社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－OS－104 (6)
2007／1／30

アドレス空間を共有するスレッドの空間的な集約を行うスケジューラの評価

山田 賢[†] 日下部 茂[††]

Linux の clone() 命令で生成されるスレッドは，一般的に生成元の親プロセスとアドレス空間を共有するという特徴を持つ．我々はこのスレッドの特徴に着目し，Linux カーネル 2.5 で導入された $O(1)$ スケジューラをベースにアドレス空間を共有するスレッドの時間的な集約を行うスケジューリングを実現し，これによるコンテキスト切り替えのオーバーヘッド削減やメモリ階層の活用の効果を確認してきた．一方，CMP や SMT 上では，このアドレス空間を共有するスレッドが同一プロセッサ上で実行されるように空間的集約を行うと，時間的な集約効果の強化が期待出来る．そこで本研究ではこれらのスレッドに対し，時間的な集約に加えて空間的集約を行い，時間的な集約方式と比べた際の評価を行った．

Evaluation on scheduler of context aware aggregation to processor

Satoshi YAMADA[†] and Shigeru KUSAKABE[††]

Threads in Linux generally share the same memory address space with their parent and peer threads. We modified $O(1)$ scheduler of Linux to aggregate threads which share the same memory address space. Thus, we can enhance the locality of reference and reduce context switch overhead. In addition, we can expect more effective use of this scheduler on CMP or SMT platforms by aggregating these threads to the same processor. Therefore, we modified our scheduler to aggregate these threads to the same processor and evaluated the effect.

1. はじめに

Linux において clone() 命令で生成されるスレッドは，2.1 節で述べるように，一般的に生成元の親プロセスとアドレス空間を共有するという特徴を持つ．つまり，同一の親プロセスを持つスレッド同士はアドレス空間を共有する．我々はこの特徴に着目し，マルチスレッド実行時にスループットやレスポンスを高めるようなスケジューリングの研究を行っている．プロセススケジューリングのオーバーヘッド低減のため，Linux ではカーネル 2.5 から定数オーダーのスケジューリングを実現する $O(1)$[1] スケジューラが導入されている．しかしながら，$O(1)$ スケジューラではプロセス間のアドレス空間共有を考慮したスケジューリングは行われていない．一方，プロセッサとメモリ速度の乖離は大きく，メモリ階層の有効な活用が性能向上に重要である．また，コンテキスト切り替えに伴うオーバーヘッドもシステムのスループットやレスポンスの向上を妨げる要因であると考えられる．そのため，我々はアドレス空間を共有するスレッドの集約が有効であると考え，Linux カーネル 2.5 において導入された $O(1)$ スケジューラをベースに，そのような集約を時間的に行うスケジューリングを定数オーダーで実現し，評価してきた[2]．一方，近年普及してきた CMP や SMT では，複数のプロセッサ上にスレッドが分散する．そのためアドレス空間を共有するスレッド同士が別々のプロセッサに分散してしまうと，時間的な集約効果の低下が考えられる．また，アドレス空間を共有するスレッドが同一プロセッサ上に存在しない場合は，スケジューリングのオーバーヘッドから $O(1)$ スケジューラよりも性能が低下してしまうことも考えられる．一方，同じアドレス空間を共有したスレッドを同一のプロセッサに集約することで，時間集約スケジューリングの効果の更なる向上が期待できる．そこで本研究ではまずスループットの向上に着目し，これらのスレッドに対し空間的集約を行うスケジューラを実装し評価を行った．

以降，2 章は準備として本研究が対象とする Linux スレッドと課題となる CMP や SMT 上でのオーバーヘッド，Linux の $O(1)$ スケジューラの問題点につい

† 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
†† 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

て述べる．3章では我々が実装評価を行ってきた時間集約スケジューリングと，本研究で評価を行う空間集約スケジューリングについて説明する．4章ではこのスケジューラの評価を行い，提案手法の効果と問題点について考察する．5章で関連研究について述べ，6章でまとめる．

2. 準　　備

2.1 Linux スレッド

Linux ではカーネルスレッドとユーザスレッドが1対1に対応するスレッドモデルを採用しており，スレッドの資源管理をカーネルで行っている．Linux におけるスレッドは，呼び出し元のスレッドの複製を作成する，Linux 固有の clone() システムコールによって実現されている．プロセスを生成する fork() システムコールでは，アドレス空間，ファイルディスクリプタ，シグナルハンドラテーブルといったプロセス資源が親プロセスから複製される．これに対し，clone() システムコールでは呼び出し時にこれらの資源を親プロセスと共有するかどうか選択することが出来るようになっている．Linux のスレッドは，一般的に親プロセスと実行コンテキストの一部を共有した特殊な子プロセスとして生成され，スケジューラはプロセスと同様にスレッドを管理する．

本研究は，clone() システムコールによって生成され，アドレス空間を共有したスレッドを集約の対象とする．アドレス空間が異なるスレッドを集約を行わずにインターリーブする場合と比較して，アドレス空間を共有するスレッド間の連続実行では，同一アドレス空間内で共有しているデータが，プロセッサのハードウェアキャッシュ上に存在する可能性が高い．このため，ハードウェアキャッシュの利用効率の向上が期待できる．また，アドレス空間を共有するスレッド間では，アドレス空間の管理を行うメモリディスクリプタが同一であるため，コンテキスト切り替え時に，このメモリディスクリプタを切り替える処理が省略でき，コンテキスト切り替えが高速となる．

2.2 オーバーヘッド

本研究が問題の対象とする CMP や SMT におけるメモリ階層とコンテキスト切り替えの具体的なコストについて，表1に示す．ここで 2p/0K, 2p/16K はそれぞれ 0K バイト，16K バイトの2つのプロセスが切り替わる際のコスト，L1，L2，Mem はそれぞれ L1 キャッシュ，L2 キャッシュ，メインメモリのアクセスレイテンシを示している．使用した OS は Linux 2.6.17 で，計測には LMBench[10] を用いた．この表より，コンテキスト切り替えのオーバーヘッドとメインメモリのアクセスレイテンシが大きく，スループット向上を妨げていると考えられる．この傾向は CMP や SMT が登場する以前のシングルコアのプロセッサからみられる特徴であり，CMP や SMT でも依然として同じ傾向が見られる．

また，CMP や SMT ではプロセッサ間の通信のオーバーヘッドが大きく影響する．そのため，プロセス間通信を行うプロセス同士が他のプロセッサ上に分散してしまうと，プロセス間通信のオーバーヘッドが大きくなることが考えられる．

表1　汎用プロセッサにおけるオーバーヘッド (clock cycles)

CPU	2p/0K	2p/16k	L1	L2	Mem
Core Duo	1095	3071	3	14	157
Core 2 Duo	3298	5798	3	14	149
Athlon 64 ×2	2300	3280	2	12	115

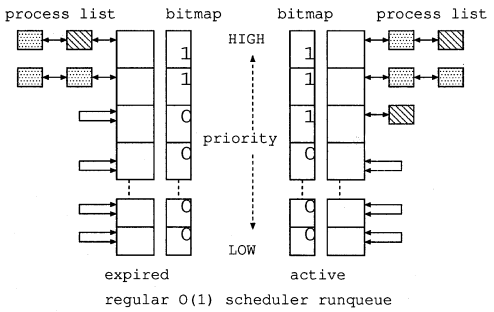

図1　$O(1)$ スケジューラの概念図

2.3 $O(1)$ スケジューラと時間集約スケジューラの問題点

Linux2.6 ではプロセススケジューラとして $O(1)$ スケジューラ[1] を採用しており，図1はその概念図である．$O(1)$ スケジューラでは，プロセッサごとに runqueue を持ち，runqueue はプロセスの優先度でソートされたプロセスリストの配列とビットマップを中心に構成されている．プロセッサごとに"active"と"expired"という2つのキューを用い，実行中のプロセスを active 側のキューに，クォンタムを使い切ったプロセスを expired 側のキューにキューイングしている．active 側が空になったら active と expired の役割を交替させる．ビットマップは対応するエントリのリストにプロセスが存在するかどうかのフラグとなっており，このビットマップを介することで次に実行すべきプロセスを定数オーダーで発見できる．

$O(1)$ スケジューラでは，スレッド起床時にスレッドをあらかじめ優先度ごとのキューへとキューイングする．この時，直前にプロセッサを割り当てられたスレッドを考慮した優先度の設定などは行わない．そのため，例えばアドレス空間を共有するスレッドを優先するといったことはなされておらず，アドレス空間の共有を利用したメモリ階層の有効利用や，コンテキスト切り替えのオーバーヘッド削減の機会が失われている．

3. スレッド集約スケジューラ

メモリ階層の有効利用とコンテキスト切り替えコスト削減を同時に実現するために，我々はアドレス空間を共有するプロセスの定数オーダーによる時間的な集約を実現してきた．本研究ではこれに加えて，プロセッサに関する集約を行うことで，より効果的な時間集約を目指す．この2つのスケジューラについて本章で説明する．

3.1 時間的な集約

本節ではアドレス空間を共有するスレッドの時間的な集約の概念について，図2を用いて説明する．図2ではA，B，C，D，Eがそれぞれスレッドを表している．また，スレッドA，C，Dが同じアドレス空間を共有し，スレッドB，Eは別のアドレス空間を共有するスレッドを示している．$O(1)$ スケジューラではA，B，C，D，Eの順で実行され，アドレス空間が異なるコンテキスト切り替えは3回実行される．一方アドレス空間を共有するスレッドを集約して実行するとA，C，D，B，Eの順で実行され，アドレス空間が異なるコンテキスト切り替えは1回に減る．

このような実行を行う機構について，図3を用いて説明する．図3は通常の $O(1)$ スケジューラのランキュー (図中左) と我々の提案するスケジューラのランキュー (図中右) を示しており，図2の例と同様，A，B，C，D，Eはそれぞれランキューにつながれたスレッドを示している．通常の $O(1)$ スケジューラでは優先度の高い順にスレッドが選択されて実行されるため，図3の例ではA，B，C，D，Eの順で実行され，アドレス空間が異なるコンテキスト切り替えは3回実行される．一方，我々の提案するスケジューラではアドレス空間ごとにランキューを用意する．そしてアドレス空間を共有するスレッドに対しては優先度を1上げるようなスケジュールを行う．この機構によりA，C，D，B，Eの順で実行され，アドレス空間が異なるコンテキスト切り替えは1回に削減される．このように時間的な集約スケジューリングを行うことで，コンテキスト切り替えのオーバーヘッド削減や，参照の局所性向上といった効果が得られると考える．一方集約されないスレッドが待たされるなどの公平性の問題が生じる．

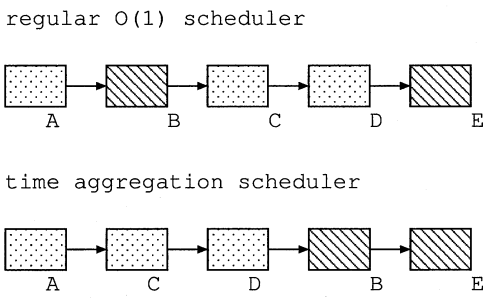

図2 時間集約の概念

3.2 空間的な集約

本節ではアドレス空間を共有するスレッドの空間的な集約について図4を用いて説明する．前節と同様，A，B，C，D，Eはスレッドを示しており，スレッドA，C，Dが同じアドレス空間を共有し，スレッドB，Eは別のアドレス空間を共有するスレッドを示している．また，CPU0でA，B，C，が実行されCPU1でD，Eが実行されるとする．この場合それぞれのCPUでアドレス空間の異なるコンテキスト切り替えが合計3回行われる．ここで同じアドレス空間を共有するスレッドを一つのプロセッサに集約すると，アドレス空間の異なるコンテキスト切り替えの回数は0回に出来る．

このような実行を行う機構について，図5を用いて説明する．まず空間的な集約に際し，本研究ではスレッド生成時のCPUマスクの設定に着目した．CPUマスクとはスレッドの実行を許すCPUを指定する設定であり，タスク構造体のcpu_allowedメンバに記述される．通常のスレッド生成処理においては，copy_process()関数中でスレッド生成元の親プロセスのCPUマスクの設定がそのままコピーされる．これに対し本研究では，CPUマスクをスレッドの生成時に決定した1つのCPUに設定することで空間的な集約を実現する．準備としてタスクのメモリ構造体に集約用のCPUマスクを設定するメンバを追加する．CPUマスクの設定は以下の手順で行われる．まず，生成されるスレッドのメモリ構造体に空間的な集約のためのCPUマスクが設定されていないかを，新たに追加したメンバを用いてチェックする．親プロセスが最初にスレッドを生成するはこのCPUマスクが設定されていない．設

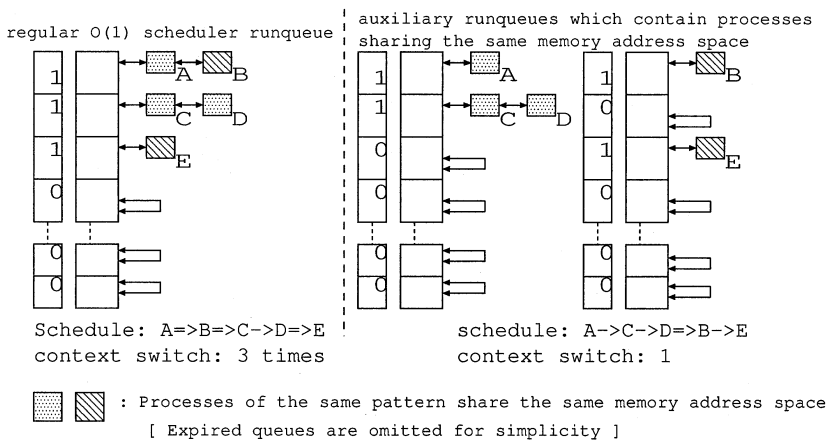

図 3　時間的なスレッド集約機構の概念図

定がされていない場合は実行する CPU を決定し，その情報をメモリ構造体に新たに追加したメンバに書き込む．そしてこの情報を元に，スレッドに CPU マスクの設定を行う．親プロセスが次にスレッドを生成する際には，すでにメモリ構造体の新たに追加したメンバに CPU マスクが設定されているので，これに従い cpu_allowed の設定を行う．このあと，3.1 節で示した時間集約スケジューリングを行うことで，コンテキスト切り替えのオーバーヘッド削減やメモリ階層の活用がより強化されることが期待できる．

しかし，アドレス空間の異なるスレッド同士が通信を行う場合など，アドレス空間ごとに集約してスループットが低下する場合も考えられる．本研究では，4.1 節に示すように CPU マスクの設定を行い，その空間的な集約の効果について計測した．

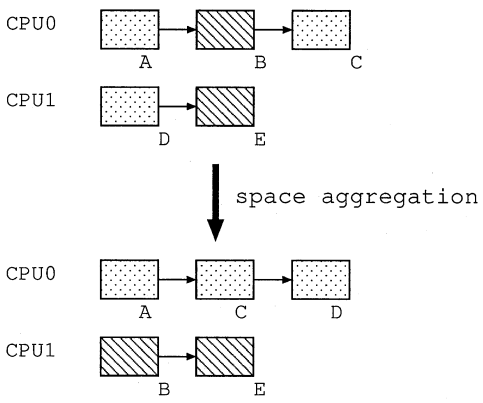

図 4　空間集約の概念

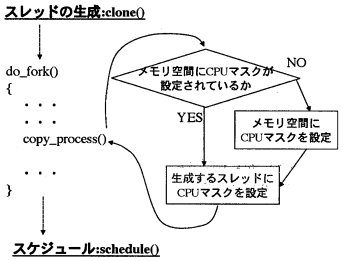

図 5　空間的なスレッド集約機構の概念図

4. 評　　価

本研究の評価には chatroom benchmark(以下 chat)[8] を用いた．chat を用いる理由としては，生成するスレッド数が管理できること，サーバとクライアントがそれぞれ別のアドレス空間を共有しており，考察が行いやすいといったことが挙げられる．

4.1 方　　法

chat の概要を図 6 に示す．chat はチャットルームをシミュレーションするベンチマークで，チャットルーム 1 つあたりデフォルトで 20 人のメンバがおり，TCP ソケット通信によって各メンバが 1 メッセージあたり 100 バイトのメッセージの送受信を行う．また，チャットルームの数はデフォルトで 10 に指定されるため，デフォルトの設定では 200 メンバがそれぞれメッセージの送受信を行うことになる．この時 1 メンバが送受信

のためにそれぞれ2つのLinuxのスレッドを生成する．そのため，クライアント側だけで $10 \times 20 \times 2 = 400$ スレッドが生成される．サーバ側でも同様にスレッドが生成され，合計で800スレッドが生成される．この時，クライアントとサーバでそれぞれ1つメモリアドレス空間をもつ．また，chatでは1スレッドの仮想記憶上のプロセスの大きさは約14MBで，実際に消費するメモリの大きさは約3.6MBである．

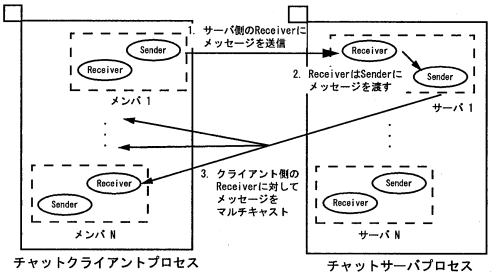

図 6 chatroom benchmarkの概要

本研究ではクライアント数=サーバ数=1の時とクライアント数=サーバ数=2の時で評価を行った．評価の際には，Linux-2.6.17のデフォルトのスケジューラ，Linux-2.6.17を修正して時間集約のみを行うスケジューラ，この時間集約に加えて空間集約を行うスケジューラでの比較を行う．クライアント数=サーバ数=1の場合，chatアプリケーションはサーバ用とクライアント用の2つのアドレス空間を用いる．この場合，空間集約の際にはそれぞれのアドレス空間ごとに別のプロセッサを割り当てるようにした．つまり，サーバ側とクライアント側で異なるプロセッサを用いるような集約を行った．この時，時間的な集約スケジューラはchatとchat以外のスレッドの優先度を比較してスケジューリングを行う．

続いて，クライアント数=サーバ数=2の場合はchatを実行する際に，サーバ用とクライアント用のアドレス空間が2つずつ存在するため，chatアプリケーションは合計4つのアドレス空間を用いる．この場合，空間集約の際には，あるクライアントとそれに対応するサーバが同じプロセッサを用いるような集約を行った．この時，時間的な集約スケジューラはchat内でも異なるアドレス空間をもつスレッドが同じプロセッサ上に存在するため，これらを含めた比較をしてスケジューリングを行う．

本研究では上の2つの場合において，部屋数やメンバ数をデフォルトの設定のまま，メッセージ数を変化させて計測を行った．

評価にはデュアルコアプロセッサであるIntel Core 2 Duoを用いた．評価環境を表2に示す．Intel Core 2 DuoではL1データキャッシュ，TLBを各コアごとに持ち，L2キャッシュを2コアが共有している．

表 2 実験環境

processor	Intel Core 2 Duo 1.86 GHz
L1 data cache size	32 KB
L2 cache size	2 MB
Memory size	1 GB

4.2 結　　果

4.1節の実験の結果を図7，8に示す．どちらも結果は時間集約のみを行うスケジューラのスループットを1とした時の，各スケジューラのスループットの相対値を示す．図中の"default"はLinux-2.6.17のデフォルトのスケジューラ，"time aggregation"が時間集約のみを行うスケジューラ，"space aggregation"が時間集約に加えて空間集約を行うスケジューラでのスループットを示す．

まずクライアント数=サーバ数=1の場合についてスループットの割合の変化を図7に示す．これより集約を行うことによりデフォルトのスケジューラよりもスループットが向上することを確認した．さらに，空間集約を行うことにより，時間集約のみのスケジューリングよりも最大で1.48倍スループットが向上することを確認した．

続いてクライアント数=サーバ数=2の場合についてスループットの変化を図8に示す．この場合も集約を行うことによりデフォルトのスケジューラよりもスループットが向上することを確認した．また，空間集約を行うことにより，メッセージ数が少ない時は時間集約のみのスケジューリングよりもスループットが向上することを確認したが，メッセージ数が2000を超えるあたりで空間集約のスループットが時間集約のみの場合よりも下回っている．

4.3 考　　察

クライアント数=サーバ数=1の時，空間集約によるスループットは時間集約のみの場合よりも最大で1.48倍高くなった．一方クライアント数=サーバ数=2の時，メッセージ数が2000を超えると空間集約による効果が見られなくなり，逆に悪化する結果となった．

この原因を考察するため，まずコンテキスト切り替えのオーバーヘッドに着目する．表3はメッセージ数2000の時のchatの所要クロックサイクル数を示している．同様にメッセージ数2000のときのカーネル内部で実行されたschedule()関数の実行時間を表4

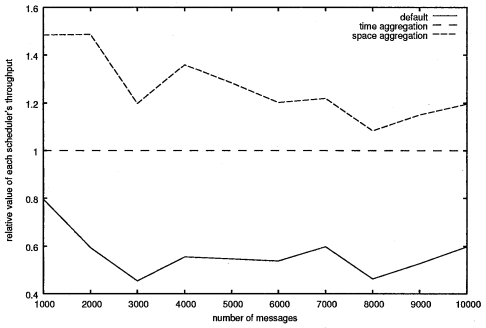

図 7　クライアント数=サーバ数=1 の場合のスループットの割合

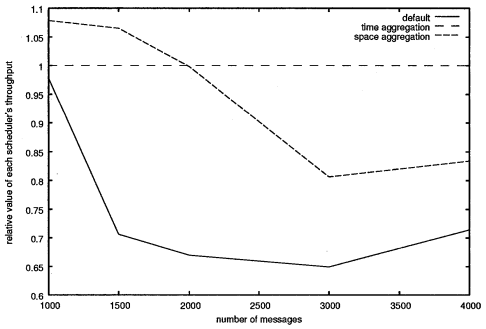

図 8　クライアント数=サーバ数=2 の場合のスループットの割合

に示す．schedule() 関数は次に実行するスレッドを選択しコンテキスト切り替えを行う．つまり schedule() 関数に要する実行時間はコンテキスト切り替えに要するオーバーヘッドを含む．計測には readprofile を用いた．表 4 によると，空間集約を行うことで，クライアント数=サーバ数=1 の時 150M クロックサイクル，クライアント数=サーバ数=2 の時 370M クロックサイクルのオーバーヘッド削減がなされたことがわかる．但し，コンテキスト切り替えの削減クロック数自体は全体の所要クロック数にくらべると 1%程度であり，全体的なスループットの変化にはあまり効果がなかった言える．

表 3　全体のクロックサイクル数 (単位：million clock cycles)

クライアント:サーバ	default	時間集約のみ	空間集約
1:1	46,494	28,601	18,083
2:2	45,216	30,281	30,328

続いて計測時に発生するキャッシュミスなどのイベント数について表 5，表 6 に示す．計測には perfctr[9] を用いた．これらの表では，L1，L2，dtlb，itlb がそれぞれ L1 キャッシュミス，L2 キャッシュミス，DTLB ミス，ITLB ミスの回数を示す．計測時の部屋数などのパラメータはデフォルトの設定と同じで，コンテキスト切り替えの計測時と同様，メッセージ数は 2000 の時の値を示した．

表 4　schedule() 関数の所要クロックティック数 (単位：million clock cycles)

クライアント:サーバ	default	時間集約のみ	空間集約
1:1	2418	781.2	539.4
2:2	4166.4	762.6	390.6

クライアント数=サーバ数=1 の時，空間集約スケジューラは時間集約スケジューラに比べて L2 キャッシュミスの回数が少ないことがわかる．しかし，L1，DTLB，ITLB ミスの回数は増加していることがわかる．クライアント数=サーバ数=2 の場合は空間集約スケジューラの L2 キャッシュミス回数は他のスケジューラとほぼ同じで，L1 キャッシュミス回数が増加している．

表 5　クライアント数=サーバ数=1 のときの発生イベント数

スケジューラ	L1 $	L2 $	dtlb	itlb
default	10083780	222113	10887647	750
時間集約のみ	10674243	239305	14977431	819
空間集約	14075166	174708	23335891	1049

表 6　クライアント数=サーバ数=2 のときの発生イベント数

スケジューラ	L1 $	L2 $	dtlb	itlb
default	9638346	196038	30561318	1555
時間集約のみ	9848060	202312	27478254	1378
空間集約	21430341	205589	31743215	1358

以上の結果から，クライアント数=サーバ数=1 のとき，空間集約を行うことによって，schedule() 関数内のコンテキスト切り替えのオーバーヘッドが削減され，L2 キャッシュミス数が減少したことがわかった．L2 キャッシュミス数の削減により，表 1 で示したメインメモリへのアクセスレイテンシの節約が出来ると考える．また，[6] などで示されているように，L2 キャッシュのバンド幅に比べてメインメモリのバンド幅は半分以下である．よって L2 キャッシュをミスしてメインメモリにアクセスする際は，アクセスレイテンシ以上のオーバーヘッドがかかる．以上のような理由により全体的なスループットの向上が計測できたと考える．

一方，クライアント数=サーバ数=2 の場合，コンテキスト切り替えのオーバーヘッド削減は確認できたが，L2 キャッシュミス数の削減は確認できなかった．よって，コンテキスト切り替えの削減と L2 キャッシュ

ミス数とは関係がないと考える．また，L1キャッシュミス回数が極端に大きくなっている．これも[6]にあるように，L2キャッシュのバンド幅はL1キャッシュのバンド幅に比べて半分以下である．これらの理由に伴いスループットの向上が見られなかったと考える．

5. 関連研究

Fedorovaら[3]は，CMPのそれぞれのコアがSMTであるCMP(chip multithreading, CMT)上でのボトルネックがL2キャッシュミスである，という問題[4]に対処するべく，スレッドのL2キャッシュミス率を実行時に計測し，この情報を元に動的なスケジューリングを行っている．このようにして，L2キャッシュミス率を下げてスループットを向上させる手法の評価を行っている．この研究ではL2キャッシュサイズが可変である独自のシミュレータを用いて，L2キャッシュサイズとスループットの関係を検証している．本研究との関連としてはスループットを妨げる対象としてメモリ階層を挙げており，それにスケジューラで対処している点である．本研究はスケジューリングの対象としてアドレス空間を共有するスレッドのみを考慮したが，この研究ではCMT上で実行するすべてのスレッドに対してスケジューリングを行う．そのため，スレッド情報の計算の負荷が大きくなってしまうが，より一般的な状況で有効なスケジューリングを想定している．本研究で用いたスケジューリングもCMT上での検証を考えている．

DeVuystら[5]は，Fedorovaらと同じくCMT上で，パフォーマンス向上と電力消費削減の観点から，スレッドのコアへの割当に関する研究を行っている．DeVuystらは負荷によっては，すべてのコアにスレッドを分散させるのではなく，スレッドの親和性を考慮しつつ，限られた数のコアに集約して未使用のコアを増やすことで，パフォーマンスの向上と消費電力の削減を実現している．本研究とはスレッドの情報から特定のコアにスレッドを集約するという点で類似しているが，場合によってコアを使用しない場合を想定するという発想は本研究にはなかった．今後このような観点からも考察を行っていく予定である．

6. まとめと今後の課題

本研究ではマルチスレッド実行時において，アドレス空間を共有するスレッドを空間的に集約する効果について，従来の時間的な集約と合わせてその効果を検証した．ベンチマークアプリケーションとしてchatを用いた結果，最大で1.48倍のスループット向上を確認した．しかし，chatのメッセージ数やクライアント，サーバ数を変化させると時間集約のみを行うスケジューラよりスループットの低下がみられた．

以上をふまえて今後の課題としては他のアプリケーションでの効果の計測がある．例えばWebサーバのように異なるコンテキストのスレッドグループを複数生成するようなアプリケーションを考えている．また，今回のようにスレッドの実行特性によっては集約の効果があまり見られない場合においても有効な空間的集約の方針についても考慮する必要があると考える．さらに，集約スケジューリングを行う際にはいつも公平性の問題が生じる．これに対しては同じアドレス空間を共有するスレッドの連続実行回数をカウントし，ユーザが設定した閾値で集約数の制限を行うことを考えている．

参考文献

1) Ingo Molnar; *Ultra-scalable O(1) SMP and UP Scheduler*, http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html, (2002)
2) Takuya Kondoh and Shigeru Kusakabe: *Enhancing Throughput of Threaded Servers on Off-the-shelf Linux Platforms*, The 3rd IASTED International Conference on Communications, Internet, and Information Technology (CIIT), (2004)
3) Alexandra Fedorova and Margo Seltzer and Christopher Small and Daniel Nussbaum: *Throughput-Oriented Scheduling On Chip Multithreading Systems*, Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University, (2004)
4) N. Tuck and D. Tullsen: *Initial Observations of the Simultaneous Multithreading Pentium 4 Processor*, PACT, (2003)
5) Matthew DeVuyst and Rakesh Kumar and Dean M. Tullsen: *Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors*, International Parallel and Distributed Processing Symposium, (2006)
6) Detailed Platform Analysis in RightMark Memory Analyzer. Part 10: Intel Core Duo (Yonah) http://www.digit-life.com/articles2/cpu/rmma-yonah.html
7) Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/design/
8) chat benchmark : Linux benchmark suite homepage. http://lbs.sourceforge.net/.
9) perfctr: Linux performance-monitoring coun-

ters driver.
http://user.it.uu.se/mikpe/linux/perfctr/.
10) Lmbech - tools for performance analysis.
http:/www.bitmover.com/lmbech/.