卒業論文

題目

マルチプロセッサ用のハードウェアスケジューラのオーバヘッド低減手法に関する研究

指導教員

佐々木　敬泰助手

2010年

三重大学 工学部 情報工学科
計算機アーキテクチャ研究室

小林　美喜 (407816)

内容梗概

近年，マルチプロセッサ・システムが普及し，プログラムを分割して並列に実行する並列処理手法が広く用いられるようになっている．中でもプログラムを粒度の細かいスレッド等の処理単位で，大多数に分割する中・細粒度並列処理は，実行するマルチプロセッサ・システムの形態に強く依存しない手法として注目されている．しかしながらこの手法では，プログラムを細かく分割する事によるタスク数の増加により，スケジューリングやコンテキスト・スイッチ，同期処理等に起因するオーバーヘッドが増大し，大幅な性能低下を招く危険性がある．

この問題を低減する手法として，ハードウェアでスケジューリングを支援するSSH(Scheduling Support Hardware)，およびハードウェアでコンテキスト・スイッチ処理を支援するCSS(Context switch Support System)が提案されている．これらの手法を用いる事で，スケジューリング及びコンテキスト・スイッチに起因するオーバーヘッドが低減され，高速化が実現された．しかしながら，従来のCSSではコンテキスト・データの退避にスケジューラバスを利用しているため，PE数が増加すればスレッドのスケジューリング時間にオーバヘッドがかかってしまうため，これを解決する必要がある．

そこで，本研究では，コンテキスト・スイッチを行う際に，前のスレッドで使用されていないレジスタは送信せず，使用されたレジスタのみを送信することでコンテキスト・データを圧縮しスケジューリング・オーバヘッドを低減させることにより，CSSを効率よく利用し更なる高速化を目指す．

Abstract

Today, multiprocessor systems are widely used in every usage. Particularly, middle grain parallelism, dividing program into a large number of small threads, is paidattention to. Because it hardly relies on multiprocessor system environment. However, the overhead of task scheduling, context switching and synchronization process becomes large compared to the execution time of threads and it often makes performance down greatly.

In order to reduce these overheads, the Scheduling Support Hardware architecture(SSH) and Context switch Support System(CSS) are proposed. SSH accelerates the performance of the OS with hardware by scheduling threads concurrently with the thread execution on the CPU. CSS saves the registers the former thread used and fetches the registers used in next thread in parallel with the thread execution on the CPU. By using them, the overhead is reduced. However, the scheduling overhead increases as the number of PE increases. Because it transmits the context data by using the scheduler bus. Then, in the present study aims at further speed-up by efficiently using CSS by decreasing scheduling overhead by compressing the context data by not transmitting the register not used in the previous thread, and transmitting only the register used when the context switch is done in the present study.

目 次

図 目 次

表 目 次

1 はじめに

近年，マルチプロセッサ・システムが普及し，プログラムをDOループやサブルーチン，関数といった粒度の粗い単位で分割して，並列に実行する粗粒度並列処理が広く用いられるようになっている．しかし，粗粒度並列処理ではタスクの粒度が粗いため，均等な負荷分散が難しく，プロセッサ台数が異なるマルチプロセッサ・システムにおいて，常に性能を最大限引き出せるようなプログラムを記述するのは極めて困難である．この問題を解決する方法の1つとして存在するのが，中・細粒度並列処理である．中・細粒度並列処理は，粗粒度並列処理に対して，プログラムを非常に小さく分割する．また，プロセッサ台数と比較しても圧倒的に多い数のスレッド集合に分割する．そして，実行条件の整ったスレッドから動的にアイドルプロセッサに割り当てて実行することで，プロセッサ台数に強く依存しない並列処理を実現出来る．
しかし，このような中・細粒度並列処理を従来のOSを用いて実行するには問題がある．何故なら，扱うタスク数が増える事により，スケジューリングやコンテキスト・スイッチの回数が増加する．そのため，これらに起因するオーバーヘッドが増大し，中・細粒度の並列性を有効に利用出来ないばかりでなく，大幅な性能低下を招く危険性があるからである．つまり中・細粒度並列処理を有効に利用するには，効率的なスケジューリング，コンテキスト・スイッチの実現が不可欠となる．
これらの問題を低減するための手法として，本来OSの機能の一部であるスレッドのスケジューリングや，CPU資源の割り当て／解放の機能をハードウェアで実行する事で細粒度な並列性を有効利用し，高速なスケジューリングの実現を目指すスケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) を用いたマルチプロセッサ・アーキテクチャが提案されている．SSHはスケジューリング専用ハードウェアを導入することで，CPU上のスレッド実行とスケジューリングを並列に行い，スケジューリングに要する時間を隠蔽している．また同様に，コンテキスト・スイッチによるオーバーヘッドの問題を低減するため，コンテキスト・スイッチに伴うレジスタの退避／復帰処理をハードウェア化し，高速なコンテキスト・スイッチの実現をするためのコンテキスト・スイッチ支援システム (Context switch Support System; CSS) を用いたアーキテクチャが提案されている [1]．SSHとCSSにより，スケジューリングやコンテキスト・スイッチに起因するオーバーヘッドによる性能低下を抑え，高速化が実現されている．しかしながら，従来のCSSではコンテキスト・

データの退避にスケジューラバスを利用しているため，PE数が増加すればスレッドのスケジューリング時間にオーバヘッドがかかってしまうため，これを解決する必要がある．
そこで，本研究では，コンテキスト・スイッチを行う際に，前のスレッドで使用されていないレジスタは送信せず，使用されたレジスタのみを送信することでコンテキスト・データを圧縮しスケジューリング・オーバヘッドを低減させることにより，CSSを効率よく利用し中・細粒度並列処理の更なる高速化を目指す．
以降，本論文では次のように構成される．まず，次章では研究の背景となる中・細粒度並列処理の概要とその問題点を説明し，研究の目的を述べる．3章では，SSHとCSSについて述べ，4章では，提案する改良案のシステムの詳細なアーキテクチャ及び動作について述べる．そして5章にて，改良後の性能評価について述べ，最後に6章で結論と今後の展望について述べる．

2 研究背景

2.1 中・細粒度並列処理

近年，マルチプロセッサ・システムが普及し，プログラムを並列に実行するための技術である並列処理手法に関する研究が広く行われている．この並列処理手法のうち，プログラムをDOループやサブルーチン，関数といった粗い単位に分割して，並列に実行する手法を粗粒度並列処理と呼ぶ．これに対して，CPU数より圧倒的大多数の細かいスレッド等に分割して，並列に実行する手法を中・細粒度並列処理と呼ぶ．
並列処理手法を用いた処理で現在の主流となっているのは，粗粒度並列処理である．しかしながら粗粒度並列処理では，分割したタスクの粒度が粗いため均等な負荷分散が難しく，プロセッサ台数やネットワーク形態等が異なる様々なマルチプロセッサ・システムにおいて，性能を常に最大限引き出せるようなプログラムを記述する事は，極めて困難である．そこで，注目されているのが中・細粒度並列処理である．
中・細粒度並列処理ではプログラムを細かく，そしてCPU数よりも圧倒的大多数のスレッドに分割するため，均等な負荷分散が容易になり，実行するマルチプロセッサ・システムに強く依存しない性能が得られるとして期待されている．しかしながら問題点も存在しており，中・細粒度並

列処理の利点を充分に活用出来ていないのが現状である．その問題を引き起こす要因となる，代表的な処理を以下に挙げる．

- スケジューリング
- コンテキスト・スイッチ
- スレッド間同期

本章では，これらの処理についての簡単な説明と共に，中・細粒度並列処理におけるそれぞれの処理の問題点を次節以降より説明して，本研究の背景と目的を述べる．

2.2 スレッドについて

ここで，各処理とその問題点を述べる前に，スレッドについて簡単に説明する．スレッドとは，プログラムの実行単位であるプロセス内に１つ以上存在する並列実行単位である．プログラムが起動されると，プロセスが生成される．その際，記憶領域やディスク資源がプロセス毎に割り当てられる．これに対してスレッドは，プロセスとして実行中のプログラム内で生成される．そのため，スレッド生成時にスレッドに割り当てられる記憶領域等の資源は，そのプロセスに割り当てられた資源の一部から割り当てられる．また，スレッドはCPU 割り当ての単位であり，実行可能状態になると，スケジューリング対象となる．
スレッドを用いる利点として，スレッド間の通信コストの低さがある．プロセスでは，プロセス毎に独立した記憶領域を持っている．そのため，プロセス間での通信コストは大きくなる．それに対してスレッドでは，同じプロセス内の他のスレッドと記憶領域を共有しているため，プロセス間通信より高速になる．
また，もう一つの利点として，スレッドの切り替えコストの低さがある．プロセスは切り替えるためにレジスタや，記憶領域の退避／復元等が必要となる．この内，記憶領域の退避／復元に膨大なコストがかかるが，スレッド切り替えには必要ないため，切り替えコストが低くなる．以上のような利点により，特に粒度の小さい処理を扱う並列処理技術では，スレッドが広く用いられている．

2.3 スケジューリング

スケジューリングとは，分割した処理の実行順序や，実行するCPUの割り当て等を，その処理の優先度やスケジューリング・ポリシに従って決定する処理の事である．並列処理では逐次処理と異なり，分割した処理の実行するタイミングを厳密に管理する必要がある．これは，分割した処理の実行するタイミングによって，最終的な結果を誤ってしまう危険性が存在するためである．そのため，CPU上での処理が切り替わる場合や，分割した処理が終了した場合等には再度スケジューリングを行う必要がある．つまりスケジューリングは，プログラムの開始から終了までの間に，複数回実行される．
中・細粒度並列処理では，分割数が粗粒度並列処理より大幅に多くなる．そのため，スケジューリング対象が増え，スケジューリング回数が分割数に応じて大幅に増加する．また，分割した1つの処理の開始から終了までに要する時間が短くなり，1回のスケジューリング時間が1つの処理に対して相対的に長くなる．その結果，スケジューリングに要するオーバーヘッドが増大し，大幅な性能低下を招き，問題となっている．

2.4 コンテキスト・スイッチ

コンテキスト・スイッチとは，CPU上の処理を切り替える処理の事である．CPU上で実行していた処理が終了した場合や，何らかの理由で処理を続行出来なくなった等で，他の処理へと実行権限を譲る場合にコンテキスト・スイッチが行われ，CPU上での処理が切り替わる．具体的にコンテキスト・スイッチとは，それまで実行していた処理が使用していたレジスタの内容をメモリへと退避し，次に実行する処理が必要とするレジスタの内容をメモリから復帰する処理である．また，現在普及している多くのCPUでは，ユーザが利用出来るレジスタ数が，1本あたり32bit又は64bitのレジスタが32本であるのが一般的である．つまり，コンテキスト・スイッチ処理で単純に全てのレジスタを退避／復帰するには，それぞれ32回ずつ，合計64回ものメモリ・アクセスが必要となる．
粒度に関わらず並列処理では，複数のCPUからメモリ・アクセスが同時に発生する可能性がある．本研究で想定しているような共有メモリ型アーキテクチャでは，同時にメモリ・アクセス出来るCPUはただ1つのみであるため，メモリ・アクセスが頻発すると共有メモリバスの競合が起き，メモリ・アクセス・レイテンシが大きくなる．これがオーバーヘッ

ドとなり，性能低下の原因となる危険性がある．特に中・細粒度並列処理では分割数が非常に多いため，コンテキスト・スイッチ回数も増えて，メモリ・アクセスが頻発する．その結果，メモリ・アクセス・レイテンシに起因するオーバーヘッドが，大幅な性能低下を招く危険性があり問題となっている．

2.5 研究の目的

前節までに，中・細粒度並列処理において，スケジューリングやコンテキスト・スイッチに起因するオーバーヘッドが大幅な性能低下を招く危険性がある事を説明した．これらの問題は中・細粒度並列処理を利用する際には必ず対処する必要があるため，関連する研究が広く行われている．その中で，スケジューリングに起因するオーバーヘッドの改善については，佐々木らによりスケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) が提案されている．SSH は，中・細粒度並列処理の効率的な利用を目的としたもので，CPU 上のスレッド実行と並行して専用のハードウェアによりスケジューリングを支援する．また，コンテキスト・スイッチに起因するオーバーヘッドの改善についても，村松らによりコンテキスト・スイッチ支援システム (Context switch Support System; CSS) が提案されている．CSS は，SSH がスケジューリングに用いる専用のハードウェアにコンテキスト・スイッチ機能を追加実装し，CPU 上のスレッド実行と並行してコンテキスト・スイッチを支援する．SSH と CSS については 3 章にて，より詳しく説明する．

SSH と CSS により，スケジューリングとコンテキスト・スイッチに起因するオーバーヘッドによる性能低下が低減され，逐次処理に対して高性能となる結果も得られている．しかしながら前節で述べた通り，コンテキスト・スイッチに起因するスケジューリングオーバーヘッドについても，中・細粒度並列処理においては無視出来ない問題である．そこで本研究では，コンテキスト・データの圧縮のための機構を提案して，SSH と CSS を搭載したマルチプロセッサ・アーキテクチャに追加実装する事で，中・細粒度並列処理をより効率的に利用出来る環境の構築を目的とする．

3 SSHとCSSを用いた従来のマルチプロセッサ環境

2章において，中・細粒度並列処理ではスケジューリングやコンテキスト・スイッチに起因するオーバーヘッドが，大幅な性能低下を招く危険性がある事を説明した．この問題を解決するために，様々な研究が盛んに行われている．その中の1つに，スケジューリングとコンテキスト・スイッチについての問題を扱い，従来の並列処理手法より性能を向上させている研究がある．その研究では，スケジューリングやコンテキスト・スイッチを，専用ハードウェアを用いて，CPU上のスレッド実行と並行して行っている．このスケジューリングとコンテキスト・スイッチ，それぞれの専用ハードウェアは，スケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) とコンテキスト・スイッチ支援システム (Context switch Support System; CSS) と呼ばれる．これらを用いる手法により，スケジューリングとコンテキスト・スイッチに起因するオーバーヘッドを低減し，従来の並列処理手法より5%から30%程の高速化を実現している．本章では，SSH とCSS のそれぞれについて説明する．図5.8 にSSH とCSS を用いたマルチプロセッサ・アーキテクチャを示す．このマルチプロセッサ・アーキテクチャは，複数のPE(Processor Element)，SSH-m(SSH-master)，共有メモリ (Shared Memory)，メモリ・バス調停器 (Memory Bus Arbiter)，スケジューラ・バス調停器 (Scheduler Memory Arbiter) から成る．メモリ・アーキテクチャとしては，集中あるいは分散共有メモリを想定しており，同一のアドレス空間を共有しているものとする．これは，スレッドを容易に任意のPE に割り当て可能にするためである．

PE は以下の3 つのユニットから構成される．すなわち，プログラムの実行を行うCPU と，SSH の一部であるSSH-s(SSH-slave)，そしてCSS で構成されている．SSH-s はCPU とSSH-m のインタフェースを提供するもので，CPU の動作と並行して次に実行されるスレッドの取得，新規スレッドのキューイング等を行う．CSS はコンテキスト・スイッチにおけるレジスタの退避／復帰を高速に行うものである．CPU とSSH-s，CSS はオンチップでの実装を想定しており，その内，CPU とSSH-s は32 ビットのアドレス／データ共有のバスで接続されているものとする．また，CPU とSSH-s の通信はメモリマップドI/O により実現しており，その物理アドレスを表3.1 に表す．

表 3.1: CPUから見たRegister Unit

レジスタ名	物理アドレス
Command	bff10000h
Status	bff10004h
WQ-Status	bff10008h
RQ-Status	bff1000ch
Write Queue 0	bff10100h
Write Queue 1	bff10104h
Write Queue 2	bff10108h
:	:
Write Queue 35	bff1018ch
Read Queue 0	bff10200h
Read Queue 1	bff10204h
Read Queue 2	bff10208h
:	:
Read Queue 35	bff1028ch

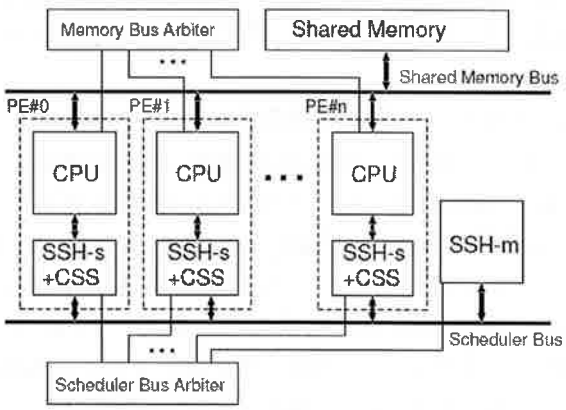


図 3.1: マルチプロセッサ・アーキテクチャ

3.1 SSHの概要

SSH では，CPU 上でのスレッドの実行と並行して，スレッドのスケジューリングを行う．SSH はSSH-m とSSH-s から成っており，スケジューリングは主にSSH-m で行う．SSH-m では，SSH-s を介して送信された新規スレッドを管理し，スケジューリング・ポリシに従ってスケジュールする．また，SSH-s から要求があった場合，最も優先度の高いスレッドの管理情報をSSH-s に返す．SSH-m 及びSSH-s の詳細については，次節以降で詳しく述べる．
SSH-m とSSH-s は専用のスケジューラ・バスにより接続される．スケジューラ・バスの調停は専用の調停器を用いる．このスケジューラ・バス調停器は，固定プライオリティとラウンドロビン型の変動プライオリティを組み合わせたもので，SSH-m からのバス使用要求を最優先とし，それ以外のSSH-s からの要求はラウンドロビンにより決定する．

3.2 SSHの詳細

SSH の構成要素には，スケジューリング等の処理を行うSSH-m と，CPU とSSH-m とのインタフェースを提供するSSH-s がある．また，SSH はサーバ／クライアント・モデルになっており，SSH-m はサーバとして，SSH-s はクライアントとして機能する．本節ではSSH-m 及びSSH-s の詳細なアーキテクチャについて述べる．

3.2.1 SSH-m

図5.9 にSSH-m のブロック図を示す．SSH-m はスレッドのスケジューリングを行うHardware Scheduler，及び実行可能キュー等を保持するGlobal Queue から成る．
各モジュールの機能
各モジュールの機能について，以下に述べる．
Hardware Scheduler: スレッドのスケジューリングを行う．スケジューリング・ポリシとしては，FIFO，ラウンドロビン，Other を実装する．スケジューリング・ポリシのOther とは，Pthread のSCHED OTHER に相当するものである．
Global Queue: 各スケジューリング・ポリシ毎の実行可能キュー(Ready

Queue)，及び待ちキュー (Wait Queue) から成り，SRAM を用いて実装する．

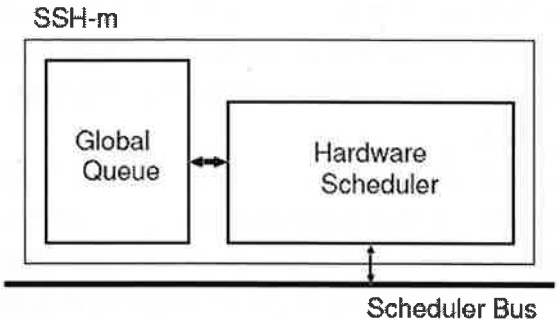


図 3.2: SSH-m のブロック図

3.2.2 SSH-s

図 3.3 に SSH-s のブロック図を示す．SSH-s は CPU と SSH-m のインタフェースを提供するもので，CPU とのインタフェースを提供する CPU-SSH Interface Unit，スケジューラ・バスを介して SSH-m と通信を行う SSH-SSH Interface Unit，及び Global Queue から先行してロードしたスレッドや Global Queue に登録すべきスレッドをバッファリングするための Register Unit から成る．

各モジュールの機能

各モジュールの機能について，以下に述べる．

CPU-SSH Interface Unit: CPU の発行した Load ／ Store 命令のメモリ・アドレスを Register Unit にマッピングし，内部レジスタの書き込み，読み出しを仲介する．

SSH-SSH Interface Unit: Register Unit を監視し，Read Queue が空になると，SSH-m に要求を出し，次に実行すべきスレッドを取得する．また，Write Queue に有効なデータが入ると，スケジューリングさせるために SSH-m にスレッドを送信する．

Register Unit: SSH-s 及び SSH-m へのコマンドや，次に実行すべきス

レッドを格納する Read Queue，新規に生成したスレッド，あるいは，実行権限を譲ったスレッドが SSH-m に送信されるまで保持しておく Write Queue から成る．

Register Unit の詳細

ここでは，Register Unit の詳細について述べる．図 4.5 の Register Unit は 76 本の 32 ビット・レジスタから成る．CPU から各レジスタにアクセスする場合，各レジスタを特定のアドレスにマッピングされた 1 ワードのメモリと見なして，Load ／ Store 命令を用いてアクセス出来る．各レジスタの機能を以下にまとめる．

Command: SSH-s 及び SSH-m への命令レジスタ．SSH-s に対する命令は，初期化，停止，タイムスライス設定の 3 つで，SSH-m に対する命令は，Global Queue の初期化のみである．

Status: Command レジスタの状態を表すもので，0 か 1 の値をとる．0 のときは，命令受け付け状態，1 のときは命令実行中を表す．CPU は，Command レジスタに命令を書く前に，Status レジスタが 0 であることを確認する必要がある．また，Command レジスタに命令を書き込み後，当該レジスタを 1 にする．

WQ-Status: Write Queue の状態を示すもので，Status レジスタ同様，0 か 1 の値をとる．Write Queue に未送信のデータが入っている場合は 1，空の場合は 0 である．Status レジスタ同様，Write Queue にデータを書き込み後，当該レジスタを 1 にする．

RQ-Status: Read Queue の状態を示すもので，Status レジスタ同様，0 か 1 の値をとる．Read Queue に有効なデータがある場合は 1，空の場合は 0 である．CPU は Read Queue を読み込み後，当該レジスタを 0 にする．

Write Queue: 新規に生成されたスレッドや実行権限を譲ったスレッド等，SSH-m に送信し，スケジュール対象となるスレッドを保持する．Write Queue は Write Queue0-3 までの 4 本で，1 つのスレッドを保持する．Write Queue4-35 までのレジスタは SSH では使用していないが，後述する CSS のために，スレッドのコンテキスト情報を保持出来るように準備している．また，このレジスタは，CPU-SSH Interface Unit を介して CPU により書き込まれ，SSH-SSH Interface Unit に読み出される．

Read Queue: 次に実行すべきスレッドを保持する．Read Queue は，Read Queue0-3 までの 4 本で，1 つのスレッドを保持する．Read Queue4-35 までのレジスタは，Write Queue 同様，SSH では使用していない．また，

このレジスタは，SSH-SSH Interface Unit により書き込まれ，CPU-SSH Interface Unit を介して，CPU により読み出される．

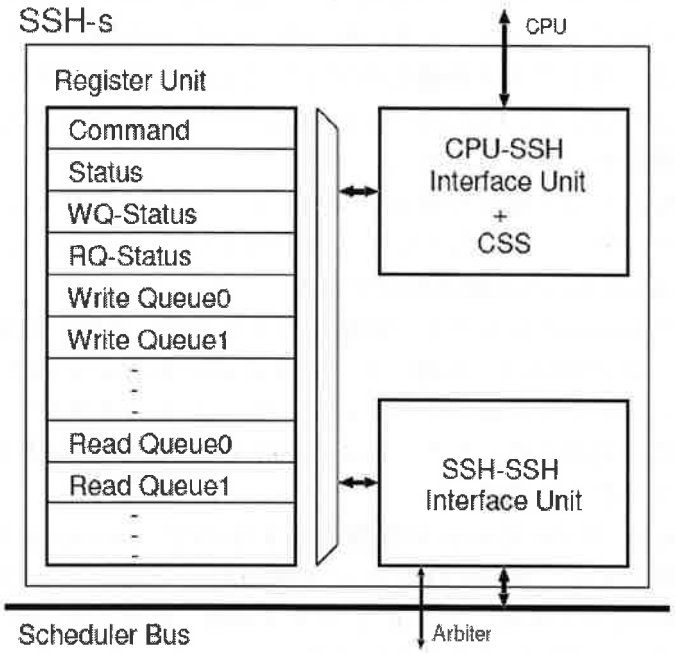


図 3.3: SSH-s のブロック図

3.3 CSS の概要

CSS は SSH で扱うスレッド管理情報にコンテキスト情報を付加する事により，コンテキスト・スイッチ時に伴うレジスタの復帰／退避先を共有メモリではなく，SSH-m にある Global Queue にしている．また，コンテキスト・スイッチ専用のハードウェアにより，CPU 上のスレッド実行と並行してコンテキスト・スイッチを高速に実行する．従来は主記憶で行っていたコンテキスト情報の管理をスケジューリング・バスを介して SSH-m で行う事により，メモリ・バスの使用頻度が高くなる事による性能

低下を回避している．CSS ではコンテキスト情報を図 3.3 にある，Write Queue4-35 及び Read Queue4-35 を用いて保持している．

3.4 CSS の詳細

図 3.4 に，SSH 及び CSS を搭載した PE のブロック図を示す．CSS では SSH-s が保持しているスレッド情報のうち，コンテキスト情報にあたる Read Queue4-35 及び Write Queue4-35 を用いる．SSH-s の Write Queue が空で，次に実行されるスレッドの情報が Read Queue に保持されている場合，CSS では現在実行中のスレッドが実行権限を譲る等して，スレッドが切り替わる際に CPU にある Register Switch を制御する事によりコンテキスト・スイッチを高速に行う．これにより，スレッド切り替え時のコンテキスト情報の退避／復元処理に要するサイクル数を削減している．なお，図の簡略化のため，バス・コントローラ，及び CPU 内部のメモリ管理ユニット (Memory Managing Unit :MMU)，TLB，乗除算ユニット，システム制御コプロセッサの一部は除いてある．

3.5 SSH 及び CSS の動作

ここで，SSH 及び CSS の動作を簡単に説明するために動作の一例を示す．初期条件として，実行開始から十分に時間が経過しており，Global Queue には実行待ちのスレッドが既に存在し，また，当該 SSH-s の Read Queue 及び Write Queue 共に空であるとする．

STEP1: CPU 上ではユーザのスレッドが実行されている．一方，SSH-s は Read Queue が空であるので，スケジューラ・バスの使用要求を出し，バスの使用権を待つ．

STEP2: スケジューラ・バス調停器よりバスの使用権を得る．

STEP3: スケジューラ・バスを介して，SSH-m に次に実行すべきスレッドの情報を要求する．

STEP4: SSH-m は，SSH-s からの要求に従い，Global Queue から最も優先度の高いスレッドの情報を取り出し，SSH-s に返す．

STEP5: SSH-s は次に実行すべきスレッドの情報を SSH-m から受け取り，Read Queue に格納する．

STEP6: CPU 上で実行中のスレッドが，実行権限を譲る等して，スレッドの実行が切り替わるタイミングになると，CPU 上ではスレッド・ライ

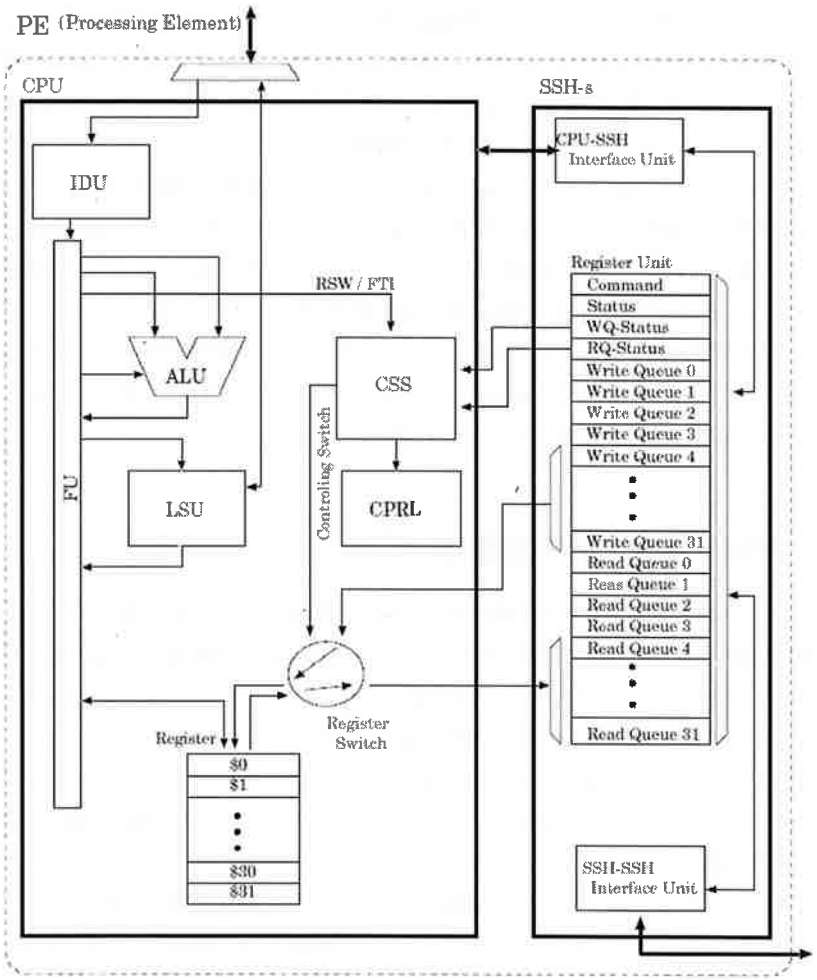


図 3.4: CSS のブロック図

ブラリへと制御が移る.
STEP7: スレッド・ライブラリにより Write Queue が空であるのを確認すると，それまで実行していたスレッドのコンテキスト情報以外の情報を Write Queue0-3 に書き込む.
STEP8: Write Queue0-3 へ書き込み後，スレッド・ライブラリは Read Queue に次に実行すべきスレッドの情報が書き込まれている事を確認し，Read Queue0-3 の情報を読み出し，CSS にコンテキスト情報の退避／復元を命令する.
STEP9: CSS はスレッド・ライブラリからの命令に従い，それまで実行されていたスレッドのコンテキスト情報を，Write Queue4-35 に書き込み，これから実行されるスレッドのコンテキスト情報を Read Queue4-35 から読み出し，コンテキスト情報の退避／復元を高速に行う．これにより Write Queue に全てのスレッド情報を書き終え，Read Queue は空になる.
STEP10: CPU 上では，スレッド・ライブラリが CSS の動作完了を確認すると，Read Queue から読み出したスレッドの実行を再開する．一方，SSH-s では Write Queue にデータが書き込まれたのを検知し，スケジューラ・バスの使用要求を出し，バスの使用権を待つ.
STEP11: スケジューラ・バス調停器よりバスの使用権を得る.
STEP12: スケジューラ・バスを介して，SSH-m に Write Queue のデータを送信する.
STEP13: SSH-s からスレッドの情報を受け取った SSH-m は，当該スレッドを再スケジューリングし，適切なキューへ追加する．一方，SSH-s は Read Queue が空であることを検知し，再びスケジューラ・バスの使用要求を出し，バスの使用権を得た後，次に実行すべきスレッドの情報を要求する.
SSH 及び CSS を用いたアーキテクチャでは，以上のような動作を繰り返し，高速にスレッドを実行していく.

3.6 SSH 及び CSS の問題点

しかしながら従来案手法である CSS では，コンテキスト・スイッチ処理が行われたあと，32 本すべてのレジスタをスケジューラバスを利用してコンテキスト専用のメモリに格納していた．その間，スケジューラ・バ

スをコンテキストの退避に占有されるため，PE 数が増加すれば，スレッドのスケジューリング時間にオーバヘッドがかかってしまうという問題がある．

4 コンテキスト・スイッチの転送データ圧縮手法の提案

4.1 手法の概要

SSH および CSS を用いることで、スケジューリングやコンテキスト・スイッチに起因するオーバヘッドが低減され，高速化が実現された．しかしながら，従来の CSS ではコンテキスト・データの退避にスケジューラバスを利用しているため，CPU の数が増加するとコンテキスト情報の送受信に時間がかかってしまう．そこで，提案手法ではコンテキスト・スイッチを行う際に，前のスレッドで使用されていないレジスタは送信せず，使用されたレジスタのみ送信することでコンテキスト・データを圧縮しスケジューリングオーバヘッドを低減させる手法を提案，実装する．

4.2 提案手法の構成

4.2.1 従来手法からの改造点

前章で述べたようにスケジューラバスを使用して全てのレジスタの退避処理を行うと，CPU の数が増えるにつれスケジューラバスの競合が大きくなる．そこで，本研究ではスケジューラバスの使用を抑えるために，SSH で扱うスレッド管理情報にスレッドの生成・復帰時点から中断されるまでの間に使用されたレジスタを区別するためのダーティビットの情報を付加することによりコンテキスト・スイッチに伴うレジスタの退避を選択的に行う手法を提案する．この手法では，コンテキスト・スイッチが発生すると，CPU はコンテキスト・データとダーティビットを SSH-s へ専用線を用いて 1 サイクルで送信する．SSH-s は受け取ったコンテキスト・データをダーティビットを元に必要なレジスタのみ SSH-m へ送信する．本手法を実現するためには，従来の SSH-m，SSH-s およびレジスタファイルに改造を加える必要がある．この改造に伴い，前スレッドで使用されたレジスタを記録するために，レジスタファイルにダーティビッ

トを追加する．また、SSH-s から SSH-m へコンテキスト・データを送信する構造も変化させる必要がある．改良した SSH・レジスタファイルを図 4.5 に示す．新しく改造されたものはドットで塗りつぶしてある箇所である．また，右上にはレジスタファイルの詳細を示す．

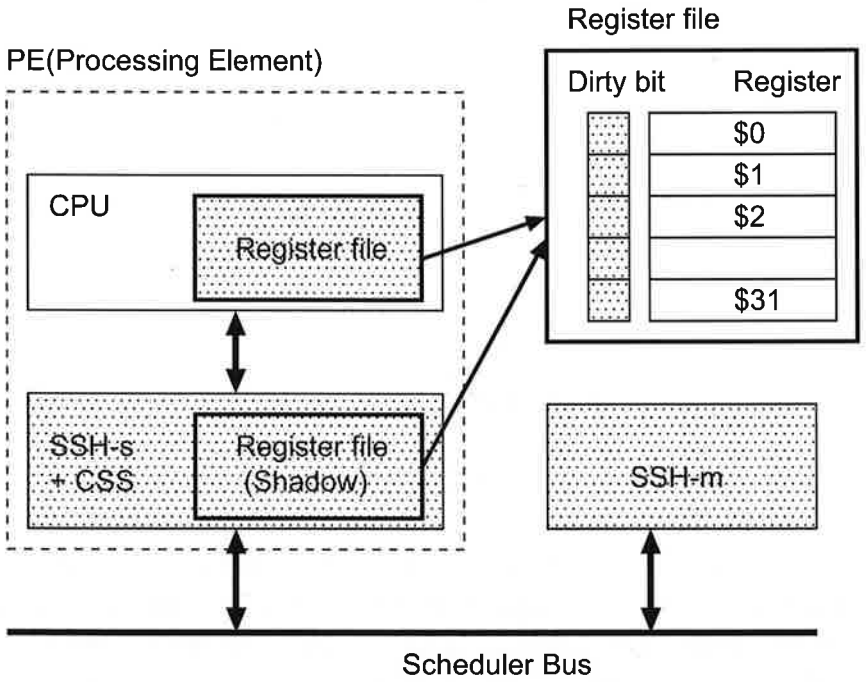


図 4.5: 改良された SSH ＋ CSS

4.2.2 動作

提案手法の詳細な動作を図 4.7 に示す．同図は，CPU でユーザ・スレッド T1 が実行され，実行が完了する前にタイムスライスを使い切って，次のスレッド T2 へ切り替わり，実行されたユーザ・スレッド T1 のコンテキストが退避される時の各ユニットの動作を示す．また、CSS は，図 4.6 の 3 つのステート・マシンに従って動作する．CSS は，図 4.6 の 3 つの状態からなっている．その 3 つの状態について述べる．

IDLE: CSS の初期状態．CSS がリセットされた時に，この状態から動作を開始する．SSH-s の RQ-Status が 1 になった場合，次の動作である

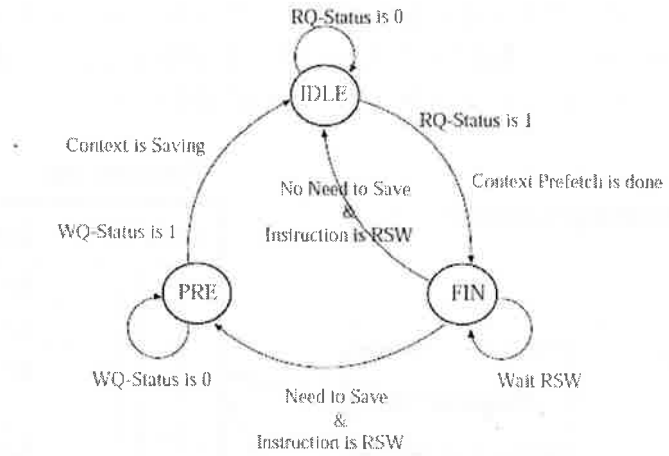


図 4.6: CSS のステートマシン

「FIN」に状態が遷移する．この時，SSH-s には次に実行するスレッドのコンテキストが各 Context Register に格納されている．FIN: プリフェッチ完了状態．RSW 命令発行後から FTI 命令が発行されるまでの間に，実行権限を譲ったスレッドの退避の必要性により状態が遷移する．FTI 命令発行後，退避が必要であれば「PRE」へ必要でなければ,「IDLE」へ状態遷移が行われる．PRE: 退避準備状態．WQ-Status が 1 になるまで繰り返される．WQ-Status が 1 になった場合,「IDLE」へ状態が遷移する．このとき，SSH-s から SSH-m へ前のスレッドで実行されていたコンテキストの情報を送信する．

STEP1: 初期状態 (a) では，CPU でスレッド T1 を実行しており，SSH は平行してスケジューリングを行っている．そして，SSH はスケジューリングを完了すると，SSH は SSH-m から SSH-s へ次に実行すべきスレッド情報を送信し (状態 (b))，最後に次のスレッドのコンテキスト・レジスタの値を Context ReadRegister へ送信し始める．SSH-s は Context Read Register の値がすべて受信し終わる (状態 (c)) と，RQ-Status の値を 1 にして状態 (d) となる．STEP2: CSS では，RQ-Status の値が 1 になったのを確認すると State を IDLE 状態から FIN 状態となる (状態 (d))．FIN 状態では，T1 が CPU の使用権限を譲渡，あるいはタイムスライスを使い切ると RSW が発行され，メイン・レジスタが Context Write Register へ，Context Read Register がメイン・レジスタへ入れ替えられメインレ

ジスタで実行していたスレッド情報がSSH-sのWrite Queueへ，CPUのダーティビットはSSH-sに用意されているダーティビット保持用のレジスタへ移され，コンテキスト・スイッチが行われる(状態(e))．その後，レジスタの退避の必要があればFIN状態からPRE状態へ(状態(f))，退避の必要がなければFIN状態からIDLE状態へ遷移する．ここでCPUのダーティビットはリセットされる．STEP3: 状態(f)では，SSHは，スケジューリングを行っている．CSSでは，WQStatusが1になるのを待つ．WQ-Statusが1になれば，SSH-sのWrite Queueの値とダーティビットの値をSSH-mへ送信を開始する(状態(g))．次にSSH-sはContext Write Registerの値をSSH-mへ送信を始める(状態(h))．CSSはSSH-mへCashe Write Registerの値を送信完了するとPRE状態からIDLE状態と遷移する(状態(i))．

5 性能評価

本章では，提案手法の有効性を示すためシミュレーションによる性能評価を行う．まず，一般的な並列処理プログラムをモデル化したものを取り上げ，従来手法を用いて実行した場合，提案手法を用いて実行した場合の比較評価を行い，提案手法の有効性を示す．以降本論文では，従来手法であるスケジューラ・バスを用いてコンテキストの退避・復帰処理を行うCSSを用いた方式を「CSS」，提案手法であるコンテキストの退避処理を選択的に行う方式を「NEW CSS」と示す．

5.1 評価環境

評価は，Verilog-HDLシミュレータを用いて行う．評価環境を表5.2に示す．シミュレーション時間を短縮する為，評価はビヘイビア・レベルの記述を用いて行う．また評価プログラムはC言語を用いて作成する．本研究の特徴の一つとして，一般的なスレッド・ライブラリの一つであるPthreadのAPIに準拠しているということがある．その為，本研究で使用した評価用プログラムは，Pthreadを実装している環境であればコンパイル，及び実行することが可能である．また，本研究で用いたCPUコアはMIPS R3000互換の命令セットを実装しているので，C言語を用いて作成した評価用プログラムをクロス開発環境を用いてDELL Optiplex

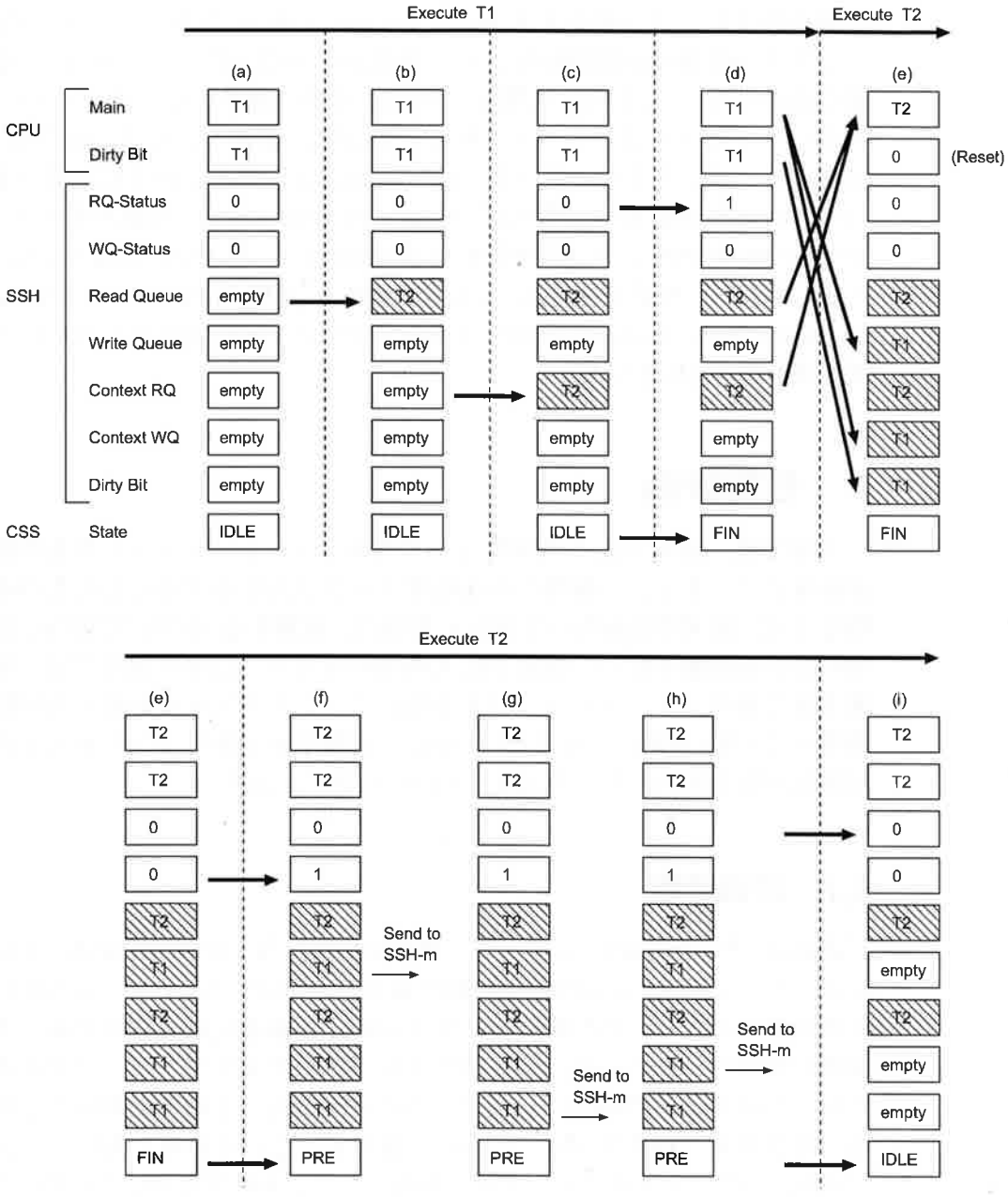


図 4.7: 提案手法の詳細な動作

GX260 上でコンパイルし，シミュレーションに利用する．表5.3 に使用したクロス開発環境を表す．

表 5.2: シミュレーション評価環境

CPU	MIPS R3000
Memory	512MB
Verilog-HDL Simulator	Synopsys Verilog Compiler Simulator 7.0

表 5.3: 使用したプログラム開発ツール

用途	ツール名	開発元
C compiler	GCC Ver 3.3.5	GNU
Assembler	GNU assembler Ver 2.15	GNU
Linker	GNU id Ver 2.15	GNU

5.2 対象とするマルチプロセッサ環境

マルチプロセッサ環境は，SSH を搭載したものを対象とする．PE は1〜16 台までの任意の台数に設定できる．共有メモリは全PE から等距離にあるUMA(Uniform Memory Access) モデルで，アクセス時間は均一である．Memory Bus の調停は，専用の Memory Bus Arbiter で行い，各PE の優先順位はラウンド・ロビンにより動的に決定される．バスの優先順位は全PE とも同一である．Scheduler Bus も Memory Bus と同様にラウンド・ロビンにより優先順位が決定されるが，SSH-m からの要求には最も高い順位の優先度が割り当てられている．命令メモリは共有メモリに置かれており，データ・バス，命令・バスはともに 32 ビットのアドレス／データ分離型である．

5.3 一般的並列処理モデルによる評価

5.3.1 評価内容

図5.8 に示すようなモデルを用いて各方式についてシミュレーション評価を行う．その際用いたパラメータは表5.4 のとおりである．図5.8 中

の大括弧内は，処理に要するサイクル数を示す．プログラムは初期化処理を行った後，スレッド Task() を *THR NUM* 個生成する．Task 関数の中では並列処理部分を 2 分割し，スレッド実行中に同期をとり，必ずスレッドの切替えが起こるようにした．このようにして，コンテキスト・スイッチに伴うレジスタ退避処理部分のオーバヘッドを考慮した評価を行う．Pthread には，スレッドを複数個同時に生成する API はない為，ループ文を用いて逐次的にスレッド生成を行う．親スレッドは，生成した順番に子スレッドと結合していき，全スレッドと結合した後,「終了処理」を行ってプログラムを完了する．本研究では並列処理部分のみの高速化に関する研究であるため，逐次処理部分，すなわち，*INIT*，*TERM* は 0 サイクルとする．処理に要するサイクル数の内，*CREATE* と *JOIN* は SSH，及びライブラリに依存する．本研究で実装したライブラリでは，表 5.5 に示す値となった．図 5.9 に評価プログラムのタスク・グラフを示す．

表 5.4: 評価パラメータ

パラメータ	意味
PE	利用できる PE 数
THR_NUM	生成するスレッド数
INIT	初期化処理に要するサイクル数
CREATE	スレッド一つ生成するのに要するサイクル数
JOIN	スレッド一つと結合するのに要するサイクル数
TERM	終了処理に要するサイクル数
THR_LOAD	Task() を逐次実行した場合に要するサイクル数

表 5.5: SSH における CREATE，JOIN のオーバヘッド

パラメータ	サイクル数
CREATE	1,028
JOIN	463

このモデルを用いて，PE 数，及びスレッド数に対する実行時間を計測する．なお，本評価では処理の実行時間を示す指標としてサイクル数を用いる．これは評価を実機ではなく，Verilog-HDL を用いたシミュレーションにより行っているためである．以降サイクル数と記述した場合に

int main(){
　　初期化処理 [$INIT$]
　　スレッド Task() 生成 [$CREATE \times THR\_NUM$]
　　スレッドの結合 [$JOIN \times THR\_NUM$]
　　終了処理 [$TERM$]
}

void Task(){
　　並列処理 [$\frac{THR\_LOAD}{THR\_NUM}$]
　　同期
　　並列処理 [$\frac{THR\_LOAD}{THR\_NUM}$]
　　同期
　　並列処理 [$\frac{THR\_LOAD}{THR\_NUM}$]
}

図 5.8: 並列処理モデル

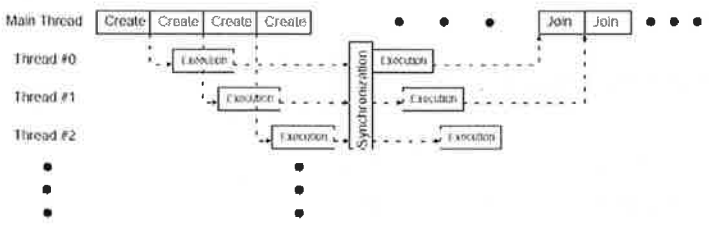
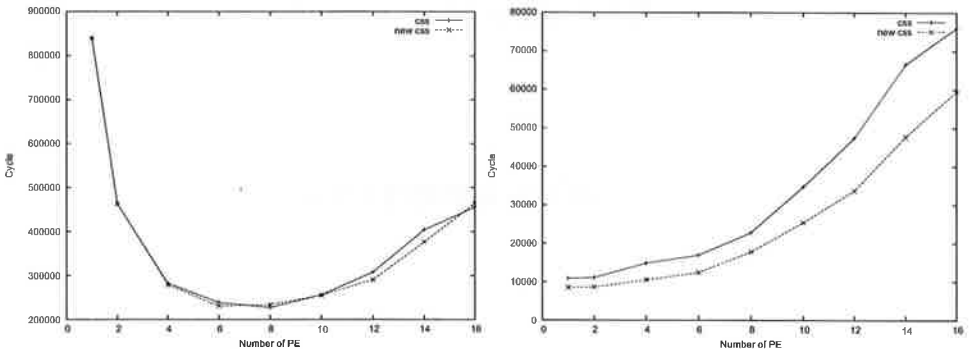


図 5.9: タスク・グラフ

は，プログラムの終了までにかかるサイクル数を表す．なお本研究では，マルチプロセッサ環境全体の設計，評価を Verilog-HDL を用いて RTL レベルで設計を行っているため，この値を動作周波数で割った値は，実機で動作させた場合の実行時間と等価である．

5.3.2 PE 数に対する処理時間の変化（台数効果）

本項では，スレッド数を 50 個に固定した上で，プログラムの様々な粒度に於ける台数効果を評価することで，各方式の比較評価を行う．図 5.3，図 5.4，図 5.5 にスレッド当りのサイクル数がそれぞれ 400，1,200 の場合の PE 数に対する処理時間の変化を示す．同図 (a) は処理に要したサイクル数を，(b) はコンテキスト・スイッチに要したサイクル数を示す．また同図 は横軸が PE 数，縦軸がサイクル数を表す．



(a) サイクル数　　(b) コンテキスト・スイッチのサイクル数

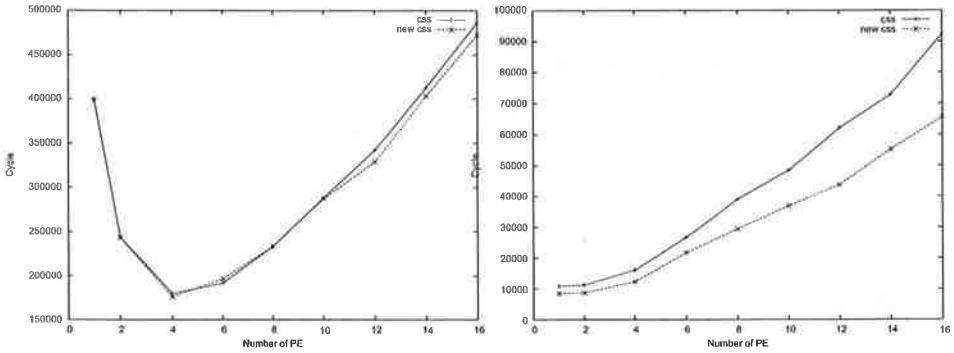
図 5.10: PE 数に対するサイクル数（スレッド当りのサイクル数 1,200）

中粒度の並列処理における評価結果

図 5.10 はスレッド当りのサイクル数が 1,200 の結果である．同図より，各方式とも PE 数が 6 台までは性能が上がっている．

細粒度の並列処理における評価結果

図 5.11 はプログラムの粒度をさらに細かくして，細粒度にしたときの処理時間の変化である．PE 数が 4 台と 12 台以降提案手法の方がサイクル数が少いのでコンテキスト・スイッチに起因するオーバヘッドが削減されたためだと考えられる．また，コンテキスト・スイッチのみのサイクル数で比較を行うと，提案手法では従来手法の平均 77 %に抑えること



(a) サイクル数　　(b) コンテキスト・スイッチのサイクル数

図 5.11: PE 数に対するサイクル数（スレッド当りのサイクル数 400）

ができた．PE 数が 4 台の時には，コンテキスト・スイッチのサイクル数が提案手法では従来手法の 76.6 %に抑えることが出来ている．

また図 5.10,5.11 ともコンテキスト・スイッチのサイクル数は PE 数にかかわらず提案手法のほうがサイクル数を削減できている．しかし，プログラムの処理に要したサイクル数では提案手法のほうが従来手法よりも劣る場合も確認できる．これは，最適なスケジューリングが行われておらず同期をとるタイミングがずれたため，CPU にタスクを効率的に送ることができなかったため CPU では，IDLE 時間が発生したと思われる．

6　結論

本研究では，スケジューリング支援ハードウェア (SSH) および、コンテキスト・スイッチ支援システム (CSS) を用いたマルチプロセッサ環境において，コンテキスト・スイッチにおける退避する転送データを圧縮することで，スケジューリング・オーバヘッドを低減し，中粒度並列処理をより有効に利用可能なシステムの構築を目指した．5 章で行った評価によると，一般的な並列処理モデル化したプログラムを用い，プロセッサ台数 12 台で並列処理を行った場合，コンテキスト・スイッチにおけるサイクル数は従来手法と比較して 30 %の性能向上を得ることができた．しかしながら，中粒度並列処理を行うにはまだ改良の余地がある．具体的には次の方法が考えられる．

コンテキスト退避の専用線の追加：コンテキスト・スイッチに伴うレジスタの退避専用線を追加し，スケジューラバスを使用せずにコンテキ

スト専用のメモリにコンテキストを直接格納できるようにする．

スレッドのパイプライン処理化:コンテキスト情報をいくつか保持し，スケジューラバスが使われていない時のみコンテキストを退避させる．使用されているときは退避させず保持されているコンテキストを再スケジューリングしたあとコンテキスト・スイッチを行うようにする．

謝辞

本研究の機会を与えて頂き，ご指導頂いた近藤利夫教授に深甚なる謝意を表します．コンピュータ・アーキテクチャという非常に興味深い分野へ導いて頂き，また常にご指導頂いた大野和彦講師、佐々木敬泰助手に深く感謝いたします．

参考文献

[1] 村松 満雄, マルチプロセッサ環境におけるコンテキスト・スイッチ支援システムの設計と評価, 三重大学工学部情報工学科卒業論文, 2004.