情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 3　235–247 (Sep. 2010)

集約制御機構を持つ コア間時間集約スケジューラの実装と評価

山田　賢[†1]　日下部　茂[†2]

本論文では Chip Multi-Processor（以下，CMP）向けに我々が提案したコア間時間集約スケジューラ（以下，IAS）対して，集約制御機構を実装したうえでその評価について述べる．IAS は，メモリアドレス空間を共有するスレッド（以下，シブリングスレッド）間では参照するデータを共有する可能性が高い，という前提の下に，シブリングスレッドを同一コア上，および，異なるコア間で集約して実行する．集約実行により，スレッド間で共有するデータに関するキャッシュミスを削減することができると考える．我々はこれまでにシブリングスレッドの集約機構のみを実装し，汎用のマルチスレッドプログラムを実行した際にスループットを向上させる効果があることを確認した．本論文では従来の IAS に対し，集約の是非や集約回数の制限，集約の強度をメモリアドレス空間ごとに設定できるように拡張する．汎用の CMP を用いた実測により，まず集約回数の制限がスループットに与える効果を示す．さらにプログラムの性質の静的な分析に基づいて集約の是非を設定することで，システム全体のスループットをさらに向上させることができることを示す．

Implementation and Evaluation of Inter-core Time Aggregation Scheduler with an Aggregation Control Mechanism

Satoshi Yamada[†1] and Shigeru Kusakabe[†2]

In this paper, we implement and evaluate an aggregation control mechanism of Inter-Core Time Aggregation Scheduler (IAS) for Chip Multi-Processor (CMP). We assume that there exists locality of reference between threads sharing the same memory address space (sibling threads). IAS aggregates sibling threads on both a single processing core (Core) and different Cores at the same time to utilize the locality of sibling threads and reduce cache misses on CMPs. Previously, we implemented only an aggregation mechanism of IAS and showed its effect in enhancing the throughput of commodity multi-threaded programs. We extend previous IAS with the control mechanism of the aggregation of sibling threads such as the strength and the limitation of the aggregation per single memory address space. We show the effect of the extended IAS to control the throughput of programs by limiting the aggregation counts on a commodity CMP platform. Additionally, we show that we can enhance the throughput of a system by setting the strength of the aggregation based on a static analysis of executed programs.

1. はじめに

本論文では Chip Multi-Processor（以下，CMP）向けに我々が提案したコア間時間集約スケジューラ（以下，IAS）に対して，集約制御機構を実装したうえでその評価について述べる．IAS は Chip Multi-Processor（以下，CMP）上で，マルチスレッドプログラムにおけるスレッド間の参照の局所性を活用する，カーネルレベルのスレッドスケジューラである．本論文におけるマルチスレッドプログラムとは，メモリアドレス空間を共有するカーネルレベルスレッド（以下，シブリングスレッド）を複数生成し，スレッドレベルの並行/並列な処理を行うプログラムを意味する．我々は複数のマルチスレッドプログラムを同時に実行するような状況を想定し，スレッドスケジューリングによるプログラム実行の性能向上を目指す．具体的な事例としては，マルチスレッド化された Web サーバやアプリケーションサーバなどを同時に実行する Web アプリケーションなどがある．

IAS は，メモリアドレス空間を共有するスレッド間では参照するデータを共有する可能性が高い，という前提の下に，シブリングスレッドを同一コア上と異なるコア間で集約して実行する．IAS の効果はキャッシュの活用という観点から，以下の 2 つに分類できる．1 つはプロセッサコア（以下，コア）ごとに持つキャッシュの活用である．シブリングスレッドを単一コア上で連続に実行する場合，実行するプログラムの性質によっては，スレッド間で共有するデータをキャッシュ上に集約し，スループットの向上や実行時間の削減といった効果を期待できる．特に Intel の x86 といったプロセッサでは，エントリの無効化をメモリアドレス空間の切替えを機会にハードウェアが自動的に行う．このようなプロセッサ上では，シブリングスレッドの集約実行によりメモリアドレス空間切替え頻度を削減することで，実行性能を向上させることができる可能性が高い．IAS の効果の 2 点目としては，コア間で

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

共有するキャッシュの活用である．近年では CMP や Chip Multi-Threading といった単一チップ上に複数のコアを持つプロセッサが普及している．マルチコアプロセッサにおいてはコア間でキャッシュなどの計算資源を共有していることから，異なるコア間で同時に実行するスレッドの組合せが性能に影響を与えることが知られている[2)–7)]．我々は従来の時分割方式[16)] のように，参照の局所性が存在すると考えられるシブリングスレッドを同時に異なるコア間で実行することにより，スレッド間で共有するデータをコア間で共有するキャッシュ上に集約し，キャッシュミスを削減することができると考える．

我々はこれまでに，単一プロセッサ上でシブリングスレッドを時間軸上で集約して実行する時間集約スケジューラ（以下，TAS）を，Linux を用いて実装しその効果について検証を行ってきた[1)]．さらに TAS を拡張して複数のコアでのシブリングスレッドの時間集約実行を促す方式で IAS における集約機構の実装を行った．IAS の特徴は各コアで独立したスケジューラを動かしつつも，単一のフラグのみを介した軽量なコア間の協調スケジューリングを $O(1)$ で行うことが可能であるという点である．我々はこれまでに，データベースサーバのような汎用のマルチスレッドプログラムを実行した際にスループットを向上させる効果があることを確認した[9)]．

本論文では従来の IAS に対し，集約の是非や強度，集約回数の制限といった集約制御機能をメモリアドレス空間ごとに設定できるように拡張する．これにより，プログラムの特性に合わせた効率的な集約状況の設定や，過剰な集約の抑制を行うことが可能となる．IAS はスレッドレベル並列処理の最適なスケジューリングを保証するものではないが，マルチコアプロセッサの普及にともなうプログラムのマルチスレッド化を考慮した際に，現実的かつ効率的なスケジューラとして有用であると考える．

本論文では集約制御機構を持つ IAS を，汎用的かつ現実的なワークロードの実装である DaCapo ベンチマーク[12)] を用いて検証する．具体的には，集約制限数の設定により，スループットの向上効果が制御可能であることを示す．さらに，プログラムの性質の静的な分析に基づいて集約の是非を設定することで，画一的にシブリングスレッドを集約するよりもシステム全体のスループットを向上させることができることを示す．

本論文の構成を以下に述べる．2 章では，関連研究を紹介する．3 章では，集約制御機構を付加した IAS の実装について示す．4 章で予備評価として SysBench[11)] の `memory` プログラムを用いた評価について示す．`memory` プログラムは IAS の効果を示すベストケースの 1 つであり，スレッド間で共有するデータサイズとコア間で共有する L2 キャッシュサイズに対する IAS の効果の関連を示すことができる．5 章で DaCapo を用いて付加した集約制御機構の観点に基づいた評価を行う．6 章でまとめる．

2. 関 連 研 究

1 章で述べたように，マルチコアプロセッサにおいては従来の共有メモリ型マルチプロセッサと異なりコア間でキャッシュを共有することが一般的であることから，これを活用するためのスレッドスケジューリング手法について研究がさかんに行われている．この中で多く提案されているスケジューラは，実行時間をスレッドの実行時情報のサンプリングとこの実行時情報に基づいたスケジューリングのフェーズに分割し，2 つのフェーズを繰り返す手法である[4),5),7),13)]．たとえば，Parekh らは実行可能なスレッドの各組合せにおける L2 キャッシュや TLB のミス数を実際に実行しつつ計測し，これに基づいてスループットを向上させる組合せの決定を行う手法を提案している[5)]．これらの手法は理論的にはどのようなプログラムの実行時にも適用可能であるという利点がある一方で，サンプリングにともなうオーバヘッドを考慮する必要がある[14)]．また，実行時情報を得られたとしても，これを基に行うスケジューリングの最適化問題は，2 コアの場合クラス P，3 コア以上で NP 完全問題であることが知られている[6)]．そのため，我々が想定するような多くのスレッドの同時実行時においては，スレッド情報のサンプリングおよびスケジューリングのオーバヘッドが高くなり，現実的ではないという指摘がある[2),14),15)]．これに対して，我々の手法は最適なスケジューリングを保証する手法ではないが，CFS に付加して動的に管理する情報は各スレッドのメモリアドレス空間のみであり，また，スケジューリングも $O(1)$ で行えるという利点がある．

我々の手法はシブリングスレッド間での参照の局所性を利用することで，キャッシュ上への共有データの集約と実行性能の向上を図る．Ziemba らは，Web アプリケーションサーバ実行時にシブリングスレッドごとに実行コアを固定する手法を提案している．この手法は従来より空間分割方式として知られる手法であり[16)]，共有メモリ型マルチプロセッサ上でプロセッサごとのローカルキャッシュを活用する効果が知られている．Ziemba らの研究では，マルチコアプロセッサ上においても空間分割方式による性能向上効果が期待できることを示した[17)]．ただし，空間分割方式においてはシブリングスレッドごとの実行コアの固定を適切に行うことができなかった場合に，コア間の負荷が偏るという問題がある．これに対して，我々の手法はスレッドの実行コアを固定することなく，シブリングスレッドの時分割，および，空間分割実行の効果が期待できる．

我々の手法はシブリングスレッド実行時にのみ適用可能という条件があるが，近年では

POSIX や Java，Perl といった言語，および，OpenMP，MPI，Open64 といったコンパイラで生成されたスレッドがネイティブスレッドと 1 対 1 に対応し，これらの並列実行がサポートされていることから，我々のスケジューラが適用できる機会が増加していると考える．また，我々のスケジューラの効果はプログラムのスレッドの挙動によるところが大きく，マルチスレッド化されたプログラムであっても集約実行によりその性能が下がる場合がある[18]．これは I/O の待ち時間を隠蔽するためにスレッド化したものの，扱うデータのサイズが大きいために，コア間でメモリの競合が発生しやすいプログラムなどで顕著である．しかし，マルチコア環境におけるメモリ階層の活用を意図したプログラムのスレッド化に関する研究が近年さかんであり，このようなマルチスレッド化の技術が向上すれば我々のスケジューラが有効となる機会も増えると考える．たとえば，Anderson らはマルチコア環境におけるスレッド間の適切なデータ共有をチェックするツールの開発を行っている[19]．また，Chen らは本研究のようなコア間での集約スケジューリングを行う際に，共有キャッシュを活用するうえで効果的なスレッドの粒度を決定するコンパイラの開発を行っている[20]．上記のようなツールを用いてもなお集約の効果が現れないプログラムに対しては，本論文で付加する IAS の集約機能により対処できると考える．あるプログラムを集約することでシステム全体の性能が落ちるような場合，そのプログラムのみ集約しないよう設定を行うことでシステム全体の性能を向上させることができると考える．

3. コア間時間集約スケジューラ（IAS）の集約制御機構の実装

本項では IAS の集約制御機構の実装について説明する．IAS は提案済みの TAS の拡張であり，Linux 2.6.24 のスレッドスケジューラである CFS を基に実装した．本章ではまず，従来の IAS の実装について，CFS，TAS，集約機構のみ実装した IAS の順に 3.1 節で説明する．次に，集約制御機構を付加した IAS の実装について 3.2 節で述べる．

3.1 従来の IAS の実装について

3.1.1 Completely Fair Scheduler（CFS）におけるスケジューリングとロードバランス

本章では最初に CFS における単一コア上のスケジューリング方針を述べ，次にコア間でのロードバランスについて述べる．

CFS は名前のとおりスレッド間の公平な CPU 時間の割当てを意図して実装されており，同時刻に始まった静的優先度（`nice` 値）が等しいすべてのスレッドに対して，任意の期間に CPU 時間を等しく割り当てることを目標としている．CPU 時間とはあるスレッドの実行中に実際に CPU を用いた演算を行っている時間を秒単位で示したものであり，ランキュー内での待機時間や I/O 待ちなどで経過する時間は CPU 時間には含まない．CFS では CPU 時間をナノ秒単位でカウントし，スレッドごとの累積した CPU 時間と `nice` 値を基に *vruntime* という優先度を計算している．CFS では *vruntime* が少ないほど割り当てられた累積の CPU 時間が少ないことを意味する．つまり，*vruntime* が少ないスレッドの優先度が高くなり，スレッドが CPU を割り当てられるとその CPU 時間に応じた値が *vruntime* に加算される．こうしてスレッド間での CPU 時間の公平な割当てを実現している．CFS ではランキューはコアごとに存在し，各コア上で独立したスケジューラが実行されている．また，CFS ではスケジューリングの際に各スレッドのメモリアドレス空間を考慮しない．

CFS におけるコア間のロードバランスもコアごとに独立に行われる．CFS においては，カーネルが負荷の均衡を保つべき CPU の集合としてスケジューリングドメインを定義している．スケジューリングドメインはコアやノードといった階層で構成され，各ドメインは物理コアや論理コアの集合であるグループに分割される．CFS は各ドメインでグループ間の負荷の均衡を保つようにロードバランスを行う．CFS では各コア上のスケジューラが，それぞれ各ドメインでのグループ間の負荷の均衡を一定周期ごとに計算する．そして各スケジューラは自らのコアのランキュー（以下，ローカルランキュー）上に，一番負荷が高い他のグループからスレッドを移動することで，負荷の均衡を回復できるかどうかを判断する．ローカルランキューへのスレッドの移動で負荷の均衡を回復すると判断した場合は，実際にスレッドを移動し，また一定周期ごとにこれを繰り返す．負荷の計算時には，各スレッドが持つ `weight`（重み）を考慮する．重みはスレッドの `nice` 値によってカーネル内で定義されている．CFS では，ロードバランスの際にランキュー上で待機しているスレッド数の重みの和を考慮し，各スレッドのメモリアドレス空間を考慮しない．また，スレッドが実行を行うべきコアを明示的に指定することもできるが，実行コアの指定を行わない場合にはすべてのコアで実行を許す．

3.1.2 時間集約スケジューラ（TAS）の概要

TAS は単一コア上で，シブリングスレッドを集約実行を行う．シブリングスレッドの集約実行を実現するために，TAS では現在実行されているスレッドのシブリングスレッドを時間集約候補スレッドとし，これに対して動的に優先度ボーナスを与える．ただし，各スレッドの優先度の設定基準は CFS に従う．もし時間集約候補スレッドとランキュー上で一番優先度が高いスレッドの *vruntime* の差が優先度ボーナス未満であれば，TAS は次に実行するスレッドとして時間集約候補スレッドを選択する．3.1.1 項で述べたように，CFS では

vruntime の値が小さい方が優先度が高い．そのため，TAS のための優先度ボーナスは，現在実行されているスレッドのシブリングスレッドの *vruntime* を削減するように作用する．

実装に際しては以下のメンバを Linux のスレッドやメモリアドレス空間を管理する構造体に追加し，実行時の管理対象とする．

- アドレス空間を共有するスレッドが存在するかどうか認識するためのフラグ `sib_flag`
- シブリングスレッドどうしのリンクをつなぐためのリスト構造体 `mm_link`
- シブリングスレッドどうしのリンクの起点へのポインタ `mm_init`
- 優先度ボーナスを格納する変数 *agg_bonus*

集約実行のために追加する必要があるのは上記の 4 つのメンバのみであり，簡潔な実装が可能である．まず Linux におけるスレッドごとの状態を管理する構造体である `task_struct` 構造体に，アドレス空間を共有するスレッドを認識するためのフラグとなるメンバ `sib_flag` を追加する．あるスレッドが子スレッドを生成する際には，OS が子スレッドの `sib_flag` を立てる．次に，同じくスレッドごとの状態を管理する構造体 `task_struct` 構造体に，同じメモリアドレス空間を共有するスレッドどうしをつなぐリスト構造体である `mm_link` を追加する．`mm_link` には要素数がコアの数だけある配列構造を持たせることで，シブリングスレッドどうしのリンクをコアごとに生成する．シブリングスレッドどうしのリンクがコアごとに存在することで，あるコア上のランキューに存在するあるスレッドのシブリングスレッドの探索時間を削減することができる．カーネルは `sib_flag` が立っているスレッドをランキューに挿入する際に，ランキューが存在するコアの ID をインデックスとする `mm_link` のリンクにそのスレッドをつなぐ．この際，シブリングスレッドのリンクは *vruntime* の昇順にソートされるように生成する．最後にメモリアドレス空間を管理する構造体である `mm_struct` 構造体ごとに，シブリングスレッドどうしのリンクの起点となるポインタ `mm_init` を生成する．`mm_init` も要素数がコアの数だけある配列構造を持たせることで，シブリングスレッドどうしのリンクの起点をコアごとに生成する．スケジューラはスケジューリング時に実行中のスレッドの `sib_flag` をチェックする．フラグが立っていれば実行中のスレッドのメモリアドレス空間に存在する `mm_init[コア ID]` に格納されたポインタをたどって，シブリングスレッドを探索する．シブリングスレッドが同じコア上で存在する場合に，そのスレッドを時間集約候補スレッドとして考慮し，一番優先度の高いスレッドとの比較を行う．このとき用いる優先度ボーナスをカーネル内変数 *agg_bonus* として定義する．

TAS の具体的な動作例を，**図 1** を用いて説明する．図 1 では，あるマルチコアプロセッサのコア ID1 上における時間集約実行の様子を示している．図 1 中の円はそれぞれスレッドであり，スレッド中の数値は *vruntime* を示す．スレッドは四角で示すランキューの中に *vruntime* 順でソートされている[⋆1]．スレッド中のパターンはそれぞれのメモリアドレス空間，破線のリンクはシブリングスレッドの `mm_link` 間のリンクを示しており，リンクの起点は `mm_init[1]` に格納されたポインタである．現在実行されているスレッド（以下，`curr_thrd`）A はランキューから外されており，CFS がスレッド A の次に実行するスレッドは *vruntime* が一番小さいスレッド B である．TAS では，スレッド B のほかに，スレッド A とメモリアドレス空間を共有するスレッド C を次に実行するスレッドの候補とする．このとき，式 (1) を満たした場合，スレッド C を次に実行するスレッドとして選択する．ただし式 (1) において，*B.vruntime*，*C.vruntime* はそれぞれスレッド B，スレッド C の *vruntime* を表し，以降は他のスレッドに関しても同様に示す．

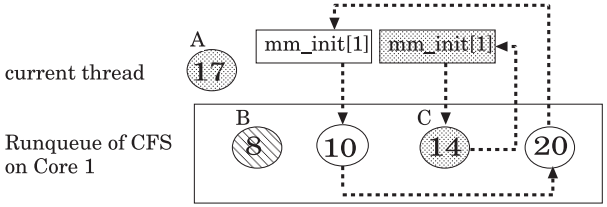

図 1　時間集約スケジューラ（TAS）の動作例
Fig. 1　Case example of Time Aggregation Scheduler.

$$B.vruntime > C.vruntime - agg\_bonus \tag{1}$$

式 (1) を満たさない場合は，スレッド B を次に実行するスレッドとして選択する．図 1 の場合，もし *agg_bonus* を 7 以上に設定すれば，式 (1) を満たすことになりスレッド C を次のスレッドとして実行する．*agg_bonus* の値は我々が実装したシステムコールにより，OS の実行中に変更可能である．

3.1.3　集約機能のみ実装したコア間時間集約スケジューラ（IAS）の概要

TAS から IAS への拡張のポイントは大きく 2 つある．1 つはコア間で同時にシブリングスレッドを集約実行させる機構の追加である．もう 1 つはメモリアドレス空間ごとに優先度ボーナスや集約制限値を設けて，集約を制御する機構の追加である．

コア間で同時にシブリングスレッドを集約実行させる機構についてまず述べる．コア間での集約機能の追加の際に必要なのは，コア間で集約するシブリングスレッドのメモリア

⋆1 CFS のランキューは赤黒木構造を持つが，説明の簡易化のために横並びで示す．

ドレス空間へのポインタを格納する変数 ia_mm のみであり，簡潔に実装可能である．IAS では各コアで TAS を動かし，各コアを master コアと slave コアに分別する．master コアとして指定されたコア上で時間集約されているスレッドのシブリングスレッドが，slave コアとして指定されたコア上に存在する場合に，このスレッドをコア間集約候補スレッドとする．IAS ではコア間集約候補スレッドに slave コア上で一番優先度の高いスレッドや curr_thrd のシブリングスレッドに対する優先度ボーナスを与える．そこで，master コアが時間集約を行った際に，カーネル内の変数（以下，ia_mm）に集約するスレッドどうしが共有するメモリアドレス空間へのポインタを格納する．ia_mm の内容を操作できるのは master コアのみで slave コアは ia_mm を参照するだけである．ia_mm が master コアによってあるメモリアドレス空間にセットされたとき，slave コア上で動くスケジューラは ia_mm が指すメモリアドレス空間を共有するスレッドを，各自のローカルランキューから探す．ia_mm が指すメモリアドレス空間を共有するスレッドが見つかった場合，IAS はそのスレッドを次に実行するスレッドの候補と見なし，IAS のための優先度ボーナスを使って，そのスレッドを実行しようとする．

3.2　コア間時間集約スケジューラ（IAS）への集約制御機構の実装

本節では 3.1.3 項で説明した従来の IAS に対して，メモリアドレス空間ごとの集約制御機構の付加の実装について説明する．メモリアドレス空間ごとに集約を制御する機構については，以下の 3 つの変数をメモリアドレス空間を管理する構造体のメンバとして追加する．

- 集約用の優先度ボーナスを格納する変数 *agg_bonus*
- 単一コア上での累積集約回数を格納する変数 *agg_count*
- 集約回数の制限値を格納する変数 *agg_limit*

集約制御機構を付加する IAS ではメモリアドレス空間ごとの集約度の調整を行うために，TAS で定義した *agg_bonus* をメモリアドレス空間のメンバとして再定義する．スケジューラは時間集約の際には現行スレッドのメモリアドレス空間から，コア間集約の際には ia_mm のメモリアドレス空間からそれぞれ *agg_bonus* を参照する．*agg_count* と *agg_limit* は同一コア上での集約回数を制限するための機構である．コア間時間集約スケジューラは同一コア上での集約のたびに *agg_count* をインクリメントし，*agg_limit* にあらかじめ設定された値との比較を行う．*agg_count* の値が *agg_limit* を超えた場合はシブリングスレッドに対して優先度ボーナスの付与を行わない．*agg_bonus* を調節することにより，スレッド間の消費 CPU 時間の差を一定のレベルに保つことができる．しかし，*agg_bonus* の適用対象である *vruntime* は，実行時の状況に応じてナノ秒単位で計算される値であるため，ユーザがこの

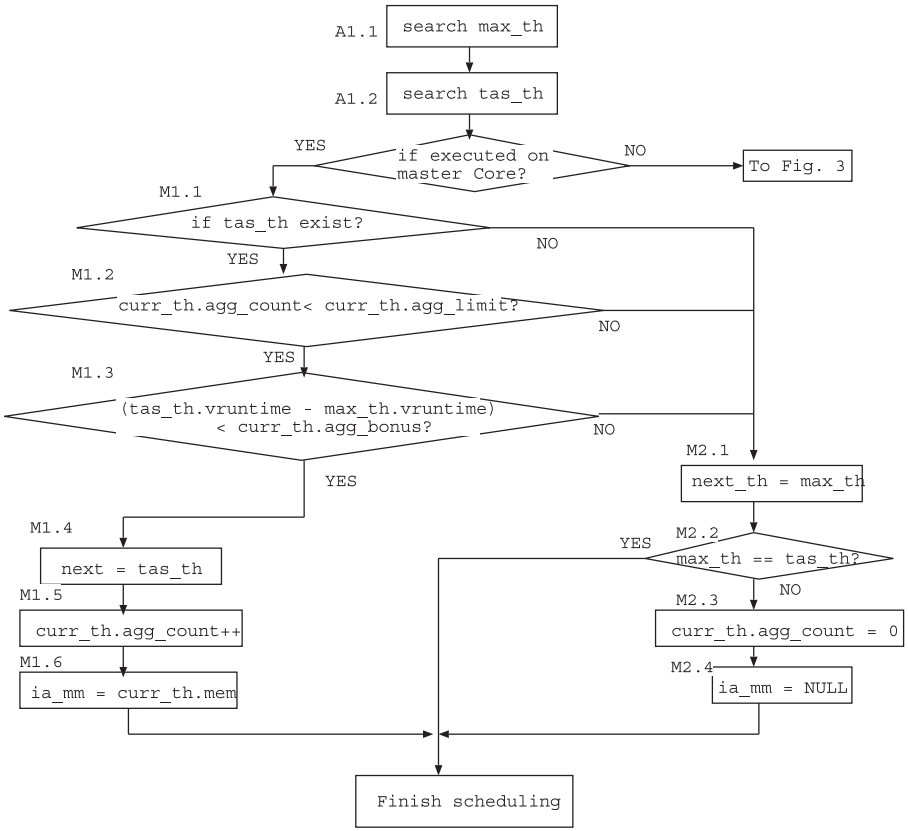

図 2　集約機構を追加したコア間時間集約スケジューラ（IAS）の動作フロー（master）
Fig. 2　Flow of Inter-Core Time Aggregation Scheduler with an aggregation control mechanism (master).

値を調節することは難しいと考える．これに対し，*agg_count* や *agg_limit* を用いることで集約の回数を基準にした集約レベルの制限を行い，スレッド間の CPU を割り振られる回数や消費する CPU 時間といった観点での公平性を保つことができる．

以上の制御機構を実装したスケジューリングのフローチャートを**図 2** と**図 3**，さらに具体的な動作例を**図 4** を用いて示す．ただし，図中および以下の説明では，ランキュー上で一番優先度の高いスレッドを max_th，時間集約候補スレッドを tas_th，コア間集約候補ス

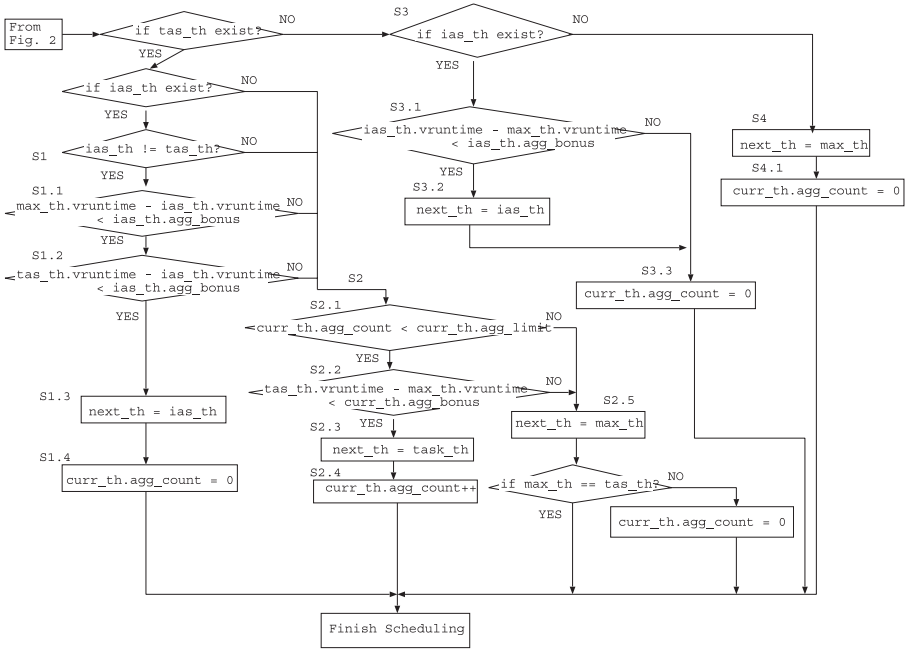

図 3　集約機構を追加したコア間時間集約スケジューラ（IAS）の動作フロー（slave）
Fig. 3　Flow of Inter-Core Time Aggregation Scheduler with an aggregation control mechanism (slave).

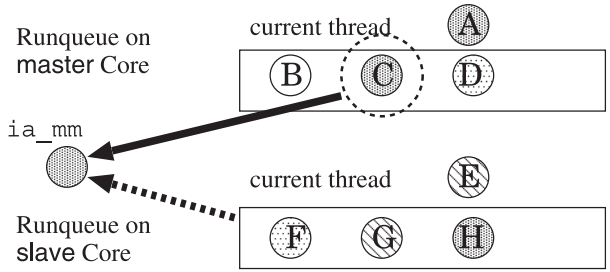

図 4　コア間時間集約スケジューラ（IAS）の動作例
Fig. 4　Case example of Inter-Core Time Aggregation Scheduler.

レッドを `ias_th`，次に実行するスレッドを `next_th` とする．図 1 と同様，図 4 でも円がスレッドを示し，スレッド中のパターンでメモリアドレス空間を表す．図 4 ではデュアルコアマシンでの実行を想定し，図中の四角が 2 つのコア上のそれぞれのランキューの状態を示している．また，図 1 と同様に，優先度の高いスレッドが一番左に位置するものとする．図 4 においては `master` コア上で `tas_th`，`slave` コア上で `tas_th` と `ias_th` が存在する場合を想定する．`master` コア上での `curr_thrd` がスレッド A，`max_th` がスレッド B，スレッド A とメモリアドレス空間 α を共有する `tas_th` がスレッド C である．`slave` コア上では `curr_thrd` がスレッド E，`max_th` がスレッド F，スレッド E とメモリアドレス空間 β を共有する `tas_th` がスレッド G，そしてメモリアドレス空間 α を共有する `ias_th` がスレッド H である．さらに，メモリアドレス空間 α に設定された *agg_limit* を $\alpha.agg\_limit$，`master`，`slave` コア上に設定された *agg_count* をそれぞれ $\alpha.agg\_count.master$，$\alpha.agg\_count.slave$ と表し，メモリアドレス空間 β に関しても同様に表す．また括弧内は図 2，図 3 中の各番号に対応する．

図 2 に示すように，IAS ではまずすべてのコアにおいて `max_th` と `tas_th` を検索する（A1.1，A1.2）．次に `master` コアでは後述の式 (2)，(3) をともに満たすかを判定する（M1.1，M1.2，M1.3）．満たす場合は $\alpha.agg\_count$ をインクリメントして `ia_mm` を α に設定し，スレッド C を次スレッドとして実行する（M1.4，M1.5，M1.6）．満たさない場合はスレッド B を次スレッドに決定し，$\alpha.agg\_count$ を 0，`ia_mm` を NULL にそれぞれ設定する（M2.1，M2.3，M2.4）．ただし，`agg_limit` を超えている状況で `max_th` が `tas_th` と同一となる場合は，`agg_count` および，`ia_mm` の修正は行わない（M2.2）．以降は式 (2)，(3) をともに満たす場合を想定する．

$$\alpha.agg\_count.master < \alpha.agg\_limit \tag{2}$$

$$B.vruntime > C.vruntime - \alpha.agg\_bonus \tag{3}$$

`slave` コアでは（図 3），`tas_th` と `ias_th` の検索を行い，`max_th` との比較を行う．図中では `tas_th` と `ias_th` が別個に存在する場合（S1），`tas_th` のみ存在する場合（S2），`ias_th` のみ存在する場合（S3），さらに集約対象のスレッドが存在しない場合（S4）に分類される．図 4 のような `tas_th` と `ias_th` がともに存在する場合を考慮すると，以下の式 (4)，(5) をともに満たすか判定する（S1.1，S1.2）．ともに満たす場合はスレッド H を次スレッドとして実行（S1.3）し，同時に $\beta.agg\_count.slave$ を 0 に設定しておく（S1.4）．

$$F.vruntime > H.vruntime - \alpha.agg\_bonus \tag{4}$$

$$G.vruntime > H.vruntime - \alpha.agg\_bonus \tag{5}$$

上記の条件を満たさない場合は tas_th と max_th を比較し，以下の式 (6)，(7) を満たすか判定する（S2.1，S2.2）．ともに満たす場合は $\beta.agg\_count.slave$ をインクリメントし，スレッド G を次スレッドとして実行する（S2.3，S2.4）．満たさない場合は $\beta.agg\_count.slave$ を 0 に設定し，スレッド F を次スレッドとして実行する（S2.5）．tas_th のみ存在し，ias_th が存在しない場合にも同じ挙動を行う．

$$\beta.agg\_count.slave < \beta.agg\_limit \tag{6}$$

$$F.vruntime > G.vruntime - \beta.agg\_bonus \tag{7}$$

また，ias_th のみ存在する場合には，図 3 の S3 からのフローチャートで示すように，ias_th と max_th との比較を行って次のスレッドを決定した後，$curr\_th.agg\_count$ を 0 に戻す（S3.1，S3.2，S3.3）．ias_th も tas_th も存在しない場合は，次に実行するスレッドに max_th を指定し，$curr\_th.agg\_count$ を 0 に戻す（S4.1，S4.2）．

シブリングスレッドの検索においては，3.1.2 項で示したシブリングスレッドごとのリンクを用いる．このリンクは $vruntime$ の昇順にソートされているため，検索およびスケジューリングの計算量は CFS と同じく $O(1)$ である．

スレッドの各コアへの分散に関しては，3.1.1 項で示した CFS のロードバランス機能をそのまま用いる．上述したように，CFS ではロードバランスの際にメモリアドレス空間を考慮せず，特に実行コアを指定しない限りすべてのコアで実行を許すように設定されている．そのため，通常のプログラムの実行時にはシブリングスレッドが各コアに分散され，IAS による効果が期待できる．

agg_bonus や agg_limit の値，および，master，slave，ia_mm の設定は我々が実装した API により，OS の起動中に変更可能である．API の仕様については文献 10) に譲る．agg_bonus と agg_limit の値は，実行するプログラムの ID を用いて指定することができ，本論文ではこの効果について，5 章で検証する．一方，各コアの master/slave への割当て，および，ia_mm の数は，任意に設定可能であり，より多くのコアや複雑なキャッシュ構造を持つプロセッサに対しては，ia_mm の数を増やすことで従来の空間分割方式の効果が期待できる[10]．本研究ではデュアルコア上での実行を想定して，master にコア 0，slave にコア 1 を設定し，単一の ia_mm を介した場合のみ考慮する．

4. SysBench の memory プログラムを用いた評価

本章では予備評価として SysBench[11] の memory プログラムを用いた評価について示す．memory プログラムは IAS の効果を示すベストケースの 1 つであり，IAS の効果の上限や理想的なスケジューリング手法との比較を示すのに適していると考える．4.1 節で memory プログラムと評価方法について説明し，4.2 節で結果と考察を述べる．

4.1 memory プログラムと評価方法

memory プログラムでは生成されたスレッドが，指定されたデータブロックに対して繰り返しアクセスを行い，各スレッドのデータブロックへのアクセス回数の合計が指定した値に達するまでの実行時間などを測定する．単一の memory プログラムが生成するスレッドはすべてシブリングスレッドである．memory プログラムでは各スレッドがアクセスするデータブロックを共有する global モードと，アクセスするデータブロックを固有に持つ local モードを選択できる．本研究ではスレッド間の参照の局所性に注目するため，global モードを選択した．アクセスパターンは，共有するデータブロックに格納された値をスレッドごとの単一ローカル変数に逐次的に読み込んでいく場合と（read），スレッドごとの単一ローカル変数の値を共有するデータブロックに逐次的に書き込んでいく場合（write）が指定できる．write の場合，データブロックに対してはセマフォやロックを用いたスレッド間のアクセス制御は行わない．read，write ともに逐次的なアクセスを行うため，ハードウェアによるプリフェッチと相性が良い．特に read に関してはプリフェッチしたデータをスレッドごとに存在する単一の変数に格納していくため，ライトバックを行う頻度も少ない．そのため，シブリングスレッドを集約しなくてもキャッシュのヒット率を高く維持できると考える．一方で，write を用いた場合，プリフェッチしてきたデータブロックに対して書き込みを行う必要があることから，キャッシュに格納されていたデータブロックがキャッシュからパージされるたびにライトバックが発生する．write を用いた場合には，IAS による集約を行うことで，キャッシュ上にすでに格納されているデータブロックに対する書き込みを集約することができ，ライトバックの頻度を下げることができると考える．以上の理由により，本論文ではライトバック時におけるキャッシュの活用効果に着目し，write を選択する．さらに，デフォルトの memory プログラムでは 1 度スレッドが CPU を与えられると，タイムスライスが切れるまで共有データへのアクセスを繰り返すが，本研究ではアクセスがいったん終了するたびに CPU をイールドし再度スケジューリングを行わせた．これにより，ループ実行による局所性の向上が実行時間に与える影響を排除し，スレッド間のデータ共有によるメモリ階層活用の効果を示すベストケースの 1 つと考えることができる．

本研究では memory プログラムを 10 個同時に実行し，すべて終了するまでの実行時間と発生するイベント数について，CFS，IAS を比較する形で評価を行った．さらに実行時間と発生イベント数の観点から理想的なスケジューリング（以下，OPT）における結果との

表 1　本研究における評価環境
Table 1　Specification of our experimental platform.

Processor	Intel Core 2 Duo
L2 Cache Size/Latency	2 MB/14 ns
Memory Size/Latency	1 GB/149 ns
OS/kernel	Fedora 7/Linux 2.6.24

比較を行った．1 章で述べたように，IAS は最適なスケジューリングを保証するものではないが，その実装が効率的であることを示すために最適なスケジューリングとの比較を行う．OPT に関して，各 `memory` プログラムはマルチスレッド化されており，かつ，単純なメモリアクセスを行うだけなので，単独で実行する際にも CPU 利用率が 100%となる．この場合，実行時間を考慮した OPT は，各プログラムを同時に動かすのではなく 1 つずつ実行することで，キャッシュの競合を避ける方式であると考える．そこで，OPT では各プログラムを単独で連続に 10 回実行する際の実行時間とイベント発生数とする．

評価環境を**表 1** に示す．本研究ではスレッドがアクセスする共有データのサイズ（以下，AS）をパラメータとし，各 `memory` プログラムに同じパラメータを指定した．イベントに関しては MEM_LOAD_RETIRED.L2_LINE_MISS（以下，L2 キャッシュミス）と RESOURCE_STALLS.ANY（以下，ストール）[21] を計測した．L2 キャッシュは本評価環境において各コアが共有する最大のキャッシュである．また，L2 キャッシュとメモリとのアクセスレイテンシの差はメモリとストレージ間を除くメモリ階層の中で最も大きいため，実行時の性能に大きく影響を与える[4),13),20)]．一方でストール数は L2 キャッシュだけでなく L1 キャッシュや TLB などの複合的な影響から発生したストール数を計測している．各プログラムが生成するスレッド数は 100，AS は 16 KB から 16 MB までとした．各メモリアドレス空間に設定する *agg_bonus* は，これまでの計測から IAS での効果がそれ以上増やしても増加しない 100 millions とし，*agg_limit* はスレッド数に合わせて 100 とした．シブリングスレッドが各コアに均等に分散すると想定すると，各コアでは各プログラムから 50 スレッドずつランキューにキューイングされる．この場合，あるスレッドが集約されないでランキュー上で待たされている間に，集約されるスレッドには 2 回集約候補となる機会が与えられることになる．

4.2　`memory` プログラム実行時の実測結果と考察

本節では `memory` 実行時の実測結果と考察を述べる．**図 5** に IAS と OPT における実行時間を，CFS での実行時間を 1 としたときの比で示す．

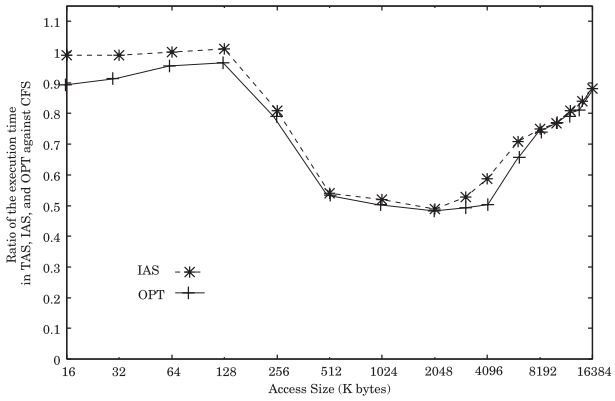

図 5　CFS に対する IAS と OPT における各実行時間の比較
Fig. 5　The ratio of the execution time in IAS and OPT against CFS.

IAS における効果についてであるが，AS によって実行時間削減効果が変化することから，集約による効果が共有データサイズによることが分かる．さらに，$AS \leq 128\,KB$ では実行時間削減効果がほとんどない．これは $AS \leq 128\,KB$ の場合，実行するプログラムの全データサイズが 1.28 MB 以下となり，並行に実行するカーネルやプリフェッチなどの影響を考慮しても，2 MB の L2 キャッシュにすべて格納することができるためであると考える．さらに効果が最大となるのは $AS = 2\,MB$ のときであり，実行時間を 2 分の 1 程度に削減することができる．IAS ではコア間でシブリングスレッドの集約実行を行うことから，全キャッシュサイズ分のデータを共有キャッシュ上に集約可能である．そのため，$AS = 2\,MB$ 付近で IAS の効果が最大になったと考える．さらに，IAS による効果が $AS \geq 2\,MB$ でも持続する理由としては，各コアでアクセスするデータ範囲が共通している場合にキャッシュ上にデータを集約でき，実行時間が削減されるためであると考える．最後に IAS と OPT を比較した場合に，IAS が OPT に近い効果を達成していることが分かる．OPT に対する IAS での増加した実行時間の割合は，すべての AS で 4%以内であった．これより IAS が 2 つのコア上での時分割集約を効率的に行っているといえる．

図 6 に，`write` 実行時において各集約スケジューラが L2 キャッシュミス数とストール数に与える効果を，CFS での発生数を 1 としたときの比で示す．図 6 から，実行時間と同じく AS によってその効果が異なり，特にストール数と実行時間の削減効果の相関が強いことが分かる．さらに，$AS \geq 2\,MB$ の場合は L2 キャッシュミス数がストール数との相関が強

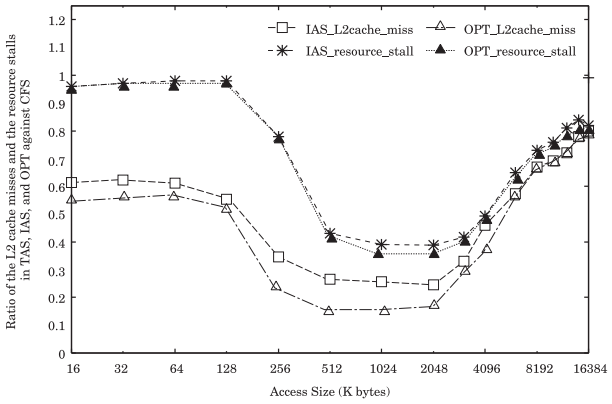

図 6　CFS に対する IAS と OPT における L2 キャッシュミスとストール数の比較
Fig. 6　Ratio of the L2 cache misses and the resource stalls in IAS and OPT against CFS.

く，L2 キャッシュミスがストール数および実行時間に大きく影響したといえる．このように共有 L2 キャッシュミスを活用するうえで IAS は有用であるといえる．また，図 6 においても，IAS と OPT の L2 キャッシュミス数とストール数がほとんど同じであることから，IAS の有用性を示している．

以上をふまえて，汎用的なマルチスレッドプログラムの実行時においても適用できる結果として以下の 2 点がある．

- 各プログラムの総データサイズがキャッシュサイズ以下の場合，IAS の効果はほとんどない．
- IAS ではスレッドがアクセスするデータサイズがキャッシュサイズを超えても，実行時間削減効果が期待できる．

1 点目のような場合は，IAS による集約はスレッド間の CPU を与えられる回数や消費する CPU 時間の公平性を悪化させる可能性があるため，集約回数や優先度ボーナスを下げるといったチューニングが必要であると考える．2 点目に関しては多くのマルチスレッドプログラムにおける IAS の有効性が期待できることを示している．一般にスレッド間でアクセスするデータをすべて共有することは少ないが，スレッド間で参照の局所性が存在すれば，スレッドが扱うデータが大きくても IAS による効果が期待できる．本研究ではこの効果を検証するために，次章で DaCapo ベンチマークを用いた評価を行う．

表 2　DaCapo のマルチスレッド化された各プログラムの特徴
Table 2　Characteristics of each multi-threaded workload of DaCapo.

Benchmark	スレッド数	%hot	Heap Volume		
			Alloc (MB)	Live (MB)	Alloc/Live
eclipse	19	0.4	5,582	30	186.0
hsqldb	20	8.6	142	72	2.0
lusearch	32	16.2	1,780	10.9	162.8
xalan	20	11.1	60,235	25.5	2,364

5. DaCapo ベンチマークを用いた評価

本章では DaCapo ベンチマークを用いて本論文で拡張した IAS の集約制御機構の検証を行う．具体的には，集約制限数の設定によるスループットの向上効果とプログラムごとの集約是非の設定効果について述べる．集約の強度の評価については，その自律的な最適化を行う機構の実装も含め，今後の課題とする．5.1 節に DaCapo ベンチマークの中から，IAS の効果の検証に用いるプログラムについて説明する．5.2 節では評価手法を示し，5.3 節に結果を示す．

5.1 DaCapo ベンチマークについて

DaCapo は 11 のプログラムからなるベンチマークスイーツであり，SPECjvm よりも現実的かつ汎用性のあるベンチマークを目指して Java で実装されている[12]．本研究では DaCapo の中でスレッドを用いて内部実行の並列化を行っている eclipse，hdsqld，lusearch，xalan の 4 つのプログラムを用いて評価を行う．各プログラムはそれぞれ統合開発環境，Java で実装された SQL データベースエンジン，テキスト検索ツール，XML ドキュメントの整形ツールをシミュレートしている．DaCapo の各ベンチマークの特徴に関して，文献 22) では消費するメモリの量や振舞いの多様性といった観点から，Jikes RVM を用いて分析している．ただし，スレッド間の共有データサイズなど IAS の効果に直接関連性がある項目については分析の対象外となっている．ここでは文献 22) から，上記の 4 プログラムの特徴について IAS との関連がありうる項目について説明し，その要約を**表 2** に示す．まずスレッド数に関してはすべてのプログラムで実行中に多少上下するが，およそ表 2 に示す結果となった．さらにコンパイラによって実行頻度が多いと認識されているメソッドの割合を%hot 列として示す．%hot 列より eclipse では実行するメソッドが多岐にわたる一方で lusearch では同じメソッドを実行する割合が高い．よって%hot からは lusearch，xalan，hsqldb，eclipse の順でコア間時間集約実行による効果が期待できる．次に実行中に割り当てられる最大ヒープ

サイズと最大 Live オブジェクトサイズ，さらにその比をそれぞれ “Alloc”，“Live”，“Alloc / Live” として示す．“Alloc/Live” 値が小さいほど同一のオブジェクトへの参照を繰り返すと考えられるため，IAS の効果も期待できる．表より hsqldb，lusearch，eclipse，xalan の順で IAS の効果が期待できる．以上より lusearch や hsqldb での効果が大きいと期待できる．一方で xalan や eclipse では集約により，キャッシュへの負荷が大きくなり，同時に実行するプログラムの性質によっては，システム全体の性能の低下が考えられる．

5.2 評 価 方 法

DaCapo を用いた評価においては，表 1 に示した環境の上で Sun の Java Version 6 を用いて実測を行った．評価の際には Linux の CFS と IAS での各項目の結果を比較し，OPT に関しては考慮しない．これは DaCapo の各プログラムはその挙動が大きく異なり，単独で順に実行することが必ずしもスループットや実行時間を最適化することにならないためである．以下に集約回数の設定による効果と集約の是非の設定による効果の各評価手法について述べる．

5.2.1 集約回数の設定による効果

集約回数の設定による IAS の効果測定に際しては，5.1 節で示した 4 つのプログラムに対して同じ *agg_count* の値を設定したうえで，同時に実行した際の各プログラム，および，全体のスループット，実行時間，イベント発生数を計測する．スループットは各スレッドが消費した CPU 時間の和の逆数，実行時間は各プログラムの終了までの経過時間により評価を行い，CPU 時間と経過時間に関しては GNU の `time` コマンドで計測した．イベント発生数の計測には Oprofile を用い，本節では 4.2 節で実行時間との相関が一番強かったストール数にのみ着目した．IAS の優先度ボーナス *agg_bonus* は 400 millions *vruntime* とした．この値は事前の計測からボーナス値の不足が生じない値に設定している．また，集約制限値 *agg_limit* は表 2 のスレッド数を考慮して 5，10，20 の 3 つの値を適用し，その効果を計測した．

5.2.2 集約是非の設定による効果

集約是非の設定による IAS の効果測定に際しては，5.1 節で示した 4 つのプログラムの静的な分析に基づいて集約の是非を決定したうえ，同時に実行した際の各プログラム，および，全体のスループットと実行時間を計測する．ここでは 5.1 節での予測に基づき，eclipse の集約の効果が低いと考え，eclipse を集約した場合と集約しない場合を比較する．計測の際に用いた *agg_bonus* の値は 400 millions *vruntime*，*agg_limit* の値は 20 とした．

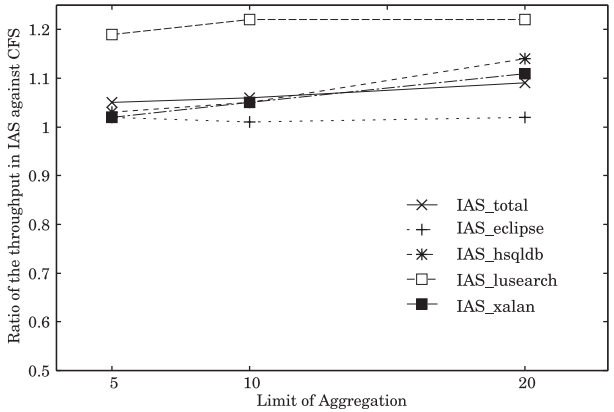

図 7　集約制限値がスループットに与える効果
Fig. 7　The effect of limiting the aggregation countson the throughput.

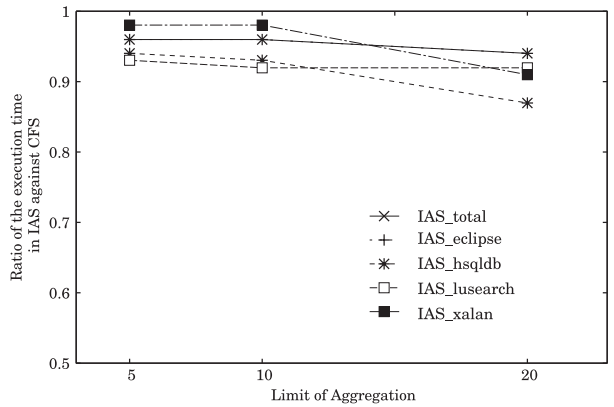

図 8　集約制限値が実行時間に与える効果
Fig. 8　The effect of limiting the aggregation counts on the turn around time.

5.3 結　　果

5.3.1 集約制限数の設定による効果

IAS がスループットに与える影響を**図 7**，実行時間に与える影響を**図 8** に示す．それぞれの図において横軸に集約制限数，縦軸に CFS とそれぞれのスケジューラにおけるスルー

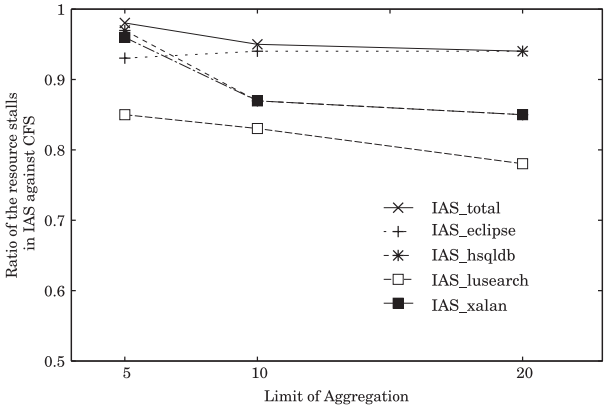

図 9　集約制限値がストール数に与える効果
Fig. 9　The effect of limiting the aggregation counts on the resource stalls.

プットと実行時間の比を，各プログラムと実行時全体（total）別に示している．

図 7，8 から分かるようにプログラムによって程度の差はあるものの，すべてのプログラムでスループットが向上し実行時間が削減されている．効果が最も大きかったのは lusearch でスループットが 20%以上増加した．これは表 2 に示したように lusearch での%hot の割合が大きかったためと考える．逆に効果が一番小さかったのは eclipse であり，これも表 2 に示したように，%hot な割合が少なく，かつ，Alloc / Hot の割合が高かったことから，スレッド間の参照の局所性が低かったと考えられる．さらにスループットと実行時間に大きく影響していると考えられるストール数に関して，それぞれのスケジューラでの発生数の CFS との比を，各プログラムと実行時全体別に**図 9** に示す．図 9 から分かるように，すべてのプログラムでストール数が削減されており，特に lusearch において削減数が 20%以上削減されている点で，スループットでの効果と関連していることが確認できた．

またほぼすべてのプログラムで，集約制限数の増加にともないスループットの向上と実行時間の削減効果が大きくなった．これは集約制限数を上げることで，集約回数が増加し，その分集約スケジューリングの効果が大きくなったためと考える．IAS の効果を制御する方式としてはほかに優先度ボーナスの値を変更が考えられるが，3.1.1 項で述べたように *vruntime* の値はナの秒単位で計測されたクォンタム時間から計算された値であるため，粒度が細かすぎて適切な値の設定が難しい．本項で示したように，集約の効果を集約制限数で

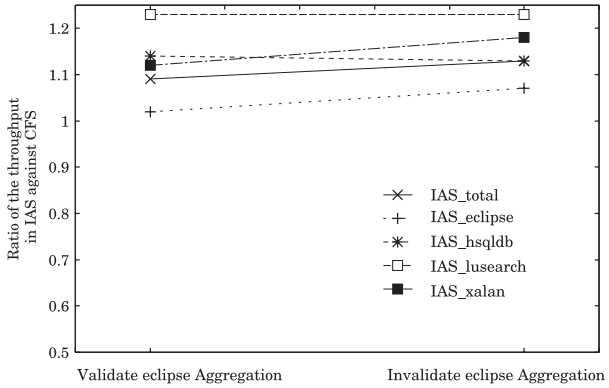

図 10　集約是非がスループットに与える効果
Fig. 10　The effect of validating and invalidating the aggregation on the throughput.

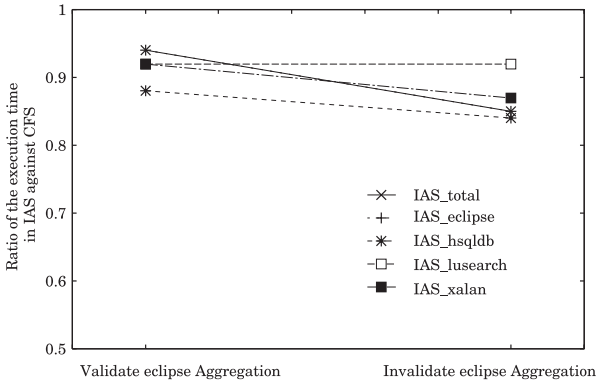

図 11　集約是非が実行時間に与える効果
Fig. 11　The effect of validate and invalidate the aggregation on the turn around time.

設定することにより，より直感的に IAS の効果を制御できると考える．

5.3.2　集約是非の設定による効果

DaCapo の eclipse プログラムを集約する場合としない場合での各プログラム，および，全体のスループットを**図 10**，実行時間を**図 11** に示す．図 10 では eclipse プログラムのスレッドを集約した場合と集約しない場合の比較を，CFS での値との割合という形で示す．

図 10 で示すように，lusearch や xalan といった 5.3.1 項で効果が大きいと述べたプログラムに対する影響はほとんどなかった．一方で，eclipse や hsqldb の実行スループットは上がる傾向がある．この原因として，eclipse では各スレッドが参照するワーキングセットが大きく，シブリングスレッドを集約してもキャッシュ活用の効果などが現れていなかったためであると考える．むしろ各スレッドのワーキングセットが大きいために，集約によりメモリを圧迫し，実行時間が大きくなってしまったことが考えられる．また，eclipse ではスレッドが集約されずに後回しにされるため，1 度 CPU を与えられた際のクォンタムの長さが長くなると考えられ，その分スループットが向上すると考える．図 11 においても同様に eclipse を集約したかどうかでの比較を，CFS での値との割合で示す．図 11 で示すように，eclipse や xalan などのプログラムを中心に実行時間が削減され，全体での実行時間も削減できている．

このように eclipse のように，静的な分析により集約の効果が得られにくいと考えられるプログラムに関しては，初めから集約の対象から外すことで，システム全体の性能を向上させることができると考える．今回は静的な分析に Jikes RVM を用いたが，2 章で紹介したようにスレッド実行にともなう様々なツールの開発が進んでいる．Jikes RVM などのツールを用いることにより，IAS の集約機構を活用することができると考える．

6. おわりに

本論文では CMP 向けに我々が提案した，IAS に対して集約制御機構を実装したうえでその評価について述べた．集約制御機構を用いることで，集約の是非や集約回数の制限，集約の強度をメモリアドレス空間ごとに設定できる．本論文では汎用的かつ現実的なワークロードの実装である DaCapo を用いた実測のなかで，集約回数の設定によりスループットへの効果を制御可能であることを示した．さらにプログラムの性質の静的な分析に基づいて集約の是非を設定することで，システム全体のスループットをさらに向上させることができることを示した．

今後の課題としては IAS 用の優先度ボーナスや集約回数の制限値の自動的なチューニングを行うシステムの構築がある．具体的には実行時にストール数などのイベントを計測し，集約による効果を動的に分析したうえでチューニングを行うヘルパスレッドの実装を行う．また，本研究では共有 L2 キャッシュを持つデュアルコアプロセッサ上での効果にのみ着目したが，コア数の増加にともないメモリ階層の構造も複雑化してきている．そこで複数の `master`，`slave` を用いた場合のマルチコアプロセッサ上での空間分割効果の検証なども今後行っていく．

参考文献

1) Yamada, S., et al.: Effect of Context Aware Scheduler on TLB, *Proc. Workshop on Multi-Threaded Architectures and Applications* (2008).
2) Chandra, D., et al.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, *Proc. HPCA*, pp.340–351 (2005).
3) Kim, S., et al.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, *Proc. PACT*, pp.111–122 (2004).
4) Fedorova, A., et al.: Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design, *Proc. USENIX 2005 Annual Technical Conf*, pp.395–398 (2005).
5) Parekh, S., et al.: Thread-Sensitive Scheduling for SMT Processors, Technical Report, Univ. Washington (2000).
6) Jiang, Y., et al.: Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors, *Proc. PACT*, pp.220–229 (2008).
7) Snavely, A., et al.: Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor, *Proc. International Conference on Measurement and Modeling of Computer Systems*, pp.66–76 (2002).
8) Yamada, S., et al.: Development of a Thread Scheduler for Global Aggregation of Sibling Threads, *Research Reports on Information Science and Electrical Engineering of Kyushu University*, Vol.1, No.2, pp.69–74 (2008).
9) Yamada, S., et al.: Impact of Inter-Core Time Aggregation Scheduler on a Database Server Workload, *Asian Journal of Information Technology* Vol.8, No.4, pp.94–103 (2009).
10) Yamada, S., et al.: Proposal and Evaluation of APIs for utilizing Inter-Core Time Aggregation Scheduler, *Proc. Workshop on Job Scheduling Strategies for Parallel Processing* (2010).
11) SysBench: A system performance benchmark. http://sysbench.sourceforge.net/
12) DaCapo benchmark suite, http://dacapobench.org/
13) 小川周吾，ほか：プロセスの実行時情報を用いたスケジューラによる高速化手法，情報処理学会論文誌コンピューティングシステム，Vol.46, No.SIG 12 (ACS 11), pp.161–169 (2005).
14) 近藤正章，ほか：トラクションコントロール実行：CMP 向けプロセス実行制御方式の提案，情報処理学会論文誌コンピューティングシステム，Vol.1, No.2 (ACS 12), pp.111–123 (2008).
15) DeVuyst, M., et al.: Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors, *Proc. International Parallel and Dis-*

tributed Processing Symposium (2006).
16) 福田　晃：並列オペレーティングシステム，コロナ社 (1997).
17) Ziemba, S., et al.: Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters, *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture* (2008).
18) Anderson, J., et al.: Parallel Task Scheduling on Multicore Platforms, *ACM SIGBED Review*, Vol.3, Issue 1, The work-in-progress (WIP) session of the RTSS 2005, pp.1–6 (2006).
19) Anderson, Z., et al.: SharC: Checking Data Sharing Strategies for Multithreaded C, *Proc. 2008 ACM SIGPLAN*, pp.149–158 (2008).
20) Chen, S., et al.: Scheduling Threads for Constructive Cache Sharing on CMPs, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pp.105–115 (2007).
21) Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2. http://www.intel.com/products/processor/manuals/index.htm (2010 年 1 月 24 日最終確認)
22) Blackburn, et al.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proc. 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages and Applications* (2006).

(平成 22 年 1 月 26 日受付)
(平成 22 年 6 月 12 日採録)

山田　賢（正会員）

平成 16 年九州大学工学部電気情報工学科卒業．平成 18 年九州大学大学院システム情報科学府情報工学専攻修士課程修了．平成 21 年同博士後期課程修了．計算機アーキテクチャ，並列処理，クラウドコンピューティング等に興味を持つ．ACM 会員．博士（工学）．

日下部　茂（正会員）

1989 年九州大学工学部情報工学科卒業．1991 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．同専攻助手を経て，現在，九州大学大学院システム情報科学研究院情報工学専攻准教授．関数型言語，マルチスレッド処理，オペレーティングシステム，クラウドコンピューティング，ソフトウェア工学と形式手法，応用行動分析等に興味を持つ．ACM，IEEE-CS，電子情報通信学会，ソフトウェア技術者協会各会員．博士（工学）．