オン・チップ・マルチスレッドアーキテクチャ向けシステムソフトウェア

佐藤 未来子 (Mikiko Sato) †　笹田 耕一 (Koichi Sasada) †　加藤 義人 (Norito Kato) †
大和 仁典 (Masanori Yamato) †　中條 拓伯 (Hironori Nakajo) †　並木 美太郎 (Mitaro Namiki) †

オンチップマルチスレッドアーキテクチャプロセッサ（以下，OCMT プロセッサ）を搭載したシステムにおいて，複数の命令流をチップ上で高効率に並列実行させるためのシステムソフトウェアを検討している．OS がプロセッサの計算実体を管理する従来のシステムソフトウェアを適用した場合，命令流制御にかかるオーバヘッドや，チップ上で共有するメモリ資源のアクセス効率に問題が生じる．そこで本研究では（1）ユーザレベルでの軽量な命令流制御と，(2) OCMT プロセッサ上で共有するメモリ資源を意識した命令流管理により OCMT システム全体の性能向上を目指す．現状では，独自に検討している OCMT プロセッサ「*OChiMuS*」アーキテクチャの単一プロセシング・エレメント（PE）搭載システム向けに，OS「*Future*」およびユーザレベルスレッドライブラリ「*MULiTh*」の基本機能の検討と実装を行い，*MULiTh* から直接命令流を制御するプロセスモデルを実現した．本論文では，シミュレータを用いた単一 PE 版 *OChiMuS* 向けシステムソフトウェアの実装結果について述べ，複数 PE 版 *OChiMuS* 向けシステムソフトウェアの構想を述べる．

1. はじめに

今日，トランジスタ集積技術の向上および，スーパスカラアーキテクチャの高性能化アプローチである命令レベル並列性（ILP : Instruction Level Parallelism）抽出の限界などを背景に，スレッドレベルの並列性（TLP:Thread Level Parallelism）の向上を目的とするアーキテクチャプロセッサの研究開発が盛んである．複数の CPU コアを 1 チップに搭載して複数の命令流（スレッド）処理を実現する Chip MultiProcessor（CMP）や，CPU 内の演算器等のハードウェアリソースを共有しながら複数の命令流を同時実行させる Simultaneous Multithreading (SMT) アーキテクチャプロセッサ[1),2)] などが市場に登場してきている．さらに今日では，SMT アーキテクチャを CPU コアとする CMP アーキテクチャとして IBM の Power5 が発表され，チップのマルチスレッド化が加速してきている．筆者等は，このような 1 チップで同時に複数の命令流を扱うことを目的としたアーキテクチャをオン・チップ・マルチスレッドアーキテクチャ（OCMT:On Chip Multithreaded architecture）と呼んでいる．これらの OCMT プロセッサを搭載したシステムにおいて，チップ上でより多くの命令流を高効率に並列実行させるためのシステムソフトウェアの役割は大きいと考えている．

† 東京農工大学大学院工学研究科
Graduate School of Technology, Tokyo University of Agriculture and Technology

本研究では OCMT の利点を十分に活かすために，軽量な命令流制御や，OCMT プロセッサ上の共有メモリ資源を有効活用するためのシステムソフトウェアを検討し，システム全体の性能向上を目指している．本論文では，これらシステムソフトウェアの構想，およびシステムソフトウェアとともに検討を進めている独自の OCMT プロセッサ「OChiMuS」向けに実装したオペレーティングシステム（以下 OS）およびユーザレベルライブラリについて述べ，シミュレーションによる評価結果や今後の展望を述べる．

2. 本研究で指向する OCMT プロセッサシステム

Intel 社の Xeon や Pentium4 などを搭載した従来の OCMT プロセッサシステムでは，従来のシステムソフトウェアモデルを対象として命令流の並列性を高めるというアプローチをとっている[8)]．この方法は，アプリケーションプログラムの互換性が高いことや，システムソフトウェアにも大きな改変がないという利点がある．一方で，OCMT プロセッサでは，チップ上で実行する命令流がチップ上の資源を共有しながら実行するという特徴がある．従来の UNIX 流プロセスをチップ上で並列実行させた場合，別アドレス空間で実行する命令流をチップ上で同時に実行させる可能性がある．それぞれの命令流の性能を十分に出すために，従来の OCMT プロセッサでは数百から数メガバイトのキャッシュメモリをチップ上に備えている[8)]．現状では 1 チップで同時に数命令流しか扱わないが，今後

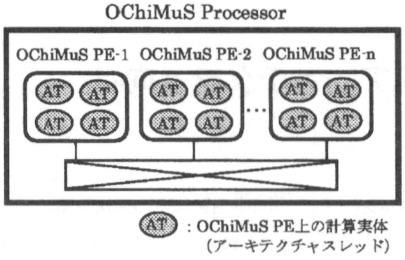

図 1 OChiMuS プロセッサの概念図

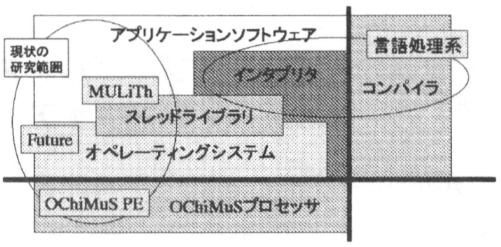

図 2 OChiMuS プロセッサ向けシステムの全体像

チップ上のマルチスレッド化が進んだ場合は，共有するメモリ資源の不足による実行性能の頭打ちも予想され，それを補うための検討が必要である．

また，従来の並列プログラムは，科学技術計算向けの超並列計算機の他，汎用システムとして SMP（Symmetric MultiProcessor）システム，クラスタや Grid などの分散システム上で実行されている．これらのシステムでは，個々の命令流がバスやネットワークを経由して，CPU などの実際に計算を行う実体 (以下，計算実体と呼ぶ）へ割当てられる．これに対し OCMT プロセッサの場合，チップ内の配線を利用した高速通信が可能であるため，命令流の実行制御に係るオーバヘッドが非常に軽いという特徴がある．今後チップ上のマルチスレッド化が進んだ場合，従来の並列計算機で行っていた細粒度の科学技術計算や，専用チップなどで行っていた中流度のマルチメディア系計算なども汎用計算機のチップ内部で行える可能性がある．その場合，OCMT プロセッサ上で並列プログラムを効率よく実行させるためのシステムソフトウェアの役割は大きく，重要な検討課題であると考えている．

本研究では，従来の粗粒度の命令流だけを処理対象とするのではなく，細粒度から中粒度の命令流を OCMT プロセッサ上でできるだけ多く並列実行させ，プログラム全体の実行性能を高めたいと考えている．そのために，単純・軽量な SMT プロセッシングエレメント（PE）を数多く搭載する OCMT プロセッサ「OChiMuS」のアーキテクチャを独自に検討している[6]．**図 1** に筆者らが指向している *OChiMuS* プロセッサの概念図を示す．*OChiMuS* プロセッサでは，複数の命令流を扱う SMT アーキテクチャのプロセッサエレメント（PE）をチップ内に複数個搭載し，1 チップで数十の命令流を扱うことを目標としている．また，本 *OChiMuS* プロセッサでマルチプロセッサ構成システムを構築し，システム全体で数百もの命令流を扱うことを視野に入れている．本 *OChiMuS* プロセッサにおいて，より多くの PE をチップ上に実装するために，PE 単体をできる限りシンプルにしている．例えば PE 内ではキャッシュメモリ，TLB，外部割込み系回路，タイマー等を共有するような設計としている．

図 2 に本研究が目指すシステムの全体像を示す．筆者らは，前述のようなシンプルな PE を数多く搭載した *OChiMuS* プロセッサを指向し，細粒度から粗粒度のアプリケーションプログラムを対象とした最適な命令流実行制御によるプログラムの高速実行を目指している．本研究では，*OChiMuS* アーキテクチャ向けのシステムソフトウェアとして OS，ライブラリ，言語処理系の検討および考察を行っている．

3. OCMT 向けシステムソフトウェアの課題

図 3 に，従来の OCMT プロセッサを搭載するシステムにおけるシステムソフトウェアモデルの一例を示す．従来システムでは，SMP 用の汎用 OS を一部改変し，OCMT プロセッサ上の個々の計算実体を SMP の CPU 単体に見立てて管理し，既存のアプリケーションプログラムを実行させている[8]．OS が制御する命令流の単位は，Linux・Unix 流でいえばプロセスもしくはスレッドである．

この方法はシステムソフトウェアの改変が少ないという利点はあるが，次に示すような問題がある．第一に，従来 OS では OS レベルで命令流を制御するため，命令流操作の度に OS レベルへ遷移する必要があり，命令流制御のオーバヘッドが大きくなるという問題がある．このオーバヘッドの影響により，計算実体へ割り当てる命令流は粗粒度に限定されてしまう．すなわち，粗粒度に比べて命令流制御を頻繁に行う細粒度や中粒度のアプリケーションプログラムの場合，OS レベル遷移に伴う命令流制御のオーバヘッドが大きく影響し，OCMT による並列実行を性能向上に活かせない．第二に，従来 OS による命令流制御では，別アドレス空間で実行する命令流を OCMT プロセッサ上へ割当てることとなり，その場合に OCMT プロセッサで扱うワーキングセットが従来に比べて広がり，キャッ

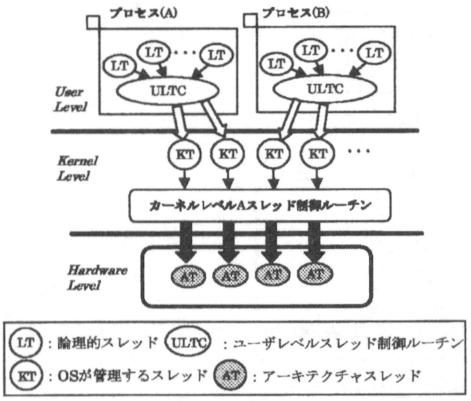

図 3 従来 OS におけるプロセス管理

シュミスや TLB ミスなどを引き起こし実行効率を下げる原因となる[3]．この問題に関しては 2 章で述べたようなメモリ資源の補強という解決手段も考えられる．しかし，OCMT プロセッサのマルチスレッド化が進むにつれメモリが大容量化した場合には，OCMT プロセッサの高コスト化が懸念される．

筆者等はシステムソフトウェアのアーキテクチャを見直すことにより，以上の問題を解決する方針とした．第一の問題に関しては，アプリケーションプログラムの粒度にかかわらず OCMT プロセッサを活用できるように，命令流の制御・管理に関する処理の軽量化を課題とした．第二の問題に関しては，OCMT プロセッサへ割り当てる命令流のメモリアクセス効率を可能な限り向上させるための命令流制御を課題とした．

また細粒度命令流を OCMT プロセッサ上でより効率よく実行させる場合，OS やライブラリでは制御しきれない細かい命令流制御が必要になると考えている．本研究では，OCMT プロセッサ上で細粒度命令流を生成するアプリケーションプログラムの実行性能を向上させるための言語処理系のサポートを課題とする．具体的には，OCMT プロセッサ向けに最適化を行う自動並列化コンパイラや，命令流の並列性を活かすインタプリタなどを考えている．

4. 目標とする OCMT 向けシステムソフトウェア

筆者等は OCMT 向けシステムソフトウェアの検討の第一段階として，まずは単一の PE を実装した *OChiMuS* プロセッサ（以下これを *OChiMuS PE* と呼ぶ）を対象とし，OS およびユーザレベルライブラリによる命令流制御および管理に関する基本機能の検討を行った．

本研究では 3 章に示した課題を解決するために，システムソフトウェアのモデルを見直した[4]．複数の計算実体を持つ *OChiMuS PE*，仮想 I/O 資源，仮想メモリ資源を割り当てる単位として「プロセス」を定義し，OS 「*Future*」がこれを管理する．プロセスに割り当てた資源を使って *OChiMuS PE* 上で実行する命令流の管理と制御は，すべてユーザレベルライブラリ「*MULiTh* (Userlevel Thread Library for Multi-threaded architecture)」[5] で担い，軽量な命令流制御を実現する．*Future* や *MULiTh* で制御しきれない細粒度命令流の制御に関しては，コンパイラやインタプリタで対応する．

Future が *OChiMuS PE* へプロセスを割り当て，*MULiTh* が同一プロセスに属する命令流を *OChiMuS PE* 上の計算実体へ割り当てるという役割分担により，命令流制御の軽量化とともに同一プロセスに属する命令流をプロセッサ上へ集約する．このことにより，ワーキングセットの広がりを緩和したり，データ共有する命令流同士がある命令流によってすでにキャッシュメモリへロードされたデータを他の命令流でも利用することで結果としてキャッシュミスを減らすという「建設的干渉 (constructive interference)[2),3)]」を期待できるようにする．また言語処理系に関しても，*OChiMuS PE* 上の複数の計算実体を活用する最適な命令コードを生成もしくは直接実行することができるようになり，細粒度命令流の実行をサポートすることが可能となる．

しかし，ユーザレベルですべての命令流の制御・管理を担う場合，I/O アクセスなどによる OS レベルでの命令流のブロッキングが発生しても，OS は実行可能な別の命令流へ切り替えることができず，*OChiMuS PE* 上の計算実体の利用効率に影響が出てしまうことが知られている．これに対しては，Scheduler Activations[9] などの従来手法を用いて，OS 内で発生した命令流制御にかかわる事象をユーザレベルへ通知することが有効である．本研究では，Kernel Notification と呼ぶ軽量な事象通知機構を設け，*Future* が *MULiTh* の命令流制御をサポートする．

5. OChiMuS PE

OChiMuS PE は，MIPS プロセッサアーキテクチャをベースとした SMT アーキテクチャプロセッサであり，マルチスレッドをサポートするためのモジュールや命令を新たに拡張している．*OChiMuS PE* アーキテクチャについての詳細は文献 6) を参照されたい．以下，*OChiMuS PE* の概要を述べる．

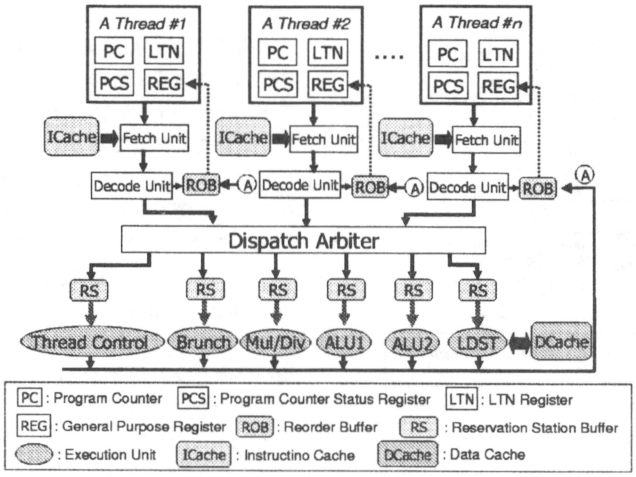

図 4 *OChiMuS PE* アーキテクチャの概要（A スレッド数 3 個搭載時の例）

表 1 OChiMuS PE のスレッド制御命令

命令	説明
`PALLC dr,sr0,sr1`	LTN を sr0，開始番地を sr1 として，論理スレッドを割り当てる
`PDALL dr,sr`	sr で指定した論理スレッドを開放する
`PBLK dr,sr`	sr で指定した論理スレッドを一時停止させる
`PUBLK dr,sr`	sr で指定した論理スレッドの一時停止を解除する
`FWD dr,sr0,sr1`	sr0 で指定した論理スレッドに sr1 のレジスタ値を転送する

* dr : destination register. 制御命令の結果が返る.　　sr : source register

図 4 に *OChiMuS PE* のアーキテクチャ概要図を示す．*OChiMuS PE* では，1 つのプロセッサで複数の命令流を並列実行可能であり，1 つの命令流を処理する単位を**アーキテクチャスレッド**（Architecture Thread：以下，**A スレッド**）と呼ぶ．各 A スレッドは，プログラムカウンタ（以下，PC），汎用レジスタなど，並列実行に必要なハードウェア資源を個別に持ち，演算器などのパイプライン資源やキャッシュメモリ，TLB を A スレッド間で共通に利用する．

OChiMuS PE では A スレッドを仮想化する機能を有しており，システムソフトウェアが管理する命令流（以下，これを**論理スレッド**と呼ぶ）を *OChiMuS PE* 内のどの A スレッドに割り当てているか，という対応付けを *OChiMuS PE* 内部で管理している．*Future* や *MULiTh* などのシステムソフトウェアは，システムソフトウェアで管理する論理スレッドの識別子（LTN:Logical Thread Number）を指定して，A スレッドの生成・削除・一時停止等を制御する．各 A スレッドは**図 5** で示す状態をもち，**表 1** で示す命令によってこの状態を制御する．

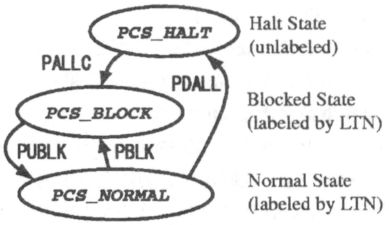

図 5 実スレッドの状態遷移

これらのスレッド制御命令はユーザレベルで利用可能であり，1 サイクルで実行できる．A スレッドは `PALLC` 命令の実行により LTN が設定され，一つの論理スレッドとして割り当てられる．ソフトウェアはその LTN を指定して A スレッドを操作する．例えば，`PBLK` 命令は指定した LTN を持つ A スレッドを一時停止状態（*PCS_BLOCK*）にし，実スレッドの実行を一時的に停止させる．`PUBLK` 命令は命令フェッチ可能状態（*PCS_NORMAL*）へ復帰させ，実行を再開させる．また，`PDALL` 命令は A スレッドを完全停止状態（*PCS_HALT*）へ戻し，A スレッドを解放する．`FWD` 命令は汎用レジ

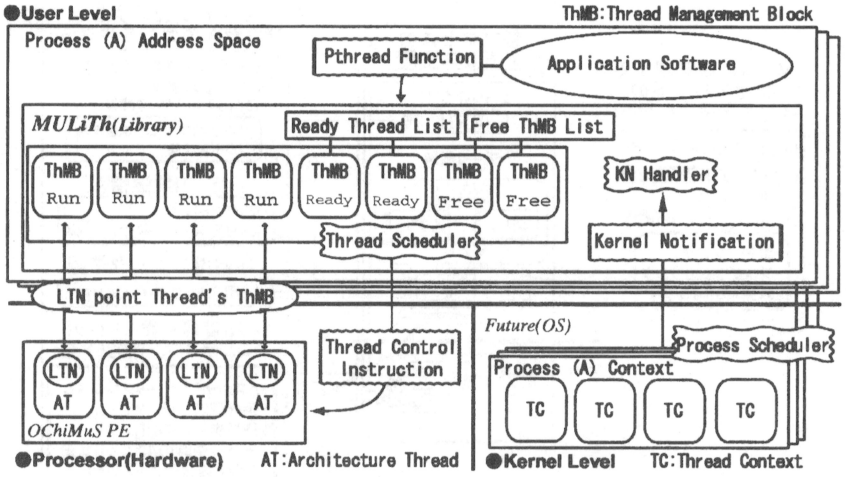

図 6 スレッドライブラリと OS，プロセッサを含めた全体像

スタ値を転送する．これらの命令実行時に，指定した LTN を保持する A スレッドがない場合，デスティネーションレジスタにエラーの値を書き込む．LTN は各 A スレッドの専用レジスタに保持されており，汎用レジスタへのアクセスと同様にユーザレベルにおいて LTN の値を参照・設定することができる．

OChiMuS PE の例外に関しては，外部割込みを固定 A スレッドで発生させ，その他すべての例外☆を例外要因の命令コードを実行させた個々の A スレッドで発生させるという仕様である．

6. OChiMuS PE 対応システムソフトウェア

本章では，*OChiMuS PE* を対象として設計・実装した *MULiTh* と *Future* について述べる．

6.1 MULiTh および Future の概要

図 6 に，*OChiMuS PE* を対象として設計・実装しているシステムソフトウェアの全体構成を示す．*MULiTh* は論理スレッドをプログラマに，いわゆるスレッドとして提供するユーザレベルスレッドライブラリである．従来のスレッドライブラリとは違い，プロセッサ上の複数の A スレッドを *MULiTh* が管理し，生成された論理スレッドの割り当て制御を担当する．プログラミングインターフェースは，POSIX スレッド (Pthread)[11] の仕様に準拠したものを提供する．*MULiTh* に関する詳細は文献 5) を参照されたい．

一方，*Future* は，仮想 I/O，仮想アドレス空間，および *OChiMuS PE* 内のすべての A スレッドをプロセスへ割り当てるという役割を担う．従来 OS では，プロセスに割り当てるハードウェアコンテキストが一つであったため，OS がプロセスごとに管理するハードウェアコンテキストも一つであった．*Future* の場合，*OChiMuS PE* をプロセスへ割り当てるため，プロセスごとに A スレッド個数分のハードウェアコンテキストを管理するという特徴がある．*Future* に関する詳細は文献 7) を参照されたい．

また，*Future* と *MULiTh* 間に備える Kernel Notification では，MULiTh に備える事象受信用ハンドラである Kernel Notification Handler（以下 **KNH**）へ *Future* が検出事象を通知する．KNH において OS からおくられた事象やブロックした論理スレッド情報をもとに論理スレッドの再スケジューリングを可能にしている．*Kernel Notification* に関する詳細は文献 5)，7) を参照されたい．

6.2 プロセスと論理スレッドのコンテキスト管理

図 7 に示すように，*Future* は，プロセスに割当てたアドレス空間や I/O 資源情報，プロセススケジューリングに必要な管理情報，KNH 関連の情報，OS 用スタックの情報，A スレッドコンテキストの情報などを管理する．特に，(4) の KNH 専用のスタックや (5) の OS 用スタックや (6) の A スレッドコンテキスト情報を A スレッド個数分管理することで，6.5 節に示す Kernel Notification の処理や例外処理などを各 A スレッドで同時に実行できるようにしている．

また，*MULiTh* では，各論理スレッドごとにスタックと論理スレッド用データ管理ブロック（ThMB:Thread

☆ MIPS アーキテクチャで扱う割込み例外以外の例外を指す．例えば，システムコール例外，アドレスエラー例外，TLB 関連例外，予約命令例外などである．

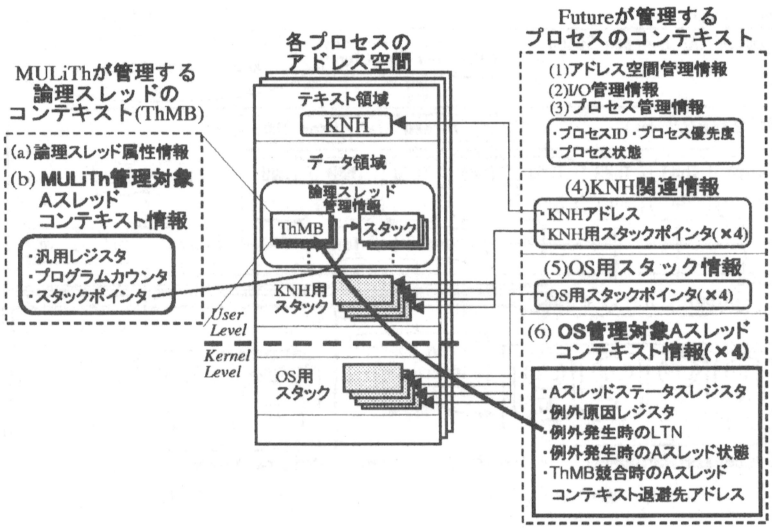

図 7 各プロセスのアドレス空間上の情報とプロセスコンテキストの関係
(OChiMuS PE 内 A スレッドが 4 個の場合)

Management Block) を管理し，各論理スレッドの属性情報や，実行を中断した論理スレッドの PC や汎用レジスタの内容を保持する．*MULiTh* では，この ThMB の先頭アドレスを論理スレッドの識別子 (LTN) として使用しており，*Future* からも実行中の A スレッドの LTN を調べることにより ThMB を参照可能としている．これを利用し *Future* は例外発生の際，実行途中の A スレッドコンテキストの汎用レジスタと PC の情報を ThMB へ退避し，その他の復帰に必要な情報をプロセスコンテキスト (6) へ退避し，例外復帰時もプロセスコンテキストに退避した LTN の値から ThMB を参照し，汎用レジスタや PC の情報を復帰する．この方法により，6.5 節に示す KNH への事象通知を効率よく行っている[5]．

6.3 プロセス切替え方式

Future では従来の OS とは異なり，プロセス切替えの際にプロセスへ割り当てた複数の A スレッドを一斉にかつ効率的に退避・復帰する機構の実現が課題となる．本研究では SMT アーキテクチャの並列処理を活かし，例外処理の枠組みで各 A スレッドのコンテキストを並列に退避・復帰処理する方式とした[7]．プロセス切替えの契機となる例外を受け付けた A スレッドが他の全 A スレッドでプロセス切替え用例外を発生させ，各 A スレッドが例外処理の枠組みで並列に A スレッドコンテキストの退避を行う．ユーザレベルへ復帰する際も各 A スレッドで同時にユーザレベルへの復帰処理を行う．本方式は OS 実行中の A スレッドに対しても従来の多重例外処理の枠組みでプロセス切替えに対処できるなど OS の実装も容易としている．

6.4 論理スレッド制御方式

MULiTh では，論理スレッドの生成，削除制御を *OChiMuS PE* のスレッド制御命令を直接利用し高速に実行する．`pthread_create` 関数によるスレッド生成時には，ThMB のアドレス (LTN) を一つ得て，*OChiMuS PE* の A スレッド割り当て命令 (`PALLC`) を実行する．これが成功した場合，A スレッドへ割り当てた論理スレッドを実行させるために，論理スレッド間レジスタ転送命令 (`FWD`) で A スレッドの初期値を設定し，論理スレッドの一時停止を解除させる命令 (`PUBLK`) で実行を開始させる．`pthread_exit` 関数などによるスレッド削除時には，まず空き LTN リストへ自スレッドの LTN を戻し，次に動作させる論理スレッドをスケジューリングにより決定し切り替えるという手順である．次に実行する論理スレッドがない場合，論理スレッドを完全停止させる命令 (`PDALL`) で自らの処理を停止させるため高速に動作する．

論理スレッド間排他制御や同期機構に関しては，従来のスレッドライブラリではスピンロックやスレッド切替えにより，`pthread_mutex_lock`，`pthread_join`，`pthread_cond_wait` 関数などを実装するのが一般的である．しかし，*OChiMuS PE* のような SMT アーキテクチャでは，スピンロックのロック変数に対する頻繁なメモリ参照が他の A スレッドの実行を阻害する可能性があるため，スピンロックやスレッド切り替

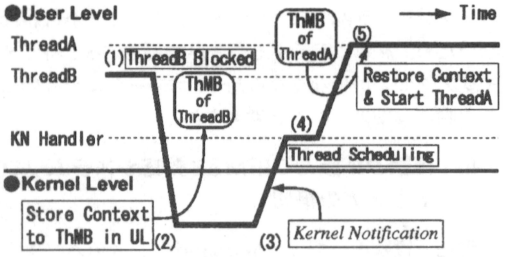

図 8 Kernel Notification の処理の流れ

えを用いない方式を検討した[5]．*MULiTh* では，ロック獲得待ちによりブロックする論理スレッドが PBLK 命令により自らを *PCS_BLOCK* 状態にし，別の論理スレッドがロックを解放する際，前記の論理スレッドに対して PUBLK 命令を発行することで，ロック獲得を再開させる．本方式により，スピンロックで発生する CPU 資源の浪費を無くし，PBLK，PUBLK 命令による高速なブロック制御を実現している．

6.5 OS 事象通知方式

Kernel Notification により *Future* が *MULiTh* へ事象通知する場合とは，(1) システムコールによる I/O アクセス要求やページフォールトなど，OS 内でブロックする必要がある場合や，(2) I/O リクエストの完了やプロセス切り替えなど OS 内でブロックしていた論理スレッドが再開可能となった場合などである．*MULiTh* は KNH のアドレスを，専用システムコールを用いてあらかじめ *Future* へ通知しておく．*Future* は 6.2 節で示したプロセスコンテキスト内に KNH のアドレスを保持しておく．*Future* は通知すべき事象を検出した場合に，OS 遷移時に実行していた論理スレッドの A スレッドコンテキストを復帰せずに，発生した事象に関する情報，KNH のアドレス，KNH 用スタックのアドレス，OS 内でブロックした論理スレッドの LTN などの情報を必要に応じてコンテキストに設定しユーザレベルへ復帰する．このとき従来の事象通知方式では，ユーザレベルスケジューラが OS から受け取ったコンテキストを格納し直す処理が必要であった．Kernel Notification の場合，*Future* が OS 遷移時に，実行途中の論理スレッドの A スレッドコンテキストを直接 ThMB へ格納しているので，効率的な事象通知を可能にしている（**図 8**）．

7. 実装と評価

筆者等は，本論文で述べた *Future* および *MULiTh* の実現方式について，実行駆動型シミュレータ「MUTHASI(MultiTHreaded Architecture Simulator)[6]」を用いて評価した．本シミュレータは *OChiMuS PE* をシミュレートし，A スレッドの個数やキャッシュなどのパラメータを設定可能である．筆者等は，GNU の binutils-2.13☆，gcc-3.2☆☆，newlib1.9.0 を用いて，テストプログラムの作成，*Future* のプロセス管理および *MULiTh* の論理スレッド制御の基本機能について実装した．

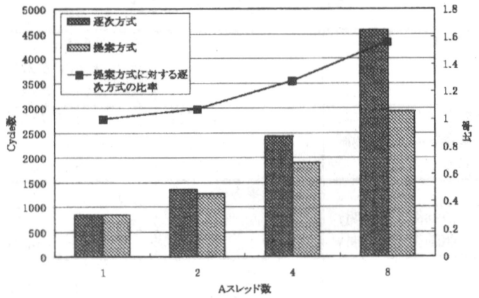

図 9 プロセス切替えサイクル数の比較

7.1 プロセス切替えの評価

図 9 に，シミュレータの A スレッド数を 1，2，4，8 に増加させて筆者等が提案するプロセス切替え方式に要するサイクル数をメモリオーバヘッドの無い状態で計測した結果と，別途比較検討した全 A スレッドコンテキストを逐次的に退避・復帰する方式（以下，逐次方式と呼ぶ）のサイクル数を机上計算で求めた結果を示す☆☆☆．プロセス切替えに要する時間とは，プロセス切り替えを引き起こす例外による OS 遷移後から KNH における次の論理スレッドのディスパッチまでの時間とする． 提案方式の場合，A スレッド数 1 個のプロセス切替え実行サイクル数に対し，A スレッドが 4 個の場合には約 2.3 倍，8 個の場合には約 3.5 倍の割合でサイクル数が増加している．一方，逐次方式は提案方式に比べて，特に A スレッドが 4 個の場合に 1.3 倍，8 個の場合に 1.6 倍の実行時間を要するという結果となっている．特に逐次方式は A スレッド数にほぼ比例して増加する処理があるため，複数の A スレッドコンテキストを扱うプロセス切替え方式としては，本提案方式が性能面で有効であることを示している．

7.2 スレッドライブラリの評価

表 2 は，シミュレータの A スレッド数を 4 個にしてスレッドライブラリの各制御処理を測定した結果で

☆ OChiMuS PE のスレッド制御命令を利用可能にしたもの．
☆☆ 最適化オプションは -O3 を設定した．
☆☆☆ 机上計算に用いた式などの詳細は文献[7] を参照されたい

表 2　論理スレッド制御の性能

評価項目	(1) 提案方式	(2) 従来方式 または (1) が利用 できない場合 (*)
論理スレッド生成	84 (74)	135 (102)
論理スレッド削除	51 (42)	223 (126)
同期	202 (95)	847 (515)
	(1) 提案方式	(2)Spin Lock
排他制御 (短時間)	972	982
排他制御 (長時間)	41461	46656
	(1) 提案方式	(2) 従来方式
OS からの通知	373 (256)	522 (418)

* スレッド制御命令が失敗した場合，または利用できない場合
* 単位は総実行サイクル数 (かっこ内は実行命令数)

ある．なお，評価に利用した Scheduler Activations 機構については，カーネルからの通知機構部分のみを実装した．

論理スレッドの生成に関して，(1) は PALLC 命令が成功した場合，(2) は従来のスレッドライブラリの動作または PALLC 命令が失敗したときの動作である．(2) では実際に生成したスレッドが動作するには，スレッド切り替えのオーバーヘッドが加わるが，(1) では処理の多くがレジスタアクセスとなり，高速に論理スレッドを生成できている．

論理スレッドの削除に関して，(1) は待ちスレッドがない場合の PDALL 命令によるスレッド削除，(2) は従来のスレッド削除における他のスレッドへ切り替え，または *MULiTh* においてのスレッド切り替えを示している．PDALL 命令を実行する (1) では，スレッドの削除が 1 命令ですむので非常に軽量である．

同期制御に関して，(1) は *OChiMuS PE* のの *PCS_BLOCK* 状態を利用した方式，(2) は従来のスレッド切り替えによる方式の結果である．(1) ではコンテキストの退避・復帰を行わないので，(2) に比べて 4 倍の性能向上となった．

排他制御制御に関して，ロック獲得のための待ちに，(1) は *OChiMuS PE* の *PCS_BLOCK* 状態を利用した方式，(2) はスピンロックを利用した方式である．短時間，長時間とは，クリティカルセクションの長さである．短時間の処理では，違いはあまり見られない．しかし，長時間排他制御による保護を行うと，(2) ではスピンロックによって他の実スレッドの実行を妨げてしまい性能が落ちることが確認でき，提案する方式の有用性が示せた．

OS からの事象通知方式に関して，(1) は提案する Kernel Notification による通知，(2) は Scheduler Activations 機構，特にそれを効率的に行う C-area[10] による通知を実装し，比較した．Kernel Notification ではスケジューラ内でのコンテキストのコピーが行われないため，高速に実行することができている．特にメモリアクセスが少なくなるため，キャッシュを有効にした際には効果があると考えられる．

8. 複数 PE 版 OChiMuS 対応システムソフトウェアの構想

現在，複数 PE 版 *OChiMuS* 対応システムソフトウェアの構想を進めている．まず筆者らは，複数 PE 構成の *OChiMuS* プロセッサの CPU 資源の割り当て方法について検討している．単一 PE 版 *OChiMuS* プロセッサの場合，プロセスに対して *OChiMuS PE* を割当てた．複数 PE 版 *OChiMuS* プロセッサの場合，以下の方法が考えられる．

- *OChiMuS* プロセッサ単位でプロセスへ割当てる．
- *OChiMuS PE* 単位でプロセスへ割当てる．

OChiMuS プロセッサに多くの *OChiMuS PE* が搭載されることを念頭に置いた場合，並列度のそれ程高くない（すなわち生成する論理スレッド数が少ない）プロセスに対して，*OChiMuS* プロセッサ（すなわち全 PE）を割当てることは得策ではない．しかし，細粒度・高並列なプログラムを対象とした場合には，プロセスに対して OChiMuS プロセッサを割当てるほうが，チップ上のメモリ資源の利用効率などの点で有効だと予想される．そこで筆者等は，*OChiMuS PE* 単位で適量のプロセッサ資源をプロセスへ割り当て，*OChiMuS* プロセッサのスレッドレベル並列性 (TLP) を活かしたいと考えている．

図 10 に，複数の PE を持つ *OChiMuS* プロセッサと *Future* と *MULiTh* の概念図を示す．*Future* が複数の *OChiMuS PE* を管理し，プロセスに対して *OChiMuS PE* 単位でプロセッサ資源を割り当てていく．またプロセスに属する論理スレッドの制御は，単一 PE の時と同様に *MULiTh* が担う．本方式を実現するためのより具体的な検討課題を示す．

(1) 複数 PE 上 A スレッド制御方法
MULiTh が割当てられた *OChiMuS-PE* の物理的な位置（番号）や数を知らなくとも 6.4 節で示した論理スレッド制御を行えるようにしたいと考えている．

(2) *OChiMuS-PE* 数の制御方法
プロセスから生成される論理スレッド数と *OChiMuS* プロセッサの CPU 資源の量を把握し，*OChiMuS-PE* の割り当て数を増減させる方法を考える必要がある．

(3) プロセスコンテキストの管理方法

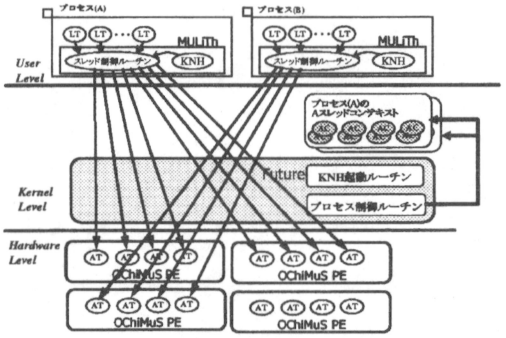

図 10 複数 PE の OChiMuS プロセッサの PE 管理

割り当てた PE 数に応じてプロセスごとに管理する A スレッドコンテキスト数も異なるため，6.3 節に示したプロセス切り替え方法やプロセスコンテキストの管理方法について再検討する必要がある．

今後はこれらの課題解決に向け *OChiMuS* アーキテクチャおよびシステムソフトウェアの検討を進めていく．

9. ま と め

本論文では，スレッドレベル並列性（TLP）の高い OCMT プロセッサを指向し，細粒度から粗粒度に至る命令流を扱うためのシステムソフトウェアに関し，従来の問題点を示し，筆者らの目標とする OCMT システムおよび提案するシステムソフトウェアアーキテクチャについて述べた．OCMT の利点を十分に活かすために，筆者らはユーザレベルでの軽量な命令流制御と，OCMT プロセッサ上の共有メモリ資源を有効活用するためのプロセス管理，さらにユーザレベルでのより効率的な命令流制御を実現するための Kernel Notification と呼ぶ軽量な事象通知機構などを検討した．システムソフトウェアとともに検討を進めている独自の OCMT プロセッサ「OChiMuS」の単一 PE 向けに実装した OS およびユーザレベルライブラリについて述べ，シミュレーションによる評価結果により本論文で示した方式の有効性を示した．今後は，複数 PE 版 OChiMuS プロセッサ対応のシステムソフトウェアの検討を進めることと，より実用的なアプリケーションを動作させた場合の本提案方式の効果を明らかにしていきたいと考えている．

参 考 文 献

1) D.M.Tullsen, S.J.Eggers, and H.M.Levy : Simultaneous multithreaing : Maximizing on-chip parallelism, Proc. of the 22nd Annual International Symposium on Computer Architecture, pp. 392–403 (1995).
2) Jack L. Lo, Luiz A. Barroso, Susan J. Eggers, Kourosh G., Henry M. Levy and Sujay S. Parekh : An analysis of database workload performance on sisimultaneous multithreaded processors, Proc. of the 25th Annual International Symposium on Computer Architecture, pp. 39–50 (1998).
3) 山崎真矢，本多弘樹，弓場敏嗣: マルチスレッドアーキテクチャにおけるデータキャッシュ構成方式の提案，情処研報 HPC-73-14, pp. 79–84 (1998).
4) 佐藤未来子，河原章二，中條拓伯，並木美太郎: SOC 時代に向けた SMT 用 OS の構想，情処研報，Vol. 2002, No. 79, pp. 31–38(2002).
5) 笹田耕一，佐藤未来子，河原章二，加藤義人，大和仁典，中條拓伯，並木美太郎：マルチスレッドアーキテクチャにおけるスレッドライブラリの実装と評価，情報処理学会論文誌：コンピューティングシステム，Vol. 44, No.SIG 11 (ACS3), pp. 215–225(2003).
6) 河原章二，佐藤未来子，並木美太郎，中條拓伯: システムソフトウェアとの協調を目指すオン h シップマルチスレッドアーキテクチャの構想，コンピュータシステムシンポジウム，Vol.2002, No. 18, pp. 1–8(2002).
7) 佐藤未来子，笹田耕一，河原章二，加藤義人，大和仁典，中條拓伯，並木美太郎：マルチスレッドアーキテクチャ向け OS「Future」におけるプロセス管理，情報処理学会論文誌，ACS Vol. 5 (掲載予定) (2004).
8) Intel Technology Journal(Hyper-Threading Technology), http://developer.intel.com/ technology/itj/2002/volume06issue01/index.htm
9) Anderson, T., Bershad, B., Lazowska, E., and Levy, H.: Scheduler Activations : Effective Kernel Support for the User-level Management of Parallelism, Proc. the 13th ACM Symp. On Operating System Principles, pp. 95–109 (1991).
10) 猪原, 益田: ユーザとカーネルの非同期的な協調機構によるスレッド切替え動作の最適化, 情報処理学会論文誌, Vol. 36, No. 10, pp. 2498–2510(1995).
11) IEEE: *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1* (1996).