動コヒーレンシ制御を実現した．科学技術計算およびメディア処理の4つのアプリケーションプログラムを用いて，上述のRP2において4コア使用時の自動並列化性能の評価を行ったところ，ハードウェアコヒーレンシ制御で1コアの逐次実行と比較して平均3.03倍，提案するソフトウェアコヒーレンシ制御においても同様の平均3.13倍の性能向上が得られ

た．さらに，ハードウェアではコヒーレンシ制御できない 8 コア使用時に平均 4.88 倍の性能向上が得られた．

最後に，第 7 章「結論」では，本研究により得られた成果と今後の課題について述べる．

第2章

OSCAR自動並列化コンパイラ

2.1 はじめに

マルチコアプロセッサを含む，マルチプロセッサシステムを用いて高性能・低消費電力を実現するためには，ループレベルの並列性のみならずプログラム中の複数レベルの並列性を利用した並列処理，キャッシュやローカルメモリを有効活用するためのデータローカリティ最適化，および低消費電力化のための周波数・電圧制御やクロック・電源制御が必要となり，並列化を含む高度な最適化技術を駆使したソフトウェア開発が必須となる．しかしながら，並列処理の高度な専門知識を用いたプログラミングは，一般のアプリケーションプログラム開発者には難易度が高く，長期間の並列チューニングを要する．

並列プログラミングを行う場合には，共有メモリ向けのpthreadやOpenMP，分散メモリ向けのMPI等のように，ユーザプログラマが明示的に並列処理箇所やデータの割り当て，プロセッサ間の通信を記述するものが主流である．このような明示的な並列処理の指定はデッドロックやレースコンディション等の並列バグを誘発しやすく，正しい並列プログラムを記述するのを困難にしている．さらに，並列処理による性能向上を得るためには，並列処理オーバーヘッドやデータローカリティを考慮する必要があり，一般のアプリケーションプログラマには難易度が高いものとなっている．また，所望の性能を得るために，対象のコア数やアーキテクチャに特化した記述が必要となることが多く，プログラムの異なるアーキテクチャ間でのポータビリティが阻害されてしまっている．そのため，シングルプロセッサ用のプログラムから，コンパイラが自動的に並列性を抽出して，マルチプロセッサ上での効率的な並列処理を可能とする自動並列化コンパイラの利用が強く期待されている．

従来より，プログラム全域からの適切な粒度の並列性抽出，プロセッサ近傍の高速キャッシュメモリあるいはローカルメモリの最適利用を実現するための自動並列化コンパイラの研究・開発が行われてきた [Wol96, EHP98, HAA^{+}96]．これらの

研究・開発の大部分はプログラム中のループ部分の並列化を対象としたものであり，現在までに様々なループ並列性解析手法やリストラクチャリング手法が開発されている．このようなループ並列化手法は商用コンパイラにも導入され，IBM XL コンパイラや，Intel コンパイラ等は自動並列化機能を備えている．

ループ並列化手法は大きな進歩を遂げたが，現在では既に成熟期に至り今後の大幅な性能向上は見込めないと考えられている．そのため，マルチプロセッサシステムの更なる実効性能向上のためには，従来のループ並列性に加え，ループ間やサブルーチン間といった粗粒度タスク並列性や，基本ブロック内での命令・ステートメント間の近細粒度並列性などの複数レベルの並列性を利用することが必須である．

早稲田大学で開発されている OSCAR コンパイラ [本多 90, KHM+92, 笠原 03] では，粗粒度タスク並列処理，ループレベル並列処理，近細粒度並列処理を組み合わせたマルチグレイン並列処理を実現している．さらに OSCAR コンパイラは抽出したマルチグレイン並列性に応じ，プログラムの各所毎に並列性に見合った適切な計算資源 (プロセッサ) の割当てや粗粒度タスク間にまたがる広域的なキャッシュメモリ最適化を実現している．本研究では，この OSCAR コンパイラを利用して様々なマルチコアプロセッサにおける C プログラムの自動並列化および低消費電力化に関する研究・開発を行った．本章ではこの OSCAR コンパイラの概要について述べる．

本章の構成は以下の通りである．まず，第 2.2 節では OSCAR コンパイラが実現しているマルチグレイン並列処理について述べ，第 2.3 節では OSCAR コンパイラの構成と，OSCAR コンパイラが並列コード生成に利用している並列処理用 API である OSCAR API について述べる．そして，第 2.4 節では関連研究について述べ，最後に第 2.5 節でまとめを述べる．

2.2　マルチグレイン並列処理

本節では，OSCAR コンパイラで実現されているマルチグレイン並列処理について述べる．マルチグレイン並列処理は粗粒度タスク並列性，ループ並列性，近細粒度並列性を組み合わせ，プログラム全域から並列性を抽出する技術である．本論文では，粗粒度タスク並列性とループ並列性を用いたマルチグレイン並列処理を行う．

2.2.1　マクロタスク生成

粗粒度タスク並列処理では，プログラムは基本ブロックまたはその融合ブロックで構成される疑似代入文ブロック BPA[笠原 03]，DO ループや後方分岐により生じるナチュラルループで構成される繰り返しブロック RB[笠原 03]，サブルーチンブロック SB[笠原 03] の 3 種類の粗粒度タスク (マクロタスク MT[笠原 03]) に分割される．繰り返しブロック RB やサブルーチンブロック SB はその内部をさらにマクロタスクに分割し階層的なマクロタスク構造を生成する．

図 2.1 にマクロタスクの階層的定義の模式図を示す．まずプログラムのメインルーチン，すなわち第 1 階層を 3 種類のマクロタスクに分割し，マクロタスクの種類に応じて，第 2 階層，第 3 階層と階層的にマクロタスク分割する様子を示している．

2.2.2　粗粒度タスク並列性抽出

マクロタスク生成後，各階層においてマクロタスク間のデータ依存と制御フローを解析し，マクロタスク間のデータと制御のフローを表すマクロフローグラフ[本多 90, 笠原 03] を生成する．マクロフローグラフの例を図 2.2(a) に示す．マク

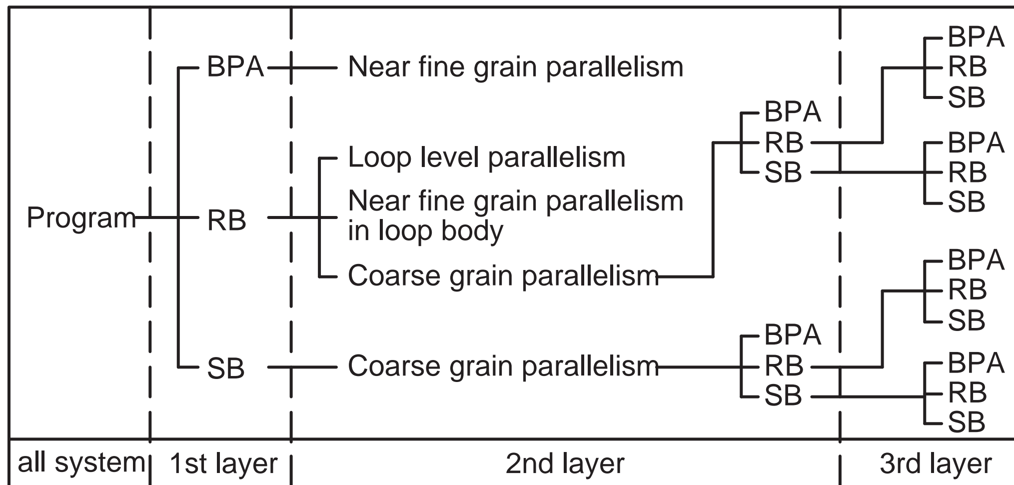

図 2.1: 階層的マクロタスク定義

ロフローグラフ中，ノードはマクロタスク，ノード中の番号はマクロタスク番号，ノード内の小円は条件分岐ノード，ノード間の実線はデータフロー，点線は制御フローをそれぞれ表す．このマクロフローグラフは，マクロタスク間の制御フロー，データフローを表すだけであり，マクロタスク間の並列性を表してはいない．

次に，階層的に生成されたマクロフローグラフに対し最早実行可能条件解析[本多 90, 笠原 03] を適用し，階層的なマクロタスクグラフ MTG[本多 90, 笠原 03] を生成する．最早実行可能条件とは，制御依存とデータ依存を考慮したマクロタスクの最も早く実行を開始してよい条件であり，マクロタスクグラフは粗粒度タスク並列性を表す．マクロタスクグラフの例を図 2.2(b) に示す．マクロタスクグラフでは，ノード，ノード中の番号，ノード中の小円はマクロフローグラフ同様マクロタスクと条件分岐を表しているが，ノード間の実線はデータ依存，点線は拡張された制御依存を表している．拡張された制御依存とは，通常の制御依存の他に，マクロタスクの非実行確定条件を含んでいることを表している．また，実線アーク (弧) は，アークが束ねるエッジが AND 関係にあり，点線アークは束ねるエッジが OR 関係にあることを表している．例えば，図 2.2(b) 中の MT6 の最早

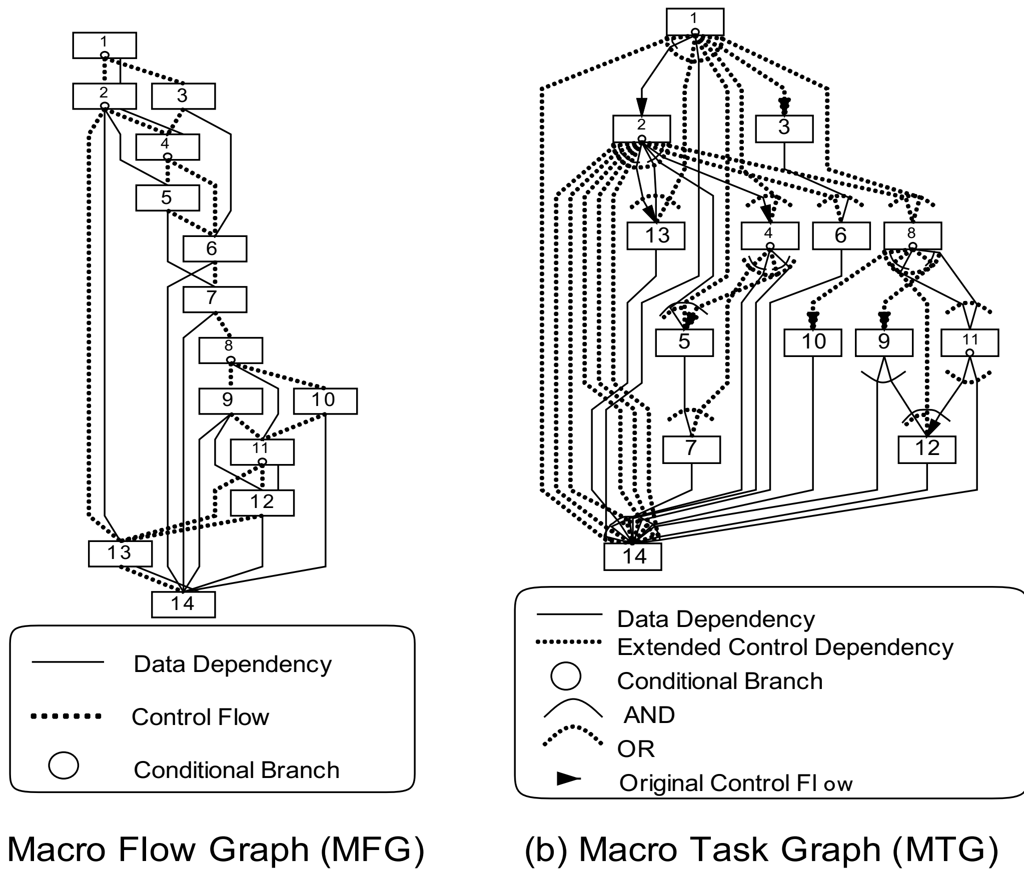

図 2.2: マクロフローグラフとマクロタスクグラフ

実行可能条件は,「MT2 が MT4 に条件分岐することが決定するか，MT3 の実行が終了」という条件になる．このマクロタスクグラフは，マクロタスク間の粗粒度タスク並列性を直接表現したグラフとなっている．

2.2.3 データローカライゼーション

プロセッサとメモリの速度差の拡大によりキャッシュメモリやローカルメモリを有効利用することがマルチプロセッサシステムの性能向上にとって重要となっている．OSCAR コンパイラでは並列性とデータローカリティの両方を考慮したデー

タローカライゼーション手法 [吉田 94, 石坂 02] により複数粗粒度タスク間でキャッシュあるいはローカルメモリ上のデータを効果的に用いる．

データローカライゼーション手法では，まず複数ループ間のデータ依存を解析し，データ依存する分割後の小ループ間におけるデータ授受がキャッシュあるいはローカルメモリを介して行われるようにそれらのループを整合して分割するループ整合分割 [吉田 94] を行う．分割されたループのうち同一データにアクセスする複数のマクロタスクは，データローカライザブルグループ（DLG）と呼ぶタスク集合にグループ化される．図 2.3 にループ整合分割を適用したマクロタスクグラフを示す．図中 (b) の同じ網掛けで塗られたマクロタスクが DLG に属するマクロタスクである．

整合分割後の粗粒度タスクスケジューリングでは，粗粒度タスク間の並列性を考慮しながら，同一 DLG に属するマクロタスクが可能な限り同一プロセッサ上で連続的に実行されるようにスケジューリングを行う．このようにループ分割と DLG 内タスクの連続実行を組み合わせることにより，複数のループに渡り再利用することを可能とすることでメインメモリアクセスを削減し，タスク間のデータ授受をキャッシュあるいはローカルメモリを用いて高速に行うことが可能となる．

2.2.4 プロセッサグループへのマクロタスク割り当て

粗粒度タスク並列処理では複数のプロセッサエレメント PE[小幡 03] をソフトウェア的にグループ化したプロセッサグループ PG[小幡 03] に割り当てる．割り当てられた MT 内で更に MTG が定義されている場合は，プログラム中の階層的 MTG の並列性を有効に利用するため，内部 MTG の並列性に応じて PG 内の PE を階層的にグループ化する．

マクロタスクグラフがデータ依存のみから構成される場合はコンパイル時に静的にスケジューリングが行われ，各プロセッサグループの処理するマクロタスクが

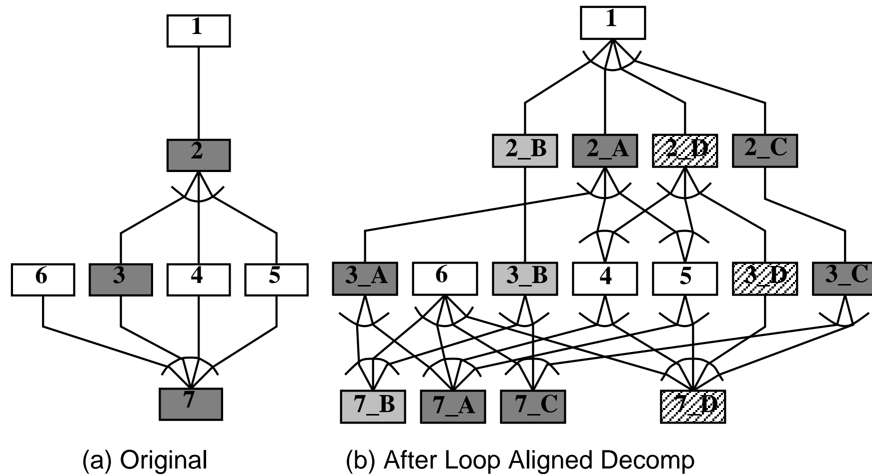

図 2.3: データローカライゼーションにおけるループ整合分割

決定される．マクロタスクグラフが条件分岐等の実行時不確定性を含む場合は実行時にスケジューリングを行うダイナミックスケジューリングルーチンをコンパイラが自動生成し，実行時にマクロタスクを PG に割り当てる．ダイナミックスケジューリングには，1 つのスケジューリング専用プロセッサを用いる集中型と，マクロタスクコードの実行とスケジューリングを全プロセッサが協調して行う分散型があり，実行するターゲットシステムのプロセッサ数やデータ通信オーバヘッド等の各パラメータに応じて適切な手法を選択する必要がある．各マクロタスクは階層的にスタティックスケジューリングあるいはダイナミックスケジューリングされる．生成されたスタティックスケジューリングコード及び実行時スケジューラはユーザコードであり，OS のシステムコールによるスケジューラに比べ極めて低オーバーヘッドなスケジューリングが可能である．

図 2.4 に階層的なコード生成のイメージ図を示す．まず，第 1 階層 (図中 1st layer) では 8 つの PE を 4PE を持つ PG0，PG1 の 2PG にグループ化している．この階層はおいては条件分岐等の実行時不確定性を含まないため，スタティックスケジューリングされ，MT1_1，MT1_2，MT1_4 は PG0 に，MT1_3 は PG1 に割り当てられ

ている．スタティックスケジューリングを適用した場合，コンパイラは各 PE に対してスケジューリング結果に従い，異なるコードを生成する．

次に，サブルーチンブロックである MT1_3(SB) の内部に第 2 階層 (図中 2nd layer) が定義されており，MT1_3_3 から MT1_3_5，MT1_3_6 に対する条件分岐が存在するため，ダイナミックスケジューリングが行われ，この例では MT1_3 を実行する 4 つの PE をそれぞれ PG とし，集中スケジューリングを適用している．集中スケジューリングの場合，一つの PE がスケジューラとして働き，その PE はマクロタスクの実行は行わず，その他の PE へのマクロタスクの割り当てのみを行う．したがって集中スケジューラとなる PE7 用のコードにはスケジューリングルーチンが生成される．一方，スケジューラ以外の PE はマクロタスクの実行を担当するが，実行するマクロタスクは実行時にスケジューラによってきめられるため，各 PE のセクションには PE4 から PE6 に示すように，全ての第 2 階層のマクロタスクのコードが生成され，各 PE は実行時にスケジューラの割り当て結果に従い，これらのマクロタスクを選択的に実行する．

一方，繰り返しブロックである MT1_4(RB) の内部には第 3 階層 (図中 3rd layer) が定義されており，この例では分散ダイナミックスケジューリングを適用している．また，この階層では MT1_4 を実行する 4 つの PE を 2PE ずつにグループ化している．分散ダイナミックスケジューリングでは，各 PG がスケジューリングとマクロタスク実行の両方を行うので，各セクションには全マクロタスクのコードとスケジューリングルーチンの両方が生成される．各 PE はスケジューリングルーチンを実行し，自身が次に実行するマクロタスクを決定した後，そのマクロタスクを実行する．マクロタスクの実行終了後はマクロタスクコードの後に生成されたスケジューリングルーチンを実行し，次に実行するマクロタスクを決定する．

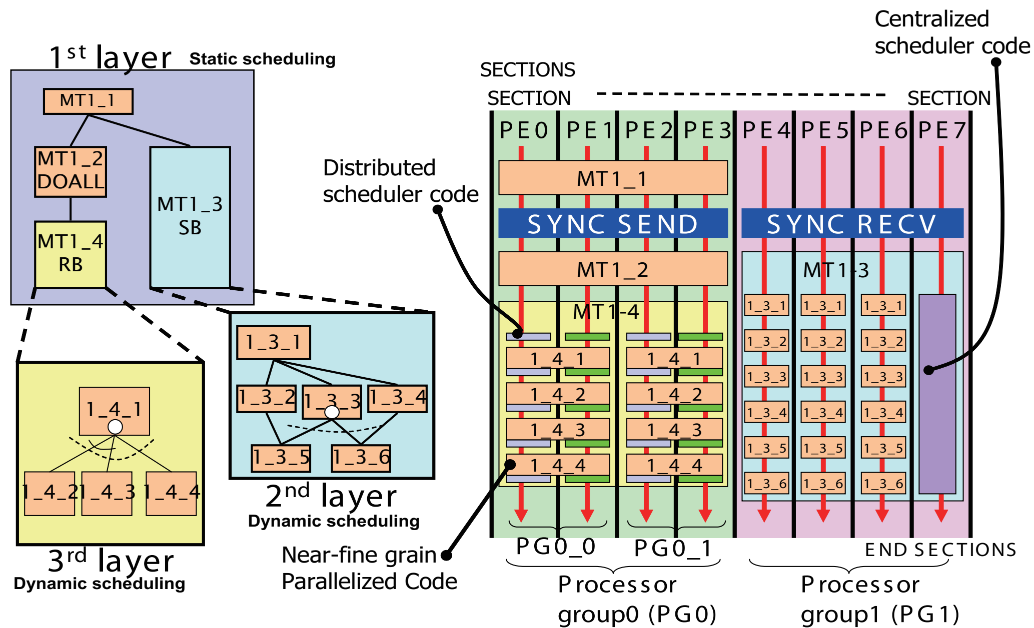

図 2.4: 階層的コード生成イメージ

2.3 OSCAR コンパイラの構成と OSCAR API

OSCAR コンパイラはソースコード変換により標準的な並列化指示文を含んだ C または FORTRAN プログラムを生成することで，多様なプラットフォームに対するポータビリティを実現している．OSCAR コンパイラはこの並列化指示文として OSCAR API[OSC, KMM+09] を利用している．

OSCAR API は NEDO “リアルタイム情報家電用マルチコア技術” プロジェクトにおいて，早稲田大学，東芝，日本電気，パナソニック，日立製作所，富士通研究所，ルネサステクノロジによって開発された情報家電用マルチコア向け並列化 API である．OSCAR 自動並列化コンパイラにより逐次プログラムから OSCAR API を用いた並列プログラムを自動生成することが可能であり，既存の逐次コンパイラに API 解釈部を追加することにより，各社マルチコア上での並列処理が実現可能となる．OSCAR API を用いることで，OSCAR コンパイラによる最適化を様々

なマルチコアで実現することができ，並列プログラムの大幅な生産性向上が期待できる．

2.3.1 OSCAR マルチコアアーキテクチャ

OSCAR API では標準的なアーキテクチャとして OSCAR マルチコアアーキテクチャを定めている．本節では OSCAR マルチコアアーキテクチャ[木村 01] の概要を述べる．

図 2.5 に示すように，OSCAR マルチコアアーキテクチャは複数のプロセッサコア (PE) および複数バスやクロスバ等の内部接続ネットワーク，オンチップおよびオフチップの集中共有メモリ (CSM) を持つ．各 PE は CPU コア，ローカルプログラムメモリ (LPM) または命令キャッシュ，ローカルデータメモリ (LDM) またはデータキャッシュ，分散共有メモリ (DSM)，およびデータ転送コントローラ (DTC) を持つ．コンパイル時の情報に応じてこれらのメモリを適切に使い分けることにより，スタティック及びダイナミックスケジューリング時の両方において効率の良い並列処理を行うことを特徴とする．さらに，コンパイラによる低消費電力制御を実現するために，周波数・電圧制御レジスタ (FVR) が追加されている [白子 06]．また，多くの実在するマルチコアプロセッサは OSCAR マルチコアのモデルで近似することができ，機能の有無やパラメタを変更することで，多様なマルチコアプロセッサへの対応が可能となる．

2.3.2 OSCAR API 指示文

OSCAR API は C と FORTRAN に対応しており，共有メモリマルチプロセッサを対象とした OpenMP[ope05] のサブセットに，OSCAR マルチコアアーキテクチャ用の新たな指示文を追加したものとなっている．以下に指示文の構成を示す．

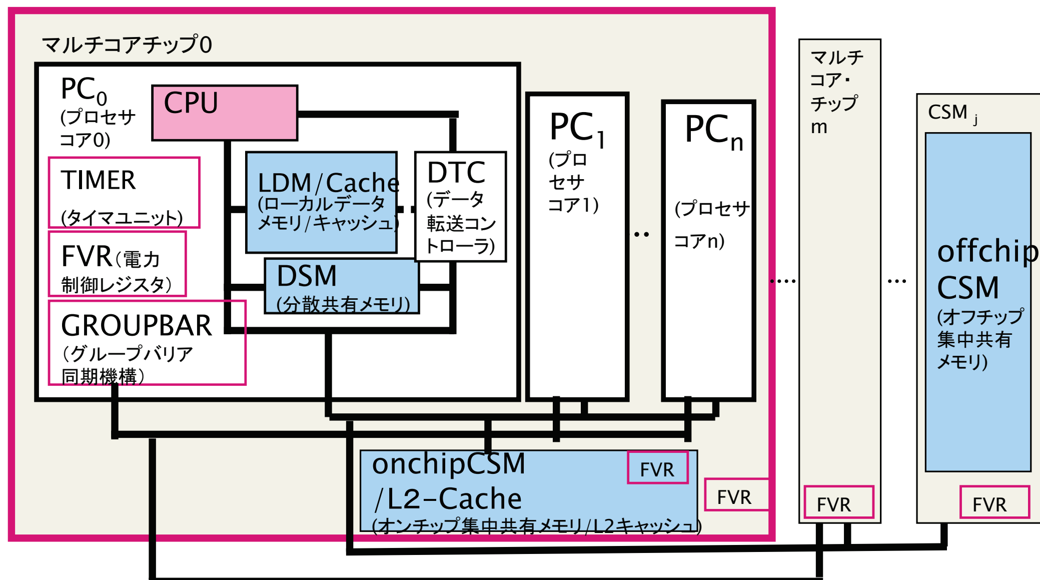

図 2.5: OSCAR マルチコアアーキテクチャ

OpenMP のサブセット

スレッド生成，同期，排他制御ディレクティブは OpenMP 互換となっており，スレッド生成には “parallel sections”，メモリアクセス順の保証には “flush”，排他制御には “critical” を用いる．これら 3 つがサポートされている環境であれば，SMP モードにおけるマルチグレイン並列処理を簡単に実現できる．

独自定義のディレクティブ

独自定義のディレクティブとしては，データのローカルメモリ配置，DTC(DMAC) によるデータ転送，周波数・電圧・電源制御，タイマ，同期用の API が策定されている．

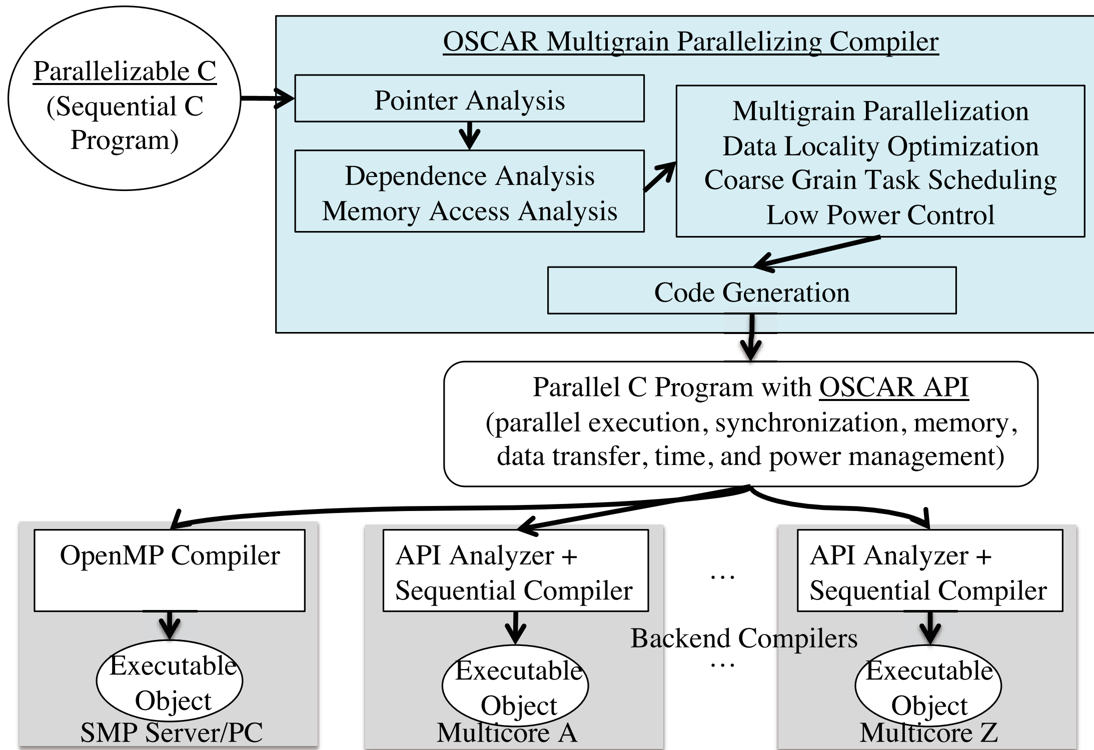

図 2.6: OSCAR コンパイラを用いたコンパイルフロー

2.3.3 OSCAR API を用いたコンパイルフロー

OSCAR 自動並列化コンパイラおよび並列化 API を用いたプログラムのコンパイルの流れを図 2.6 に示す．まず，C 言語等で記述された逐次型アプリケーションプログラムを OSCAR 自動並列化コンパイラでコンパイルし，OSCAR API で記述された並列プログラムを生成する．次に，この並列化されたプログラムをコード生成コンパイラ (通常の 1 プロセッサ用のコンパイラに API 解釈部を加えたもの) でコンパイルしマシンコードを生成する．これにより OSCAR コンパイラを用いて異なる企業のマルチコアプロセッサ用の並列コードを自動生成することが可能となる．

2.4 関連研究

プログラム全域からの適切な粒度の並列性抽出，プロセッサ近傍の高速キャッシュメモリあるいはローカルメモリの最適利用を実現するための自動並列化コンパイラの研究・開発が行われてきた．

イリノイ大学の Polaris コンパイラ [EHP98] ではシンボリック解析，Array Privatization，実行時データ依存解析，レンジテスト，インタープロシージャ解析といった強力な依存解析手法を用いたループ並列化を実現している．またスタンフォード大学の SUIF コンパイラ [HAA+96] では unimodular transformation や affine partitioning [WL91, LCL99] といった種々のループリストラクチャリングを内包した並列化手法を用いることでループレベルの並列性解析とデータローカリティ最適化を総合的に行っている．これらのコンパイラの研究により，ループ並列化技術は飛躍的に進化を遂げたが，ループ並列処理はすでに成熟期となっており，さらなる性能向上のためにはループ並列性のみならずタスク並列性やステートメントレベルの並列性の併用が必要不可欠と考えられる．

マルチレベルの並列性を利用するコンパイラには NANOS コンパイラ [GMO+00]，PROMIS コンパイラ [SSP99]，そして OSCAR コンパイラ [本多 90, KHM+92, 笠原 03] が挙げられる．カタルーニャ大学の NANOS コンパイラでは，拡張した OpenMP API によって粗粒度タスク並列性を含むマルチレベル並列性を抽出しようとしている．また PROMIS コンパイラはフロントエンドとバックエンドで共通の中間表現を用いてループ並列性と命令レベル並列性を統合しようとしている．それに対して，早稲田大学で開発されている OSCAR コンパイラでは，粗粒度タスク並列処理，ループレベル並列処理，近細粒度並列処理を組み合わせたマルチグレイン並列処理を実現している．

2.5　まとめ

本章では本研究で開発したOSCAR自動並列化コンパイラの機能と構成ならびにコンパイラの解析や最適化の処理フローについて述べた．

OSCARコンパイラは逐次CまたはFortranプログラムを入力して並列処理用の指示文を含む並列CまたはFortranプログラムを自動生成する．OSCARコンパイラは複数レベルの並列性を利用したマルチグレイン自動並列化を行い，プログラムの階層 (関数やループネスト等) ごとに並列性の抽出を行い，それぞれの階層の持つ並列性に応じて，ベーシックブロック，ループ，サブルーチン等の粗粒度タスクをプロセッサにスケジューリングしていく．さらに，プロセッサ近傍のキャッシュやローカルメモリを有効利用するためのデータローカリティ最適化を適用する．並列コード生成にはOpenMPの持つ４つの指示文に加えて，電力制御，メモリ配置，データ転送，グループバリア同期，時間管理をサポートするOSCAR APIの指示文を利用する．このOSCARコンパイラにより自動生成されたOpenMPおよびOSCAR APIを含む並列プログラムを各マルチコア向けのコード生成コンパイラを用いてさらにコンパイルすることで実行バイナリを生成する．これにより，様々なマルチコアアーキテクチャ上で自動並列化および低消費電力化を実現できる．

OSCARコンパイラを利用することにより，既存の並列処理手法を様々なマルチコアプロセッサにおいて実現しつつ，さらに新たな手法の研究・開発が可能となる．本論文の以下の章で提案するプログラムの解析や最適化手法は，このOSCARコンパイラに実装されている．

第3章

ポインタ解析

3.1 はじめに

組込み機器から PC，スーパーコンピュータに至るあらゆる情報機器において，マルチコアプロセッサが主流となってきており，並列ソフトウェア開発の生産性向上のため自動並列化コンパイラの実用化が期待されている．従来より科学技術計算分野の FORTRAN プログラムに対しては，Polaris[EHP98]，SUIF[HAA+96]，OSCAR[笠原 03] といった高性能な自動並列化コンパイラが開発され，多くの並列マシン上でその有効性が確認されてきた．より広く利用されている C 言語においても自動並列化コンパイラの利用が望まれているが，C 言語ではポインタを多用した自由な記述が可能であり，コンパイラがこれらのポインタの指し先を一意に解析できない場合，依存解析の精度が劣化し自動並列化の阻害要因となってしまう．

現状では多くの研究用あるいは商用の自動並列化コンパイラにおいて，C プログラムに対する効果的な自動並列化や自動ベクトル化を期待する場合は，全てのポインタに対する restrict 修飾 [C9999] や，ポインタエイリアスの扱いに関する独自のコンパイラオプションを利用することが推奨されており，事実上全くポインタを使用しないプログラムを記述する必要がある．しかしながら，コンパイラによるポインタ解析の精度を高めることにより，プログラム記述に対する制約条件が緩和され，高度なコンパイラ最適化技術の利用によるマルチコア，メニーコア向けのソフトウェア生産性の向上が可能と考えられる．

ポインタ解析に関しては多くの研究 [Hin01] が行われているが，C 言語のあらゆる記述に対してポインタの指し先を静的に特定するのは事実上不可能となっている．そのため，科学技術計算やメディア処理アプリケーションのアルゴリズムは潜在的に高い並列性を持っていながら，従来のポインタ解析技術では並列性の自動抽出にはしばしば不十分なことがある．

そのような例の一つに，アルゴリズム上は多次元配列として扱うことが可能なデータ構造を，ポインタへのポインタとメモリ動的確保を行うループの記述によ

```
/* 多次元配列のようなデータ構造を確保 */
BB1:
a = (int **)malloc(n * sizeof(int *))
LOOP2:
for (i = 0; i < n; i++) {
    a[i] = (int **)malloc(m * sizeof(int *));
}
/* 確保したデータ構造に計算結果を代入 */
LOOP3:
for (i = 0; i < n; i++) {
LOOP3_1:
    for (j = 0; j < m; j++) {
        a[i][j] = i + j;
    }
}
```

図 3.1: 従来のポインタ解析では識別できないプログラム例

り実装する場合がある．多次元配列と同様の機能を実現する際に，C 言語では図 3.1 のように malloc によって各次元に相当する領域を確保し，ポインタの参照によってアクセスすることがある．図 3.1 の例では，添え字とメモリアドレスが 1 対 1 の関係になっており，i および j を変数としたとき $a[i][j]$ は必ず一意なメモリアドレスへの参照となっている．一般に，アルゴリズム上は多次元配列となっていても，各次元の要素数が入力によって決定される場合には，C 言語ではこのようなデータ構造が静的な配列の代替として利用されることが多い．

この例では，LOOP3 の外側ループにおいてイタレーションごとの並列処理が可能である．しかし，このようなポインタへのポインタとヒープで確保されたデータ構造に対して従来のポインタ解析の適用を考えた場合，多くのポインタ解析では配列の個々の要素の指し先を独立には解析しないため，ポインタ配列の各要素の指し先情報は配列全体で単一の情報に縮退される．そのため，図 3.1 における LOOP3 の外側ループでは，a[i] の示す各要素の間のエイリアスの関係が不明とな

ることから，ループイタレーション間の依存解析ができず，自動並列化が阻害されてしまう．

そこで本章では，ポインタの配列において各要素の指し先のエイリアス関係を識別可能な Element-Sensitive ポインタ解析を提案する．ポインタ配列の各要素に指し示されるオブジェクトに重なりが無い場合は，このようなデータ構造に対するアクセスを含むループに対し，依存距離と依存ベクトルによるイタレーション間依存解析が適用可能となり，並列性の抽出が可能となる．

提案する Element-Sensitive ポインタ解析では，既存のポインタ解析手法に対して簡単な追加情報を加えるだけで，自動並列化に有用なポインタ解析精度を得ることができる．Element-Sensitive ポインタ解析の利点としては，(1) 既存の逐次プログラムが自動並列化可能となることによる速度向上，(2) 既存の逐次プログラムを書き換えることで自動並列化による速度向上を得ようとした場合のコード書き換え量の削減，(3) ループリストラクチャリング等のプログラム最適化過程におけるポインタ指し先情報の表現として利用，そして (4) 解析アルゴリズムがシンプルであり軽量で効率的な実装が可能，といったことが挙げられる．

本章の構成を以下に示す．従来のポインタ解析に関する研究成果を利用し，Flow-Sensitive[EGH94, WL95, Kah08, HL09], Context-Sensitive[EGH94, WL95, WL04, NKH04, LLA07, Kah08, HL09], Heap-Sensitive[NKH04, LLA07], Field-Sensitive[PKH04, LLA07] ポインタ解析を基準となるポインタ解析精度として採用し，さらにこれらに加えて，従来あまり議論されていなかった Cycle-Sensitivity[RRH08], Element-Sensitivity について焦点を当てる．まず第 3.2 節では基準となる既存のポインタ解析手法について述べ，第 3.3 節では新たに提案する Element-Sensitive ポインタ解析について述べる．次に第 3.4 節では Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用について述べ，第 3.5 節でポインタ解析結果の自動並列化への利用方法について述べる．そして，第 3.6 節では Element-Sensitive ポ

インタ解析を利用して自動並列化を適用した際のマルチコアシステム上での処理性能およびさらなるポインタ解析精度向上の可能性について述べる．最後に，第3.7節で関連研究について述べ，第3.8節でまとめを述べる．

3.2 基準となるポインタ解析

ポインタ解析とは，プログラム中に現れるポインタ変数がメモリ上のどの領域を指すかを解析するものである．データ依存の解析，データフロー解析，データアクセス範囲解析等の並列性抽出のためのプログラム解析 [Wol96] の精度はポインタ解析の精度に大きく依存する．そのため，ポインタ解析ではポインタの指し先を一意に決定できることが望まれる．ポインタ解析はプログラムの内部状態や指し示される領域に関する情報の持たせ方によって分類され，解析精度と解析コストのトレードオフが問題となる [Hin01]．

本節では，本章において基準となるポインタ解析精度として採用した，Flow-Sensitive，Context-Sensitive，Heap-Sensitive，Field-Sensitive ポインタ解析について述べる．

3.2.1 Points-to 解析

ポインタ解析では，メモリ上のあるポインタオブジェクトとそのポインタが指し示す可能性があるオブジェクトを解析するものが主流であり，Points-to 解析とも呼ばれる．この Points-to 解析で対象とするオブジェクトは，プログラム中で宣言されるスカラ変数や配列変数あるいはヒープ上の領域等を示し，しばしば複数のメモリ位置を一つの名前に抽象化して扱うため，抽象化されたメモリ位置 (abstract memory location) とも呼ばれる．Points-to 解析のイメージを図 3.2 に示す．図中では，ポインタの指し先関係を → を用いて表し，$(a \rightarrow b)$ はポインタオブジェク

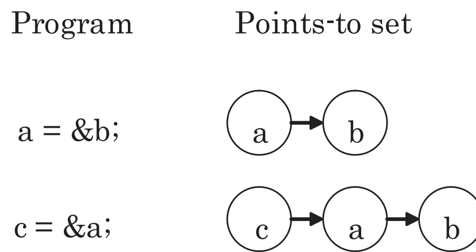

図 3.2: プログラムと Points-to 集合の例

ト a がオブジェクト b を指す可能性があることを示している．このようなポインタの指し先関係の集合を Points-to 集合と呼び，Points-to 解析は Points-to 集合を解析する．Points-to 集合におけるポインタの指し先情報は，各オブジェクトをノード，指し元オブジェクトから指し先オブジェクトへの関係をエッジで表現した有向グラフ (Points-to グラフ) として扱うことができる．

3.2.2 Flow-Sensitive ポインタ解析

Flow-Sensitive ポインタ解析はプログラムのコントロールフローを考慮して，プログラムの各地点ごと，一般的にはステートメントごとに個別の解析情報を生成する．Flow-Insensitive 解析ではステートメントの順序を考慮せず，サブルーチンやプログラム全体で有効である単一の解析情報を生成するため，Flow-Sensitive よりも保守的な解析結果となる．

一般的に，Flow-Sensitive ポインタ解析は，データフロー解析 [ALSU07] の枠組みを用いて実装される．すなわち，対象となる束 (lattice) L，束の交わり (meet) 演算子 $\wedge$，伝達関数 (transfer function) F で定義される．ポインタ解析においては，束 L は Points-to 集合，交わり演算子 $\wedge$ は和集合 $\cup$ となり，伝達関数は各ステートメントにおける入力から出力に対する Points-to 集合の変化を計算する．また，ポインタ解析はコントロールフローグラフに沿って，順方向に行われる．

コントロールフローグラフにおける各ノードを k とすると，各ノードは入力するポインタ解析情報 IN_k と出力するポインタ解析情報 OUT_k の二つの Points-to 集合を保持する．それぞれのノードにおいて，IN_k から伝達関数 F_k を用いて OUT_k を計算する．このときのノード k におけるデータフロー方程式は以下のようになる．

$$IN_k = \bigwedge_{x \in pred(k)} OUT_x \tag{3.1}$$

$$OUT_k = F_k(IN_k) \tag{3.2}$$

このときの伝達関数 F_k は，以下の通りである．

$$F_k = gen_k \wedge (IN_k - kill_k) \tag{3.3}$$

ここで，gen_k にはそのステートメントによって新たに生成される Points-to 集合が登録される．また，$kill_k$ は更新されるポインタに対して強い更新 (strong update) を行うか，弱い更新 (weak update) を行うかによって内容が異なる．新たに生成される Points-to 集合において，あるポインタの指し元オブジェクト v が確実に更新され，v が単一のポインタに対応することが決定できる場合は，強い更新が行われる．すなわち v の更新前の指し先情報が $kill$ 集合に登録されて，その結果データフロー方程式により除去されることになる．一方，ポインタの指し元オブジェクト v が確実に更新されるとは限らない，あるいは v が単一のポインタに対応することが決定できない場合は，弱い更新が行われる．すなわち $kill$ 集合を空集合とすることでそれまでのポインタ指し先情報は全て保持される．

本章のポインタ解析は Emami らの Flow-Sensitive，Context-Sensitive ポインタ解析のアルゴリズム [EGH94] をベースとして実装している．Flow-Sensitive ポインタ解析は解析コストが大きいことが問題となっていたが，近年は現実的な時間で解析可能なアルゴリズムが提案されてきている [Kah08, HL09].

3.2.3　Context-Sensitive ポインタ解析

Context-Sensitive ポインタ解析では関数呼び出しの関数のコールパスごとに個別に解析情報を生成することで，解析精度を高めている．Context-Insensitive ポインタ解析では，コールパスを区別せず，ポインタの引数間のエイリアスの関係が異なる場合は，それらの情報が縮退され，保守的な解析結果となってしまう．

Emami らの Flow-Sensitive，Context-Sensitive アルゴリズム [EGH94] では，引数におけるポインタ解析情報の関係に応じてサブルーチンをクローニングすることで Context-Sensitive 解析を実現している．関数呼び出しにおいてポインタを引数として渡す場合，呼び出された関数では引数のポインタを介して，スコープ外の変数にアクセスすることがある．そのような場合は，スコープ外の変数を不可視変数 (invisible variable) として定義し，解析情報を表現する．インタープロシージャ解析では，Mapping によって呼び元の関数の変数を呼び出される関数の不可視変数に対応付けを行い，呼び元における解析情報から呼び先における解析情報を合成する．そして，呼び先の関数の解析後に Unmapping を行う．すなわち，呼び先の関数における解析情報から呼び元における解析情報を合成する．

3.2.4　Heap-Sensitive ポインタ解析

ヒープ領域の抽象化についてもこれまでに多くの議論が行われてきたが，多くのポインタ解析では，メモリ動的確保を行うステートメントごとに別のオブジェクトとしている [Kah08, HL09]．しかし，メモリ確保用のラッパ関数を用意していた場合，同じデータ型のオブジェクトを確保すると，それらが全て同一のオブジェクトとして縮退してしまい，解析の精度が低下する．

これを解決するために，ヒープを確保した際にメモリ動的確保が行われたコールパスごとに別名のオブジェクトとする手法があり，これらは Heap-Sensitive，あ

るいは Heap-Cloning と呼ばれる [NKH04, LLA07].

本章ではこの Heap-Sensitive ポインタ解析を採用している．この Heap-Sensitive 解析はインタープロシージャ解析時に，呼び出された関数の解析情報を呼び出し元の関数に Unmapping する際に，その都度，別名のオブジェクトとして扱うことで実現できる．

3.2.5　Field-Sensitive ポインタ解析

Field-Sensitive ポインタ解析 [PKH04] では構造体の各メンバを区別して，それぞれ別のオブジェクトとして解析を行う．Field-Insensitive ポインタ解析では，構造体全体を単一のオブジェクトとして扱い，各メンバの指し先情報が縮退される．構造体をプログラムの主要なデータ構造として利用している場合は，高い解析精度を得るために Field-Sensitive 解析が重要となるため，本章では Field-Sensitive ポインタ解析を採用している．

3.3　Element-Sensitive ポインタ解析

本節では，新たに提案する Element-Sensitive ポインタ解析について述べる．多くのポインタ解析では，ポインタの配列に対して各要素の指し先を区別せず，各要素が指す可能性がある全ての領域を，単一の指し先情報に縮退して表現していた．しかし，これでは図 3.1 で示すようなポインタへのポインタとメモリ動的確保を用いたようなデータ構造において十分な解析精度が得られず，自動並列化の阻害要因となってしまう．このような例では，ポインタの配列において各要素ごとのポインタの指し先を識別することが必要となる．

そこで，本節では既存のポインタ解析の各オブジェクトに対して *element alias* 属性という真偽値型の変数を追加するだけで効率的に Element-Sensitive 解析を実現

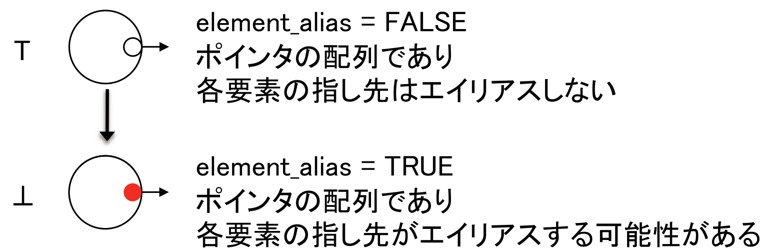

図 3.3: *element alias* 属性の束 (lattice)

する方法を述べる．この Element-Sensitive ポインタ解析のデータ構造およびデータフロー解析におけるアルゴリズムを示す．

3.3.1 *element alias* 属性

本章で提案する Element-Sensitive ポインタ解析は，ポインタの配列における任意の 2 要素が指し示すオブジェクトのエイリアスの有無を，*element alias* 属性を持たせることで区別する．図 3.3 のイメージ図のように，*element alias* 属性は真偽値型の変数であり，各要素の指し先にエイリアスがある場合は真，無い場合は偽の値を保持する．このポインタ型の配列オブジェクトに追加される *element alias* 属性がデータフロー解析の枠組みにより解析される．その際，*element alias* 属性の束 (lattice) は，⊤ が偽 (FALSE)，⊥ が真 (TRUE) となり，各オブジェクトの *element alias* 属性は偽の値で初期化され，後述の伝達関数により解析が行われる．

3.3.2 伝達関数

Element-Sensitive ポインタ解析の伝達関数では，*element alias* 属性を計算するために，従来の Flow-Sensitive ポインタ解析における gen の計算に若干の変更を行う．従来の gen の計算を gen_{fs} とすると，Element-Sensitive 解析における gen_{es}

は以下のようになる．

$$gen_{es} = calc_element_alias(gen_{fs}, IN) \tag{3.4}$$

すなわち，gen_{fs} と IN を入力として，*element alias*属性の計算 $calc_element_alias$ を行う．データフロー解析において，*element alias*属性を計算するためのアルゴリズム $calc_element_alias$ を図 3.4 に示す．

*element alias*属性の計算を行う関数 calc_element_alias は，各ステートメントにおける既存の Flow-Sensitive ポインタ解析の Points-to 集合 gen と Points-to 集合 IN を入力とし，*element alias*属性を更新された Points-to 集合 gen を返り値として返す．calc_element_alias では，gen に含まれるポインタの指し先関係 ($lh_ptr \rightarrow rh_object$) における，各ポインタの指し元オブジェクト lh_ptr について，lh_ptr がポインタの配列である場合に lh_ptr の *element alias*属性を計算する．lh_ptr がポインタの配列である場合，更新されるポインタ要素が一意とはならないため強い更新が行われることはなく，$kill$ 集合は必ず空集合となる．すなわち，伝達関数による処理後の lh_ptr の指し先は，ステートメントの IN に含まれる lh_ptr の指し先の集合 $rh_objects_IN$ と，gen 集合で生成された lh_ptr の指し先の集合 $rh_objects_gen$ の和集合となる．ここで，lh_ptr のそれまでの指し先のオブジェクトの集合 $rh_objects_IN$ と，新たなポインタの指し先として登録されるオブジェクトの集合 $rh_objects_gen$ に重なりがある場合は，ポインタの配列の各要素の指し先がエイリアスする可能性があるため，gen における lh_ptr の *element alias*属性を真とする．一方，$rh_objects_gen$ と $rh_objects_IN$ に重なりがない場合は，この更新によってポインタの配列オブジェクト lh_ptr の各要素の指し先にエイリアスが発生する可能性は無いため，gen における lh_ptr の *element alias*属性は偽のままとなる．

function calc_element_alias (Points_to_set *gen*, Points_to_set IN)
　return : Points_to_set
begin
　for each $lh_ptr \in \{lh_ptr \mid \exists rh_object\,(lh_ptr \rightarrow rh_object) \in gen\}$
　　if lh_ptr is array of pointer **then**
　　　$rh_objects_IN = \{rh_object \mid (lh_ptr \rightarrow rh_object) \in IN\}$;
　　　$rh_objects_gen = \{rh_object \mid (lh_ptr \rightarrow rh_object) \in gen\}$;
　　　if $rh_objects_IN \cap rh_objects_gen \neq \phi$ **then**
　　　　$lh_ptr.element_alias$ = TRUE;
　　　endif
　　endif
　endfor
　return *gen*;
end

図 3.4: gen における *element alias* 属性計算アルゴリズム

3.3.3 交わり (meet) 演算子 $\wedge$

データフロー解析の交わり (meet) 演算子 $\wedge$ は，*element alias* 属性に関しては図 3.3 のように真偽値の 2 通りの値のみを取り得るため，あるオブジェクトについて全てのパスでその *element alias* 属性が偽の場合は偽，一つでも真であれば真となる．

各ステートメントの *gen* の計算では，IN における Points-to 集合と *element alias* 属性を入力として *element alias* 属性を計算していたが，コントロールフローの交わり $\wedge$ ではオブジェクトの指し先の関係は考慮せずに各オブジェクトの *element alias* 属性のみを入力として演算を行う．これは，実際にプログラムを実行する際には，交わり後のポインタオブジェクトが指し示すのはその先行パスのいずれかで指し示していた指し先となるためである．このとき，オブジェクトのエイリアス関係には指し先のオブジェクトの重複は影響しないため，各パスの Points-to 集合に重なりがあった場合でも，全ての先行パスの *element alias* 属性が偽であれば，

交わり後の *element alias* 属性は偽とすることができる．

3.3.4 インタープロシージャ解析

インタープロシージャ解析では，複数のオブジェクトを単一の不可視変数に Mapping する場合に，*element alias* 属性の Mapping について考慮する必要がある．この場合，交わり (meet) 演算子 $\wedge$ と同様に，呼び出し元におけるそれぞれのオブジェクトの *element alias* 属性が 1 つでも真の場合は Mapping された不可視変数の *element alias* 属性は真，そうでなければ偽とする．

呼び先におけるオブジェクトから呼び元におけるオブジェクトへの Unmapping 時には，解析情報の合成は起こらないため，呼び先の情報から対応する呼び元におけるオブジェクトの *element alias* 属性を設定する．

3.3.5 Cycle-Sensitive 不使用時の Element-Sensitive ポインタ解析の適用例

図 3.1 のコード例について，Cycle-Sensitive を使用せずに Element-Sensitive ポインタ解析のみを適用した場合の解析イメージを図 3.5 に示す．この例ではまずポインタへのポインタ a の指し先として，ポインタの配列オブジェクトがヒープ領域に確保され，ポインタ解析においては h1 という名前で識別される．次に，a[i] の各要素の指し先として，ループ中でそれぞれヒープ領域を確保しており，これらをまとめて一つのオブジェクトとして扱い，h2 という名前が付けられる．すなわち，プログラム実行時には a[i] の指し先は各イタレーションで確保されたそれぞれ独立した領域を指し示すことになるが，ポインタ解析におけるヒープの名前付けはプログラム中の静的なステートメントの場所によって行われるため，各イタレーションで確保されるヒープは全て h2 という名前に縮退されて表現される．

データフロー解析では，ループについては反復的に解析を行い，解析結果が収束するまで計算を続ける．この例ではまず 1 イタレーション目では「h1 が h2 を指す」という解析情報が gen として生成される．この時点では，IN は h1 は未初期化状態のため，このイタレーションでは h1 の指し先が IN と gen で重複することはなく，OUT は h1 が h2 を指し，h1 の $element\ alias$ 属性は偽という情報となる．

次にデータフロー解析の 2 イタレーション目でも，h1 が h2 を指すという解析情報が gen として生成される．しかし，このときは既に IN において「h1 が h2 を指す」という情報が存在するため，IN と gen において h1 の指し先が重複する．そのため，gen における h1 の $element\ alias$ 属性は真となり，OUT は h1 が h2 を指し，h1 の $element\ alias$ 属性は真となる．3 イタレーション目は 2 イタレーション目と同様に処理が進み，OUT が 2 イタレーション目の OUT と同一になるため，h1 が h2 を指し，h1 の $element\ alias$ 属性が真という結果でデータフロー解析が収束する．

この例においては，各イタレーションで確保されたヒープオブジェクトが単純に同一の名前のオブジェクト h2 として扱われるため，各イタレーションで確保する領域が別の領域であることを認識できず，$element\ alias$ 属性を正しく偽と設定できなかったことになる．そこで，第 3.4 節では Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析を併用することで，$element\ alias$ 属性を解析する方法を示す．

3.4 Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用

ループのイタレーションを識別する Cycle-Sensitive 解析を実現する方法の一つとして，Aging[RRH08] という手法がある．この手法では，ヒープオブジェクトの

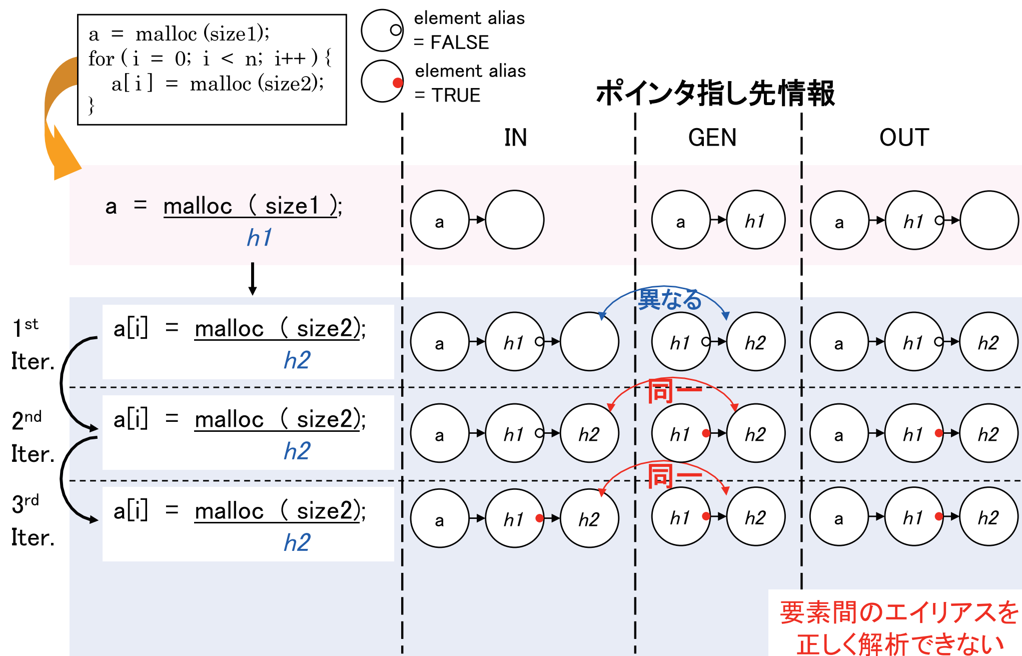

図 3.5: Cycle-Sensitive を利用しない Element-Sensitive ポインタ解析の適用例

名前付けを行う際に，確保されたヒープオブジェクトについて，現在のイタレーションで確保されたものなのか，前のイタレーションで確保されたものなのかを区別することで，簡易な実装で Cycle-Sensitive 解析を実現する．Aging により各イタレーションで確保するヒープ領域を識別することで，Element-Sensitive ポインタ解析において，*element alias* 属性を適切に設定することが可能となる．

3.4.1 Element-Sensitive ポインタ解析と Aging の併用例

図 3.1 におけるデータ構造構築部分のコードを例に，Element-Sensitive ポインタ解析と Aging を併用した場合の解析イメージを図 3.6 に示す．

この例においても a が h1 を指すという情報を生成する地点までは，図 3.5 と同様であるが，a[i] の指し先を動的確保するループの部分について，Aging を利用す

ることで解析の流れが変わってくる．

まず 1 イタレーション目では「h1 が h2 を指す」という解析情報が gen として生成される．このとき，Aging を用いる場合は，新たに確保されたヒープオブジェクトについては，NEW という属性が設定され，現在のループイタレーションで確保されたオブジェクトであることが明示される．この時点では，IN において h1 は未初期化状態のため，このイタレーションでは h1 の指し先が IN と gen で重複することはなく，OUT は h1 が h2 を指し，h1 の *element alias* 属性は偽という情報となる．

次にデータフロー解析の 2 イタレーション目の処理の開始時に Aging が適用され,「h1 が h2 の NEW のオブジェクトを指す」という情報が「h1 が h2 の OLD のオブジェクトを指す」という情報に変化する．すなわち，IN の情報は h1 が h2 の OLD を指す，という情報となる．ここで gen の生成は 1 イタレーション目と同様に h1 が h2 の NEW を指すという解析情報となるため，h1 の指し先は IN では h2 の OLD，gen では h2 の NEW となり，異なるオブジェクトなので h1 の *element alias* 属性は偽となる．そして，OUT は h1 が h2 の NEW または OLD を指し，h1 の *element alias* 属性は偽となる．3 イタレーション目の処理の開始時に再度 Aging が行われ,「h1 が h2 の NEW または OLD を指す」という情報が「h1 が h2 の OLD を指す」という情報に変化する．その後は 2 イタレーション目と同様に処理が進み，OUT が 2 イタレーション目の OUT と一致するため，h1 が h2 の NEW または OLD を指し，h1 の *element alias* 属性は偽という結果でデータフロー解析が収束する．このように，Aging の併用により正しく *element alias* 属性を解析することができる．

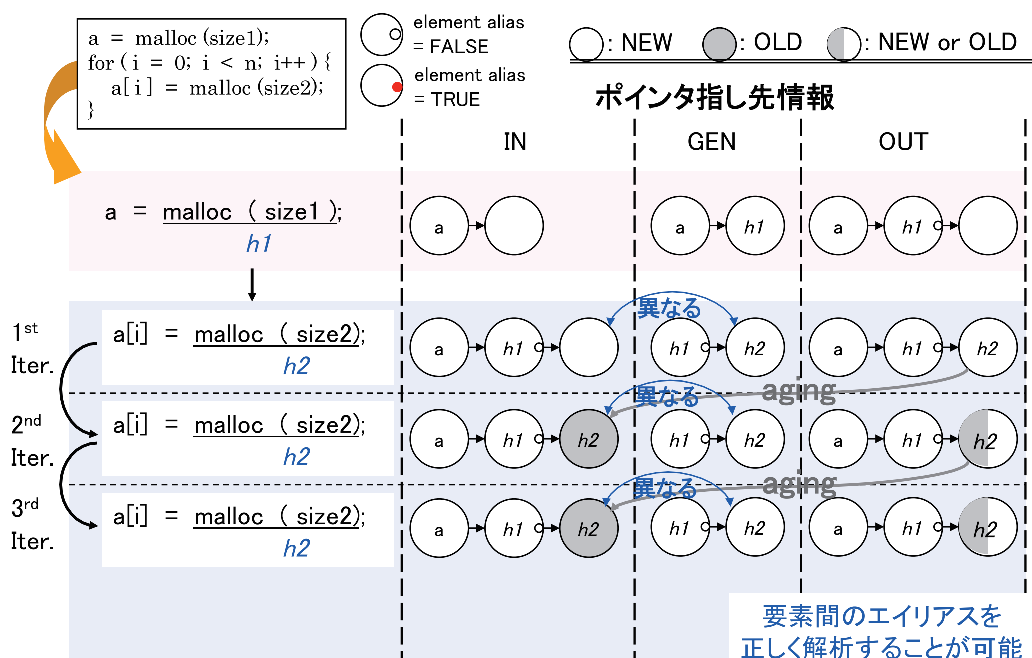

図 3.6: Element-Sensitive ポインタ解析と Aging の併用例

3.5 Element-Sensitive ポインタ解析の自動並列化への利用

本節では Element-Sensitive ポインタ解析による解析情報の自動並列化への適用について述べる．Element-Sensitive ポインタ解析における *element alias* 属性の情報は，ポインタへのポインタを介したアクセスに対して，配列ベースのプログラム向けの自動並列化技術 [Wol96, LLL01, 吉田 94] を適用するための条件となっている．これにより多次元配列をデータ構造としてプログラムを記述した場合と同様に自動並列化が適用可能となる．

3.5.1　イタレーション空間における依存解析

自動並列化のためのループレベル並列性やデータローカリティ最適化はイタレーション空間におけるループ依存解析結果を元に行う．多次元配列においては，配列の添字とメモリアドレスが一意に対応可能なことから，ループ制御変数と依存ベクトルを用いてイタレーションスペースにおける依存解析が可能である．

ポインタアクセスされる領域についても，対象のループにおいてそのポインタを用いて添字アクセスを行う場合に，その添字とアクセスするメモリアドレスが一意に対応可能なことが保証できれば，多次元配列と同様にイタレーション空間における依存解析が可能となる．

3.5.2　イタレーション空間における依存解析の例

図3.1の例と同様のコードサンプル (図3.7(a)) を用いた場合の Element-Insensitive および Sensitive ポインタ解析結果と，それぞれの場合のイタレーション空間における依存解析結果を示す．

図 3.7(b) の Element-Insensitive ポインタ解析を適用した場合は，保守的にポインタの配列 a[i] の各要素が指しうるオブジェクトに重複がある可能性があるものとして解析する．指しうるオブジェクトに重複がある可能性がある場合は，Points-to グラフの葉ノードにあたるオブジェクト，すなわち添字の最右側次元のアクセスに対応するメモリ領域については，添字とメモリ領域との対応が一意となるため，安全に依存解析を行うことができる．よって，図中 a[i][j] の j によるアクセスではオブジェクトの先頭からのオフセットは j の値により一意に決定される．しかし，左側次元の添字については，メモリ上の領域との対応関係が不明なため，多次元配列のように依存解析を行うことはできない．

それに対し，Element-Sensitive ポインタ解析を適用した場合 (図 3.7(c)) は，ポ

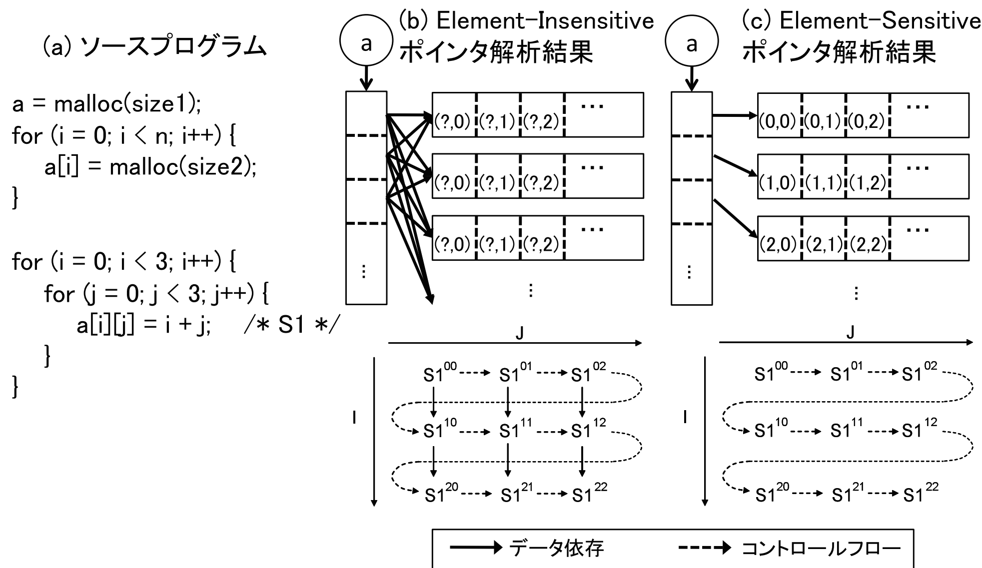

図 3.7: ポインタ解析を利用したイタレーション空間における依存解析結果

インタの配列 a[i] の各要素が指す領域が独立していることを保証できるため，a[i][j] の各次元の添字のパターンとそのアクセス先のメモリ位置の対応付けが可能となり，多次元配列と同様に扱うことができる．

3.6　マルチコアシステム上での性能評価

本節では，Element-Sensitive ポインタ解析を適用した際の，コンパイラによる自動並列化結果について述べる．Element-Sensitive ポインタ解析の有無による処理性能の違いについて評価する．

第 3.2 節で述べた基準となる Flow-，Context-，Heap-，Field-Sensitive ポインタ解析，および第 3.3 節で述べた新たに提案する Element-Sensitive ポインタ解析と第 3.4 節で述べた Aging(Cycle-Sensitive ポインタ解析) を，OSCAR 自動並列化コンパイラ [笠原 03] に実装し評価を行った．OSCAR コンパイラを用いて自動並列化した際の 8 コア構成の SMP サーバである IBM p5 550Q における使用するコア

数と処理性能について評価を行う．

3.6.1 自動並列化を適用する際のコンパイル手順

OSCAR コンパイラは C 言語等のソースコード変換をサポートしており，逐次の C 言語で記述されたプログラムに対して，OSCAR コンパイラは自動並列化を適用し，OpenMP[ope05] で並列化された C プログラムを自動生成する．このときに利用する OpenMP 指示文は parallel sections，flush，critical の 3 種類の指示文のみであり，OpenMP 仕様のうちこれらの構文をサポートしている環境であれば，並列コード生成が可能である．この 3 つのディレクティブは情報家電向けのマルチコア用並列処理 API である OSCAR API[OSC] においても，並列処理に必要な OpenMP 指示文のサブセットとして定義されている．

この OSCAR コンパイラにより自動生成された OpenMP C (OSCAR API C) プログラムを対象プラットフォームの OpenMP あるいは OSCAR API に対応したネイティブコンパイラによりコード生成を行う．

3.6.2 評価対象プログラム

SPEC2000 より art のオリジナルコード，equake を並列処理向けに修正を行ったコード，および SPEC2006 より hmmer，lbm をポインタ解析精度を考慮して修正を行ったコード (Parallelizable C[間瀬 09]) に対して自動並列化を適用した．この Parallelizable C コードは，Element-Sensitive ポインタ解析を用いてもオリジナル C コードではポインタの指し先が解析できない場合に，ポインタ解析精度を考慮したソースコードの修正を行うことで自動並列化を適用可能としたものである．

表 3.1: 評価環境

System	IBM p5 550Q
CPU	Power5+ (1.5GHz × 2 × 4)
L1 D-Cache	32KB for 1 core
L1 I-Cache	64KB for 1 core
L2 cache	1.9MB for 2 cores
L3 cache	36MB for 2 cores
Native Compiler	IBM XL C/C++ for AIX Compiler V10.1
Compile Option	OSCAR: -O5 -qsmp=noauto Native: -O5 -qsmp=auto
備考	SMT: disabled

3.6.3 評価環境

本評価においては，8 コア搭載の SMP サーバである IBM p5 550Q において評価を行った．評価環境のパラメータを表 4.2 に示す．

ただし，本章の評価では equake のみ IBM XL C/C++コンパイラの最適化レベルを-O4 としている．これは，OSCAR コンパイラで自動生成した OpenMP プログラムを IBM XL C/C++コンパイラで-O5 でコンパイルした際に正常に実行できなかったためである．本評価では並列処理による速度向上を評価するため，equake の全評価において最適化レベルを-O4 に統一した．

3.6.4 自動並列化による処理性能

オリジナルコードと Parallelizable C に書き換えたコードそれぞれに対して，OSCAR コンパイラで自動並列化を適用した際の処理性能を示す．IBM p5 550Q における処理性能を図 3.8 に示す．

図中，横軸が各アプリケーションと使用するポインタ解析における Element-

Sensitive および Cycle-Sensitive の有無を示す．Baseline は基準となる Flow-，Context-，Heap-，Field-Sensitive ポインタ解析を利用して自動並列化を適用した場合，Element Sensitive は基準となるポインタ解析に加え，Element-Sensitive ポインタ解析と Aging を併用した場合を示す．縦軸はオリジナルコードをネイティブコンパイラの 1 コアで逐次実行した場合に対する速度向上率を示す．横軸の各項目内のバーは使用しているプロセッサコア数を示し，それぞれ左から 1 コア，2 コア，4 コア，8 コアとなる．

IBM p5 550Q における処理性能

art，equake については Element-Sensitive ポインタ解析により自動並列化に必要なポインタの指し先情報を解析可能であり，図 3.8 に示すように，8 コア使用時に逐次実行時と比較して，art で 4.96 倍，equake で 5.61 倍の速度向上が得られた．hmmer, lbm についてはオリジナルコードは Element-Sensitive ポインタ解析を用いても十分なポインタ解析結果を得ることができなかったが，ポインタの利用法について若干の書き換えを行うことで，オリジナルコードの逐次実行時と比較して 8 コア使用時に hmmer で 6.06 倍，lbm で 5.35 倍の速度向上が得られた．これらの，ポインタ解析の解析精度を考慮して書き換えを行ったプログラムを含めると，Element-Sensitive ポインタ解析を用いた自動並列化により，逐次実行と比較して 8 コア使用時に平均 5.50 倍の性能向上が得られた．

このうち，Element-Sensitive ポインタ解析の利用によって大きな並列化効果が得られたのは equake と hmmer である．equake と hmmer については，基準となるポインタ解析では自動並列化による速度向上が全く得られていなかったが，Element-Sensitive ポインタ解析を利用することで自動並列化による大きな速度向上が得られた．

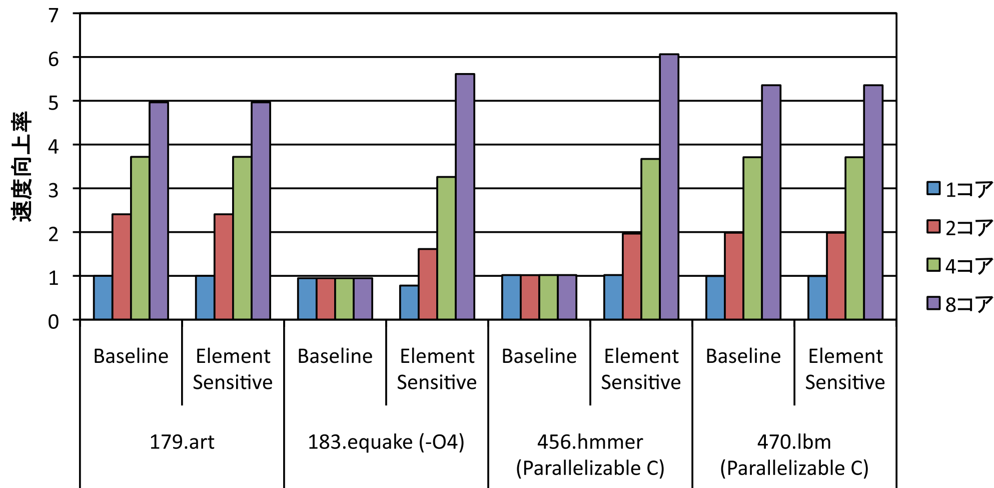

図 3.8: IBM p5 550Q における OSCAR コンパイラによる自動並列化結果

3.6.5 OSCAR コンパイラによる自動並列化結果

評価を行った art，equake，hmmer，lbm の各アプリケーションプログラムについて，適用された自動並列化の概要およびポインタ解析結果を以下に示す．

SPEC2000 art

art では，オリジナルの C ソースコードを修正すること無く，自動並列化による速度向上を得ることができた．art では，Element-Insensitive の基準となるポインタ解析において，OSCAR コンパイラの自動並列化により 8 コア使用時に 4.96 倍の性能向上が得られている．

art の主要なデータ構造は，複数のスカラ変数をメンバに持つ構造体の配列がヒープ領域に確保され，ポインタを介してアクセスされる形状となっている．すなわち，主要なデータ構造は構造体の 1 次元配列となっており，主要なループは構造体の配列の各要素に対して処理を行う．そのため，基準となるポインタ解析でも十分にポインタの指し先が解析されていた．

SPEC2000 equake

equake では Element-Insensitive ポインタ解析では自動並列化による速度向上を得ることができなかったが，Element-Sensitive ポインタ解析により 5.61 倍と大きな速度向上が得られた．

これは，equake の主要なデータ構造が図 3.1 で示したようなポインタへのポインタとヒープ領域を用いたデータ構造となっており，解析には Element-Sensitive ポインタ解析が必要不可欠となるためである．

SPEC2006 hmmer

オリジナルコードでは Element-Sensitive ポインタ解析を用いても，自動並列化による速度向上を得ることができなかったため，自動並列化可能となるように Parallelizable C に書き換えたコードに対する自動並列化結果を示している．Parallelizable C で記述することで，OSCAR コンパイラによる自動並列化により IBM p5 550Q で 8 コア使用時に 6.06 倍の速度向上が得られた．

hmmer のソースコード中にはポインタの配列は存在しないが，自動並列化の過程で，ループディストリビューションおよびスカラエキスパンションが適用され，その後に Element-Sensitive ポインタ解析が行われることで，大きな速度向上が得られた．すなわち，自動並列化の過程において，ソースコード上でスカラ変数であったポインタ変数を，コンパイラの中間表現ではポインタの配列として扱っている．このように，コンパイラにおけるプログラムリストラクチャリング適用後の解析情報の表現に Element-Sensitive ポインタ解析が有用となる．

SPEC2006 lbm

lbm においても，オリジナルコードでは Element-Sensitive ポインタ解析を用いても，自動並列化による速度向上を得ることができなかったため，図 3.8 では自動並列化可能となるように Parallelizable C に書き換えたコードに対する自動並列化結果を示している．Parallelizable C で記述することにより，OSCAR コンパイラによる自動並列化により 8 コア使用時に 5.36 倍の速度向上が得られた．lbm の主要なデータ構造はヒープ上に確保された 1 次元化された配列であり，Parallelizable C に書き換えを行うことにより，基準となるポインタ解析で自動並列化可能となった．

3.7 関連研究

従来よりポインタ解析の精度と効率について多くの研究がなされており [Hin01]，現在では従来高精度とされてきたポインタ解析も実用的な時間で実現可能となっており，さらなる精度向上も現実的となっている．

Flow-Sensitive，Context-Sensitive のポインタ解析は古くはデータフロー解析により定式化され，Emami ら [EGH94] による関数ポインタを考慮したアルゴリズムや，Wilson ら [WL95] による Partial Transfer Function を用いたインタープロシージャ解析アルゴリズムが提案されている．Flow-Sensitive のアルゴリズムも Context-Sensitive のアルゴリズムも最悪計算量が指数関数時間となる．現在ではより効率のよいアルゴリズムが提案されているが，しかしながら，当時の Flow-Sensitive，Context-Sensitive ポインタ解析は高い解析精度を持っていたものの解析効率が低く，数万行程度のプログラム規模までしか実用的な時間で解析することができなかった．そのため，解析効率の高い Flow-Insensitive 解析手法について研究がなされ，型推論によって定式化された Andersen の Subset-Based の多項式時間のアルゴリズム [And94] と，精度が劣るがほぼ線形時間で解析可能である Steensgaard の Inclusion-

Based のアルゴリズム [Ste96] が提案された．その後はより精度の高い解析を効率的に実現する手法が研究され，Heintze ら [HT01] により Subset-Based で 100 万行のプログラムの解析までスケールアップする Flow-Insensitive，Context-Insensitive アルゴリズムが提案された．さらに，Berndl ら [BLQ+03] や Whaley ら [WL04] による BDD を用いたポインタ解析により，Subset-Based，Flow-Insensitive，Context-Sensitive 解析が実用的な時間で可能となった．そして，Kahlon[Kah08] の精度の低いポインタ解析で対象の問題を分割することで高い精度のポインタ解析を適用する対象問題を小さくする手法や，Hardekopf ら [HL09] による Semi-Sparse ポインタ解析，Yu ら [YXH+10] による Full-Sparse ポインタ解析により，現在では Flow-Sensitive 解析も実用的な解析時間で実現可能となっている．

ヒープの扱いについても精度の議論が行われており，プログラム上のヒープを確保する malloc() 等の関数呼び出しのステートメントごとに別の領域として扱う方式が主流であったが，さらにヒープを確保する関数のコーリングコンテクストごとに領域を区別して扱う Heap-Sensitive あるいは Heap-Cloning と呼ばれる手法 [NKH04] が提案されており，ヒープを確保する際にラッパ関数を用いるような場合に必要な解析精度となっている．また，構造体のメンバの扱いについても議論が行われており，メンバを全て別の領域として扱う Field-Sensitive 解析を効率的に扱う手法が Pearce ら [PKH04] により提案されている．Lattner ら [LLA07] は Heap-Cloning と Field-Sensitive を両立した Context-Sensitive 解析を実用的な時間で可能な手法を提案している．

このように，現在では Flow-，Context-，Heap-，Field-Sensitive なポインタ解析も実用的に利用可能となっており，本章ではこれらの研究成果をベースとして，Element-，Cycle-Sensitive なさらに高精度なポインタ解析を実現した．

本章で Cycle-Sensitive 解析を実現するために利用した Aging[RRH08] は，ループのイタレーションの先頭でバッファを確保し，次のイタレーションでも使い回す

ようなパターンに対する解析として提案された．このような例では，通常のデータフロー解析では，各イタレーションで確保されるバッファがデータフロー解析の収束演算により同一領域とみなされてしまう．そこで，Aging を適用することで，そのイタレーションで確保されたヒープ領域とそれ以前のイタレーションで確保されたオブジェクトを別のオブジェクトとして解析可能とすることを可能としている．Aging の提案においても Element-Sensitive 解析との補完関係について言及されていたが，実際に本章で提案する Element-Sensitive ポインタ解析と Aging を併用することで高い解析精度を得ることができた．

既存の Element-Sensitive ポインタ解析として，ポインタの配列の各要素とヒープオブジェクトを確保したループイタレーションの情報をマッピングする Element-Wise Points-to Set[WFPS02] がある．Element-Wise Points-to Set は Java の自動並列化を指向して提案されたもので，シンボリック式を用いてオブジェクトの対応関係を計算している．本章で提案する方式では *element alias* 属性という真偽値のみで表現しているが，これだけで自動並列化のための解析を行う必要条件を満たしており，軽量で効率的な実装が可能となる．

Points-to 解析ではなく，任意のポインタ変数の間のエイリアス関係を解析するポインタ解析も提案されている．Connection 解析 [GH96a] では複数のポインタオブジェクトが指し示す領域に重なりがないかどうかを解析する．本章で提案した *element alias* は複数のポインタオブジェクトの関係ではなく，単一オブジェクトの各要素の指し先の関係を解析するものであるが，Connection 解析と同様の概念を採用している．

また，リスト構造や木構造のように自身の構造体型へのポインタを持つような，再帰的なデータ構造に特化したポインタ解析が存在する．Shape 解析 [GH96b] ではポインタの指し先の具体的なオブジェクトを解析するのではなく，ポインタが指し示すデータ構造が木構造なのか DAG なのかサイクルなのか，といったデータ

構造の解析を行う．しかしながら，既存の高度な並列化あるいはデータローカリティ最適化技術 [Wol96, LLL01, 吉田 94] は主に多次元配列とループによる処理を対象としており，再帰的なデータ構造には対応していないのが現状である．そのため，再帰的なデータ構造に対する並列化技術と併せた解析手法の改良が今後の課題となる．

3.8 まとめ

本章ではコンパイラによる自動並列化のためのポインタ解析として Element-Sensitive ポインタ解析を提案した．提案した Element-Sensitive ポインタ解析は，*element alias* という真偽値型の変数のみによる表現により，軽量で効率的な実装を可能とし，既存の Flow-Sensitive ポインタ解析を容易に拡張可能である．Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用により，C 言語で頻出のデータ構造に対する自動並列化を可能とし，本手法が有効なアプリケーションプログラムが存在することを示した．科学技術計算およびマルチメディア処理を行う 4 つのプログラムに対して Element-Sensitive ポインタ解析を利用した自動並列化を適用し，マルチコアシステム上での処理性能の評価を行ったところ，8 コア構成のサーバである IBM p5 550Q において逐次実行時と比較して平均 5.50 倍の速度向上が得られた．

本章で提案した Element-Sensitive ポインタ解析により，既存の C プログラムの自動並列化による速度向上に加え，ユーザプログラマがプログラムを修正することによる自動並列化の適用可能性も広がる．ポインタ解析を利用したユーザプログラマへのプログラム書き換え支援ツールの開発等により，マルチコア向けソフトウェア開発における自動並列化の利用を促進し，ソフトウェア生産性の向上が実現可能と考えられる．

第4章

Parallelizable C

4.1 はじめに

マルチコアプロセッサを含め，マルチプロセッサシステム用の並列プログラミングは難易度が高く，所望の性能を得るために長期間の並列チューニングが必要であることで知られており，並列ソフトウェア生産性向上が大きな課題となっている．そのため，C や FORTRAN のような既存の逐次型プログラムをコンパイラで自動並列化することによる，短時間での高性能の実現が強く期待されている．

従来より科学技術計算分野の FORTRAN プログラムに対しては，Poralis[EHP98]，SUIF[HAA+96]，OSCAR[笠原 03] といった高性能な自動並列化コンパイラが開発され，多くの並列マシン上で，その有効性が確認されてきた．しかしながら，C 言語ではポインタを多用した自由な記述が可能であり，コンパイラによるポインタ解析において，ポインタの指し先を静的に特定するのは非常に困難である [Hin01]．冗長なポインタエイリアス情報が存在することにより，コンパイラによる依存解析の精度が劣化することが，自動並列化の阻害要因となっている．

そこで，C プログラムの記述を制約することで自動並列化コンパイラによる階層的な並列性の利用やキャッシュ，ローカルメモリ最適化といった高度な並列処理手法の利用を可能とするために，C プログラム記述のガイドラインとして Parallelizable C を提案する．Parallelizable C でプログラムを記述することにより，プログラマはアルゴリズムのチューニングに専念し，煩雑な並列処理向けのテクニックはコンパイラにより自動で適用可能となる．

C 言語の記述の自由度を用途に合わせて制限するコーディングガイドラインは，主に車載ソフトウェア向けの高信頼性，高安全性，高移植性を目的とした MISRA-C[MIS04] に代表されるように，産業界で広く普及している．マルチプロセッサ向けの自動並列化においても，IMEC によるマルチプロセッサ向けのコーディングガイドラインである Clean C[Cle] や，Hwu らにより assertion 等によるヒント情報を用いて自動並列化コンパイラを有効利用する暗黙的な並列プログラミングモ

デル [HRU+07] が提案されている．このようなガイドラインを設定する場合に，そのプログラム記述が現実的かどうか，つまり実際にどの程度の自動並列化の効果が得られるか，そしてその記述法がプログラマに不自由なプログラム記述を強制していないかどうかが，そのガイドラインの普及に向けた課題となる．

本章で提案する Parallelizable C は C 言語のサブセットとして定義され，C 言語の処理系で正常にコンパイルし実行可能である．Parallelizable C を利用したプログラム開発では，フルセットの C プログラムのうち，並列化を意識したい部分を Parallelizable C 言語で書き換えることで，それに応じた自動並列化を可能とすることを目指している．並列処理による高速化が期待されている，科学技術計算およびマルチメディア処理の逐次プログラムについて書き換えを行ったところ，それらは少量の書き換えにより，コンパイラによる自動並列化が適用可能であることを確認した．

本章の構成を以下に示す．まず第 4.2 節では Parallelizable C の概念について，第 4.3 節では Parallelizable C のコーディングルールについて述べる．第 4.4 節では Parallelizable C へのプログラム書き換えについて述べ，そして，第 4.5 節では性能評価に利用した OSCAR コンパイラの概要について，第 4.6 節では Parallelizable C プログラムに対して OSCAR コンパイラの自動並列化を適用した際のマルチコアシステム上での処理性能について述べる．第 4.7 節で関連研究について述べ，第 4.8 節でまとめを述べる．

4.2　Parallelizable C の概念

第 3 章で述べたポインタ解析の実装により，コンパイラによる自動並列化の適用範囲が拡大可能になっていくと考えられる．しかし，C 言語のあらゆる仕様を網羅するポインタ解析の実装は事実上不可能であり，自動並列化コンパイラを現

実的に利用していく上では，解析可能な C プログラムの記述を明確化し，その範囲でプログラムを記述する必要がある．

4.2.1 C 言語のコーディングガイドライン

Parallelizable C は C 言語のサブセットで定義されるコーディングガイドラインあり，C 言語の処理系で正常にコンパイルし実行可能である．本章で提案する Parallelizable C を利用したソフトウェア開発では，C プログラムのうち，並列化を意識したい部分を Parallelizable C の要件を満たすように書き換えることで，それに応じた並列性抽出を可能とすることを目指している．Parallelizable C の記述は，対象のコンパイラの解析器の精度に大きく左右されるため，コンパイラの解析精度が高くなるほど，プログラム記述ルールの制約が緩和され，プログラマの負荷を軽減することができる．

4.2.2 対象のコンパイラの解析精度

本章では以下のようなポインタ解析機能を持つコンパイラを想定して Parallelizable C のコーディングルールを決定する．

- ステートメントごとにコントロールフローに沿った解析を行う (flow-sensitive)
- 関数の呼び出し箇所ごとに解析を行う (context-sensitive)
- メモリ動的確保までの関数呼び出し経路ごとにヒープオブジェクトを区別する (heap-cloning)
- 構造体のメンバをそれぞれ別のオブジェクトとして扱う (field-sensitive)
- ポインタがオブジェクトの先頭を指すかどうかを判別する

- ポインタの配列について各要素の指し先に重なりがないかを判別する
- ヒープオブジェクトを確保したイタレーションを判別する

また，構造体の解析においては以下のような機能を持つものとする．

- スカラとして扱う構造体のメンバは別のオブジェクトとして扱う
- 配列として扱う構造体はそれ自体を配列オブジェクトとして扱い，配列要素内のメンバの区別は行わない

ここで，ポインタの指し先の領域についても，そのポインタからの逆参照時のオフセットが必ずゼロとなるような領域はスカラ，オフセットがゼロ以外になる場合がある領域は配列として扱うものとする．

4.3 Parallelizable C のコーディングルール

本節ではポインタ解析をサポートしたコンパイラを想定した，Parallelizable C のコーディングルールについて述べる．本仕様は C 言語記述におけるルールと，コンパイル時の外部関数に対するヒント情報から構成される．

4.3.1 C 言語記述におけるルール

Parallelizable C では本節で述べるルールに従ってプログラムの記述を行うものとする．ただし，Parallelizable C はフルセットの C 言語に対応した処理系でコンパイルされるため，プログラマの判断でルールを逸脱することも許されるが，コンパイラによる自動並列化は困難になる．

以下に，Parallelizable C の C 言語記述におけるルールを示す．

変数宣言の境界を超えたアクセスを行わない

構造体のメンバや多次元配列の各次元の境界を越えたアクセスを行わない．C コンパイラでは処理系により正常動作することもあるが，C 言語仕様上は未定義である．

ポインタのキャストを行わない

メモリ動的確保時を除いて，ポインタのキャストを行わない．ポインタ参照先のオブジェクトのオフセットを正しく解析するためである．

ポインタ算術演算を行わずに添字アクセスを行う

ポインタ算術演算は行わない．ポインタ指し先の領域にアクセスする場合にはポインタ変数の値は更新せずに，ポインタ変数に対する逆参照演算子 (*) もしくは添字アクセス ([ ]) により行うこととする．

条件分岐やループ内でポインタ変数への値の代入を行わない

メモリ動的確保時を除いて，条件分岐やループ内ではポインタ変数への値の代入を行わない．

同一領域に対する複数のポインタを関数の引数として渡さない

配列の単一次元内の異なるオフセットのアドレスを関数の引数として渡さない．ただし，同一配列でも重なりのない部分配列であればよい．これは，２つの引数の指し先を別々のオブジェクトとして扱えなくなるためである．

一時バッファとして利用するヒープ領域を使い回さない

ヒープ領域の使い回しは，通常の変数の使い回しよりも解析が困難であり，並列性抽出の阻害要因となる．ヒープ領域を使い回す例としては，ループのボディにおいてのみ利用するバッファ領域を複数ループイタレーションに渡って確保し続けて再利用する，といったことが挙げられる．

構造体の配列のメンバに対する配列アクセスを行わない

配列メンバを持つ構造体の配列を使用しない．構造体のメンバに配列がある場合は，下位の関数に構造体の１要素にあたる領域を引数として渡し，下位の関数内でそのメンバの配列へのアクセスを行う．これは配列要素間と同様に，構造体のメンバ間の識別精度も解析精度に影響するためである．

再帰的なデータ構造を利用しない

リストや木構造等，再帰的に同じ型の構造体オブジェクトへのリンクがあるデータ構造を使用しない．リストや木構造であれば並列処理可能なことがあるが，リストや木構造の解析には専用の解析が必要となる．配列ベースの記述であればコンパイラによる高度な最適化が適用できるため，配列型のデータ構造に変換してから計算処理を行うこととする．

呼び出し間に依存のある外部関数を使用しない

ファイル入出力，エラー処理等の，ライブラリ内で内部状態を持ち，呼び出しごとに依存がある外部関数を使用しない．エラー処理やログ出力は並列化したい部分の外側にくくり出す等の工夫が必要である．

関数の再帰呼び出しを使用しない

関数は直接的か間接的かに関わらず，その関数自体を呼び出さない．コールグラフを静的に決定できなくなったり，メモリ使用量の見積もりが難しくなったりするためである．

関数ポインタを使用しない

関数ポインタを使用せず switch-case を用いて関数呼び出しを記述する．これは関数の呼び出し先の静的な解析が難しくなるためである．

可変個引数を持つ関数を定義しない

可変個引数には未定義の動作が多いためである．

4.3.2 外部関数に関するヒント情報

プログラム全体に対して解析を行うことを前提としているが，ライブラリ関数等を利用する場合は，そのライブラリ関数による挙動をディレクティブやコンパイルオプションにより指示することでコンパイラの解析精度劣化を防ぐことができる．そのような指示を行う方法としては，以下のようなヒント情報が挙げられる．

- 外部関数においてポインタ引数の指し先は変化しないことを仮定する
- 外部関数からの戻り値がポインタ型の場合，そのポインタの指し先は新たに領域確保されたヒープ領域であることを仮定する
- 外部関数における入出力変数についてのデータフロー情報を指定する

表 4.1: Parallelizable C への書き換え項目とコード変更量

アプリケーションプログラム	合計 (LOC)	書き換え項目	削除量 (LOC)	追加量 (LOC)	変更率 (%)
SPEC2000 art	1270	N/A	0	0	0%
SPEC2000 equake	1513	N/A	0	0	0%
SPEC2006 lbm	1155	オブジェクト先頭以外へのポインタ ループ内のポインタ更新	-7 -10	+7 + 14	1.8%
SPEC2006 hmmer	35992	ヒープ領域の再利用	-9	+8	0.03%
Mediabench mpeg2encode	3750	アルゴリズムの変更	-640	+864	23.5%

4.4　Parallelizable C へのプログラム書き換え

本節では SPEC2000 より art, equake, SPEC2006 より lbm, hmmer, MediaBench より mpeg2encode を例に，Parallelizable C への書き換えについて述べる．

各アプリケーションの書き換えにおいては，まず最初にファイルを一つにまとめて，入出力や時間計測用のコードを整備した参照コードを作成し，それに対して書き換えを行った際の修正箇所を調べた．

書き換えの過程においては，Parallelizable C に適合するための書き換えに加えて，プログラムの持つ並列性を十分に抽出するためのプログラム構造の書き換えも行った．

4.4.1　書き換え項目とコード変更量

Parallelizable C への書き換えのコストを定量化するために，書き換え項目のリストアップおよび，書き換えの際の変更前のコードサイズを基準とした，削除量と追加量の割合を計測した．計測方法は，各ソースコードのインデントを無くしてから，行単位の diff を取り，削除側，追加側それぞれの行数をカウントした．表 4.1 にそれぞれのアプリケーションごとの，Parallelizable C への書き換え項目および削除量，追加量の割合を示す．また，このときの削除量と追加量のうちの大きい方を変更量とする．

4.4.2 コード書き換え内容

SPEC2000 の art については，オリジナルコードの時点で Parallelizable C のルールに適合していたため，書き換えは行わなかった．

SPEC2000 の equake については，Parallelizable C のルールには適合していたが，Parallelizable C コードでは，プログラム実行時間の大部分を占める関数 smvp() において，配列全体に対するリダクション処理による並列化を行うために手動でリストラクチャリングを行った．

SPEC2006 の lbm については，配列の途中を指すポインタと，ループ中におけるポインタ変数の更新があったため，Parallelizable C コードではそれらの除去を行った．図 4.1(a) のようなループ中でのポインタ更新については，図 4.1(b) のようにフラグ変数と条件分岐を用いて修正した．

SPEC2006 の hmmer については，図 4.2(a) のように，複数イタレーションに渡ってバッファ領域を使い回していたため，Parallelizable C コードでは図 4.2(b) のように各イタレーションごとに領域を確保して解放するように修正した．

mpeg2encode では，MPEG2 エンコードアルゴリズムが持つマクロブロックレベルの並列性とデータローカリティが利用不可能なプログラム構造となっているため，並列性抽出のためのプログラム構造の変更 [小高 05] を併せて行った．

4.5 OSCAR 自動並列化コンパイラの概要

OSCAR コンパイラ [笠原 03] は C や FORTRAN 言語で記述された逐次ソースプログラムを入力し，並列プログラムを自動生成する自動並列化コンパイラである．従来は，FORTRAN77 向けに開発されていたが，新たに C フロントエンド，C ソースコード生成部，ポインタ解析器の実装等の拡張を行うことで，C 言語の自動並列化にも対応した．

```
for( t = 1; t <= param.nTimeSteps; t++ ) {
        …
        LBM_performStreamCollide( *srcGrid, *dstGrid ); /* 主要計算部 */
        LBM_swapGrids( &srcGrid, &dstGrid ); /* srcGrid と dstGrid のポインタ値を交換する */
        …
}
```

(a) オリジナルコード

```
flg = 0;
for( t = 1; t <= param.nTimeSteps; t++ ) {
    …
    if (flg % 2)
        LBM_performStreamCollide( *dstGrid, *srcGrid ); /* 主要計算部 */
    else
        LBM_performStreamCollide( *srcGrid, *dstGrid ); /* 主要計算部 */
    flg++;
    …
}
```

(b) Parallelizable C コード

図 4.1: lbm におけるループ中でのポインタの更新の除去

OSCAR コンパイラではループイタレーションレベルの並列処理を行うのみでなく，ループ・手続き間の粗粒度タスク並列処理 [本多 90]，ステートメント間の近細粒度並列処理 [木村 01] を組み合わせたマルチグレイン並列処理 [笠原 03]，メモリウォール問題に対処するための複数ループにわたるキャッシュあるいはローカルメモリの最適利用 [吉田 94, 石坂 02] が実現されている．さらに，プログラム中の各並列処理部に対する適切なリソース割り当てや，各リソースの周波数・電圧・電源制御による消費電力の自動削減 [白子 06] が実現されている．

4.6 マルチコアシステム上での性能評価

本節では，Parallelizable C プログラムを OSCAR コンパイラで自動並列化した際の PC，サーバ，組込み機器向けのそれぞれのマルチコアシステム上における使用するコア数と処理性能について評価を行う．

```
static void main_serial_loop() {
    mx = CreatePlan7Matrix(1, hmm->M, 25, 0); /* P7Viterbi() で使用するデータ構造を malloc */
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(..., mx);  /* 主要計算部 */
        ...
    }
    ...
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm, struct dpmatrix_s *mx) {
    ResizePlan7Matrix(mx, ...); /* 作業領域を realloc により使い回す */
    ... /* score の計算 */
    return score;
}
```

(a) オリジナルコード

```
static void main_serial_loop() {
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(...);  /* 主要計算部 */
        ...
    }
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm) {
    struct dpmatrix_s *mx;
    mx = AllocPlan7Matrix(...); /* イタレーションごとに作業領域を malloc */
    ... /* score の計算 */
    FreePlan7Matrix(mx); /* イタレーションごとに作業領域を free */
    return score;
}
```

(b) Parallelizable C コード

図 4.2: hmmer におけるバッファの使い回しの除去

4.6.1 評価方法

OSCAR コンパイラによる並列化の際には，データローカリティ最適化を含めたマルチグレイン並列化を適用し，OpenMP[ope05] で並列化された C プログラムを出力する．このときに利用する OpenMP 指示文は OSCAR API[OSC] で規定されている OpenMP 指示文のサブセット部分にあたる．

自動生成された OpenMP C (OSCAR API C) プログラムを対象プラットフォームの OpenMP あるいは OSCAR API に対応したネイティブコンパイラによりコード生成を行う．また，ネイティブコンパイラにおいて自動並列化をサポートしている環境については，ネイティブコンパイラによる自動並列化結果も併せて示す．

4.6.2　評価対象プログラム

第4.4節で述べた，SPEC2000よりart，equake，SPEC2006よりlbm，hmmer，MediaBenchよりmpeg2encodeの5種類のCプログラムについて，オリジナルCコードとParallelizable Cコードそれぞれに対し，OSCARコンパイラによる自動並列化を適用した際の処理性能を示す．また，本章では書き換えを行っていないが，Parallelizable Cを満たすように実装されている音声圧縮プログラムであるAACエンコード(AACencode)についても，併せて評価を行う．AACエンコードは株式会社ルネサステクノロジ提供のプログラムであり，関数の引数ポインタ以外のポインタ・構造体を原則的に使用せずに製品レベルのミドルウェア仕様を参照実装したものである[間瀬07]．

4.6.3　評価環境

本評価においては，8コア搭載のSMPサーバであるIBM p5 550Q，Intelの4コアCPUであるCore i7を1基搭載したPC，ルネサステクノロジ，日立製作所，早稲田大学で共同開発した情報家電用マルチコアRP2[IHY+08]におけるSH-4Aを4コア構成によるSMPモードの3つの環境で評価を行った．各環境におけるパラメータを表4.2に示す．IBM p5 550Qでは，1プロセッサあたり2スレッド実行のSimultanemous Multi-Threading(SMT)が可能であるが，本評価でSMTは用いないものとした．同様に，Intel Core i7においても1プロセッサあたり2スレッド実行のHyper-Threadingが可能であり，またチップ温度を計測してハードウェアで自動的にオーバークロックするTurbo Boostが利用可能であるが，本評価では共に用いないものとした．

表 4.2: 評価環境

System	IBM p5 550Q	Intel Core i7 920	Renesas/Hitachi/Waseda RP2
CPU	Power5+ (1.5GHz × 2 × 4)	Nehalem (2.66GHz × 4)	SH-4A (600MHz × 4)
L1 D-Cache	32KB for 1 core	32KB for 1 core	16KB for 1 core
L1 I-Cache	64KB for 1 core	32KB for 1 core	16KB for 1 core
L2 cache	1.9MB for 2 cores	256KB for 1 core	
L3 cache	36MB for 2 cores	8MB for 4 cores	
Native Compiler	IBM XL C/C++ for AIX Compiler V10.1	Intel C/C++ Compiler verion 11.0	SH C Compiler + OSCAR API Parser
Compile Option	OSCAR: -O5 -qsmp=noauto Native: -O5 -qsmp=auto	OSCAR: -fast -openmp Native: -fast -parallel	

4.6.4 自動並列化による処理性能

オリジナルコードと Parallelizable C に書き換えたコードそれぞれに対して，OSCAR コンパイラで自動並列化を適用した際の処理性能を示す．IBM p5 550Q における処理性能を図 4.3 に，Intel Core i7 における処理性能を図 4.4 に，ルネサステクノロジ／日立製作所／早稲田大学 RP2 における処理性能を図 4.5 に示す．

図中，横軸が各アプリケーションとコードの種類を示し，縦軸はオリジナルコードをネイティブコンパイラの 1 コアで逐次実行した場合に対する速度向上率を示す．横軸の各項目内のバーは使用しているプロセッサコア数を示し，それぞれ左から 1PE，2PE，4PE，8PE となる．

IBM p5 550Q における処理性能

OSCAR コンパイラでは各アプリケーションのオリジナルコードに対して，8 コア使用時に逐次実行時と比較して，art で 4.96 倍，equake で 1.69 倍，mpeg2encode で 1.53 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジナルコードの逐次実行時と比較して equake で 5.61 倍，lbm で 5.35 倍，hmmer で 6.06 倍，mpeg2encode で 5.12 倍の性能向上が得られた．また，元々Parallelizable C で記述されている AAC エンコーダでは 6.16 倍の性能向上が

得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して平均 5.54 倍の性能向上が得られた．

なお，IBM p5 550Q の評価においては，equake のみ最適化レベルを-O4 としている．これは，Parallelizable C コードを OSCAR コンパイラで生成した OpenMP プログラムを IBM XL コンパイラで-O5 でコンパイルした際に正常に実行できなかったためである．本評価では並列処理による性能向上を評価するため，equake の全評価において最適化レベルを-O4 に統一した．

Intel Core i7 における処理性能

OSCAR コンパイラではオリジナルコードに対して，4 コア使用時に逐次実行時と比較して art で 1.51 倍，equake で 1.15 倍，mpeg2encode で 1.81 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジナルコードの逐次実行時と比較して equake で 1.45 倍，lbm で 1.28 倍，hmmer で 3.34 倍，mpeg2encode で 3.60 倍の性能向上が得られた．また，元々Parallelizable C で記述されている AAC エンコーダでは 3.48 倍の性能向上が得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して 4 コア使用時に平均 2.43 倍の性能向上が得られた．

ルネサステクノロジ／日立製作所／早稲田大学 RP2 における処理性能

OSCAR コンパイラではオリジナルコードに対して，4 コア使用時に逐次実行時と比較して art で 2.53 倍，equake で 1.33 倍，mpeg2encode で 1.61 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジ

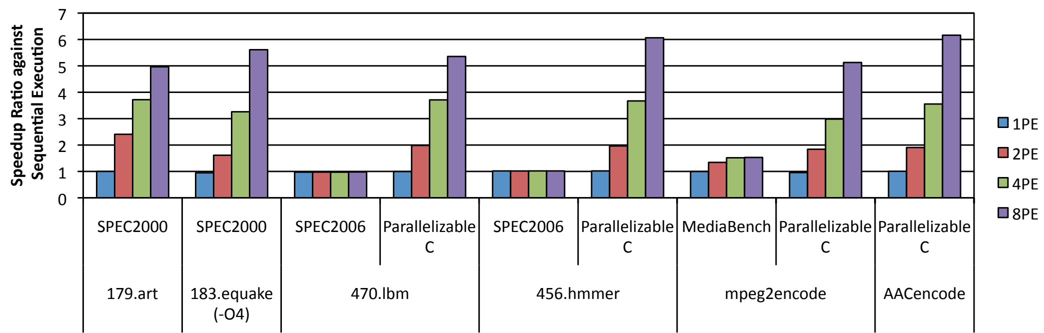

図 4.3: IBM p5 550Q における OSCAR コンパイラによる自動並列化結果

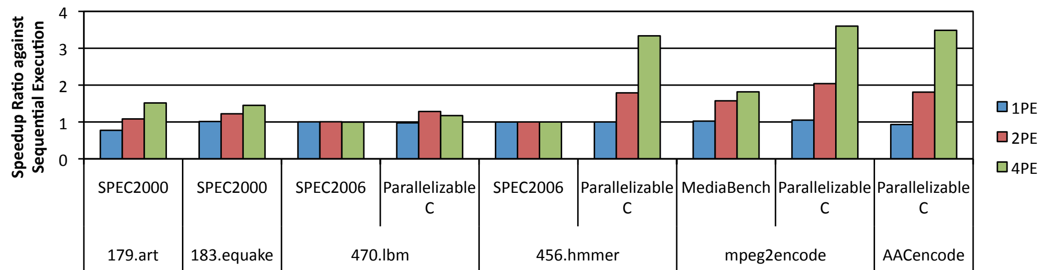

図 4.4: Intel Core i7 920 における OSCAR コンパイラによる自動並列化結果

ナルコードの逐次実行時と比較して equake で 1.95 倍，mpeg2encode で 3.27 倍の性能向上が得られた．また，元々Parallelizable C で記述されている AAC エンコーダでは 3.34 倍の性能向上が得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して 4 コア使用時に平均 2.78 倍の性能向上が得られた．

なお，SPEC2006 のアプリケーションについては，本章では RP2 上での評価を行っていない．

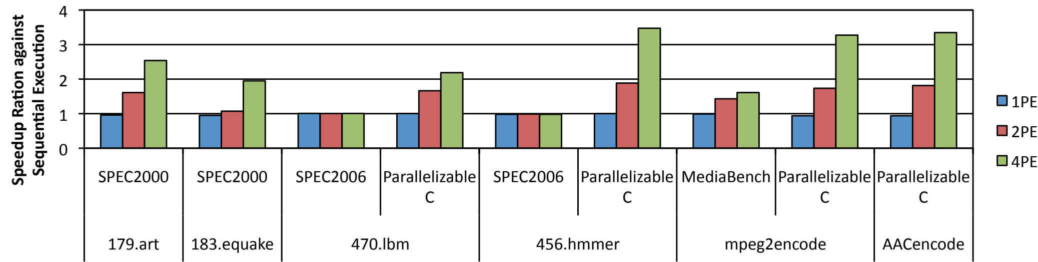

図 4.5: ルネサステクノロジ／日立製作所／早稲田大学 RP2 における OSCAR コンパイラによる自動並列化結果

4.6.5 OSCAR コンパイラによる自動並列化結果

本章で書き換えを行った art, equake, lbm, hmmer, mpeg2encode について，適用された自動並列化の概要を以下に示す．その際に，IBM XL コンパイラ，Intel コンパイラによる自動並列化結果についても併せて示す．

SPEC2000 art

art では，オリジナルコードにおいて，OSCAR コンパイラの自動並列化により IBM p5 550Q において 8 コア使用時に 4.96 倍の性能向上が得られている．art では実行時間の 95% 程度は train_match と match という 2 つの関数によって占められており，ともに処理の大部分は 7 つの並列化可能なループの収束演算によって占められている．OSCAR コンパイラではこれらのループを並列化することにより性能向上が実現された．

IBM XL コンパイラ，Intel コンパイラでは自動並列化による性能向上は得られなかった．

SPEC2000 equake

オリジナルコードでは IBM p5 550Q において 8 コア使用時に 1.69 倍と性能向上が限定的であったが，書き換えを行うことで 5.61 倍と大きな性能向上が得られるようになった．書き換えを行った smvp() のループでは，配列全体に対するリダクション計算が行なわれている．各プロセッサごとに対象の配列の一時配列を用意し，各プロセッサで配列の全要素について，部分和の計算結果を一時配列に格納する．その後，各プロセッサで部分和を格納した一時配列について，その総和を元の対象配列に格納することで，並列処理を実現している．このため，配列全体のコピーを行うオーバーヘッドにより逐次処理時間が 20%ほど増加しているが，プログラムの大部分が並列処理可能となり，プロセッサコア数に応じた性能向上が得られた．

XL コンパイラ，Intel コンパイラではオリジナルコード，Parallelizable C コードともに自動並列化による性能向上は得られなかった．

SPEC2006 lbm

オリジナルコードでは自動並列化による性能向上を得ることができなかったが，Parallelizable C に書き換えることで，OSCAR コンパイラによる自動並列化により IBM p5 550Q で 8 コア使用時に 5.36 倍の性能向上が得られた．

lbm では LBM_performStreamCollide という関数が実行時間の 90% 程度を占めており，この関数は単一の並列化可能なループで構成されている．オリジナルコードでは，この関数の呼び出し部を含むループにおいてポインタの更新が行われており (図 4.1)，このポインタ更新により，ポインタの指し先情報が曖昧になっていたことが並列性抽出の阻害要因となっていた．これを，Parallelizable C への書き換えによって，ポインタの更新を除去することにより並列性抽出が可能となり，大

きな性能向上が得られた．

IBM XL コンパイラでは自動並列化による性能向上は得られなかったが，Intel コンパイラではオリジナルコード，Parallelizable C コード共に自動並列化による性能向上が得られており，OSCAR コンパイラによる Parallelizable C コードの自動並列化時と同等の性能となっていた．

SPEC2006 hmmer

オリジナルコードでは自動並列化による性能向上を得ることができなかったが，Parallelizable C に書き換えることで，OSCAR コンパイラによる自動並列化により IBM p5 550Q で 8 コア使用時に 6.06 倍の性能向上が得られた．

456.hmmer の全プログラム実行時間の 90%以上を占めるループは図 4.6 で示される構造となっている．逐次ループ内部にイタレーション間の依存がある部分とない部分が存在するため，並列処理を行うためにはループのリストラクチャリングが必要となる．OSCAR コンパイラによってループディストリビューションを行い，データーローカライゼーション手法 [吉田 94] を適用した．このイメージを図 4.7 に示す．

この例では，まず図 4.6 のループに対してスカラエキスパンションが適用される．このとき，図 4.6 のランダムシーケンス生成部とシーケンス数値化部の関数戻り値はポインタであり，これらに対してスカラエキスパンションを適用するとポインタの配列となる．これらのポインタ配列の各要素間にエイリアスがないことが解析され，ループディストリビューションが適用される．その後，粗粒度タスク並列化におけるデーターローカライゼーション手法が適用され性能向上を得ている．

オリジナルコードに対しては図 4.6 のループ内部で，ヒープ領域に確保したバッファをイタレーションを超えて再利用しているため，ポインタ解析を利用しても並列性の抽出ができず，性能向上は得られなかった．

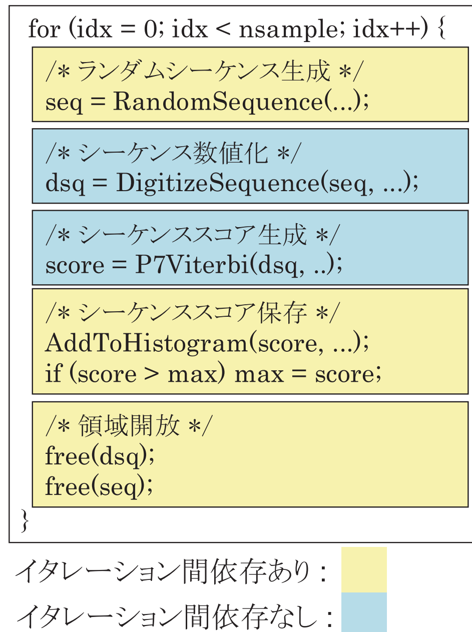

図 4.6: hmmer における主要処理ループ構造

IBM XL コンパイラ，Intel コンパイラによる自動並列化では，オリジナルコード，Parallelizable C コードともに性能向上は得られなかった．

MediaBench mpeg2encode

オリジナルコードでは，IBM p5 550Q で OSCAR コンパイラによる自動並列化により 4 コア使用時に 1.53 倍の性能向上が得られた．この性能向上は動き推定処理中の関数である frame_estimation() が粗粒度タスク並列処理された結果である．コンパイラにより解析された並列度は 2.5 となっており，3 プロセッサが割り当てられ並列処理が行われた．そのため，4 コア使用時と 8 コア使用時では性能の差が見られなかった．

Parallelizable C コードでは OSCAR コンパイラの自動並列化により 8 コア使用時に 5.13 倍の性能向上が得られた．これは，プログラムの書き換えにより，OS-

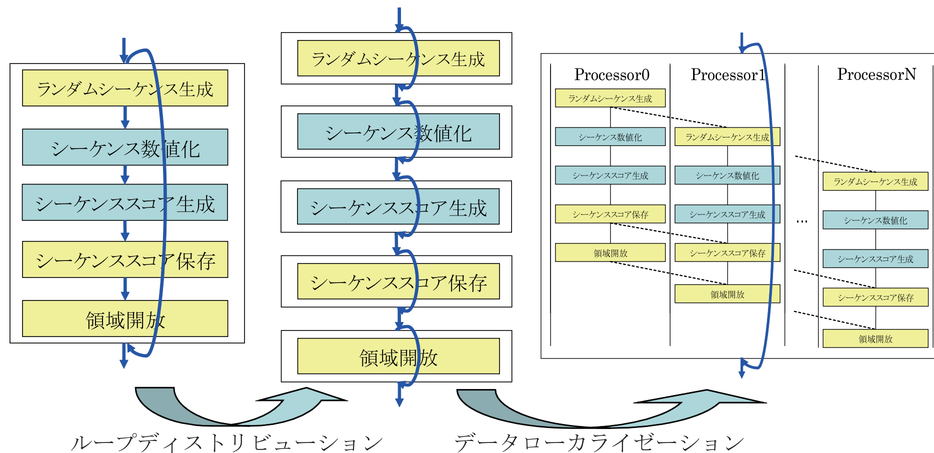

図 4.7: hmmer における主要処理ループのデータローカライゼーション

CAR コンパイラによるマクロブロックレベルの並列性とデータローカリティの抽出 [小高 05] が可能となったためである．

IBM XL コンパイラ，Intel コンパイラによる自動並列化では，オリジナルコード，Parallelizable C コードともに性能向上は得られなかった．

4.7 関連研究

並列ソフトウェア開発のためのプログラミングモデルは大きな課題となっており，並列プログラミングを容易化するための並列プログラミング言語の提案や，逐次プログラムから自動的に並列化するための逐次プログラミングモデルの改良が提案されている．

新しい並列プログラミング言語としては，ストリーム処理に特化した StreamIt[TKA02] や，ロックを用いずに並列性を記述可能であり PGAS(Partitioned Global Address Space) メモリモデルをサポートする X10[CGS+05]，階層的なメモリを対象とした並列性とデータローカリティをプログラム可能な Sequoia[FHK+06]，並列処理タスクの宣言的な記述とそのタスク内部処理の命令的な記述を可能とする Molatomium[高山 10]

等が提案されている．このような新しい並列プログラミング言語の導入により，並列ソフトウェア開発の生産性が飛躍的に向上する可能性がある．しかしながら，新しいプログラミング言語の開発現場への導入には，プログラマの教育や既存の資産の移植が必要であり，導入障壁が高くなってしまっている．そこで本論文では，既存の逐次プログラミング言語の記述方法のガイドラインを設定することで自動並列化の適用を可能とし，効果的な並列処理と多様なアーキテクチャへの移植性を実現する．本研究では実際に自動並列化による高い性能を得ることを目的として，自動並列化フレームワークを構築し，性能評価を行った．

自動並列化については，近年では高い精度のポインタ解析を効率的に行う手法が確立されてきており，それらを利用した自動並列化により，ユーザプログラマの並列ソフトウェア開発の負担を大きく低減できると考えられる．Ribeiro ら [RC07] は SPEC CINT2000，MediaBench の各 C プログラムに対して context-, flow-sensitive ポインタ解析を適用した際の解析精度の定量的な評価を行っており，その結果，C プログラムのポインタには静的な解析が不可能なものが含まれる一方で，十分に解析可能なプログラムが存在することを示唆した．Ryoo ら [RUR+07] はコンパイラによる解析で MPEG4 エンコーダのリファレンスプログラムに含まれる粗粒度並列性が抽出可能かどうか分析しており，インタープロシージャ配列解析, heap-sensitive, field-sensitive ポインタ解析，変数の取りうる値の範囲解析等を組み合わせることで並列性の抽出が可能と結論づけている．これらの研究成果は逐次の C プログラムの自動並列化の実現可能性を示唆しているが，実際に自動並列化による性能向上までは確認されていない．また，C 言語のあらゆる記述に対してポインタの指し先を静的に特定するのは事実上不可能 [Hin01] となっており，自動並列化コンパイラのようなツールを有効活用するためには，用途に応じてコーディングガイドラインにより記述の自由度を制限することが必要となる．

MISRA-C[MIS04] は主に車載ソフトウェア向けの高信頼性，高安全性，高移植

性を目的としたコーディングガイドラインであり，産業界で広く普及しておりコーディングガイドラインを遵守しているかどうかを確認するチェッカツールも市販されている．MISRA-C は組込みシステム向けの高信頼性のためのガイドラインであり，Parallelizable C とは用途が異なるが，共通するコーディングルールを含んでいる．

マルチプロセッサ向けの自動並列化においては，IMEC によるマルチプロセッサ向けのコーディングガイドラインである Clean C[Cle] が提案されている．Clean C は Parallelizable C と同様にマルチコア向けの自動並列化のためのガイドラインであり，共通するコーディングルールを含んでいるが，Parallelizable C のルールよりも抽象的なルールとなっている．また，Hwu ら [HRU+07] により assertion 等によるヒント情報を用いて自動並列化コンパイラを有効利用する暗黙的な並列プログラミングモデルが提案されており，逐次プログラミングのガイドラインと補完的に利用可能と考えられる．しかしながら，これらの研究では実際に自動並列化フレームワークを用いた評価は行われておらず，コンセプトの提案にとどまっている．そこで，本論文ではコーディングガイドラインとして Parallelizable C を提案するとともに，実際にプログラムの書き換えと様々なマルチコアプロセッサにおいて自動並列化結果の性能評価により有用性を示した．

Bridges ら [BVZ+07] は逐次プログラミングモデルを拡張し，逐次プログラムにおいて一定の幅のある計算結果を許容することで，並列プログラミング時と同様に並列性を利用することを提案している．実行時非決定性を導入することで，並列化のための自由度が向上する一方で，並列プログラミングと同様に結果の再現性がなくなりデバッグが難しくなると考えられる．そのため，本研究では逐次プログラミングモデルの拡張は行わず，既存のプログラミング言語の仕様を制限するコーディングガイドラインを導入することで自動並列化を行う．

スレッドレベル投機実行はハードウェアサポートを利用して，従来のハードウェ

アの想定よりも多くの並列性を利用するものであり，Bridges ら [BVZ+07] の研究においても TLS のハードウェアサポートを想定していた．しかしながら，現在のハードウェアでは TLS の実現は困難であり，本論文では既存の一般的なマルチコアプロセッサで実現可能な自動並列化を行う．Prabhu ら [PO05] は TLS 向けに手動でプログラミングを行い，TLS 向けのプログラミングガイドラインを提案しているが，その内容には Parallelizable C 等のガイドラインとの類似性が見られる．このプログラミングガイドラインも静的なコンパイラで解析できるものがよいという結論となっており，TLS のようなハードウェアサポートを利用する際にも Parallelizable C のガイドラインが有効と考えられる．

4.8 まとめ

本章ではコンパイラによる自動並列化を可能とするための C 言語の記述方法として Parallelizable C を提案した．Prallelizable C で記述した科学技術計算およびマルチメディア処理の逐次プログラムに対して自動並列化を適用し，マルチコアシステム上での処理性能の評価を行った．その結果，2 コア集積のマルチコアである IBM Power5+ を 4 基搭載した 8 コア構成のサーバである IBM p5 550Q において逐次実行時と比較して平均 5.54 倍，4 コア集積のマルチコアである Intel Core i7 920 プロセッサを搭載した PC において平均 2.43 倍，SH-4A コアを集積した情報家電用マルチコア RP2 の 4 コアを使用した SMP 実行モードにおいて平均 2.78 倍の性能向上が得られた．

本章で提案した Parallelizable C と OSCAR コンパイラのような自動並列化コンパイラを組み合わせることで，マルチコア向けソフトウェア開発におけるソフトウェア生産性の向上を実現し，今後のマルチコア・メニーコア技術の研究開発の促進につながると考えられる．

第 5 章

低消費電力化制御

5.1 はじめに

半導体集積度向上に伴ったスケーラブルな処理性能と高い電力効率を実現するために，マルチコアプロセッサはプロセッサアーキテクチャの主流となりつつある．これらマルチコアプロセッサの有効利用には実行するプログラムの適切な並列化が必須であり，従来より多くの自動並列化コンパイラに関する研究が行われている [Wol96, HAA+96, EHP98]．また，マルチコアでは処理性能の向上に加え，増加する消費電力をいかに抑えるかが大きな課題となっており，プロセッサの周波数・電圧制御や電源・クロック遮断の機能を利用して，実行時に低消費電力化を適用する手法 [ABD+03, WJMC04, LMH04] や，コンパイル時に低消費電力化を適用する手法 [HK03, 白子 06] が提案されている．

OSCAR コンパイラ [本多 90, 笠原 03] では，従来の市販コンパイラで用いられていたループ並列処理に加え，粗粒度タスク並列処理，近細粒度並列処理を組合せたマルチグレイン並列処理を実現しており，プログラム全域にわたる並列化が可能である．その際，マルチグレイン並列処理による並列性抽出に加え，プログラムの各部分に対するコアごとの適切な動作周波数/電圧および電源遮断のタイミングを決定し，必要な処理性能を維持しつつ電力を低減させている [白子 06]．これらの OSCAR コンパイラによる並列化及び低消費電力化を様々な種類のマルチコアに適用する目的で OSCAR API[OSC] が提案されている．OSCAR API は NEDO “リアルタイム情報家電用マルチコア技術” プロジェクトにおいて，早稲田大学，東芝，日本電気，パナソニック，日立製作所，富士通研究所，ルネサステクノロジによって開発された情報家電用マルチコア向け並列化 API である．OSCAR API を用いることで，OSCAR コンパイラによる最適化を様々なマルチコアで実現することができ，並列プログラムの大幅な生産性向上が期待できる．

OSCAR コンパイラの並列処理性能については従来より様々なマルチプロセッサシステムにおいて評価されてきた [笠原 03]．本研究では OSCAR コンパイラによ

る低消費電力化手法をさらに発展させ，実マルチコア上で動作するフレームワークを構築し，消費電力削減効果を実測により評価を行った．

本章では，OSCAR コンパイラによって従来実現されてきた低消費電力化手法に加えて，第 5.3.2 節で述べる実行時のコスト変動によるプロセッサ間の負荷不均衡に対する低消費電力化手法，および第 5.3.3 節で述べるリアルタイムアプリケーションを対象とした低消費電力化手法を提案する．これら OSCAR コンパイラによる低消費電力化手法を OSCAR API を用いて実現し，8 コアの情報家電用マルチコア RP2[IHY+08] 上で評価を行った．コンパイラによるマルチコアの電力制御を実マルチコア上で実現し評価を行ったのは筆者等の知る限り本研究が世界で初めてである．

本章の構成は以下の通りである．第 5.2 節では提案する OSCAR コンパイラによる低消費電力化手法の前提となるマルチグレイン並列処理について述べる．第 5.3 節では OSCAR コンパイラによる低消費電力化手法について述べる．第 5.4 節では OSCAR API における電力制御用指示文について述べ，第 5.5 節で性能評価について述べる．そして，第 5.6 節で関連研究について述べ，第 5.7 節でまとめを述べる．

5.2 マルチグレイン並列処理

本節では OSCAR コンパイラによるマルチグレイン並列処理の概要を述べる．

5.2.1 マクロタスク生成

マルチグレイン並列処理では，プログラムは基本ブロックおよびその融合ブロック BPA，ループなどの繰り返しブロック RB，サブルーチン，関数を表す SB の 3 種類のマクロタスク MT [笠原 03] すなわち粗粒度タスクに分割される．RB や SB の内部は再帰的にマクロタスク分割され，階層的なマクロタスク構造となる．

5.2.2 マクロタスクグラフ生成

各階層におけるマクロタスク間のデータ依存，制御フローをもとに最早実行可能条件解析 [本多 90, 笠原 03] を適用し，マクロタスク間の並列性を抽出したマクロタスクグラフ MTG を生成する．マクロタスクグラフも入れ子構造をとり，プログラム全域にわたる階層的な粗粒度タスク並列性が抽出される．

5.2.3 マクロタスクスケジューリング

階層的に定義されたマクロタスクは，プロセッサの集合であるプロセッサグループ PG により実行される．適切な PG のグルーピングとタスクスケジューリングにより，階層的な並列性が有効利用される．この際，並列ループは複数の小ループに分割され，粗粒度タスクとしてプロセッサに割り当てられる．

5.3 OSCAR コンパイラによる低消費電力化手法

マルチグレイン並列処理によりプログラムの最大限の並列化が可能であるが，実プログラムの完全な並列化は一般的に難しく，同期やデータ送受信によりプロセッサがアイドル状態となる可能性がある．また，リアルタイム処理においてタスク実行が早く終了した場合，デッドライン時刻までの待機時間が発生する．これらプログラム中のアイドル時間（余裕度）を利用しプロセッサの動作周波数低減や電源遮断を適用することにより，効果的な電力削減が可能となる．

本節では OSCAR コンパイラによる低消費電力化手法について述べる．本手法は以下のアーキテクチャサポートを持つマルチコア上で最適に利用可能である．

- プロセッサコア毎に周波数が可変
- 周波数に応じて電圧も低減可能

- コア毎の電源/クロック遮断が可能

5.3.1 コンパイル時における低消費電力化手法

本節では，コンパイル時の情報をもとに静的に適用される低消費電力化手法の概要を述べる．本手法はプログラムの最小処理時間を維持するための最速実行モード，与えられたデッドライン制約の範囲内における電力消費量最小化を目指すデッドライン制約モードの 2 つの実行モードを持つ．マクロタスクグラフに対する一般的な低消費電力化手法の手順は以下の通りである．

(1) マクロタスクグラフの処理時間の算出

(2) 各マクロタスクにおける動作周波数の決定

(3) アイドル時間に対する電源遮断

マクロタスクグラフの処理時間

マルチグレイン並列処理により，マクロタスク MT は各プロセッサグループ PG に適切に割り当てられる．図 5.1 に示す例では，3 つの PG に対し MT がスケジューリングされ，ユーザによって指定されたデッドライン時間が定められている．図中の time が時間経過を表しており，周波数が最速の場合の 1 クロックを単位時間とする．また，MT 間の実線はデータ依存を表している．当該 MTG の実行終了時間 T_{MTG} を表すため，以下の定義を行う．マクロタスク MT_i に対して，

T_i : MT_i の処理時間

T_{start_i} : MT_i の実行開始時刻

T_{end_i} : MT_i の実行終了時刻

を定義する．また，当該 MTG の開始時刻を 0 とする．図 5.1 の入口ノードであ

る MT_1 のように，当該 PG が最初に実行する MT であり，かつ他の MT にデータ依存しない MT_i の実行開始時刻 T_{start_i} は

$T_{start_i} = 0$

となり，実行終了時刻は

$T_{end_i} = T_{start_i} + T_i = T_i$

となる．一方それ以外の MT_i に関しては，当該 PG が先行して実行するマクロタスク MT_j と MT_i がデータ依存するマクロタスク集合 $\{MT_k, MT_l, ...\}$ が終了した時点で MT_i の実行が開始されるため，実行開始時刻は

$T_{start_i} = max(T_{end_j}, T_{end_k}, T_{end_l}, ...)$

となり，終了時刻は

$T_{end_i} = T_{start_i} + T_i$

となる．図 5.1 を例として考えると MT_2，MT_3 は MT_1 が終了した時点で実行されるため開始時刻は

$T_{start_2} = T_{start_3} = T_{end_1} = T_1$ となり，終了時刻は

$T_{end_2} = T_{start_2} + T_2 = T_1 + T_2$,

$T_{end_3} = T_{start_3} + T_3 = T_1 + T_3$ である．また，MT_6 は MT_2 と MT_3 が終了した時点で実行が開始されるため

$T_{start_6} = max(T_{end_2}, T_{end_3})$ より，

$T_{start_6} = max(T_2, T_3) + T_1$

となる．同様に計算し，出口ノードである MT_8 の終了時刻は

$T_{end_8} = T_1 + T_8 + max(T_2 + T_5, T_6 + max(T_2, T_3), T_7 + max(T_3, T_4))$ と表される．

ここで一般形を考えると，出口ノード MT_{exit} の終了時刻は

$T_{end_{exit}} = T_m + T_n + ... + max_1(...) + max_2(...) + ...$

と表記される．入口ノードの実行開始時刻を 0 としたため，この $T_{end_{exit}}$ が当該 MTG の実行終了時間 T_{MTG} となる．

各マクロタスクにおける動作周波数の決定

MTG に対して与えられたデッドライン時間を $T_{MTG_deadline}$，全 MT を最速の周波数で実行した場合の処理時間を T_{MTG_fast} と定義する．T_{MTG_fast} は T_{MTG} の式を用いて算出される．これらの定義より，2 つの実行モードにおける制約条件は以下のようになる．

- 最速実行モード: $T_{MTG} = T_{MTG_fast}$
- デッドライン制約モード: $T_{MTG} \leq T_{MTG_deadline}$

本手法では，それぞれの制約条件を満たしつつ全体の電力消費量が最小となるよう各 MT の動作周波数を決定する [白子 06].

アイドル時間に対する電源遮断

図 5.2 に示すように，各 MT の動作周波数決定後も同期待ちなどによりプロセッサがアイドル状態となる場合がある．ただし，図 5.2 中の MID は最速周波数の半分の動作周波数で当該 MT を実行することを表す．本手法ではアイドル状態にあるプロセッサに対して，電源遮断およびクロックゲーティングを適用することでアイドル中の無駄な電力消費を削減する [白子 06].

5.3.2 実行時負荷不均衡に対する低消費電力化手法

本節では，実行時に生じうるプロセッサ間の負荷不均衡に対する低消費電力化手法を提案する．コンパイル時に各プロセッサに処理が均等に割り当てられたとしても，プログラムの入力などに依存する処理の不確定性により，実行時にプロセッサ間で負荷の不均衡が起こる可能性がある．特に，ループボディに条件分岐を含む並列ループを分割し各プロセッサに割り当てた場合においてそのような負

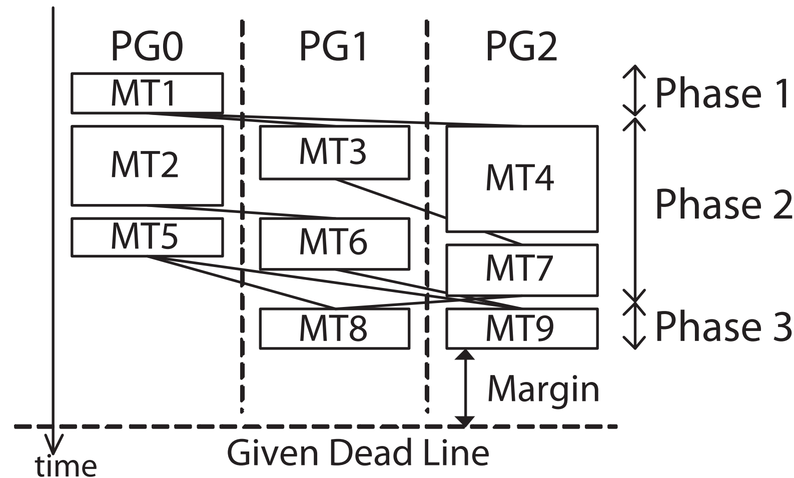

図 5.1: マクロタスクのスケジューリング結果の例

荷不均衡は発生しやすく，提案手法ではそのような並列ループをコンパイル時に検出し，電力制御命令をプログラム中に挿入することで，実行時の負荷不均衡により生じるアイドル時間に対して電源遮断を行い，無駄な電力消費を削減する．

あるループを本手法の制御対象とする必要条件は以下の通りである．

- 複数のプロセッサによって並列処理される
- ループボディに条件分岐が存在する，または回転数が変動する内部ループを持つ
- 電力制御オーバヘッドと比較して処理時間が十分に大きい

上記の条件を満たす実行時負荷不均衡ループを含む MTG 中において，負荷不均衡ループを実行した後に，プロセッサ間でバリア処理が必要な場合にビジーウェイトによって消費される電力を削減するため，先行してバリアに到達したプロセッサに対して電源遮断を適用する．ただし,「電力制御オーバヘッドと比較して処理時間が十分に大きい」とは，並列ループの逐次実行時のコンパイラによる推定処理時間を Cost，当該並列ループを処理するプロセッサ数を PE，電源遮断状態から

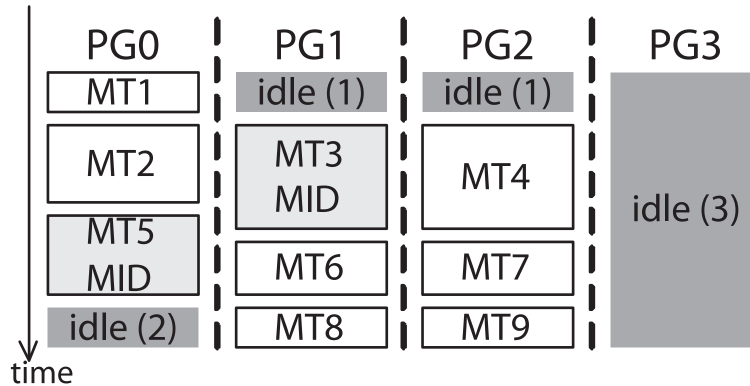

図 5.2: 周波数・電圧制御の結果の例

最速の電力状態に復帰するためのオーバヘッドを $Overhead_{poweroff}$，電力制御オーバヘッドに対するループのコスト比を Ratio としたとき，以下の式を満たすことを言う．

$$Overhead_{poweroff} \times Ratio \leq Cost/PE$$

なお，第 5.5 節の RP2 上での評価においては $Overhead_{poweroff}$ には実測値の 100[μs] を用い，Ratio は 100 とした．

提案手法では，カウンティングセマフォアを用いることで最後にバリアに到達した PG を実行時に判定し，以下の処理に移る．

- 最終 PG: 電源遮断中の他 PG を復帰
- 他 PG: 電源遮断状態に移行

図 5.3 に負荷不均衡のある並列ループに対する本手法の適用例を示す．この例では，負荷不均衡により PG3，PG2，PG1 の順にバリアに到達し電源遮断状態に移行している．最後にバリアに到達した PG0 により PG1〜PG3 に対して電源復帰命令が発行される．

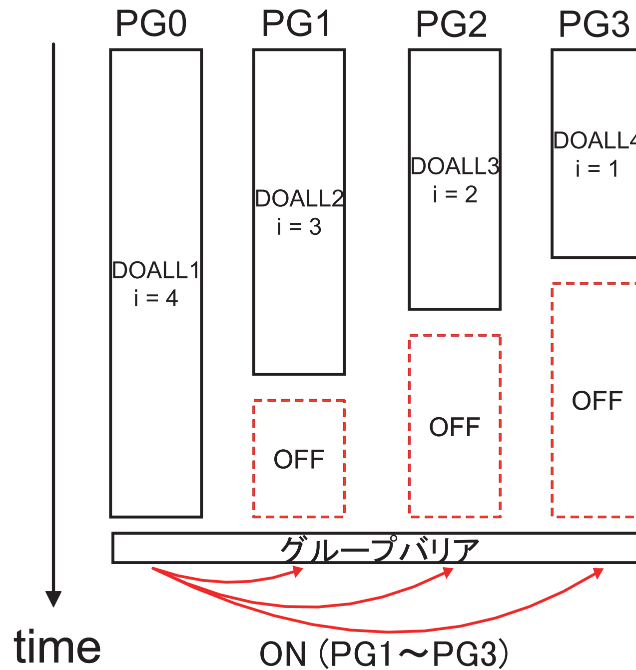

図 5.3: 負荷不均衡のある並列ループに対する低消費電力化手法の適用例

5.3.3　デッドライン時刻までの待機電力最小化

第 5.3.1 節ではデッドライン制約に応じて各 MT の処理周波数を適切に低減させる手法について述べた．しかしデッドライン時間が十分に大きい場合，全 MT を最低周波数に指定した場合でもデッドライン時刻よりも早く処理が終了することがある．特にメディアアプリケーションなどの周期的なリアルタイム処理においては，全 MT の処理終了後にプログラムへの入力待ち等による待機時間が発生する可能性がある．本節では，このような制約時刻までの待機を伴うデッドライン制約モードをリアルタイム制約モードと定義し，待機時間中の不要な電力消費を最小化する手法を提案する．本手法は第 5.4 節で述べるプログラム中での現在時刻を取得するタイマ API を用いる．

制御対象階層の検出

提案手法では，ユーザにより与えられたデッドライン時間と第 5.3.1 節の方法によってコンパイル時に推定した周波数制御後の MTG の処理時間を比較し，当該 MTG における待機時間 T_{wait} を算出する．なお，周波数制御後の各 MT の処理時間は，その周波数に応じた比率 (50%の周波数なら 2 倍，25%の周波数なら 4 倍) を MT の処理時間にかけて推定する．動作周波数を低減してもバスやメモリの速度は変わらないため，動作周波数を半分にしても実際には処理時間は 2 倍にならないこともあるが，本手法では最大の遅延を考慮することで，デッドライン時刻を超過するのを防いでいる．

また，当該 MTG に割り当てられた全プロセッサが電源遮断と動作状態への復帰を行うためのオーバーヘッドを T_{OH_power}，同様にクロックの遮断・復帰に必要なオーバーヘッドを T_{OH_clock} とする．これら MT 処理および電源／クロック遮断オーバーヘッドの推定では，対象マルチコア上でのプロファイル情報を用いることで精度を高めている [白子 06]．提案手法ではこれらのパラメータを比較し，以下の判断基準で電力制御を適用する．

- $T_{wait} \leq T_{OH_clock}$: 制御なし
- $T_{OH_clock} < T_{wait} \leq T_{OH_power}$: クロック遮断
- $T_{OH_power} < T_{wait}$: 電源遮断

タイマを用いた実行時デッドライン管理

当該 MTG がクロック／電源遮断の適用階層と判断された場合，その MTG での最後の MT を処理するプロセッサ（PG 内の先頭プロセッサ）がマスタプロセッサに指定される．マスタプロセッサ以外のプロセッサは，自身に割り当てられた最後の MT の終了後に電源／クロック遮断状態に移行する．マスタプロセッサは

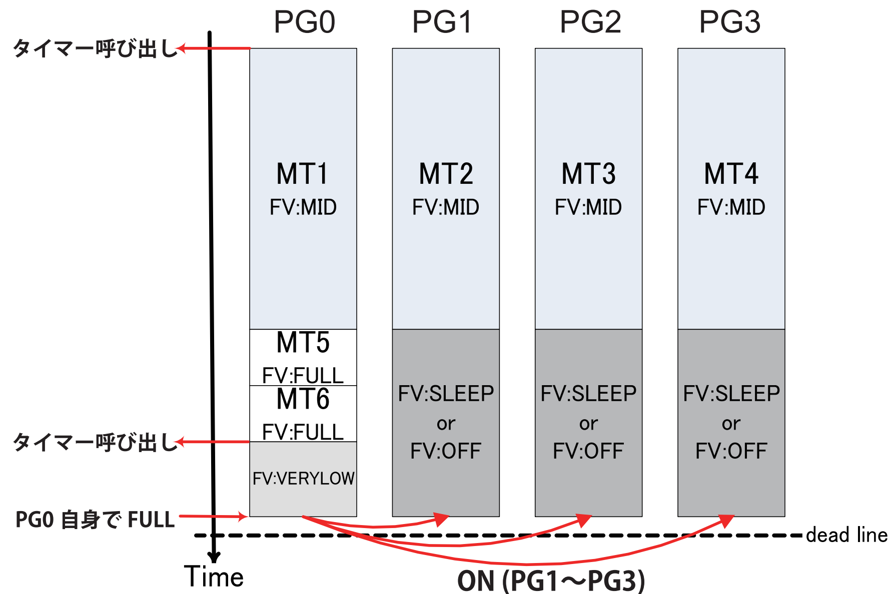

図 5.4: リアルタイム制約モードの適用例

自身に割り当てられた先頭 MT の開始時刻をタイマを用いて取得し，最後の MT の終了時に最低動作周波数に移行後，タイマ API を用いてデッドラインまで待機する．デッドラインに到達後，電源／クロック遮断状態である他のプロセッサを復帰させ次の処理に移る．

図 5.4 に提案手法の適用例を示す．ここではコア 0（PG0 内先頭プロセッサ）がマスタプロセッサに選択され，最後の MT6 終了後にデッドライン時刻まで最低の消費電力状態で待機している．

5.4 OSCAR API における電力制御用指示文

本節では提案手法を実装した OSCAR コンパイラが出力する OSCAR API における電力制御用の指示文について述べる OSCAR API は OpenMP をベースとした API で，並列実行，メモリ配置，データ転送，電力制御，タイマ，同期の 6 つのカテゴリ，15 の指示文で構成される [OSC].

```
#pragma oscar get_fvstatus(pe_no, module_id, fv_state)
#pragma oscar fvcontrol(pe_no, (module_id, fv_state))
#pragma oscar get_current_time(current_time, timer_no)
```

図 5.5: 本手法で用いた OSCAR API

5.4.1 電力制御およびリアルタイム処理向け指示文

本章では OSCAR API のうち，図 5.5 に示す 3 つの API を用いて，情報家電用マルチコア RP2 上での電力制御を実現した．

指示文 get_fvstatus によって，pe_no 番目のプロセッサの，module_id で示されたモジュールの電力状態を変数 fv_state に取得することができる．モジュールとしては，CPU，キャッシュあるいは CPU とキャッシュを含むコア全体など，マルチコアを構成する各種要素を指定できる．本手法ではこの指示文を用いて，状態遷移中の電源復帰命令発行の防止を実現した．また，指示文 fvcontrol によって，pe_no 番目のプロセッサの，module_id で示されたモジュールの電力状態を変数 fv_state で指定した電力状態に変更することができる．また，指示文 get_current_time によって，timer_no 番目のタイマーから現在時刻を取得し，変数 current_time に取得することができる．この指示文を用いて，第 5.3.3 節で述べたリアルタイム制約モードでの低消費電力化手法を情報家電用マルチコア RP2 上で実現した．

5.5 性能評価

第 5.3 節で述べた低消費電力化手法を実装した OSCAR コンパイラの性能評価を情報家電用マルチコア RP2 上で行った．OSCAR コンパイラにより自動生成したプログラムを RP2 用のコード生成コンパイラでコンパイルし，RP2 上で実行した際の RP2 チップの電流・電圧の実測値より，消費電力の評価を行った．

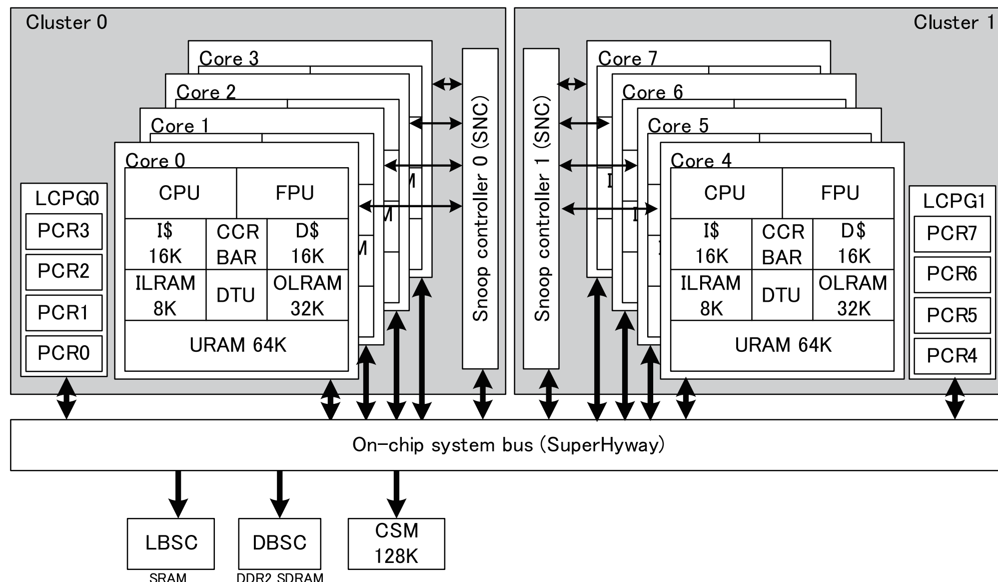

図 5.6: RP2 の構成図

5.5.1　情報家電用マルチコア RP2

本節では，提案手法の評価に用いた RP2 について述べる．RP2 は NEDO の “リアルタイム情報家電用マルチコア技術” プロジェクトにおいて，早稲田大学，ルネサステクノロジ，日立製作所によって共同開発された情報家電向けマルチコアプロセッサで，一つのチップ上に 8 つの SH-4A コアを搭載している．RP2 の構成図を図 5.6 に示す．各プロセッサコアには，CPU，キャッシュ，ローカルメモリ（命令メモリ ILRAM，データメモリ OLRAM），分散共有メモリ（URAM），データ転送ユニット（DTU）が搭載されており，4 コアまではスヌープコントローラが MESI プロトコルでキャッシュコヒーレンシを保証する SMP モードとして動作する．5 コア以上を使用する場合は，ソフトウェアが明示的にキャッシュコヒーレンシを保証する必要がある．

RP2 では低消費電力機能として，各コアの周波数を独立に 600MHz，300MHz，150MHz，75MHz に実行時に切り替えることが可能である．それに加えて，コア

表 5.1: RP2 の電力状態

状態名	クロック遮断対象モジュール	電源遮断対象モジュール	消費電力 [W]
FULL(600MHz, 1.404V)	なし	なし	5.29
MID(300MHz, 1.196V)	なし	なし	2.44
LOW(150MHz, 1.004V)	なし	なし	1.22
VERYLOW(75MHz, 1.004V)	なし	なし	0.99
Light Sleep(1.004V)	CPU	なし	1.096
Normal Sleep(1.004V)	CPU, cache, ILRAM, OLRAM	なし	0.743
Resume Standby(1.004V)	URAM	CPU, cache, ILRAM, OLRAM, DTU	0.566
CPU off(1.004V)	なし	CPU, cache, ILRAM, OLRAM, DTU, URAM	0.554

中の CPU のみのクロックを遮断する Light Sleep，コア中の URAM と DTU 以外のクロックを遮断する Normal Sleep，コア中の URAM のクロックを遮断，それ以外を電源遮断する Resume Standby，コアの電源をすべて遮断する CPU off の 4 つの低電力状態を各コア独立して取ることが可能である．動作状態への復帰は他のコアからの割り込み処理によって行われる．

また，供給電圧はチップ一括での変更が可能であり，全コアのうちで最大の周波数で動作しているコアの動作周波数によって決定される．最大の動作周波数が 600MHz の場合は 1.404V，300MHz の場合は 1.196V，150MHz 以下およびクロックまたは電源遮断状態の場合は 1.004V となる．電圧の設定はランタイムライブラリによって適切に行われる．

状態遷移のオーバヘッドは周波数変更とクロック遮断は瞬時に可能であり，電源遮断は 5μs，電源復帰には 30μs で可能である．

表 5.1 に RP2 の持つ電力状態と，8 つのプロセッサコアを各電力状態にした場合の消費電力を示す．

5.5.2 消費電力評価条件

本節では，RP2 チップにおける電流および電圧を実測することにより消費電力を評価した．この際，消費電力については室温で測定した．

電流についてはRP2 評価ボードの電流測定用の端子に銅線を接続し，チップに流れる電流を非接触式の電流プローブを用いて実測した．電圧についても RP2 評価ボードの電圧測定用の端子に電圧計のピンを接続して実測した．電流および電圧の計測は 100μs 間隔で行い，実測した電流および電圧の値からその時刻における平均消費電力を算出し，時間で積算することでプログラム全体の消費エネルギーを算出した．また，プログラム全体の平均消費電力は各時刻における平均電力の平均により算出した．

RP2 チップは 8 つのプロセッサコアを搭載している．そのため，8 コア未満のプロセッサ構成について評価を行う場合は，使用していないコアについては電源遮断 (CPU off) 状態として評価を行った．

また，各評価における電力の計測は，プログラムの実行が開始してから終了するまでの消費電力を計測した．最速実行モードでは，使用するコア数が増加することで実行時間が短縮されるが，本評価では短縮された実行時間までの消費電力について評価を行っている．

5.5.3 最速実行モードの評価

最速実行モードは第 5.3.1 節で述べたコンパイル時における低消費電力化手法と第 5.3.2 節で述べた実行時負荷不均衡に対する低消費電力化手法を用いており，本節ではこれらの手法をまとめて本手法と呼ぶ．

SPEC2000 の art，equake 及び MediaBench の MPEG2 デコーダを制約付き C[間瀬 09] に書き換えたプログラムとルネサステクノロジ提供の AAC エンコーダを用いて，

RP2 上で最速実行モードでの電力評価を行なった．なお，MPEG2 デコーダの実行条件として，画面サイズは 352 × 240，フレームレートは 30[fps]，ビットレートは 5000[kbps] のものを用いた．AAC エンコーダについては，16bit ステレオ，ビットレート 128[kbps] のものを用いた．評価には 4 コアの SMP モードを用い，使用しないプロセッサはプログラム開始時に CPU off を適用した．また，電源制御には Resume Standby を使用した．

4 コアでの並列化による速度向上としては，1 コアを用いた場合と比較して，art にて 2.49 倍，equake にて 2.67 倍，AAC エンコーダにて 3.29 倍，MPEG2 デコーダにて 2.67 倍の速度向上が得られた．

本手法を適用した場合と適用しない場合の，実行時間と RP2 チップ全体の消費エネルギーを比較した．ただし，データの入出力部分は評価から除いている．まず実行時間を図 5.7 に示す．図の横軸が各アプリケーション，縦軸が実行時間を表しており，各アプリケーションのバーは左から，1 コアを用いた場合，4 コアを用いて本手法を適用しなかった場合，4 コアを用いて本手法を適用した場合を表している．4 コアを用いた場合における本手法適用による遅延率は art にて 0.0042%，equake にて 0.62%，AAC エンコーダにて 0.11%，MPEG2 デコーダにて 2.22%であった．これらの遅延は第 5.3.2 節で述べた実行時負荷不均衡に対する低消費電力化手法によるものである．

次に消費エネルギーを図 5.8 に示す．本手法による消費エネルギー削減効果は art において 13.05%（1795.32[J] から 1560.92[J]），equake で 3.99%（3071.64[J] から 2949.07[J]），AAC エンコーダで 3.84%（3.03[J] から 2.91[J]），MPEG2 デコーダで 9.01%（25.68[J] から 23.37[J]）となった．

ここで最速実行モードにおいて，第 5.3.2 節で述べた実行時負荷不均衡並列ループに対する制御が特に有効であった art について詳しく考察する．art の処理時間の約 6 割を占める並列ループにおいて，最も処理が早く終了したプロセッサと最

も遅く処理が終了したプロセッサでは，平均で 51.1%の時間差があった．この並列ループは，コンパイル時には均等に分割され各プロセッサに処理が割り当てられたが，プログラムの入力によって並列ループの内側のループの回転数が決定されるため，実行時にプロセッサ間で負荷不均衡が生じた．本手法ではこのループを含む MTG に対して電力制御が適用された．

art を実行した際の電力波形の抜粋を図 5.9 に示す．図の横軸が時刻，縦軸が消費電力を表している．また，縦の破線に囲まれた領域がループの 1 周期を表している．図 5.9 で電力が低減しているのは，前述したループを含む MTG に対する電力制御が寄与している．図 5.9 の電力制御が適用された箇所の波形を見ると，まず 2.25[W] 付近まで電力が下がり（a），次に 1.9[W] 付近まで電力が下がっていることが分かる（b）．これは，処理時間の約 6 割を占めるループを実行する際，各プロセッサが処理終了後にバリアに到達した順に電力制御が適用されたためである．

次に，本手法で発生した遅延率について考察する．本手法では，対象 MTG のバリア同期の直前のタイミングで電源制御を適用していたため，制御オーバーヘッドに応じた遅延が生じていた．最も遅延率が大きかった MPEG2 デコーダにおいては，9.01%の消費エネルギーの削減効果が得られた一方，遅延率が 2.22%となっている．最速実行モードにおいては遅延が発生しないことが理想となるため，遅延率を低減させる必要があると考えられる．遅延を生じさせずに電力制御を行うためには，制御オーバーヘッドを考慮して電源制御をバリア同期よりも一定時間前のタイミングで行う必要があり，このような制御の最適化は今後の課題である．

5.5.4 リアルタイム制約モードの評価

リアルタイム制約モードは第 5.3.1 節で述べたコンパイル時における低消費電力化手法と第 5.3.2 節で述べた実行時負荷不均衡に対する低消費電力化手法と第 5.3.3 節で述べたデッドライン時刻までの待機電力最小化手法を用いており，周期的なリ

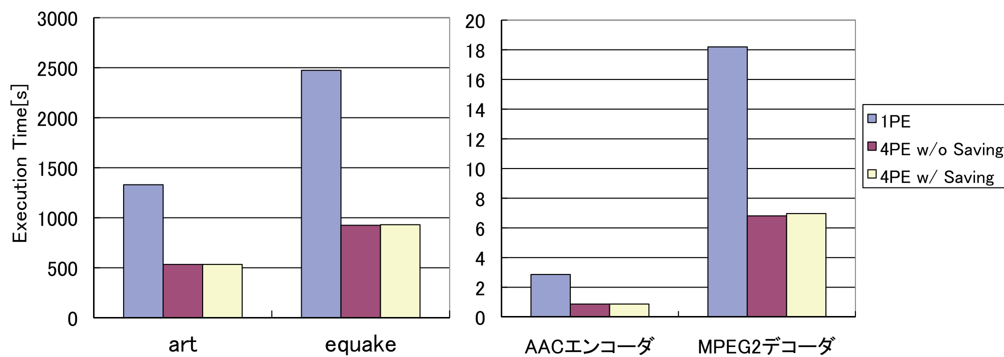

図 5.7: 最速実行モードにおける実行時間

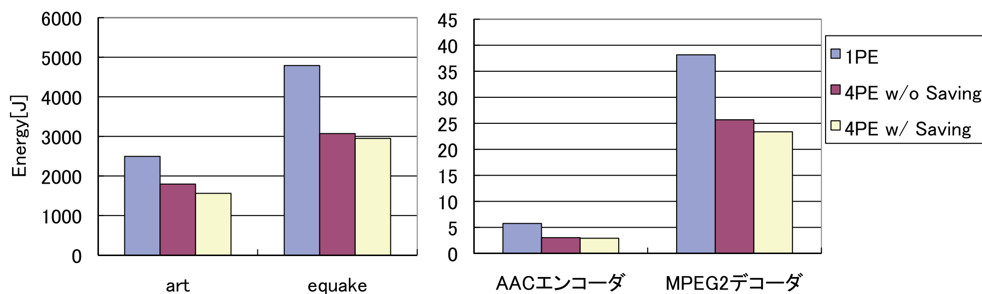

図 5.8: 最速実行モードにおける消費エネルギー

アルタイムデッドラインを持つようなプログラムを対象としたモードである．本節ではこれらの手法をまとめて本手法と呼ぶ．

ルネサステクノロジ提供の AAC エンコーダと，MediaBench の MPEG2 デコーダを制約付き C に書き換えたプログラムを用いて，RP2 上でリアルタイム制約モードでの電力評価を行なった．AAC エンコーダにおけるリアルタイム制約は 44.1[フレーム/s]，MPEG2 デコーダにおけるリアルタイム制約は 30[フレーム/s] とした．なお，評価に先立ち，RP2 上でこれらのプログラムを実行したプロファイル情報（プログラム中のループや関数の実行時間や実行回数の情報）をコンパイラに与え，

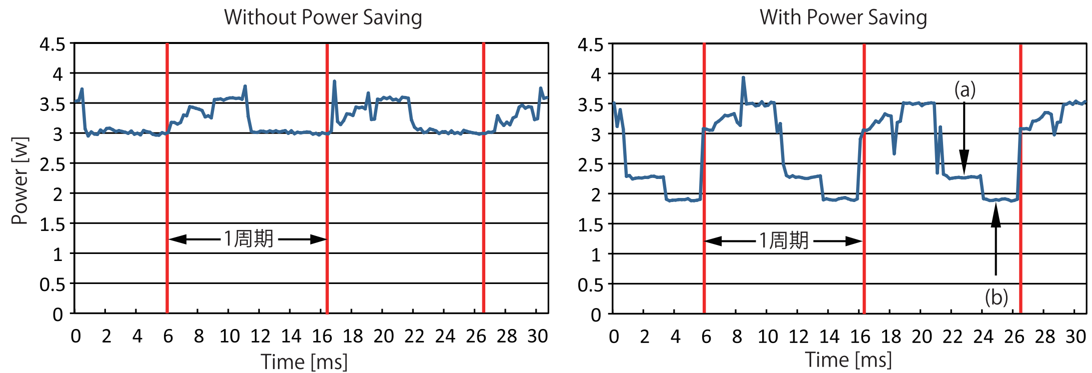

図 5.9: art の電力波形

各マクロタスクのコスト見積もり精度を高めている．

本手法を適用した場合と適用しない場合の平均電力を図 5.10 に示す．図 5.10 の各アプリケーションの左側の 2 つのバーは，リアルタイム制約を満たせる最少のコア数で実行した際の，本手法適用時と非適用時の平均電力を示している．AAC エンコーダにおいては 1 コア，MPEG2 デコーダにおいては 2 コアでリアルタイム制約を満たすことができた．右側の 2 つのバーは 8 コアを用いて実行した際の，本手法適用時と非適用時の平均電力を示している．各コア数における本手法による電力削減効果は，AAC エンコーダにおいて 1 コア使用時に 66.0%（1.88[W] から 0.64[W]），8 コア使用時に 87.9%（5.40[W] から 0.65[W]）であり，MPEG2 デコーダにおいて 2 コア使用時に 20.1%（2.44[W] から 1.95[W]），8 コア使用時に 76.0%（5.41[W] から 1.30[W]）であった．

AAC エンコーダおよび MPEG2 デコーダを 8 コアを用いて実行した際の電力波形の抜粋をそれぞれ図 5.11 と図 5.12 に示す．図の横軸が時刻，縦軸が電力を表す．AAC エンコーダにおいては，リアルタイムデッドラインまでの余裕度が十分に大きいため，エンコード処理は LOW の状態で行なわれ，その後デッドラインまで VERY LOW 及び Resume Standby が適用された．MPEG2 デコーダにおいては，リアルタイムデッドラインまでの余裕度が小さいため，主要のデコード処理につ

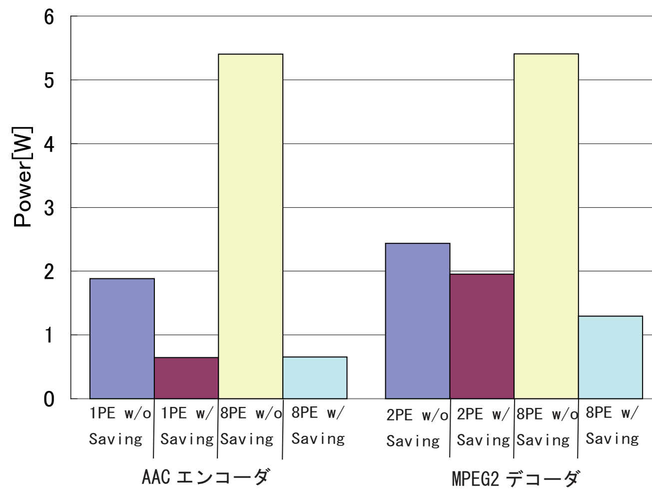

図 5.10: リアルタイム制約モードでの平均電力

いては LOW の状態で 8 コアで並列処理されたが（a），他の処理については FULL の状態と電源遮断が併用して適用された．

リアルタイム制約を満たせる最少のコア数を用いて本手法を適用した場合と，8 コアを用いて本手法を適用した場合の平均電力を比較する．AAC エンコーダにおいては，1 コアで実行した場合でもデッドラインまでの余裕度が十分にあるため、8 コアを用いた場合と同様にエンコード処理が LOW の状態で実行された．8 コアを用いた場合，同期や電源復帰のオーバヘッドが 1 コアを用いた場合よりも大きいため，1 コアを用いた場合の方が 8 コアを用いた場合よりも 1.4%平均電力が低くなった．一方，MPEG2 デコーダにおいてはデッドラインまでの余裕度が小さいため、1 コアでのリアルタイム実行は不可能で，2 コアを用いた場合でも，主要のデコード処理を最速（FULL）でない周波数状態で実行するとリアルタイム制約が満たせなくなってしまう．そのため，2 コアを用いた場合は主要のデコード処理を FULL で実行後，デッドラインまで VERY LOW 及び Resume Standby が適用された．そのため、8 コアを用いた場合の方が 2 コアを用いた場合よりも 33.7%平均電力が低くなった．

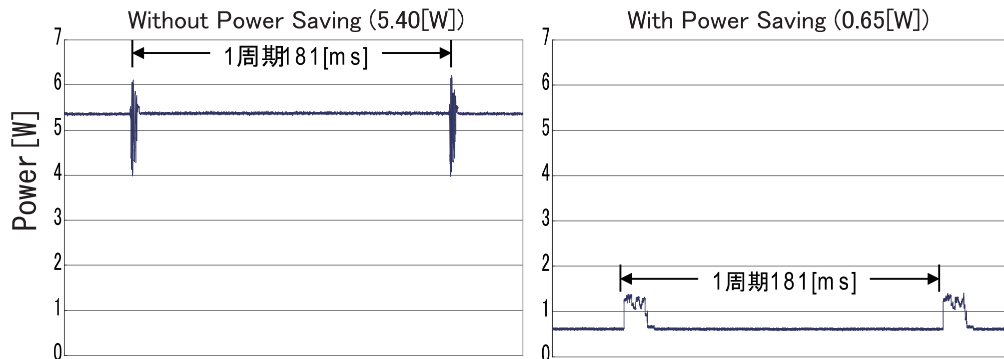

図 5.11: AAC エンコーダにおける電力波形

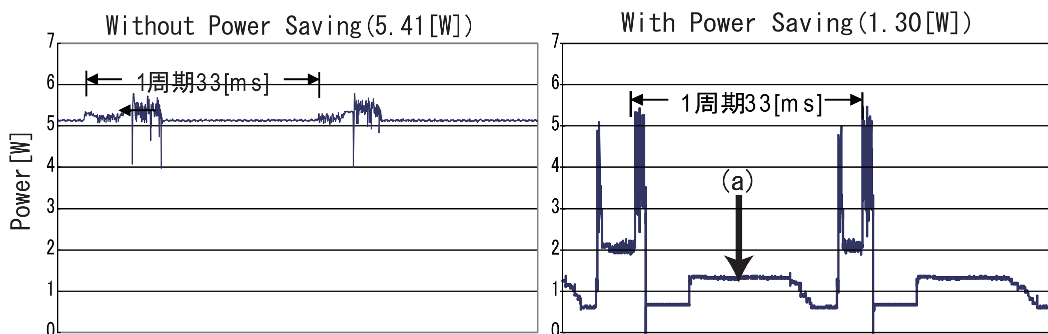

図 5.12: MPEG2 デコーダにおける電力波形

5.6　関連研究

処理性能の向上に加え，増加する消費電力をいかに抑えるかが大きな課題となっており，従来から様々な手法が提案されている．

キャッシュミス回数測定用カウンタや命令キューなどのハードウェアサポートにより実行時にプログラム中の各フェーズにおける負荷を判断し，不要なリソースを停止する Adaptive Processing[ABD+03] や，計算資源の各部分に対して実行時の負荷に応じた周波数・電圧制御（FV 制御）を行なう Online Methods for Voltage and Frequency Control[WJMC04] や，ループイタレーションレベルのプロセッサ間の負荷不均衡に対して，実行時のハードウェアサポートにより電力制御を行う

Thrifty Barrier[LMH04] などがある．しかしながら，これらの実行時の情報を利用した手法では，プログラム全域にわたるグローバルな消費電力の最適化は難しく，局所的な最適化にとどまってしまう．また，実行時電力制御のための処理やハードウェア追加が必要となるため遅延が大きいという欠点がある．

このような実行時の低消費電力化手法に対して，コンパイル時の静的な情報に基づく低消費電力化手法では，プログラムの解析による詳細な情報を利用することができるため，よりきめ細やかな電力制御が可能となり，プログラム全域にわたる消費電力の最適化が実現できる．コンパイラ制御によるシングルプロセッサの低消費電力化手法として compiler-directed DVS（dynamic voltage scaling）[HK03] が挙げられる．しかしながら，この手法はシングルプロセッサのみを対象としており，マルチコアプロセッサ向けの並列処理と電力制御との両立を実現できていない．

マルチコアプロセッサ向けの低消費電力化手法としては，OSCAR コンパイラによる粗粒度タスク並列処理における低消費電力化手法が提案されている [白子 06]．OSCAR コンパイラではマルチグレイン並列処理による並列性抽出に加え，プログラムの各部分に対するコアごとの適切な動作周波数/電圧および電源遮断のタイミングを決定し，必要な処理性能を維持しつつ電力を低減させている．従来の OSCAR コンパイラによる低消費電力化手法では，コンパイル時の簡易的なシミュレーション結果でしか性能評価が行われていなかった．それに対し，本研究では OSCAR API を利用することにより，実マルチコア上で低消費電力化手法を実現し，実際に実行時間と消費電力の性能評価を行った．このように実際のマルチコアプロセッサ上でコンパイラによる低消費電力化手法を実現したのは，筆者の知る限り本研究が初めてである．

5.7 まとめ

本章では OSCAR API を用いたマルチコアプロセッサ向けの低消費電力化手法を提案し，情報家電用マルチコア RP2 上で評価を行った．最速実行モードでは，実行時に負荷不均衡が起こりうる並列ループを含む MTG において電源遮断を適用することで，ビジーウェイト等による無駄な電力消費の削減を実現し，SPEC2000 の art，equake 及び AAC エンコーダ，MPEG2 デコーダにおいて消費エネルギーをそれぞれ 13.06%，3.99%，3.84%，9.01%削減することができた．また，メディアアプリケーションのリアルタイム処理向けのリアルタイム制約モードでは AAC エンコーダと MPEG2 デコーダにおいて，リアルタイム制約を守りつつ平均電力をそれぞれ 87.9%，76.0%削減することができた．

第6章

ソフトウェアコヒーレンシ制御

6.1 はじめに

組込み機器から PC，スーパコンピュータに至るあらゆる情報機器において１チップ上に複数のプロセッサコアを集積したマルチコアプロセッサの普及が進んでおり，チップ内に集積するコア数の増加による高性能・低消費電力化が期待されている．現在主流の 4 から 8 コア程度のマルチコアでは主記憶共有型のマルチプロセッサシステム (SMP) が一般的であるが、そのキャッシュコヒーレンシ制御機構のハードウェアは，プロセッサコア数の増加に伴い，その実装が回路規模的にも消費電力的にも困難となることが知られている [CSG99]．そのため，今後コア数が 32 コアから 64 コア以上のメニーコアとなっていくと，共有メモリ空間とコア毎のコヒーレンシ制御機能を持たないキャッシュ上でも，ソフトウェアによりコヒーレンシ制御を行うことができれば，コスト及び電力消費を抑えつつ効率的な並列処理を実現できると期待されている．特にハードウェアコストや消費電力およびリアルタイム制約に厳しい組込み系マルチコアでは，4 コアの富士通 FR1000[SKK+05]，8 コアの東芝 Venezia[NTF+08] 等において，すでにノンコヒーレントキャッシュアーキテクチャが採用されている．また，ルネサステクノロジ／日立製作所／早稲田大学の RP2[IHY+08] は 8 コア集積しているが，ハードウェアではチップ価格を抑えるためにコヒーレンシ機構は 4 コアまで対応とし，5 コア以上のコア数ではノンコヒーレントキャッシュとして利用する設計となっている．高性能計算分野においても，イリノイ大の Rigel[KJJ+09]，Intel の Single-Chip Cloud Computer (SCC)[HDH+10] は，それぞれのコアがキャッシュを持つが，チップ全体でのコヒーレンシ制御はソフトウェアにより行うことが想定されている．

しかしながら，このようなアーキテクチャにおける単一のアプリケーションプログラムの並列処理による高速化では，ソフトウェアによるコヒーレンシ制御のプログラミングが必須であるが，ソフトウェア開発は長期間に伴う非常に複雑な作業となり，ソフトウェア開発生産性が問題となる．そのため，ユーザからコヒー

レンシ制御の複雑さを隠蔽し，簡単にチップを使うことを可能とするソフトウェア開発環境を提供する必要がある．その際，単一のアプリケーションプログラムの並列処理による高速化を実現するためには，コヒーレンシ操作のオーバヘッドを最小限に抑える効率的な実装が必須となる．

コヒーレンシ制御で対処すべき問題としては，stale data の参照の回避と false sharing の回避が挙げられる．stale data とはメモリの値が更新された後もプロセッサのキャッシュに保持されたままの古いデータのことであり，これを参照してしまうと意図しない動作を起こしてしまう．一方で，false sharing とは複数のプロセッサのキャッシュが同時に同一データを更新された状態で保持する状態であり，意図しない動作となってしまう．そこで，コヒーレンシ制御においては，この stale data の参照の回避と false sharing の回避が必要となる．

本章では，ノンコヒーレントキャッシュマルチコア及びメニーコアプロセッサにおけるアプリケーションプログラムの自動並列化をターゲットとして，コンパイラがソフトウェアでコヒーレンシ制御を行う手法を提案する．本手法の特徴は，コンパイラによる高度な自動並列化とキャッシュ最適化 [Wol96, ALSU07, 笠原 03] を適用した上で，コア間で共有するデータに対する並列処理タスク単位の粗粒度なキャッシュ操作と，ノンキャッシャブル変数指定を用いた効率的なコア間通信をコンパイラが自動生成することにある．また，各コアでは自コアのキャッシュに対してのみ操作を行うため，コヒーレンシ制御のために余分なコア間トラフィックが発生しない．これにより，プログラマの手を煩わせることなくソフトウェアによるコヒーレンシ制御を実現し，多くのコア数まで性能がスケールアップするメニーコアハードウェアを低コストで容易に開発できるようになる．

提案するソフトウェアコヒーレンシ制御手法を，OSCAR 自動並列化コンパイラ [笠原 03] に実装し，4 コアまではハードウェアコヒーレント機構を持つが，5 コア以上はノンコヒーレント共有メモリ動作となる，8 コア集積の情報家電用マルチコ

ア RP2 において評価を行った．その結果，コンパイラによるソフトウェアコヒーレンシ制御を利用することで，ハードウェアによるコヒーレンシ制御をサポートする 4 コアまででコヒーレントキャッシュと同等に性能向上が得られ，さらにハードウェアではコヒーレンシ制御を行えない 5 コアから 8 コアまでスケーラブルな性能向上が得られることが確認された．

本章の構成は以下の通りである．まず，第 6.2 節で提案するソフトウェアコヒーレンシ制御の概要を述べる．次に，第 6.3 節で提案する並列化コンパイラによるソフトウェアコヒーレンシ制御手法について述べ，第 6.4 節では自動並列化コンパイラとコード生成コンパイラのインターフェースとなる OSCAR API におけるキャッシュ制御用の指示文について述べる．そして，第 6.5 節でノンコヒーレントキャッシュアーキテクチャ上での並列処理性能について述べる．第 6.6 節で関連研究について述べ，最後に第 6.7 節でまとめを述べる．

6.2　ソフトウェアコヒーレンシ制御

コヒーレンシとは，あるメモリアドレスに格納されるデータをある時刻において全てのプロセッサコアから同一の値としてアクセスできることである．並列処理のプログラムを正常に実行するためには，ソフトウェアやハードウェアを用いてコヒーレンシを維持するための枠組みが必要となる．

6.2.1　ノンコヒーレントキャッシュアーキテクチャ

対象とするノンコヒーレントキャッシュアーキテクチャでは，図 6.1 に示すように，複数のプロセッサコアが主記憶メモリを共有しており，それぞれのプロセッサコアは独立にキャッシュを持つが，コヒーレンシ制御用のハードウェア機構は持たない．各コアのローカルキャッシュのハードウェアは，一般的なマイクロプロ

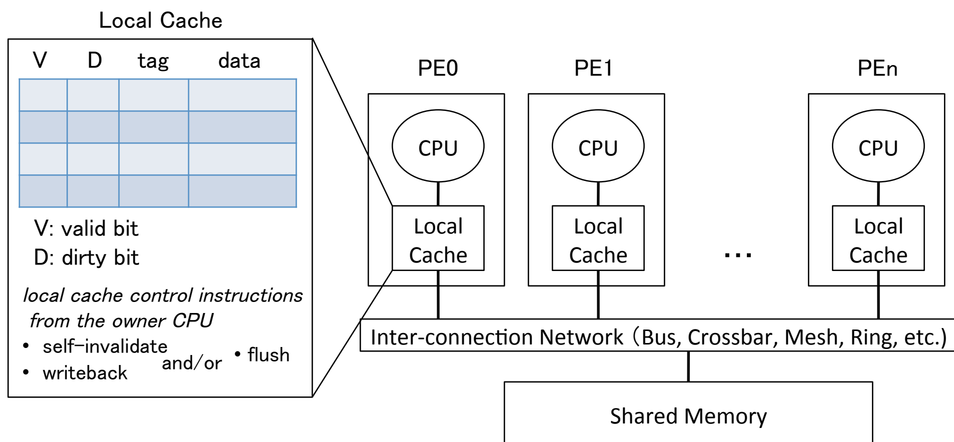

図 6.1: ノンコヒーレントキャッシュアーキテクチャ

セッサで利用されているような各キャッシュライン毎に valid bit と dirty bit を持つものを想定しており，各プロセッサコアにおいて，そのキャッシュラインが有効かどうか (valid bit)，キャッシュライン上のデータがメモリから読み込まれた後にキャッシュ上で更新されているかどうか (dirty bit) をハードウェアで管理する．すなわち，それぞれのプロセッサコアが同一のメモリアドレスに格納される値を個別にキャッシュすることがあるため，ソフトウェアで整合性を取る必要がある．

あるプロセッサコアがある変数の値をキャッシュしている場合に，他のプロセッサコアがその変数を更新するとき，更新前からキャッシュしているプロセッサコアにおいて次にその変数を参照する前に自プロセッサコアのキャッシュラインの内容をインバリデートしないと，他プロセッサコアが更新する前の古いデータを誤って参照してしまう．また，他のプロセッサコアで更新された変数を参照する前には，更新したプロセッサコアはキャッシュ上で更新したデータを主記憶メモリにライトバックして，明示的にメモリを更新することによって，他のプロセッサで参照可能となることを保証する．

その際に，キャッシュのハードウェアはプログラム中の個々の変数ごとではなく，キャッシュライン単位でメモリとのデータ一貫性の維持を行うため，プログラム上

では異なるメモリ領域として宣言されていても同一のキャッシュラインに対応するメモリアドレスに割り付けられる変数がある．そのためにキャッシュラインが意図せず複数のプロセッサコアで共有されてしまうフォルスシェアリングについても，コヒーレンシ制御で対処する必要がある．特に，ソフトウェアによるコヒーレンシ制御では，フォルスシェアリングへの対処の実行時オーバヘッドが大きくなるため，重要な課題となる．

6.2.2　ソフトウェアによるキャッシュコヒーレントプロトコル

本節では，フォルスシェアリングの回避をコンパイラで実現し，特殊なハードウェアサポートを必要とせず既存のプロセッサコアを多数集積するのみで実現可能なコンパイラによるソフトウェアコヒーレンシ制御手法を提案する．提案するソフトウェアによるキャッシュコヒーレントプロトコルを図 6.2 に示す．提案するコヒーレンシ制御のプロトコルは Modified，Valid，Invalid，Stale の 4 状態の状態遷移で実現される．各状態は以下のような意味である．

- Modified: そのコアでキャッシュ上のデータが更新されて書き戻されていない状態
- Valid: そのコアでキャッシュ上のデータが有効でありメモリ上の値と一致している状態
- Stale: キャッシュ上のデータに対応するメモリ領域が他のプロセッサコアで更新されて古いデータとなってしまっている状態
- Invalid:キャッシュが無効化された状態

ここで，状態遷移図中の括弧はキャッシュラインの dirty bit の値を表しており，Valid と Stale はハードウェアの状態としては同じ状態であり，コンパイラにおけ

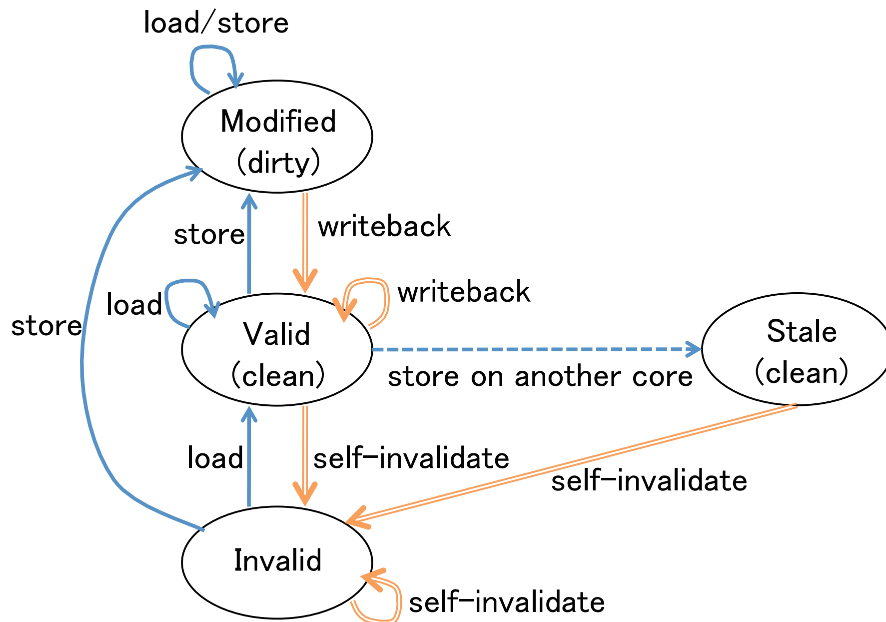

図 6.2: ソフトウェアによるキャッシュコヒーレントプロトコル

る解析時の状態管理における扱いで区別している．図中の状態遷移において，一重線はロード (load)，ストア (store) 等のプロセッサコアによるメモリアクセスを表す．二重線は自コアのキャッシュラインのライトバック (writeback) やセルフインバリデート (self-invalidate) 等のキャッシュ操作を表す．提案手法では，これらのキャッシュ操作指示をコンパイラが自動生成し，全てのコアは他のコアの動作に影響されることなくキャッシュ操作を行いながら並列処理を行うことが特徴である．コンパイラは対象のメモリアドレスのデータについて，複数のコアのキャッシュで同時に Modified の状態とならないように，また Stale 状態のときに読み込みを行わないように，プログラムの最適化およびキャッシュ操作の挿入を行う．

以下，本章では上記のソフトウェアコヒーレンシ制御を行うための，コンパイラの実装について述べる．

6.3 コンパイラによるソフトウェアコヒーレンシ制御手法

本節では提案するコンパイラによるコヒーレンシ制御手法について述べる．

6.3.1 階層的粗粒度タスク並列処理

本章では，階層的な並列性を汎用的に表現可能な階層的粗粒度タスク並列処理を対象にソフトウェアコヒーレンシ制御を適用する．

階層的粗粒度タスク並列処理では，プログラムは基本ブロック (BB)，ループ等の繰り返しブロック (RB)，関数やサブルーチン呼び出し等のサブルーチンブロック (SB) 等のマクロタスクとして分割され，マクロタスク間のコントロールフローとデータ依存より並列性を抽出したマクロタスクグラフとして表現される [本多 90]．繰り返しブロック (RB) とサブルーチンブロック (SB) はその内部についても階層的にマクロタスクを生成していく．また，ループ繰り返し間のループレベル並列性は，ループ分割により粗粒度タスク並列性に変換され，粗粒度タスク並列処理の枠組みで扱われる．

階層的粗粒度タスク並列処理のイメージを図 6.3 に示す．プログラムは階層的なマクロタスクグラフとして表現され，各マクロタスクの持つ並列性を元にコンパイラがプロセッサコアのグルーピングを行う．階層的にグルーピングしたプロセッサグループへスケジューリングを行うことで，階層的な並列処理を実現する．

6.3.2 並列処理階層の決定

コンパイラではプログラムの各部分の持つ並列性に応じて，階層的なマクロタスクグラフに対して，プロセッサグループを割り当てていく [小幡 03]．この過程

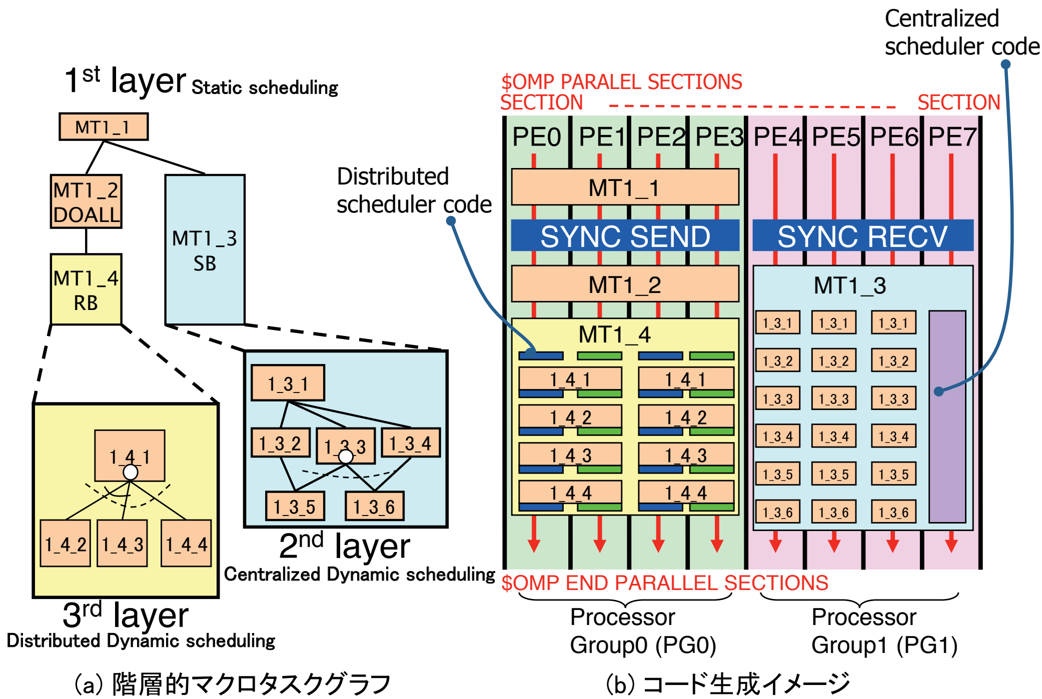

図 6.3: 階層的粗粒度タスク並列処理のイメージ

で複数のプロセッサグループが割り当てられた階層について，粗粒度タスク並列処理が行われる．そこで複数のプロセッサグループが割り当てられた階層において，並列処理される可能性があるタスクのメモリアクセス範囲から，フォルスシェアリングが発生する可能性があるかどうかを解析する．

6.3.3　変数配置のアラインメント

まず，フォルスシェアリングの回避のために，プロセッサコア間で共有する変数の先頭アドレスをそれぞれキャッシュライン境界へアラインメントして配置する．malloc 等で動的確保されるヒープ領域についても，先頭アドレスがキャッシュラインの先頭にアラインメントされるものとする．これにより異なる変数への参照によるフォルスシェアリングを回避するとともに，配列や構造体等の内部要素に

対するアクセスによるフォルスシェアリングの検出と回避を行いやすくする．

6.3.4 フォルスシェアリングの解析

本節では，各変数の先頭がキャッシュラインの境界にアラインメントされていることを前提とした場合に，その変数の内部のメモリ領域について，フォルスシェアリングが発生する可能性があるかどうかを検出する手法を述べる．

この解析は並列処理のためのタスク分割後のタスクグラフにおける各タスクのメモリアクセス範囲を解析する．並列処理のためのタスク分割が行われるタスク，すなわち並列ループやデータローカライゼーションの対象と判定されたループについては，タスク分割後のアクセス範囲を考慮してフォルスシェアリングの検出を行う．ここでは，最大分割数でタスクを分割した際，すなわちループ１イタレーションが部分タスクとなるような場合を考えて，フォルスシェアリングが発生する可能性を考える．各タスクのメモリアクセス範囲に重なりがなく，それらのアクセス範囲の境界をまたぐキャッシュラインが存在しなければ，フォルスシェアリングは発生しないことを保証できる．

図 6.4(a) のタスクグラフは，並列性抽出とデータローカリティ最適化 [吉田 94] が適用されるため，最大分割数で分割されると想定すると図 6.4(b) のようにタスク分割が行われる．ここで，プログラム上のデータ依存だけを考慮して並列処理される可能性があるタスク，例えば doall1_1 と doall1_2 や doall1_2 と doall1_3 のメモリアクセス範囲を解析すると，それぞれのタスクで同一キャッシュラインを複数プロセッサコアで当時に更新する可能性があり，フォルスシェアリングによる出力依存 (Write after Write) があることが分かる．また，loop3_1 と doall2_2 においては，doall2_2 で参照するキャッシュラインを loop3_1 で更新するため，フォルスシェアリングによる逆依存 (Write after Read) があることが分かる．さらに，loop3_2 と doall4_1 においては，doall4_1 で参照するキャッシュラインを loop3_2 で更新す

図 6.4: 分割後の部分ループに関するフォルスシェアリングの検出

るため，フォルスシェアリングによるフロー依存 (Read after Write) があることが分かる．

このように，フォルスシェアリングが発生する可能性があるタスク群と，対象の変数を解析し，フォルスシェアリングによるデータ依存を解析する．コンパイラのこの後のフェーズでは，フォルスシェアリングの回避のための最適化を行い，もし回避できなかった場合は解析されたフォルスシェアリングによるデータ依存情報に基づいて逐次的に処理される．

6.3.5　フォルスシェアリングの回避

本手法では第節で検出したフォルスシェアリングによるデータ依存に対して，配列データの形状拡張，ループの分割位置の調整，キャッシュを無効化したメモリ領域上のバッファの利用，CPU コアプライベートデータの利用と共有領域への書き戻し逐次化を行うことで，フォルスシェアリングを回避する．

データレイアウト変換

データレイアウト変換は，ある変数の変数宣言を変更することで，データレイアウトを変更する操作である．配列のパディングおよびエキスパンション，構造体のフィールドへのパディングにより行われる．パディングの例を図 6.5 に示す．プログラムの変数宣言の int a[6][6] では外側の i ループを均等分割して並列処理を行うとフォルスシェアリングが発生してしまう．そこで，配列の宣言サイズを変更し，int a[6][8] とすることで，ループ分割後のタスクのメモリアクセス範囲の境界とキャッシュライン境界が整合し，フォルスシェアリングを回避することができる．

ある変数に関する，データレイアウト変換を行う際には，その対象の変数とエイリアスする可能性のある全ての変数に対してデータレイアウト変換を適用する必要がある．そのような変数をポインタ解析結果より解析し，まとめてデータレイアウト変換を適用する．また，それらの変数を指すポインタの型についても，あわせて変更を行う．

キャッシュライン境界に合わせたループ分割

対象の変数の先頭がキャッシュラインの先頭にアラインメントされている場合は，ループ分割時のイタレーション数を適切に設定することで，分割後の部分タスクによるデータアクセス範囲の境界がキャッシュライン境界と一致し，フォルス

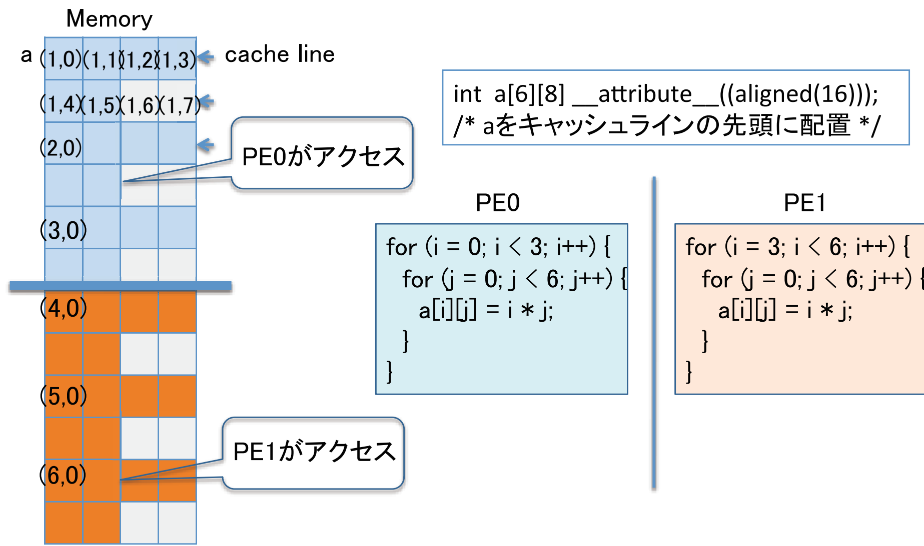

図 6.5: 配列のパディングによるフォルスシェアリングの回避

シェアリングが発生しないことを保証できる．

図 6.6 における配列 a については，PE0 に 4 イタレーション，PE1 に 2 イタレーション割り当てられるようにループ分割を適用することでフォルスシェアリングを回避している．

ノンキャッシャブルバッファを用いた通信の挿入

ループ中に複数の変数への書き込みがあり，それぞれのアクセス時のオフセットが異なるため，ループ分割のみではフォルスシェアリングを回避できない．そのような場合には，境界領域については一時的にノンキャッシャブルなバッファへ値を格納し，そのバッファを用いて通信を行いながら並列処理を行う．

図 6.6 の配列 b の例では，配列 a へのアクセスパターンに合わせてループ分割を適用した場合に，配列 b については分割後のメモリアクセス範囲とキャッシュラインの境界が整合しない．そのため，境界領域については一度ノンキャッシャブル

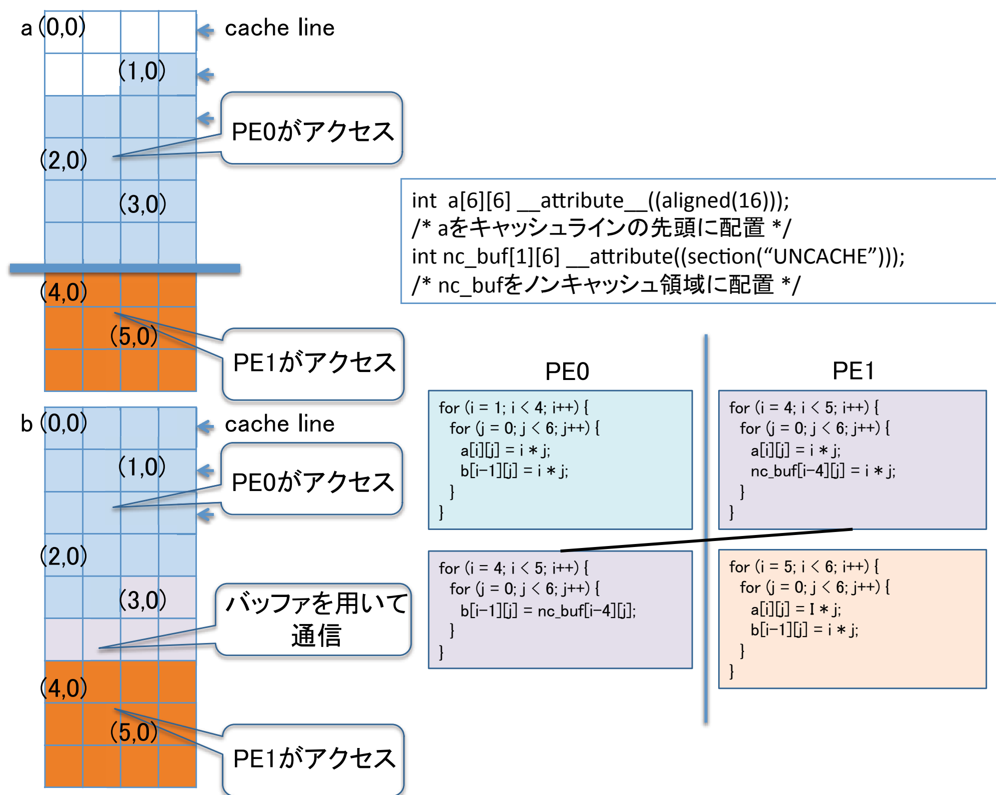

図 6.6: キャッシュライン境界に整合したループ分割とノンキャッシャブルバッファを用いた通信の生成

バッファに書き込み，そのバッファを用いて値を通信することで，フォルスシェアリングを回避している．

変数のローカル化と書き戻しの逐次化

各プロセッサコアで書き込みを行う変数におけるアクセスパターンがストライドアクセス等で各プロセッサコアがアクセスする範囲に重なりがある場合は，変数のローカル化と書き戻しの逐次化を行う．変数が関数の引数の指し先の領域等で先頭がキャッシュラインにアラインメントされているかどうか不明な場合にも，このローカル化と書き戻しの逐次化を行う．

6.3.6 キャッシュのセルフインバリデートおよびライトバック操作の挿入

本手法では，コンパイラのデータ利用状況の解析結果より，マクロタスクの実行開始前に，他のプロセッサコアで更新されて Stale 状態となっている変数について，自コアのキャッシュをセルフインバリデートすることで，それ以降で不正に参照されないようにする．また，マクロタスクの実行終了後に他コアで実行する後続マクロタスクで参照する変数がキャッシュ上に存在する場合は，そのデータを共有メモリ上にライトバックすることにより，他コアがその変数を参照できるようにする．粗粒度タスクスケジューリング結果を元にプロセッサグループをまたぐデータ依存についてコンパイラが同期コードの生成を行う際に，キャッシュ操作指示をあわせて挿入する．

マクロタスクグラフ上に存在する依存エッジは，フロー依存，出力依存，逆依存の３種類となる．このうち，フロー依存 (Read after Write) および出力依存 (Write after Write) の箇所について図 6.7 のようにキャッシュ操作指示を挿入する．ただし，逆依存 (Write after Read) については同期は必要であるが，データに対するキャッシュ操作指示は挿入しなくてよい．

6.4 OSCAR API におけるキャッシュ制御用指示文

本章で提案するソフトウェアコヒーレンシ制御手法は，対象プログラムのソースコードレベルの変換で実現することができる．本手法を実装した OSCAR コンパイラでは逐次プログラムを入力とし，プログラムのデータ依存とコントロールフローの解析結果より自動並列化を行い，OSCAR API[OSC] で記述された並列プログラムを自動生成する．ソフトウェアコヒーレンシ制御のためのキャッシュ操作や変数配置については，OSCAR API の仕様に新たな拡張を行うことで，変換さ

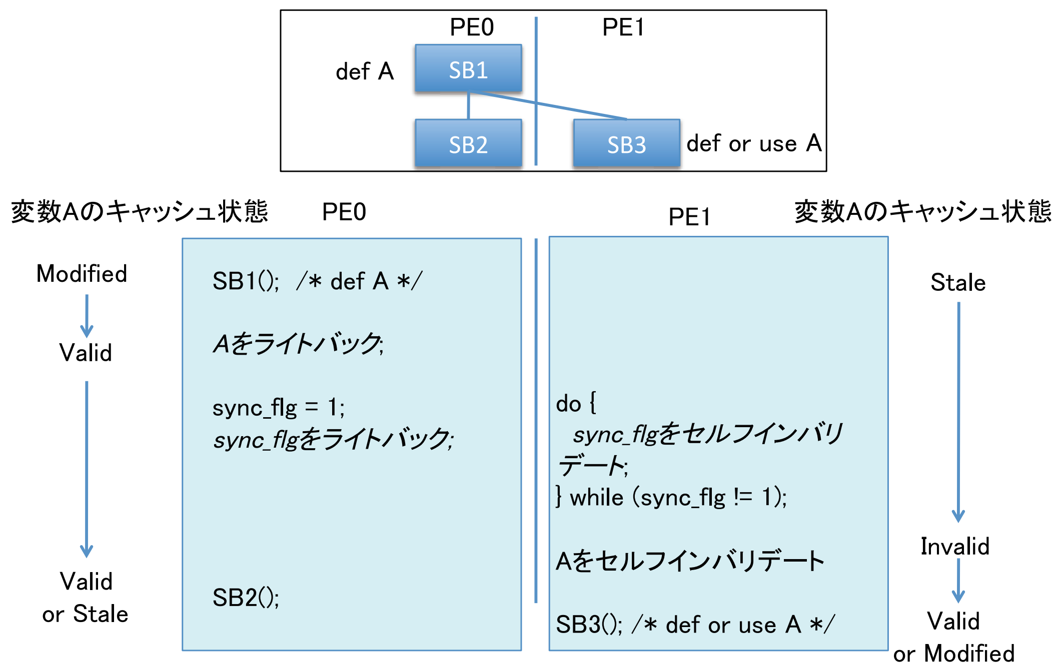

図 6.7: キャッシュのセルフインバリデート及びライトバック操作指示の挿入

れたプログラムのコード生成を行う．

6.4.1　ノンコヒーレントキャッシュ向け指示文

OSCAR API は情報家電用の並列処理 API として開発されており，OpenMP のサブセットである，parallel sections，flush，critical 指示文による並列処理向けの指示文に加えて，新たに定義したデータのメモリ配置，データ転送，電力制御，グループバリア同期およびタイマ指示文から構成される．OSCAR API の処理系においては，各スレッドの処理は各コアにバインドされていることを前提に処理を行うことを期待している．

本章で提案するコンパイラによるソフトウェアコヒーレンシ制御を実現するにあたり，以下の 5 つの指示文を新たに追加した．

- noncacheable: 変数をノンキャッシャブルにする
- aligncache: 変数の先頭をキャッシュライン境界にアラインメントする
- cache_writeback: 自コアのキャッシュ上のダーティラインの書き戻し
- cache_selfinvalidate: 自コアのキャッシュラインの無効化
- complete_memop: メモリ操作の終了

6.5 性能評価

本章で提案するソフトウェアコヒーレンシ制御手法をOSCAR自動並列化コンパイラに実装し，情報家電用マルチコアRP2を用いてその有効性を評価した．

6.5.1 情報家電用マルチコアRP2

情報家電用マルチコアRP2のブロック図を図6.8に示す．RP2はSH-4Aコアを8コア集積したマルチコアであるが，内部はクラスタ構造となっており，コヒーレンシ制御を行うハードウェアを持つ4コアのSMPがクラスタを構成しており，クラスタ間ではハードウェアによるコヒーレンシ制御を行わない．すなわち，5コア以上で使用する場合はソフトウェアによりコヒーレンシを維持する必要がある．

RP2はキャッシュを明示的に操作する命令を持っており，自コアの対象のキャッシュラインに対するライトバック(OCBWB命令)，インバリデート(OCBI命令)，ライトバックの後にインバリデートを行うキャッシュフラッシュ(OCBP命令)が利用可能である．コンパイラによるキャッシュ操作指示により，これらのキャッシュ操作命令が実行されることになる．なお，RP2のデータキャッシュは16KBであり，ラインサイズは32Byteである．

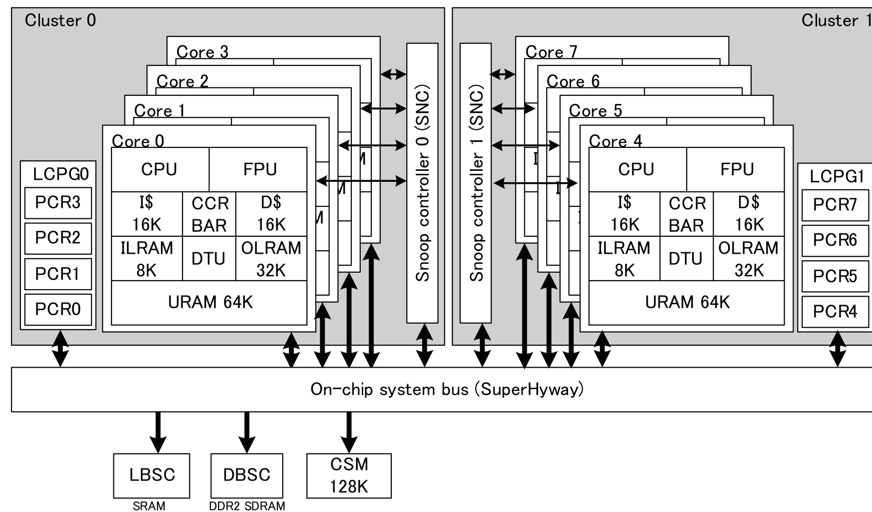

図 6.8: 情報家電用マルチコア RP2 のブロック図

また，RP2 は全コアで共有するオンチップ CSM を持っており，これを通信用のノンキャッシャブルバッファとして用いた．

6.5.2 評価条件

逐次 C プログラムを OSCAR コンパイラで自動並列化し，OSCAR API[OSC] を用いた並列コードを自動生成する．その並列プログラムを，API 解釈機能を持った SH C コンパイラでコンパイルすることで，実行形式コードを生成する．

比較評価では，4 コアまでのハードウェアによるコヒーレンシ制御と，8 コアまでのコンパイラによるソフトウェアコヒーレンシ制御の処理速度の比較を行う．ソフトウェアコヒーレンシ制御の評価時にはハードウェアによるコヒーレンシ制御機能は無効となるように設定を行った．また，同期やリダクションループの総和計算におけるコア間の通信については，いずれの評価もオンチップ CSM 上のノンキャッシャブル領域を利用するものとする．

対象のアプリケーションとして，マルチメディアアプリケーションから，AAC エ

ンコーダ，MPEG2 エンコーダと，科学技術計算から SPEC2000 より art と equake を用いて評価を行った．今回対象とするプログラムは Parallelizable C で記述され，コンパイラによる自動並列化が適用可能となっている [間瀬 09]．

6.5.3 コンパイラの実装

本章では，コヒーレンシ制御が必要なキャッシュ操作が必要な箇所において，まずはキャッシュ全体を対象にキャッシュ操作を行うように実装し，評価を行った．また，セルフインバリデートあるいはライトバックが必要な箇所について，全てキャッシュフラッシュ，すなわち対象のキャッシュラインをライトバックした後にセルフインバリデートする操作を行うものとして実装している．

各タスクにおけるメモリアクセス範囲の解析情報から，対象のメモリ領域を指定して，ライトバックやセルフインバリデートのキャッシュ操作を個別に行うことで，より効率的な実装が可能と考えられる．つまり，本章での評価はソフトウェアコヒーレンシ制御のベースラインの評価となり，選択的なキャッシュ操作の実装と評価は今後の課題となる．

6.5.4 並列処理性能

ハードウェアによるコヒーレンシ制御を有効にした場合の SMP 構成における並列処理性能および，ハードウェアによるコヒーレンシ制御を無効にしソフトウェアによるコヒーレンシ制御を適用した場合のノンコヒーレントキャッシュ構成における並列処理性能を図 6.9 に示す．図中の横軸が評価アプリケーションと使用 PE 数，縦軸がコヒーレントキャッシュ(SMP) モードの逐次実行時の性能に対する速度向上率となっている．左側のバーが SMP モードの性能，右側のバーがコヒーレンシをソフトウェアで保証するノンコヒーレントキャッシュ(NCC) モードの性能

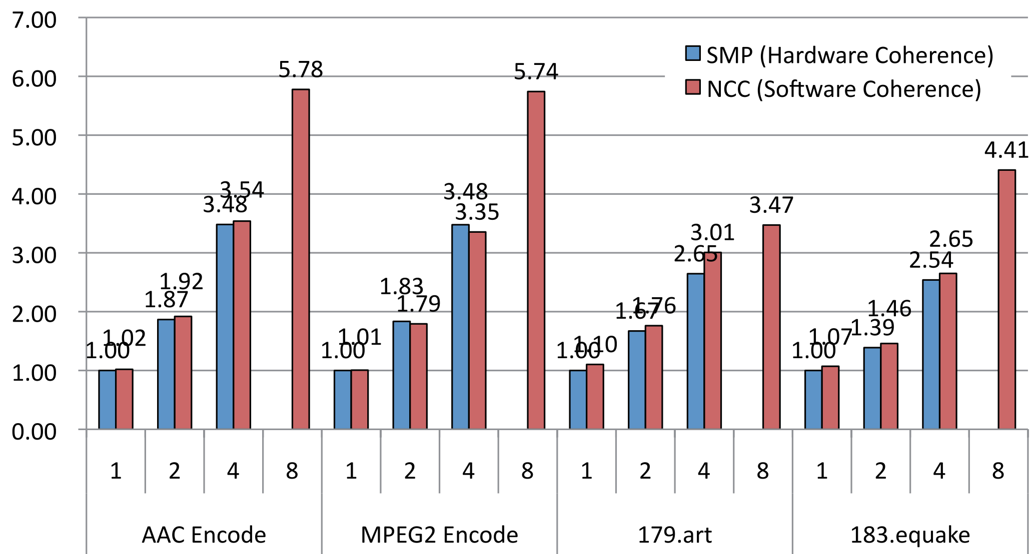

図 6.9: RP2 における並列処理性能

を示している．

まず，SMP モードと NCC モードの 1 コア使用時の性能を比べると，NCC モードでは art において最大 7%ほど性能が向上するが，これはコヒーレンシ制御ハードウェアのオーバーヘッドに起因するものである．次に，4 コア使用時の性能を比較すると，5 本のアプリケーションプログラムにおいて，MPEG2 エンコーダにおいて SMP と比較して最大 5%の性能劣化が見られるが，概ね同等の性能が得られ，art では 13%の性能向上が得られた．

NCC モードでは 8 コア使用時にもそのまま性能向上が見られ，それぞれ SMP モードの逐次実行時と比較して，AAC エンコーダで 5.78 倍，MPEG2 エンコーダで 5.74 倍，art で 3.47 倍，equake で 4.41 倍の性能向上が得られ，平均 4.88 倍の速度向上が得られた．

本節の評価では，ソフトウェアコヒーレンシ制御におけるキャッシュ操作を非常に保守的に実装したにも関わらず，ハードウェアによるコヒーレンシ制御と遜色の無い，高い性能を得ることができた．

6.5.5　ソフトウェアコヒーレンシ制御による影響

ソフトウェアコヒーレンシ制御とハードウェアによるコヒーレンシ制御の性能を比較した際の，性能に影響を与える要因について評価を行った．ソフトウェアコヒーレンシ制御を行うことによる性能変化は，(1) ハードウェアコヒーレンシ制御機構の無効化，(2) コンパイラによるフォルスシェアリング回避，および (3) 明示的なキャッシュ操作 (キャッシュフラッシュ) による影響が考えられる．上記の要因を解析するために，ハードウェアの設定とコンパイラオプションを変更した評価を行った．図 6.10 に評価結果を示す．図中の横軸がアプリケーションおよび使用コア数，縦軸がソフトウェアコヒーレンシ制御時を基準とした，各オプションの性能である．折れ線のそれぞれの系統は以下の通りである．

- NCC (software coherence) : 基準となるソフトウェアコヒーレンシ制御適用時の性能.
- NCC (hardware coherence enabled): ソフトウェアコヒーレンシ制御を行うプログラムをハードウェアコヒーレンシを有効にした状態で実行した際の性能，すなわち (1) の要因を示す.
- false sharing avoidance (hardware coherence): ソフトウェアコヒーレンシ制御の構成要素のうちコンパイラによるフォルスシェアリングの回避のみを適用し，ハードウェアコヒーレンシ制御により実行した場合の性能，すなわち (2) の要因を示す.
- stale data handling (hardware coherence): ソフトウェアコヒーレンシ制御のもう一つの構成要素であり，コンパイラによる明示的なキャッシュ操作指示の挿入のみを行った場合の性能，すなわち (3) の要因を示す.

- SMP (hardware coherence): ソフトウェアコヒーレンシ制御のためのコンパイラの機能を用いずに，コヒーレントキャッシュ向けにコンパイルした際の性能.

評価結果を見ると，いずれのアプリケーションについても，NCC (software coherence) が NCC (hardware coherence enabled) よりも高い性能となっており，ハードウェアコヒーレンシ制御を無効化したことによる性能向上が得られていることが分かる．特に art，equake ではハードウェアによるコヒーレンシ制御を無効化した場合の性能向上が大きく，結果的にソフトウェアコヒーレンシ制御の性能がハードウェアコヒーレンシ制御の性能を art で 4 コア使用時に 13.0%，equake で 4 コア使用時 4.2%上回っている.

一方で，AAC エンコーダと MPEG2 エンコーダにおいては，stale data handling (hardware coherence) が SMP(hardware coherence) に対して性能低下を起こしており，明示的なキャッシュ操作指示による影響が見えている．その結果，AAC エンコーダでは 4 コア使用時に 1.7%の性能向上にとどまり，MPEG2 エンコーダでは 4 コア使用時に 3.6%の性能低下を起こしている．ただし，本節の評価では，キャッシュ操作についてはキャッシュ全体のフラッシュを行うというナイーブな実装を行っているため，今後ライトバックとセルフインバリデートの分離やキャッシュ操作対象のメモリ領域の限定等の最適化を行っていくことで，ソフトウェアコヒーレンシ制御の性能向上が見込まれる．しかしながら，RP2 のようにデータキャッシュの容量が 16KB 程度と小容量なアーキテクチャでは，キャッシュ全体をフラッシュする実装においても 3.6%程度の性能劣化にとどまっている.

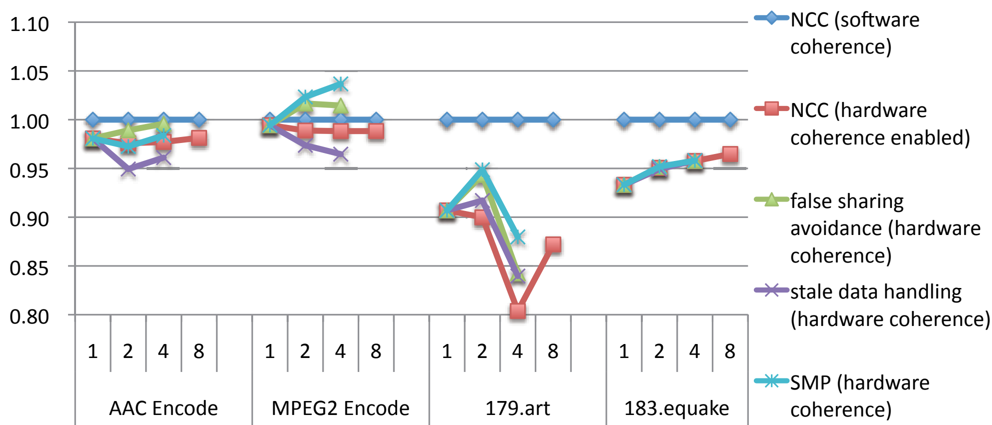

図 6.10: ソフトウェアコヒーレンシ制御における性能への影響要因

6.6 関連研究

ソフトウェアによりコヒーレンシ制御を行う手法としては OS や Runtime によるアプローチとコンパイラやプログラミングモデルによるアプローチが提案されてきた [CSG99]．OS や Runtime によりソフトウェアによりコヒーレントな仮想共有メモリ空間を実現する Shared Virtual Memory[LH89] や Treadmarks[ACD+96] が提案されている．これらの手法は一般的なプロセッサハードウェアで実現可能で，SMP 向けのソフトウェアがそのまま動作するが，コヒーレンシ維持のオーバーヘッドが大きい．

従来よりコンパイラやハードウェアサポートを用いたソフトウェアコヒーレンシ制御技術 [CY00] も提案されていたが，これらの研究ではキャッシュハードウェアにラインサイズが 1 ワードのキャッシュを想定していたり，各ワードごとに dirty bit を持つハードウェアを想定しており，現在のハードウェアでは利用することができない．

False sharing の存在はハードウェアによるコヒーレントキャッシュをサポートする共有メモリ型マルチプロセッサにおいても性能劣化の原因となることが知られ

ており，プログラマが手動で [TLH94]，あるいはコンパイラによりフォルスシェアリングを低減させる手法が提案されている [JE95, KCRB03]．ソフトウェアコヒーレンシ制御においては，プログラムを正常動作させるためには false sharing を完全に除去する必要がある．

GPGPU のようなアクセラレータ用途を想定して，Bulk Synchronous モデルのプログラミングモデルにおいてノンコヒーレントキャッシュアーキテクチャを利用する研究が行われている．Rigel Task Model[KJJ+09] では Bulk Synchronous モデルにタスクの概念を採用し，同期やデータ更新はバリア同期のタイミングで行う．そのバリア同期ごとのインターバルにおいてタスク間で共有するデータは明示的にプログラム中に共有変数として指定しておくことで，キャッシュをバイパスしてコヒーレントな下位レベルの共有キャッシュに直接ロード・ストアを行う．また，上記のタスクモデルを前提としてソフトウェアでコヒーレンシを維持するメモリモデルを併せて提案している [KJL+09]．しかしながら，Rigel ではインターバル間でのコア間通信はサポートしておらず，false sharing の対処にはワードごとの dirty bit を想定しているため，汎用のマルチコアハードウェアでそのまま適用することはできない．

Bulk Synchronous モデルにおけるノンコヒーレントキャッシュ向けのハードウェアサポートとして Sharing Tracker[TS10] が提案されている．Sharing Tracker はキャッシュコヒーレンシプロトコルを模した簡易的なハードウェアであり，コア間共有データを検出することでオフチップバンド幅を節約する．本研究ではこのような課題にも全てソフトウェアにより解決しているが，ハードウェアサポートの追加と，高度なソフトウェアによる手法のトレードオフの検討は今後の課題である．

また，これまでメモリアーキテクチャの議論においては，コヒーレントキャッシュとソフトウェア管理のスクラッチパッドメモリのトレードオフについて比較が行われてきた [LP05, LAS+07, MJCP08] が，ノンコヒーレントキャッシュアー

キテクチャにおいてソフトウェアコヒーレンシ制御を行う場合についてはあまり議論されていない．この議論は今後の課題と考えられる．

6.7 まとめ

本章では，並列化コンパイラによるソフトウェアコヒーレンシ制御手法を提案した．本手法は一般的なキャッシュハードウェアで利用可能であり，提案手法をOSCAR コンパイラに実装し，8 コア集積の情報家電用マルチコアである RP2 において，ソフトウェアによるコヒーレンシ制御により，ハードウェアによるコヒーレントキャッシュと遜色の無い，高い性能が得られた．さらに，ソフトウェアによるコヒーレンシ制御ではハードウェアがサポートしない 8 コア使用時までスケールアップし，マルチメディア処理および科学技術計算の 4 アプリケーションについて，コヒーレントキャッシュの逐次実行時の 4.88 倍の性能が得られた．

ソフトウェアコヒーレンシ制御をコンパイラで自動的に行うことにより，ハードウェアによるコヒーレンシ制御機能が不要となり，特に組込み系のマルチコア等において，コスト・開発期間とも小さく抑えることが可能になる．さらに，多くのコア数まで性能がスケールアップする，32 コアから 64 コア，さらに 128 コアを越えるメニーコアプロセッサシステムを低コストで容易に開発できるようになり，将来の大規模マルチプロセッサ構築技術の大きな柱になると考えられる．

第7章

結論

7.1 本研究により得られた成果

本研究ではマルチコア・メニーコアプロセッサに対して，短期間で高性能な並列ソフトウェア開発を可能とするための，コンパイラによる自動並列化および低消費電力化技術について大きな成果が得られた．以下に，本研究により得られた成果を総括する．

- C プログラムを自動並列化する際に必要となるポインタ解析の解析精度向上について提案した．並列処理が求められるアプリケーションプログラムで頻出するデータ構造として，ポインタのポインタにより実現される擬似的な多次元配列があるが，従来のポインタ解析では解析できず自動並列化ができなかった．このようなデータ構造を解析して自動並列化を適用するために，ポインタの配列の各要素が指しているメモリ領域のエイリアス関係を解析する Element-Sensitive ポインタ解析を新たに提案した．この Element-Sensitive ポインタ解析は，従来のポインタ解析アルゴリズムの解析情報に 1 ビットのデータ構造を追加するだけの簡易な拡張であり，軽量な実装により頻出のデータ構造を用いたプログラムを自動並列化するのに必要な解析精度を得ることができる．従来のポインタ解析では自動並列化が不可能であった SPEC2000 equake および第 4 章で述べる Parallelizable C への修正を行った SPEC2006 hmmer について，Element-Sensitive ポインタ解析を利用することで自動並列化が可能となった．自動並列化結果について 8 コア搭載の IBM p5 550Q サーバにおいて性能評価を行ったところ，8 コア使用時に逐次実行時と比べてそれぞれ equake で 4.96 倍および hmmer で 6.06 倍の性能向上が得られた．
- コンパイラによる自動並列化が可能なプログラム記述法として Parallelizable C を提案した．コンパイラの解析精度，特にポインタ解析精度を考慮して，ポインタ演算やポインタのキャストを制限するようなプログラム記述のガイ

ドラインを設定することで，C プログラム自動並列化支援の枠組みを構築した．実際にこのガイドラインに沿ってプログラムの書き換えを行ったところ，市販コンパイラでは全く並列化できなかった SPEC2000 art, equake は書き換えが不要であり，SPEC2006 lbm で 1.8%，hmmer で 0.03%の書き換えにより自動並列化可能となった．Parallelizable C で記述した 6 つの逐次プログラムに対して OSCAR コンパイラによる自動並列化を適用し，マルチコアシステム上での処理性能の評価を行った．その結果，逐次実行時と比較して，2 コア集積のマルチコアである IBM Power5+ を 4 基搭載した 8 コア構成のサーバである IBM p5 550Q において平均 5.54 倍，4 コア集積のマルチコアである Intel Core i7 920 プロセッサを搭載した PC において平均 2.43 倍，SH-4A コアベースの情報家電用マルチコア RP2 の 4 コアを使用した SMP 実行モードにおいて平均 2.78 倍の性能向上が得られた．

- 現在のプロセッサで問題となっている消費電力削減のために，最速実行時とリアルタイム実行時におけるコンパイラによる低消費電力化手法を提案した．併せて，各社のマルチコア上で低消費電力化を利用するための OSCAR API の提案を行い，自動並列化コンパイラで逐次プログラムから OSCAR API で記述された並列プログラムを自動生成することで，各社マルチコア上でコンパイラによる自動並列化と低消費電力化が利用可能となった．NEDO ”リアルタイム情報家電用マルチコア ”プロジェクトにおいて，新たに開発した 8 コア集積の情報家電用マルチコア RP2 において，マルチコアにおけるコンパイラによる消費電力制御を世界で初めて実現し，4 コアを用いた場合の最速実行モードにおいて消費エネルギーが SPEC2000 の art で 13.06%，equake で 3.99%，メディア処理の AAC エンコーダで 3.84%，MPEG2 デコーダで 9.01%削減された．また，8 コアを用いた場合のリアルタイム制約モードにおいて，平均電力が AAC エンコーダで 87.9%，MPEG2 デコーダで 76.0%削

減された．

- 今後のメニーコアプロセッサにおける重要技術となる，コンパイラによるソフトウェアコヒーレンシ制御手法を提案した．コンパイラによる自動並列化と併せて，OSCAR API を用いてキャッシュライトバック，キャッシュセルフインバリデート、および変数配置のアラインメントを指示することで，ソフトウェアによる自動コヒーレンシ制御を実現した．科学技術計算およびメディア処理の 4 つのアプリケーションプログラムを用いて，情報家電用マルチコア RP2 において 4 コア使用時の自動並列化性能の評価を行ったところ，ハードウェアコヒーレンシ制御で 1 コアの逐次実行と比較して平均 3.03 倍，提案するソフトウェアコヒーレンシ制御においても同様の平均 3.13 倍の性能向上が得られた．さらに，ハードウェアではコヒーレンシ制御できない 8 コア使用時に平均 4.88 倍の性能向上が得られた．

これらの結果より，本論文で提案したポインタ解析，プログラム記述法，低消費電力化，およびソフトウェアコヒーレンシ制御手法により，実際に性能向上や電力削減が確認され，その有効性が確認された．

7.2 今後の課題

本研究に関する今後の課題を以下にまとめる．

- リスト構造や木構造のように自身の構造体型へのポインタを持つような，再帰的なデータ構造に特化したポインタ解析が存在する．Shape 解析 [GH96b] ではポインタの指し先の具体的なオブジェクトを解析するのではなく，ポインタが指し示すデータ構造が木構造なのか DAG なのかサイクルなのか，といったデータ構造の解析を行う．しかしながら，既存の高度な並列化あるい

はデータローカリティ最適化技術 [Wol96, LLL01, 吉田 94] は主に多次元配列とループによる処理を対象としており，再帰的なデータ構造には対応していないのが現状である．そのため，再帰的なデータ構造に対する並列化技術と併せた解析手法の改良が今後の課題となる．

- Bridges ら [BVZ+07] は逐次プログラミングモデルを拡張し，逐次プログラムにおいて一定の幅のある計算結果を許容することで，並列プログラミング時と同様に並列性を利用することを提案している．計算結果の非決定性を明示的に記述可能とすることで，並列プログラミングと同様の並列性の抽出と，逐次処理と同様のプログラミングのしやすさを両立できる可能性がある．このような非決定性の導入と逐次プログラミングガイドラインの併用は今後の課題となる．また，逐次プログラムの自動並列化技術と明示的に記述された並列性の併用についても今後の課題となる．

- コンパイラによるスタティックな低消費電力化制御手法と併せて，プログラムの実行時に電力状態の指定を行う手法を組み合わせることは今後の課題となる．また，ユーザプログラムによる電力制御と，OS による電力制御を含めた資源管理が協調する枠組みの構築については今後の課題となる．

- 既存のマルチコアハードウェアで動作するソフトウェアのみによるコヒーレンシ制御手法を提案したが，ハードウェアサポートを利用することによるさらなる性能向上については今後の課題となる．

参考文献

[ABD+03] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, Vol. 36, pp. 49–58, December 2003.

[ACD+96] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, Vol. 29, pp. 18–28, 1996.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers : Principles, techniques, and tools second edition. *Pearson Education, Inc*, 2007.

[And94] Lars O. Andersen. *Program Analysis and Specialization for the C Program Language*. PhD thesis, University of Copenhagen, May 1994.

[BLQ+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM*

SIGPLAN 2003 conference on Programming language design and implementation, PLDI '03, pp. 103–114, New York, NY, USA, 2003. ACM.

[BVZ+07] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multicore. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[C9999] *ISO/IEC 9899:1999 - Programming Language C*, 1999.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek 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, New York, NY, USA, 2005. ACM.

[Cle] *Clean C*. http://www.imec.be/CleanC/.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. Parallel computer architecture - a hardware / software approach. *Morgan Kaufmann Publishers, Inc*, 1999.

[CY00] Lynn Choi and Pen-Chung Yew. Hardware and compiler-directed cache coherence in large-scale multiprocessors: Design considerations

and performance study. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 11, pp. 375–394, April 2000.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pp. 242–256, New York, NY, USA, 1994. ACM.

[EHP98] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 9, pp. 5–23, January 1998.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, p. 83, 2006.

[GH96a] Rakesh Ghiya and Laurie Hendren. Connection analysis: A practical interprocedural heap analysis for c. In Chua-Huang Huang, Ponnuswamy Sadayappan, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 8th international workshop on Languages and compilers for parallel computing*, Vol. 1033 of *LCPC '95*, pp. 515–533. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0014221.

[GH96b] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pp. 1–15, New York, NY, USA, 1996. ACM.

[GMO+00] Marc González, Xavier Martorell, José Oliver, Eduard Ayguadé, and Jesús Labarta. Code generation and run-time support for multi-level parallelism exploitation. In *Proceedings of the 8th International Workshop on compilers for parallel computing*, CPC '00, Jan. 2000.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, ISSCC '10, pp. 108 –109, feb. 2010.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on*

Program analysis for software tools and engineering, PASTE '01, pp. 54–61, New York, NY, USA, 2001. ACM.

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pp. 38–48, New York, NY, USA, 2003. ACM.

[HL09] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pp. 226–238, New York, NY, USA, 2009. ACM.

[HRU+07] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pp. 754–759, New York, NY, USA, 2007. ACM.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pp. 254–263, New York, NY, USA, 2001. ACM.

[IHY+08] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara. An 8640 mips soc with independent power-off control of 8 cpus and 8 rams by an automatic parallelizing compiler. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 90 –598, feb. 2008.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pp. 179–188, New York, NY, USA, 1995. ACM.

[Kah08] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pp. 249–259, New York, NY, USA, 2008. ACM.

[KCRB03] Mahmut Kandemir, Alok Choudhary, J. Ramanujam, and Prith Banerjee. Reducing false sharing and improving spatial locality in a unified compilation framework. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 14, pp. 337–354, April 2003.

[KHM+92] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme for oscar (optimally scheduled advanced multiprocessor). In *Proceedings of the*

4th International Workshop on Languages and Compilers for Parallel Computing, LCPC '91, pp. 283–297, London, UK, 1992. Springer-Verlag.

[KJJ+09] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 140–151, New York, NY, USA, 2009. ACM.

[KJL+09] John H. Kelm, Daniel R. Johnson, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. A task-centric memory model for scalable accelerator architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pp. 77–87, Washington, DC, USA, 2009. IEEE Computer Society.

[KMM+09] Keiji Kimura, Masayoshi Mase, Hiroki Mikami, Takamichi Miyamoto, Jun Shirako, and Hironori Kasahara. Oscar api for real-time low-power multicores and its performance on multicores and smp servers. In *Proceedings of the 22nd Internationl Workshop on Languages and Compilers for Parallel Computing*, LCPC '09, pp. 188–202, 2009.

[LAS+07] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th*

annual international symposium on Computer architecture, ISCA '07, pp. 358–368, New York, NY, USA, 2007. ACM.

[LCL99] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pp. 228–237, New York, NY, USA, 1999. ACM.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, Vol. 7, No. 4, pp. 321–359, 1989.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pp. 278–289, New York, NY, USA, 2007. ACM.

[LLL01] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pp. 103–112, New York, NY, USA, 2001. ACM.

[LMH04] Jian Li, Jose F. Martinez, and Michael C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pp. 14–, Washington, DC, USA, 2004. IEEE Computer Society.

[LP05] Mirko Loghi and Massimo Poncino. Exploring energy/performance tradeoffs in shared memory mpsocs: Snoop-based cache coherence vs. software solutions. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pp. 508–513, Washington, DC, USA, 2005. IEEE Computer Society.

[MIS04] *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*, 2004.

[MJCP08] Aqeel Mahesri, Daniel Johnson, Neal Crago, and Sanjay J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pp. 164–175, Washington, DC, USA, 2008. IEEE Computer Society.

[NKH04] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pp. 43–48, New York, NY, USA, 2004. ACM.

[NTF+08] S. Nomura, F. Tachibana, T. Fujita, Chen Kong Teh, H. Usui, F. Yamane, Y. Miyamoto, C. Kumtornkittikul, H. Hara, T. Yamashita, J. Tanabe, M. Uchiyama, Y. Tsuboi, T. Miyamori, T. Kitahara, H. Sato, Y. Homma, S. Matsumoto, K. Seki, Y. Watanabe, M. Hamada, and M. Takahashi. A 9.7mw aac-decoding, 620mw h.264 720p 60fps decoding, 8-core media processor with embedded forward-body-biasing and power-gating circuit in 65nm cmos technology. In

Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, pp. 262 –612, feb. 2008.

[ope05] *OpenMP Application Program Interface Version 2.5*, 2005.

[OSC] *Optimaly Scheduled Advanced Multiprocessor Application Program Interface (OSCAR API) version 1.0.* http://www.kasahara.cs.waseda.ac.jp/.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for c. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pp. 37–42, New York, NY, USA, 2004. ACM.

[PO05] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 142–152, New York, NY, USA, 2005. ACM.

[RC07] Constantino G. Ribeiro and Marcelo Cintra. Quantifying uncertainty in points-to relations. In *Proceedings of the 19th international workshop on Languages and compilers for parallel computing*, LCPC '06, pp. 190–204, Berlin, Heidelberg, 2007. Springer-Verlag.

[RRH08] Shane Ryoo, Christopher Rodrigues, and Wen-mei Hwu. Iteration disambiguation for parallelism identification in time-sliced applications. In Vikram Adve, María Garzarán, and Paul Petersen, editors, *Proceedings of the 20th international workshop on Languages and compilers*

for parallel computing, Vol. 5234 of *LCPC '07*, pp. 110–124. Springer Berlin / Heidelberg, 2008.

[RUR+07] Shane Ryoo, Sain-Zee Ueng, Christopher I. Rodrigues, Robert E. Kidd, Matthew I. Frank, and Wen-Mei W. Hwu. Automatic discovery of coarse-grained parallelism in media applications. *Transactions on High-Performance Embedded Architectures and Compilers I*, pp. 194–213, 2007.

[SKK+05] T. Shiota, K. Kawasaki, Y. Kawabe, W. Shibamoto, A. Sato, T. Hashimoto, F. Hayakawa, S. Tago, H. Okano, Y. Nakamura, H. Miyake, A. Suga, and H. Takahashi. A 51.2 gops 1.0 gb/s-dma single-chip multi-processor integrating quadruple 8-way vliw processors. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 194–593 Vol. 1, Feb. 2005.

[SSP99] Hideki Saito, Nicholas Stavrakos, and Constantine D. Polychronopoulos. Multithreading runtime support for loop and functional parallelism. In *Proceedings of the Second International Symposium on High Performance Computing*, ISHPC '99, pp. 133–144, London, UK, 1999. Springer-Verlag.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pp. 32–41, New York, NY, USA, 1996. ACM.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 179–196, London, UK, 2002. Springer-Verlag.

[TLH94] J. Torrellas, H. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput.*, Vol. 43, pp. 651–663, June 1994.

[TS10] David Tarjan and Kevin Skadron. The sharing tracker: Using ideas from cache coherence hardware to reduce off-chip memory traffic with non-coherent caches. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[WFPS02] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pp. 262–273, New York, NY, USA, 2002. ACM.

[WJMC04] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pp. 248–259, New York, NY, USA, 2004. ACM.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, Oct. 1991.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pp. 1–12, New York, NY, USA, 1995. ACM.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pp. 131–144, New York, NY, USA, 2004. ACM.

[Wol96] Michael Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.

[YXH+10] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pp. 218–229, New York, NY, USA, 2010. ACM.

[笠原 03] 笠原博徳. 最先端の自動並列化コンパイラ技術. 情報処理, Vol.44 No. 4(通巻 458 号), pp.384-392, Apr 2003.

[間瀬 07] 間瀬正啓, 馬場大介, 長山晴美, 田野裕秋, 益浦健, 宮本孝道, 白子準, 中野啓史, 木村啓二, 笠原博徳. 情報家電用マルチコア smp 実行モードにおける制約付き c プログラムのマルチグレイン並列化. 組込みシステムシンポジウム 2007, October 2007.

[間瀬 09] 間瀬正啓, 木村啓二, 笠原博徳. マルチコアにおける parallelizable c プログラムの自動並列化. 情報処理学会研究会報告 2009-ARC-174-15(SWoPP2009), August 2009.

[間瀬 10] 間瀬正啓, 村田雄太, 木村啓二, 笠原博徳. 自動並列化のための element-sensitive ポインタ解析. 情報処理学会論文誌プログラミング (PRO), Vol. 3, No. 2, pp. 36–47, Mar. 2010.

[吉田 94] 吉田明正, 前田誠司, 尾形航, 笠原博徳. Fortran マクロデータフロー処理におけるデータローカライゼーション手法. 情報処理学会論文誌, Vol. 35, No. 9, pp. 1848–1994, Sep. 1994.

[高山 10] 高山征大, 境隆二, 加藤宣弘, 島田智文. 並列プログラミングモデル molatomium. 情報処理学会論文誌プログラミング (PRO), Vol. 3, No. 1, pp. 54–62, Mar. 2010.

[小高 05] 小高剛, 中野啓史, 木村啓二, 笠原博徳. チップマルチプロセッサ上での mpeg2 エンコードの並列処理. 情報処理学会論文誌, Vol. 46, No. 9, Sep. 2005.

[小幡 03] 小幡元樹, 白子準, 神長浩気, 石坂一久, 笠原博徳. マルチグレイン並列処理のための階層的並列処理制御手法. 情報処理学会論文誌, Vol. 44, No. 4, Apr. 2003.

[石坂 02] 石坂一久, 中野啓史, 八木哲志, 小幡元樹, 笠原博徳. 共有メモリマルチプロセッサ上でのキャッシュ最適化を考慮した粗粒度タスク並列処理. 情報処理学会論文誌, Vol. 43, No. 4, Apr. 2002.

[白子 06] 白子準, 吉田宗弘, 押山直人, 和田康孝, 中野啓史, 鹿野裕明, 木村啓二, 笠原博徳. マルチコアプロセッサにおけるコンパイラ制御低消費電力化手法. 情報処理学会論文誌, Vol. 47, No. ACS15, 2006.

[本多 90] 本多弘樹, 岩田雅彦, 笠原博徳. Fortran プログラム粗粒度タスク間の並列性検出手法. 電子情報通信学会論文誌, Vol. J73-D-1, No. 12, pp. 951–960, Dec. 1990.

[木村 01] 木村啓二, 加藤孝幸, 笠原博徳. 近細粒度並列処理用シングルチップマルチプロセッサにおけるプロセッサコアの評価. 情報処理学会論文誌, Vol. 42, No. 4, Apr. 2001.

謝辞

本研究は，著者が早稲田大学大学院博士後期課程在学中になされたものである．本研究を進めるにあたり，終始あたたかいご指導，ご支援を賜りました，木村啓二准教授に心から感謝致します．また，本論文をまとめるにあたり，有益なご助言とご指導をいただきました，笠原博徳教授，小林哲則教授，戸川望教授に深く感謝致します．

本研究の一部はNEDO “低消費電力メニーコア・プロセッサシステム技術（グリーンIT プロジェクト）の先導研究”，NEDO “情報家電用ヘテロジニアスマルチコア技術”，NEDO “リアルタイム情報家電用マルチコア技術”，及び早稲田大学グローバルCOE “アンビエントSoC”，の支援により行われました．様々な機会でディスカッションやアドバイスをいただきました皆様に深く感謝致します．

本研究における各段階では，共にコンパイラやアーキテクチャの研究・開発を進めてきました笠原・木村研究室の皆様に数多くのご助言，ご協力をいただきました．ここに深く感謝致します．

本研究は笠原・木村研究室で研究・開発を続けているOSCARコンパイラについて，さらに機能拡張を実装しながら行われました．素晴らしいソフトウェアを残して下さいました皆様に感謝致します．

OSCARコンパイラとともに，大変有益なドキュメントを残して下さいました，石坂一久氏に感謝致します．本研究を進める上での道標となりました．

本研究において数多くのディスカッションと有益なアドバイスをいただきまし

た，中野啓史氏，白子準氏，宮本孝道氏，和田康孝氏，鹿野裕明氏，鷹野芙美代氏に感謝致します．本論文の成果は先輩方のご助言，ご協力あってのものです．

また，当時のC班，電力班，ローカライズ班，ヘテロ班等の研究グループにおいて共に研究を行いました，深津幸二氏，押山直人氏，田川友博氏，三浦剛氏，長山晴美氏，馬場大介氏，益浦健氏，山田真也氏，松本繁氏，山田海斗氏，吉田宗弘氏，池見明紀氏，中川正洋氏，神山輝壮氏，田野裕秋氏，村田雄太氏，中川亮氏，今泉和浩氏，高木翔氏，田中裕士氏，村松裕介氏，桃園拓氏，八木勇樹氏，園田訓之氏，林真人氏，渡辺岳志氏，大越久氏をはじめとする，卒業生の皆様には数多くのご助言，ご協力をいただきました．現在のOSCARコンパイラのC言語対応や情報家電用マルチコア向けの機能拡張は，皆様のご協力のおかげで実現することができました．ありがとうございました．

そして，本論文の成果は現在の笠原・木村研究室の，林明宏氏，見神広紀氏，島岡護氏，大國直人氏，小野崎雄人氏，小林直生氏，関口威氏，山本修平氏，山本哲也氏，網本達紀氏，石田瑞穂氏，大橋拓也氏，田中雄大氏，田端啓一氏，佐藤崇文氏，佐藤卓也氏，滝西哲也氏，北基俊平氏，植村淳史氏，島崎将氏，鈴木裕貴氏，春木祐香氏，三田功輔氏，師岡直輝氏，阿部高大氏，金羽木洋平氏，須田昌和氏，竹本昂生氏，三澤秀明氏をはじめとする，学生の皆様のご協力あってのものです．皆様に深く感謝致します．

最後に，研究生活を支えて下さった家族に感謝致します．

著者研究業績

著者研究業績

種類別	題名	発表掲載誌名	発表年月	著者名
論文	Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-time Heterogeneous Multicores	Proceedings of The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)	2010 年 10 月	A. Hayashi Y. Wada T. Watanabe T. Sekiguchi M. Mase J. Shirako K. Kimura H. Kasahara
○	Parallelizable C and Its Performance on Low Power High Performance Multicores	Proceedings of The 15th Workshop on Compilers for Parallel Computing (CPC2010)	2010 年 7 月	M. Mase Y. Onozaki K. Kimura H. Kasahara
○	自動並列化のための Element-Sensitive ポインタ解析	情報処理学会論文誌プログラミング (PRO), Vol. 3, No. 2, pp. 36-47	2010 年 3 月	間瀬 正啓 村田 雄太 木村 啓二 笠原 博徳
	OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers	Proceedings of The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2009)	2009 年 10 月	K. Kimura M. Mase H. Mikami T. Miyamoto J. Shirako H. Kasahara
○	マルチコア上での OSCAR API を用いた並列化コンパイラによる低消費電力化手法	情報処理学会論文誌コンピューティングシステム (ACS), Vol. 2, No. 3, pp. 96-106	2009 年 9 月	間瀬 正啓 中川 亮 大國 直人 白子 準 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
論文	マルチコアプロセッサ上での粗粒度タスク並列処理のためのコンパイラによるローカルメモリ管理手法	情報処理学会論文誌コンピューティングシステム (ACS), Vol. 2, No. 2, pp.63-74	2009 年 7 月	中野 啓史 桃園 拓 間瀬 正啓 木村 啓二 笠原 博徳
	Green Multicore SoC Software-Execution Framework with Timely-Power-Gating Scheme	Proceedings of The IEEE International Conference on Parallel Processing (ICPP2009)	2009 年 6 月	M. Onouchi K. Toyama T. Nojiri M. Sato M. Mase J. Shirako M. Sato M. Takada M. Ito H. Mizuno M. Namiki K. Kimura H. Kasahara
	マルチコア上での OSCAR API を用いた並列化コンパイラによる低消費電力化手法	先進的計算基盤システムシンポジウム (SACSIS2009), pp. 3-10	2009 年 5 月	中川 亮 間瀬 正啓 大國 直人 白子 準 木村 啓二 笠原 博徳
	Performance of OSCAR Multigrain Parallelizing Compiler on Multicore Processors	Proceedings of The 14th Workshop on Compilers for Parallel Computing (CPC2009)	2009 年 1 月	H. Mikami J. Shirako M. Mase T. Miyamoto H. Nakano F. Takano A. Hayashi Y. Wada K. Kimura H. Kasahara

著者研究業績

種類別	題名	発表掲載誌名	発表年月	著者名
論文	Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API	Proceedings of IEEE International Symposium on Advances in Parallel and Distributed Computing Techniques (APDCT-08)	2008 年 12 月	T. Miyamoto S. Asaka H. Mikami M. Mase Y. Wada H. Nakano K. Kimura H. Kasahara
	情報家電用マルチコア並列化 API を生成する自動並列化コンパイラによる並列化の評価	情報処理学会論文誌コンピューティングシステム (ACS), Vol. 1, No. 3, pp.83-95	2008 年 12 月	宮本 孝道 浅香 沙織 見神 広紀 間瀬 正啓 木村 啓二 笠原 博徳
	情報家電用マルチコア上におけるマルチメディア処理のコンパイラによる並列化	先進的計算基盤システムシンポジウム (SACSIS2008)	2008 年 5 月	宮本 孝道 浅香 沙織 見神 広紀 間瀬 正啓 木村 啓二 笠原 博徳
	An 8 CPU SoC with Independent Power-off Control of CPUs and Multicore Software Debug Function	Proceedings of IEEE Cool Chips XI: Symposium on Low-Power and High-Speed Chips 2008	2008 年 4 月	Y. Yoshida M. Ito K. Hayase T. Hayashi O. Nishii T. Hattori J. Sakiyama M. Takada K. Uchiyama J. Shirako M. Mase K. Kimura H. Kasahara

種類別	題名	発表掲載誌名	発表年月	著者名
論文	An 8640 MIPS SoC with Independent Power-off Control of 8 CPU and 8 RAMS by an Automatic Parallelizing Compiler	Proceedings of IEEE International Solid State Circuits Conference (ISSCC2008)	2008 年 2 月	M. Ito T. Hattori Y. Yoshida K. Hayase T. Hayashi O. Nishii Y. Yasu A. Hasegawa M. Takada M. Ito H. Mizuno K. Uchiyama T. Odaka J. Shirako M. Mase K. Kimura H. Kasahara
	情報家電用マルチコア SMP 実行モードにおける制約付き C プログラムのマルチグレイン並列化	組込みシステムシンポジウム 2007	2007 年 10 月	間瀬 正啓 馬場 大介 長山 晴美 田野 裕秋 益浦 健 宮本 孝道 白子 準 中野 啓史 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
講演（研究会）	OSCAR API 標準解釈系を用いた Parallelizable C プログラムの評価	情報処理学会研究会報告 Vol.2010-ARC-191-2	2010 年 10 月	佐藤 卓也 見神 広紀 林 明宏 間瀬 正啓 木村 啓二 笠原 博徳
	情報家電用ヘテロジニアスマルチコア用自動並列化コンパイラフレームワーク	情報処理学会研究会報告 Vol.2010-ARC-190-7 (SWoPP2010)	2010 年 8 月	林 明宏 和田 康孝 渡辺 岳志 関口 威 間瀬 正啓 木村 啓二 伊藤 雅之 長谷川 淳 佐藤 真琴 野尻 徹 内山 邦男 笠原 博徳
	情報家電用ヘテロジニアスマルチコア RP-X におけるコンパイラ低消費電力制御性能	情報処理学会研究会報告 Vol.2010-ARC-190-8 (SWoPP2010)	2010 年 8 月	和田 康孝 林 明宏 渡辺 岳志 関口 威 間瀬 正啓 白子 準 木村 啓二 伊藤 雅之 長谷川 淳 佐藤 真琴 野尻 徹 内山 邦男 笠原 博徳
	並列化コンパイラによるソフトウェアコヒーレンシ制御	情報処理学会研究会報告 Vol.2010-ARC-189-7 Vol.2010-OS-114-7	2010 年 4 月	間瀬 正啓 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
講演（研究会）	自動並列化技術を用いたメディア処理オフロード	情報処理学会研究会報告 Vol.2010-EMB-16-59	2010 年 3 月	石坂 一久 酒井 淳嗣 枝廣 正人 宮本 孝道 間瀬 正啓 木村 啓二 笠原 博徳
	組込み向けマルチコア上での複数アプリケーション動作時の自動並列化されたアプリケーションの処理性能	情報処理学会研究会報告 Vol.2010-ARC-188-9	2010 年 3 月	宮本 孝道 間瀬 正啓 木村 啓二 石坂 一久 酒井 淳嗣 枝廣 正人 笠原 博徳
	自動並列化のための Element-Sensitive ポインタ解析	情報処理学会第 76 回プログラミング研究会	2009 年 10 月	間瀬 正啓 村田 雄太 木村 啓二 笠原 博徳
	マルチコアにおける Parallelizable C プログラムの自動並列化	情報処理学会研究会報告 Vol.2009-ARC-174-15 (SWoPP2009)	2009 年 8 月	間瀬 正啓 木村 啓二 笠原 博徳
	メディアアプリケーションを用いた並列化コンパイラ協調型ヘテロジニアスマルチコアアーキテクチャのシミュレーション評価	社団法人 電子情報通信学会, 信学技報, ICD2008-140	2009 年 1 月	神山 輝壮 和田 康孝 林 明宏 間瀬 正啓 中野 啓史 渡辺 岳志 木村 啓二 笠原 博徳

著者研究業績

種類別	題名	発表掲載誌名	発表年月	著者名
講演（研究会）	マルチコアのためのコンパイラにおけるローカルメモリ管理手法	社団法人 電子情報通信学会, 信学技報, ICD2008-141	2009 年 1 月	桃園 拓 中野 啓史 間瀬 正啓 木村 啓二 笠原 博徳
	マルチコア上での OSCAR API を用いた低消費電力化手法	社団法人 電子情報通信学会, 信学技報, ICD2008-145	2009 年 1 月	中川 亮 間瀬 正啓 白子 準, 木村 啓二 笠原 博徳
	ポインタ解析を用いた制約付き C プログラムの自動並列化	情報処理学会研究会報告 Vol.2008-ARC-178-4	2008 年 5 月	間瀬 正啓 馬場 大介 長山 晴美 村田 雄太 木村 啓二 笠原 博徳
	階層グルーピング対応バリア同期機構の評価	情報処理学会研究会報告 Vol.2008-ARC-178-14	2008 年 5 月	山田 海斗 間瀬 正啓 白子 準 木村 啓二 伊藤 雅之 服部 俊洋 水野 弘之 内山 邦男 笠原 博徳
	マルチコアプロセッサ上でのマルチメディア処理の並列化	情報処理学会研究会報告 Vol.2007-ARC-175-15 (デザインガイア 2007)	2007 年 11 月	宮本 孝道 田村 圭 田野 裕秋 見神 広紀 浅香 沙織 間瀬 正啓 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
講演（研究会）	情報家電用マルチコアSMP実行モードにおけるマルチグレイン並列処理	情報処理学会研究会報告 2007-ARC-173-5	2007年5月	間瀬 正啓 馬場 大介 長山 晴美 田野 裕秋 益浦 健 宮本 孝道 白子 準 中野 啓史 木村 啓二 亀井 達也 服部 俊洋 長谷川 淳 佐藤 真琴 伊藤 雅樹 内山 邦男 小高 俊彦 笠原 博徳
	OSCARコンパイラにおける制約付きCプログラムの自動並列化	情報処理学会研究会報告 2006-ARC-170-01 (デザインガイア 2006)	2006年11月	間瀬 正啓 馬場 大介 長山 晴美 田野 裕秋 益浦 健 深津 幸二 宮本 孝道 白子 準 中野 啓史 木村 啓二 笠原 博徳

著者研究業績

種類別	題名	発表掲載誌名	発表年月	著者名
その他（ポスタ発表）	Multigrain Parallelization of Restricted C Programs on SMP Servers and Low Power Multicores	The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC2007)	2007 年 10 月	M. Mase D. Baba H. Nagayama H. Tano T. Masuura T. Miyamoto J. Shirako H. Nakano K. Kimura H. Kasahara
	情報家電用マルチコアのための自動並列化コンパイラ技術	STARC シンポジウム 2007	2007 年 9 月	間瀬 正啓
	OSCAR コンパイラにおける制約付き C プログラムの自動並列化	情報処理学会研究会報告 2006-ARC-170-01(デザインガイア 2006)	2006 年 11 月	間瀬 正啓 馬場 大介 長山 晴美 田野 裕秋 益浦 健 深津 幸二 宮本 孝道 白子 準 中野 啓史 木村 啓二 笠原 博徳
（招待講演）	C Language Support in OSCAR Multigrain Parallelizing Compiler using CoSy	ACE 2nd CoSy Community Gathering, Amsterdam, Netherlands	2006 年 10 月	M. Mase
（特許）	プロセッサによって実行可能なコードの生成方法、記憶領域の管理方法及びコード生成プログラム	特願 2009-285586	2009 年 12 月.	笠原 博徳 木村 啓二 間瀬 正啓

種類別	題名	発表掲載誌名	発表年月	著者名
その他 （受賞）	自動並列化のための Element-Sensitive ポインタ解析	情報処理学会 山下記念研究賞	2010 年 7 月	間瀬 正啓
	情報家電用マルチコアのための自動並列化技術	STARC シンポジウム 2007 優秀ポスター賞	2007 年 9 月	間瀬 正啓
	OSCAR コンパイラによる制約付き C プログラムの自動並列化	電子情報通信学会・情報処理学会 デザインガイア ポスタ賞	2006 年 11 月	間瀬 正啓