

ハードウェアスケジューラによる
Linux 上での近細粒度並列処理の
高速化

平成 18 年 度

三重大学大学院工学研究科
博士前期課程 情報工学専攻

大 原 一 輝

修士論文

題目

ハードウェアスケジューラによる
Linux 上での近細粒度並列処理の
高速化

指導教員

近藤 利夫 教授



2007 年

三重大学 大学院 工学研究科
博士前期課程 情報工学専攻
計算機アーキテクチャ研究室

大原 一輝(405M506)

内容梗概

近年, SMT(Simultaneous Multi Threading)やCMP(Chip Multi-Processor)プロセッサが高性能計算機だけでなく, 一般的なPCや組込み用途まで幅広く利用されるようになりつつある. ひとつのコンピュータシステムに搭載されるプロセッサコア数は今後さらに増加してゆくものと考えられる. そのような状況において, ソフトウェア開発の現場では, あらかじめプロセッサコア数が与えられなくても多数のコアを有効活用できることが強く求められる. 我々はそのような状況に際し, 近細粒度並列処理が有効と考える. 近細粒度並列処理では数千サイクル程度で終了するスレッドをプロセッサコア数よりも圧倒的に多く生成し, 実行可能なものからプロセッサコアに割り当ててゆく. これにより, アイドル状態になるプロセッサコアが発生することを防ぎ, 多数のプロセッサコアを有効活用する.

一方で, 近年, PCのみならず, スーパーコンピュータから組込みの分野までLinuxが広く普及しつつある. しかしながら, Linuxは近細粒度処理を前提として設計されていないため, 近細粒度並列処理を行った場合, タスク切り替えが頻繁し, スケジューリングを含むコンテキストスイッチのオーバーヘッドが問題となる. そこで, 本研究では佐々木らが提案しているハードウェアスケジューラSSH(Scheduling Support Hardware)をLinuxに適用することで, Linux上での近細粒度並列処理の性能を改善し, 多数のプロセッサコアを効率的に利用できる環境を実現することを目指す.

しかし, SSHはLinuxで使用することを前提に設計されたわけではないため, そのままではLinux上で利用できない. そこで, SSH, Linuxの両方を改良することで, LinuxがSSHを利用してスケジューリングを行えるようにする.

SSHはCPUと同じチップに実装することを想定しているため, PCを用いて評価を行うことはできない. そこで, SSHを組み込んだCPUを持つ評価用の計算機をVerilog HDLを用いて独自に開発し, そこへLinuxを移植して評価を行う. その結果, スケジューリング処理はSSHを使用しない場合の55~80%程度のサイクル数で行える様になったが, スケジューリング処理自体が多発し, 評価プログラムの実行は遅くなる場合があることがわかった.

Abstract

Recent years, SMT(Simultaneous Multi Threading) or CMP(Chip Multi-Processor) processors are used in not only high performance computing but also general personal computers or embedded computers. In the near future, most one computer system will have many processor cores.

Therefore, it is required for software development engineers to utilize many cores efficiently. As one method to use many processor cores efficiently, the fine grained parallel processing is promising. It divides a program into a number of threads more than the number of processor cores. And task scheduler dispatches threads to processor cores (or processing elements) from those in the runnable state. In this way, it prevents cores from being the idle state and uses many cores efficiently.

On the other side, these years, not only in PC field but also from embedded computers to super computers, Linux is being popular. But, Linux is not designed for the fine grained parallel processing, when it is used for the fine grained parallel processing, the overheads of context switches and scheduling become serious problems. So, this paper improves the fine grained parallel processing performance of Linux by using SSH(Scheduling Support Hardware) proposed by Sasaki et al, and tries to realize the environment in which many cores work efficiently.

However the SSH is designed without assuming to cooperate with Linux. In order to operate both the SSH and Linux cooperatively, this paper propose the method to modify them.

Because the SSH is assumed to be implemented in a same chip of the CPU, it is impossible to evaluate the performance of them using general personal computers. Therefore this paper designs the system, multiprocessor environment using SSH with Verilog HDL in order to evaluate its performance and ports Linux. According to the results of the evaluation, the cycles of the scheduling can save from 20% to 50%. But the scheduling occurs many times, the execution of the evaluation program itself take longer.

目次

1	はじめに	1
2	背景と提案手法	3
2.1	SMT プロセッサ, CMP の普及	3
2.2	Linux の台頭	5
2.3	Linux 上での近細粒度並列処理を行う場合の問題点と解決方法	7
3	SSH	8
3.1	SSH を利用したマルチプロセッサアーキテクチャ	8
3.2	SSH の内部アーキテクチャ	9
3.2.1	SSH-m	10
3.2.2	SSH-s	10
3.3	SSH の動作	12
4	SSH の Linux への適用	14
4.1	Linux のスレッドの扱いについて	14
4.2	SSH を Linux から利用する際の問題点	14
4.3	提案手法	15
4.3.1	Wait Queue を用いる手法	15
4.3.2	ソフトウェア待ち行列を用いる手法	16
4.3.3	Ready Queue を用いる手法	16
4.3.4	SSH を改良する手法	17
4.4	解決手法の決定	18
5	SSH の改良	19
5.1	改良型 SSH の動作	19
5.2	改良型 SSH の仕様	20
6	Linux の改造	20
6.1	Linux のスケジューリング	21
6.1.1	プロセスの状態遷移	21
6.1.2	runqueue の構造	23
6.2	SSH と runqueue の違い	25
6.2.1	Active/Expired Queue の数	25

6.2.2	Active Expired Queue のエントリ数	25
6.2.3	保持可能なプロセス情報の数	26
6.2.4	プロセス情報を取得した際の動作	27
6.2.5	プロセス情報の削除	27
6.3	SSH を利用するスケジューラの設計	28
7	性能評価	34
7.1	評価環境	34
7.2	スレッド数を変化させた場合の変化	35
7.3	スレッドの粒度を変化させた場合	38
8	関連研究	40
9	おわりに	42
	謝辞	44
	参考文献	44

目 次

3.1	マルチプロセッサアーキテクチャ	9
3.2	SSH-m	11
3.3	SSH-s	12
5.4	改良型 SSH-m	20
6.5	プロセスの状態遷移	22
6.6	runqueue の構造	24
6.7	SSH 実装でのプロセスの状態遷移	31
7.8	スレッド数を変化させた場合の平均スケジューリングサイ クル数の変化	36
7.9	スレッド数を変化させた場合の評価プログラムの実行に要 するサイクル数の変化	37
7.10	スレッド数を変化させた場合の評価プログラムの実行にサ イクル数の変化 (正規化したもの)	37
7.11	スレッド数を変化させた場合のスケジューリング回数の変化	38
7.12	粒度を変化させた場合の平均スケジューリングサイクル数 の変化	39
7.13	粒度を変化させた場合の評価プログラムの実行に要するサ イクル数の変化	39
7.14	粒度を変化させた場合の評価プログラムの実行に要するサ イクル数の変化 (正規化したもの)	40
7.15	粒度を変化させた場合のスケジューリング回数の変化	41

表 目 次

5.1	改良型SSHの仕様	21
6.2	プロセスの状態	21
6.3	新たに導入したフラグ	28
7.4	評価環境の諸元	35

1 はじめに

近年，SMT(Simultaneous Multi Threading) プロセッサや CMP(Chip Multi-Processor) が高性能計算機の分野だけでなく，PC や組込みの分野でも利用されるようになりつつある．例えば PC 用には 2 または 4 つのプロセッサコアを持つ Core2 プロセッサが Intel から出荷されている．また，組込みの分野では SONY, IBM, 東芝による Cell プロセッサが出荷されている．これはデータ処理用の 8 つのプロセッサコアと制御用のプロセッサコアを一つ備える．Cell プロセッサは家電用にデータ処理用のプロセッサを 4 つに減らしたものの計画されている．このように同じアーキテクチャでも価格や求められる性能によりプロセッサコア数が異なる製品が出荷される．今後，製品によるプロセッサコア数の違いはさらに大きくなっていくものと考えられる．

このような状況においてはソフトウェアの開発時点でシステムに搭載されているプロセッサコア数が与えられていなくても多数のコアを有効活用できることが求められる．そして，この傾向は高性能計算機やパソコンだけでなく，高性能化，高機能化の進むデジタル家電やモバイル端末，各種制御装置などの組込用途でも同様である．

この問題に対し，我々は近細粒度並列処理が有効であると考えている．

近細粒度並列処理では数百から数千サイクルで完了するスレッドをプロセッサコアの数よりも圧倒的に多く生成する。そして、これらを実行条件の整ったものからプロセッサに割り当て実行することで、アイドル状態になるプロセッサが発生することを防ぎ、多数のプロセッサコアを有効活用する。

一方で高性能計算機による科学技術計算から組込み用途まで幅広く利用できる OS として Linux が着目されている。しかしながら、Linux は近細粒度並列処理を前提として設計されてはいない。このため、Linux 上で近細粒度並列処理を行うとスレッドの生成と破棄、スケジューリング、コンテキストスイッチが多発し、これらのオーバーヘッドが大きくなる。

そこで、本研究ではこれらのうち、スケジューリングのオーバーヘッドによりハードウェアの性能を十分に引き出せないという問題がある。そこで本研究では、これらのうち、スケジューリングのオーバーヘッドに着目し、佐々木らが提案しているハードウェアスケジューラである SSH(Scheduling Support Hardware)[1, 2] を Linux に適用することで、この問題を解決し、多数のプロセッサコアを効率的に利用できる環境を実現することを目指す。

2 背景と提案手法

2.1 SMT プロセッサ, CMP の普及

従来, マイクロプロセッサは単一のプログラムから同時に実行可能な命令をできる限り多く見つけ出し (ILP¹の抽出), 多くのキャッシュメモリを積み, 高い動作周波数で高速に実行する方向に進化してきた.

ILP を向上させる過程で, マイクロプロセッサは多くの命令を同時に実行できるよう多数の演算器を備えるようになった. しかし, 同時に実行できる命令には限度があるため, 常に演算器と同じ数の命令を実行することはできず, 演算器を遊ばせてしまうことが多い.

この遊んでいる演算器を有効活用するため, SMT(Simultaneous Multithreading) プロセッサが考案された. SMT プロセッサは一つのプロセッサに複数のレジスタセットを用意し, 演算器を共有する複数台の論理 CPU として動作する. SMT プロセッサは複数の命令列を実行するので同時に実行可能な命令を非 SMT プロセッサよりも多く提供することができ, 演算器の稼働率を上げることができる.

また, 今日, プロセッサと DRAM の速度差は非常に大きくなり, キャッシュミスが発生した際には DRAM アクセスに数百サイクルを要するよう

¹Instruction Level Parallelism, 命令レベルの並列性

になった。非 SMT プロセッサではロード命令を発効した際、依存関係のある後続命令は実行することができずにパイプラインを長時間ストールさせ、演算器を遊ばせることになる²。一方、SMT プロセッサでは、この間も他の論理プロセッサの命令列を実行することができるため、演算器を遊ばせずに有効活用できる。

また、これに加え、プロセスの微細化によって顕著になったリーク電流の問題により、マイクロプロセッサの消費電力はかつて無く上昇し、これ以上の上昇は許容できない状況もマルチコア化への追い風となっている。

経験則³ではトランジスタ数を 2 倍にしても性能は $\sqrt{2}$ 倍にしか上昇しない。つまり、消費電力を 2 倍にしても性能は $\sqrt{2}$ 倍にしかない。しかし、これは逆にトランジスタ数 (消費電力) を半分にしても性能は $1/\sqrt{2}$ 倍にしか落ちないことになる。そこで従来の半分のトランジスタ数で CPU を 2 器作り、一つのチップに載せれば、消費電力は変わらないまま従来の $(1/\sqrt{2}) \times 2 = \sqrt{2}$ 倍のピーク性能を持つプロセッサを作ることができる。このような考えに基づき、複数の CPU を一つのチップに載せた CMP (Chip Multi Processor) が普及し始めている。

²アウトオブオーダー実行やノンブロッキングキャッシュを用いることでパイプラインストールを減らす技術はあるが、逐次プログラムの並列性の問題外から本質的な解決にはなっていない。

³ポラックの法則

今後は、64 の論理 CPU を搭載する Sun microsystems の UltraSPARC T2[15] や Intel が IDF Fall 2006 で公開した 80 器の CPU を搭載した CMP プロセッサ [16] にその方向性が見られるように、さらに多くの CPU を搭載したプロセッサが製品化されると思われる。

また、CMP は消費電力を押さえつつ、性能向上を期待できるため、組込の分野でも普及し始めている。代表的なものとしては SONY, IBM, 東芝による Cell[17] がある。Cell は一つの制御用汎用プロセッサ (これは SMT により 2 論理 CPU として動作する) と 8 つの信号・メディア処理用のプロセッサを搭載する。

2.2 Linux の台頭

一方で、OS の分野に目を向けてみると PC 用以外の分野では Linux の台頭が著しい。

Linux は 1991 年にヘルシンキ大学の学生であった Linus Torvals により趣味として開発が始まった。当初の目的は 386 を搭載した PC をターゲットにフリーの UNIX を開発することであった。当時、フリーの UNIX 環境の実現を目指していた GNU プロジェクトはコンパイラ、エディタ、シェルなどのアプリケーションソフトウェアを完成させていた。これらが移

植され、Linux を用いて実用的な UNIX 環境が構築できるようになった。現在では X Window System が動作し、洗練された GUI を持つ KDE や GNOME によるデスクトップ環境が実現されている。

PC で使うことを目的に開発が始まった Linux であったが、現在はデスクトップ用途よりも、その安定性、UNIX マシンの運用ノウハウを利用できること等から従来、Solaris, Tru64 UNIX 等の商用 UNIX OS の市場であったワークステーション、サーバー分野で広く使用されている。

また、組込みの分野では従来、独自開発のリアルタイム OS や、ITRON, Windows CE 等が用いられてきたが、Linux はロイヤリティが発生しないこと、オープンソースであるため高いカスタマイズ性を持つことから組込み OS としても広く利用されている。もともと PC 用 OS であったため、高機能である反面、しばしば組込み分野で求められるリアルタイム性の弱さが指摘されている。

スーパーコンピュータも Linux が広く用いられている分野である。現在では圧倒的なシェアを占め、上位 500 システムの内、82%⁴が Linux を利用する。

⁴<http://www.top500.org/stats/28/os/>, 2007 年 2 月 18 日現在

2.3 Linux 上での近細粒度並列処理を行う場合の問題点と解決方法

前述の通り、近年のマイクロプロセッサは多CPU化の流れにあり、Linux が利用される多くの分野もその例外ではない。このため、Linux も多くの CPU を有効活用できることが今後よりいっそう求められる。そのためには近細粒度並列処理が有効と考える。近細粒度並列処理では、プログラムを数千サイクル程度で終了する CPU 数よりも非常に多くのスレッドに分割する。そして、その中から実行条件の整ったものを CPU に割り当ててゆくことで、CPU がアイドル状態になることを防ぎ、全ての CPU を有効活用する。

しかし、近細粒度並列処理ではスレッドを多数生成するため、スレッドの生成、破棄、スケジューリング、コンテキストスイッチのオーバーヘッドが大きくなる。

スケジューリングのオーバーヘッドについては佐々木らがSSH(Scheduleing Support Hardware)を提案している。これはスケジューリング処理を行う専用のハードウェアを用意し、ユーザープログラムの実行と並行してスケジューリングを行うことで、CPU がスケジューリング処理に要するサイクル数を削減するものである。

そこで、本研究では、この SSH を利用して Linux のスケジューリング性能を改善する。

3 SSH

本章では提案している手法の説明に先立ち、まず本論文で利用する SSH について概括する。

3.1 SSH を利用したマルチプロセッサアーキテクチャ

SSH は集中、または分散共有メモリ型のマルチプロセッサ環境において、スレッドのスケジューリングをハードウェアで行うものである。SSH を用いたマルチプロセッサアーキテクチャを図 3.1 に示す。アーキテクチャは、CPU と SSH-s を含む複数の PE(Processing Element)、スケジューリングを行う SSH-m、SSH-s と SSH-m を結ぶスケジューラバス、スケジューラバス調停機、メモリバス調停機、共有メモリから成る。SSH-s は各 CPU と SSH-m とのインター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するスレッド情報の取得、新しいスレッド情報の送信等を行う。CPU と SSH-s はオンチップでの実装を想定しており、メモリマップド IO で接続する。

スレッドのスケジューリングは SSH-m で行う。SSH-m は SSH-s を介

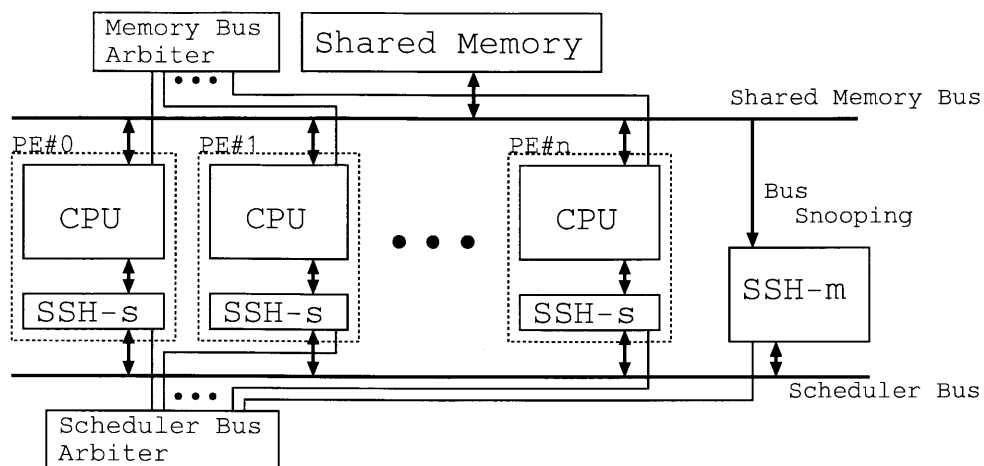


図 3.1: マルチプロセッサアーキテクチャ

して CPU から送られて来たスレッド情報を管理し，優先度に従ってスケジューリングを行う．SSH-s から要求があると，もっとも優先度の高いスレッド情報を SSH-s に返す．

SSH-m と SSH-s は調停器を備えた専用のスケジューラバスによって接続される．この調停器は，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 の内部アーキテクチャ

SSH はスケジューリングなどの処理を行う SSH-m と CPU と SSH-m のインターフェースを提供する SSH-s から構成される，SSH はクライアント・サーバーモデルになっており，SSH-m がサーバー，SSH-s がクライ

アントとして動作する．本節ではこれらの詳細なアーキテクチャについて述べる．

3.2.1 SSH-m

図 3.2 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 とスレッド情報を保持する Global Queue から成る．SSH はスレッドごとに 4 ワードのスレッド情報を保持する．この内 1 ワードは優先度やスレッドの状態等，SSH がスケジューリングを行うための情報を保持し，残り 3 ワードは OS やスレッドライブラリがコンテキスト情報へのポインタ等の任意の情報を保持させることができる．Global Queue は優先度ごとに実行可能状態のスレッドを保持する Ready Queue，待ち状態のスレッドを保持する Wait Queue を SRAM を用いて実装する．

Hardware Scheduler は SSH-s からのコマンドに応じて Global Queue へスレッド情報のエンキュー，デキューを行う．

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 のインターフェースを行う CPU-SSH Interface Unit，スケジューラバスを介し

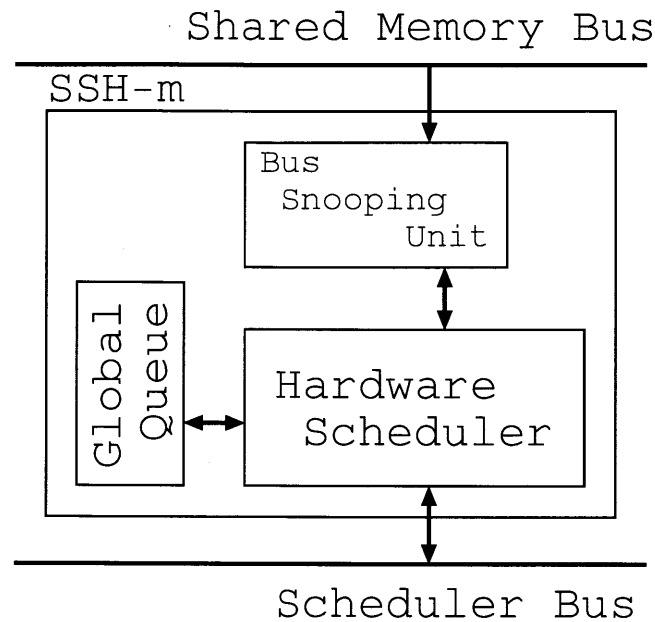


図 3.2: SSH-m

て SSH-m と通信を行う SSH-SSH Interface Unit, SSH-m から取得したスレッド情報及び, SSH-m に送信するスレッド情報を保持するための Register Unit から成る. Register Unit は SSH-s および, SSH-m へのコマンドや, 次に実行すべきスレッド情報を格納する Read Queue, 新規に作成したスレッド, あるいは実行権限を手放したスレッド情報を SSH-m が送信されるまで保持しておく Write Queue から成る.

CPU-SSH Interface Unit は CPU が発行したロード・ストア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.

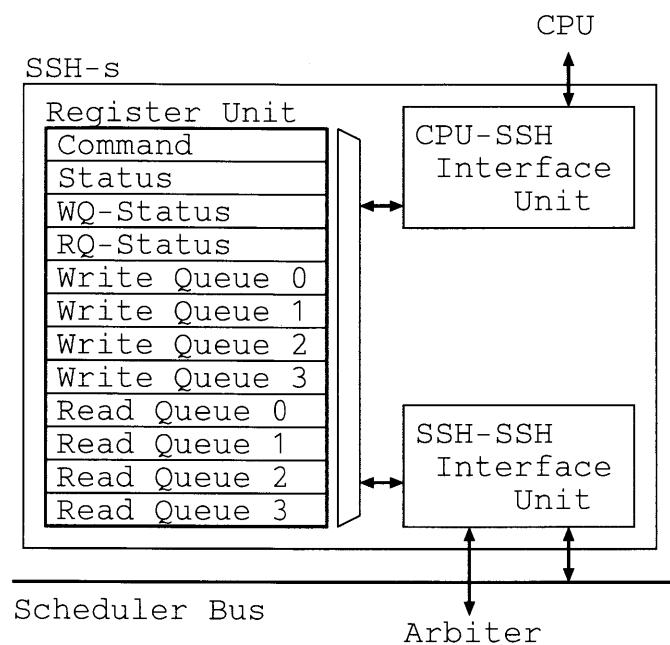


図 3.3: SSH-s

SSH-SSH Interface Unit は 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情報を取得する．また，Write Queue に有効なスレッド情報が入ると SSH-m に送信する．

3.3 SSH の動作

SSH の動作例を示す．初期条件として，実行開始から十分に時間が経っており，Global Queue には既に実行待ちのスレッドがあり，ある SSH-s の Read Queue, Write Queue は共に空であるとする．ユーザープログラムがタイムスライスを使い切り，別のスレッドにコンテキストスイッチ

する際の動作を示す.

STEP1: CPU ではユーザプログラムが動作している. これに並行して Read Queue が空であることを検出した SSH-s はスケジューラバスの使用权を確保し, SSH-m に次に実行すべきスレッド情報を要求する.

STEP2: SSH-m は Global Queue の中で最も優先度が高いスレッド情報を SSH-s に返す.

STEP3: SSH-s はスレッド情報を受け取り, Read Queue に格納する.

STEP4: OS はスレッドがタイムスライスを使い切ったことを検出すると Write Queue に現在のスレッド情報を書き込む.

STEP5: SSH-s は Write Queue にデータが書き込まれたのを検出し, バスの使用权を取得後, Write Queue のデータを SSH-m に送信する. OS は Read Queue から次に実行するスレッド情報を取得し, コンテキストスイッチを行う.

STEP6: SSH-s からスレッド情報を受け取った SSH-m は, これを優先度別に適切なキューへ追加する. (以後, 再び STEP1 からの動作を繰り返す.)

4 SSH の Linux への適用

本章では、Linux でのスレッドの扱いについて説明した後、SSH を Linux から利用する際の問題点を挙げ、その解決方法を示す。

4.1 Linux のスレッドの扱いについて

Linux ではマルチスレッドのプログラムは仮想メモリ空間や、開いているファイルなどを共有するプロセスの集合として実装される。このため、各スレッドは Linux からはプロセスとして扱われ、スケジューリングもプロセスの単位で行われる。

4.2 SSH を Linux から利用する際の問題点

SSH は Linux を前提に設計された訳では無いため、Linux に適用した際にいくつか問題となる点がある。その中でも特に深刻な問題は、タイムスライスを使い切ったプロセスの扱いの違いである。

Linux では一度 CPU を割り当てられたプロセスは I/O 資源などの待ち状態になるか、タイムスライスを使い切るか、自ら CPU を他のプロセスに譲るまで実行され続ける。Linux はあるプロセスがタイムスライスを使い切ると、その他の全てのプロセスもタイムスライスを使い切るまで再

びCPUを割り当てられることはない。この仕組みにより、優先度の高いCPUを占有してしまうことを防いでいる。SSHにはこのタイムスライスを使い切ったプロセスを保持するためのキューは用意されていないので、その扱いを検討する必要がある。

4.3 提案手法

本節では、上記問題の解決手法についていくつか検討する。

4.3.1 Wait Queue を用いる手法

SSHのWait QueueはI/O待ち状態等、実行を中断しているプロセスの情報を保持するためのキューであるが、Linuxではこのようなプロセス情報はソフトウェアで実現した待ち行列で管理しているため、このWait Queueは使用しない。そこで、タイムスライスを使い切ったプロセス情報はWait Queueに格納し、全てのプロセスがタイムスライスを使い切った時点で、Wait Queue内のプロセス情報をReady Queueに移動させるという方法が考えられる。しかし、この手法では、全てのプロセス情報がWait Queueに格納される状態が発生するため、十分な大きさを取らなければならない。また、Wait Queue内のプロセス情報をReady Queueに移動するという操作はプロセス数に比例した時間がかかってしまう。

4.3.2 ソフトウェア待ち行列を用いる手法

Wait Queue に入れる手法とほぼ同じだが，SSH の Wait Queue に入れるのではなく，ソフトウェアで実現した待ち行列に格納する方法である．この方法であれば待ち行列が大きくなってしまいうことにも対応できるが，待ち行列内のプロセス情報を Global Queue に移動させるために要するサイクル数は Wait Queue を使用する場合の数十倍になってしまう．

4.3.3 Ready Queue を用いる手法

この手法ではあるプロセスがタイムスライスを使い切ると，タイムスライスを割り当て直し，そのプロセス情報がそれまで入っていた Ready Queue よりも優先度が一つ下の Ready Queue に格納する．この操作を繰り返すとやがてそのプロセス情報は優先度が最低の Ready Queue に格納される．その後はまた元の優先度の Ready Queue に格納する．この手法ではタイムスライスを使い切ったプロセス情報を格納する行列は必要ない．また，その行列から Ready Queue に格納し直す操作も必要ないため，処理コストの面でも有利である．優先度の高いプロセスほど，多くの CPU 時間が割り当てられるという性質も持たせることができる．

しかし，この手法では各プロセスに割り当てられる CPU 時間のバラン

スが大きく崩れてしまう。Linux では各プロセスに割り当てられる CPU 時間の比は優先度に比例するが、この手法では二乗に比例した CPU 時間が割り当てられてしまうことになる。

4.3.4 SSH を改良する手法

この手法は SSH を改良し、Linux のスケジューラが採用するタイムスライスを使い切ったプロセスの管理方法をそのままハードウェア化する [20]。すなわち、SSH-m に Global Queue を 2 セット用意し、片方をタイムスライスが残っているプロセス情報を保持するキュー (Active Queue)、もう片方をタイムスライスを使い切ったプロセスを保持するキュー (Expired Queue) とする。SSH-s に送信するプロセス情報の選択は Active Queue から行う。そして、タイムスライスを使い切ったプロセスには再びタイムスライスを割り当てた後、Expired Queue に格納する。この操作を繰り返すとやがて、Active Queue は全て空になり、Expired Queue にだけプロセス情報が格納されているという状態になる。SSH-m はこれを検出し、2 つのキューの役割を交代させる。

これにより、各プロセスに定期的に CPU が割り当てられることが保証される。そして、各プロセスにタイムスライスが割り当てられる回数が等しくなり、割り当てられる CPU 時間のバランスが崩れることを防ぐ。ま

た、従来の SSH では、全スレッドが待ち状態になっても Wait Queue があふれないよう Ready Queue 一本の 3 倍の大きさを確保していた。一方、Linux では、I/O 待ちなどのプロセス情報はソフトウェアで管理するため Wait Queue は必要ない。このため、従来の Wait Queue が使用していたハードウェア資源を Active Queue に割当て、さらに Active Queue のエントリ数を従来の Ready Queue よりも減らすことで、ハードウェア量を増やすことなく Linux スケジューラを実現できる。

4.4 解決手法の決定

Linux では、バージョン 2.6 から現在実行可能なプロセス数に依らず、定数時間でスケジューリング処理を行える $O(1)$ スケジューラが採用されている。しかし、前述の Wait Queue を用いる手法、ソフトウェア待ち行列を用いる手法では、プロセス情報を Wait Queue または待ち行列から Ready Queue に移動される操作がプロセス数に比例した処理時間がかかり、 $O(1)$ スケジューラに劣る部分となってしまう。

また、Ready Queue を用いる手法では各プロセスに割り当てらる CPU 時間の比が各プロセスの優先度の 2 乗の比に比例したものになってしまう。しかし、これは本来 1 乗に比例するものである。

SSH を改良する手法では SSH に手を加える必要があるものの，その他の手法にみられるような欠点はない．そのため，本研究ではこの手法を採用する．

5 SSH の改良

5.1 改良型 SSH の動作

改良型 SSH では SSH-m が Active Queue と Expired Queue の二つの優先度別のキューを持つ．SSH-m の構造を図 5.4 に示す．改良型 SSH ではタイムスライスを使い切ったか否かという情報と共に SSH-s からスレッド情報が送れてくる．SSH-m はこれにより，タイムスライスが残っているものは Active Queue，使い切ったものは Expired Queue の該当する優先度のキューに挿入する．一方，SSH-s からスレッド情報の要求があるとき，Ready Queue の空でない最も優先度が高いキューの先頭からスレッド情報を取り出し，要求した SSH-s に返す．

この動作を繰り返すとやがて Active Queue の全てのキューが空になる．SSH-m はこれを検出し，Active Queue と Expired Queue の役割を交代させる．

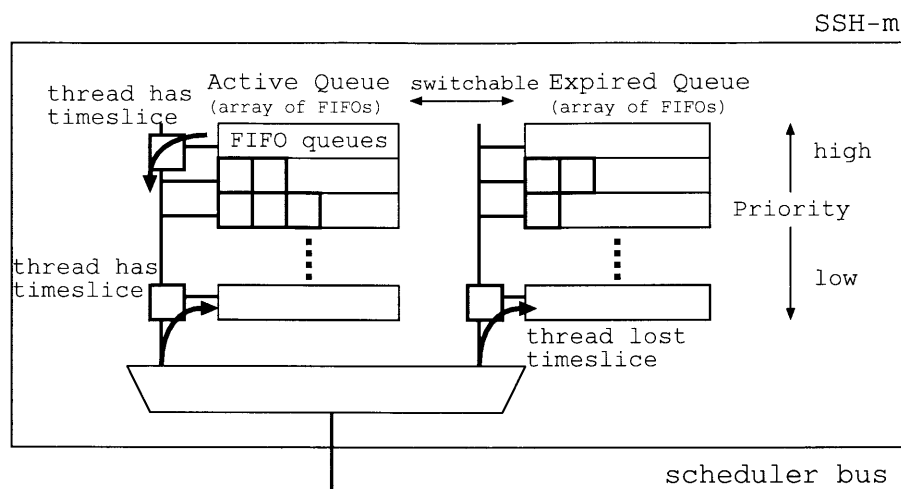


図 5.4: 改良型 SSH-m

5.2 改良型 SSH の仕様

改良型 SSH を Verilog HDL を用いて実装した。その仕様を表 5.2 に示す。

Active/Expired Queue は単純な FIFO として実装した。Active/Expired Queue は一本当たり最大 4096 のスレッド情報を保持することができ、SSH 全体では 65536 スレッドまで保持することが可能である。これらの上限を超えた際の動作は、SSH を利用する OS, スレッドライブラリ依存となる。

6 Linux の改造

本章ではまず、Linux のスケジューリングについて説明し、その SSH を利用してスケジューリングを行うために、どのような改造を行ったか説

表 5.1: 改良型 SSH の仕様

Active/Expired Queue 本数	それぞれ 16 本
Active/Expired Queue 一本当たりの最大プロセス情報 1 プロセス当たりのデータ量	4096 16 バイト (4 ワード)
Active/Expired Queue 用 SRAM 容量	2MB

表 6.2: プロセスの状態

値	意味
TASK_RUNNING	実行可能状態 (実行状態, 実行待ち状態)
TASK_INTERRUPTIBLE	待機状態. シグナルによる待機状態解除可能
TASK_UNINTERRUPTIBLE	待機状態. シグナルによる待機状態解除不可

明する. 以下, ソフトウェアで実現された Linux の標準のスケジューラ
のことをソフトウェア実装, SSH を利用してスケジューリングを行うよ
う改造した Linux のスケジューラのことを SSH 実装と呼ぶ.

6.1 Linux のスケジューリング

6.1.1 プロセスの状態遷移

Linux のプロセスのデータ構造である, `task_struct` 構造体には `state` メ
ンバがあり, これが各プロセスの状態を表す. `state` の主な値を 6.1.1 に
示す.

TASK_RUNNING は実行可能状態であり, スケジューラは TASK_RUNNING

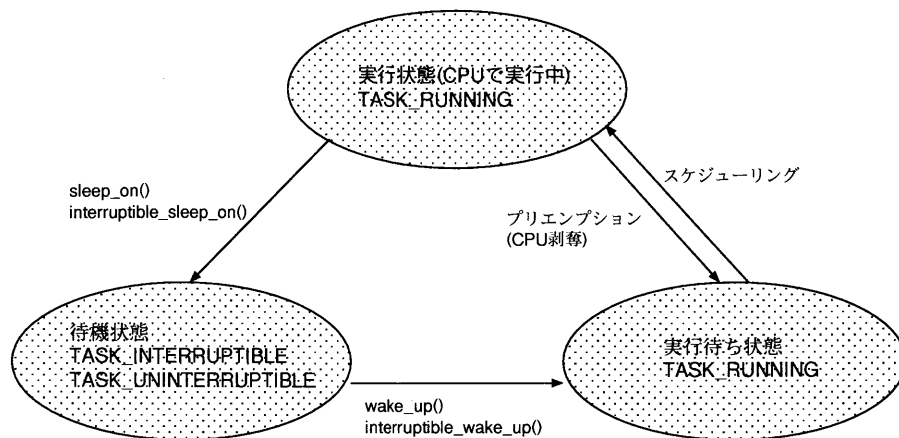


図 6.5: プロセスの状態遷移

状態であるプロセスの中から次に実行すべきスレッドを決定し，コンテキストスイッチを行う．コンテキストスイッチにより CPU を奪われたプロセス（タイムスライスを使い切ったプロセスも含まれる）も I/O 待ち等の待ち状態になるまでは TASK_RUNNING のままである．プロセスは `sleep_on()`, `interruptible_sleep_on()` 関数で，それぞれ TASK_UNINTERRUPTIBLE, TASK_INTERRUPTIBLE に遷移する．これらの違いは TASK_INTERRUPTIBLE がシグナルの受信により起床するのに対し，TASK_UNINTERRUPTIBLE は `wake_up()` 関数で起床されるまでシグナル処理を保留し続ける．

これらの動作を図 6.5 に示す．

6.1.2 runqueue の構造

ソフトウェア実装では CPU ごとに runqueue というデータ構造を持ち、TASK_RUNNING 状態のプロセスを管理している。その構造を図 6.6 に示す。runqueue は Active Queue と Ready Queue から成り、それぞれポインタの配列になっている。Active Queue の要素は優先度が同じでまだタイムスライスを使い切っていないプロセスで作ったリンクリストの先頭を指しており、Ready Queue の要素はタイムスライスを使い切ったプロセスのリンクリストの先頭を指す。

スケジューラは次に実行するスレッドを決定する際、Active Queue の空で無く最も優先度の高いリンクリストの先頭のプロセスを取り出す。そして、今まで実行していたプロセスがタイムスライスを使い切った場合には、再度タイムスライスを与え、そのプロセスを Active Queue から Ready Queue に移す。この動作を繰り返すとやがて Active Queue のプロセスが全て Ready Queue に移動する。するとスケジューラは Active Queue と Ready Queue を入れ替え、再びこの動作を繰り返す。

バージョン 2.6 から採用されたこのスケジューラは次に実行するスレッドの決定を TASK_RUNNING 状態のプロセス数によらず定数時間で行えるため $O(1)$ スケジューラと呼ばれる。

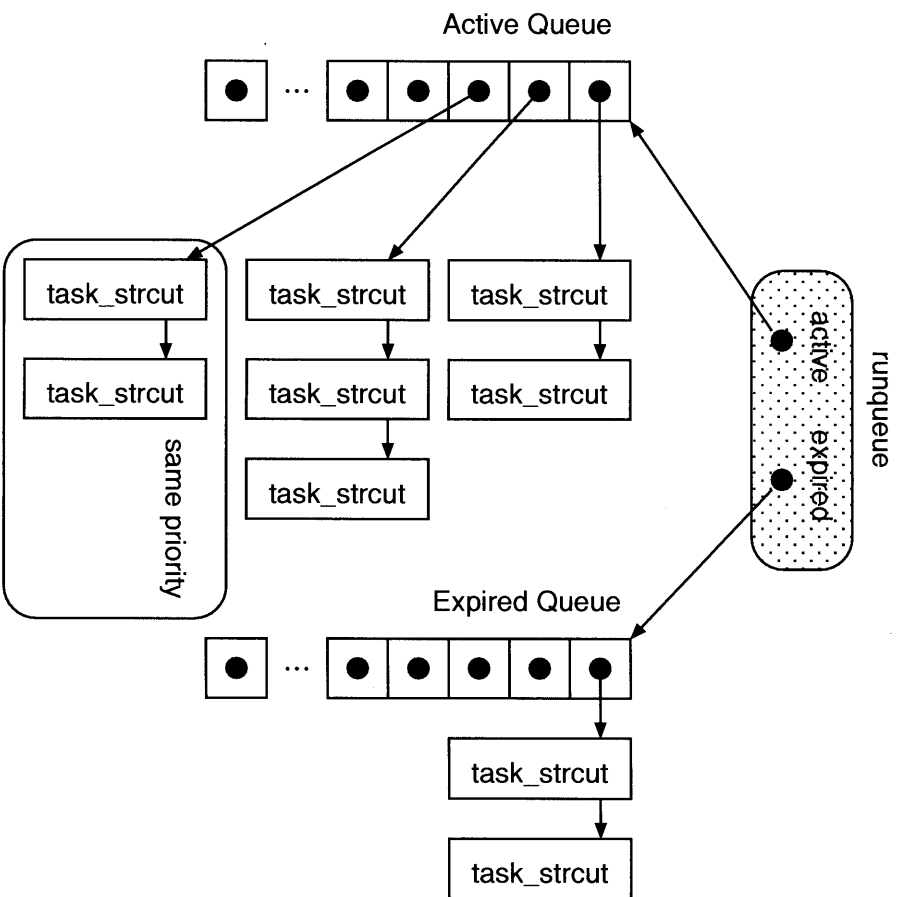


図 6.6: runqueue の構造

6.2 SSHとrunqueueの違い

4.3.4 で述べたように改良型 SSH は Linux の runqueue の構造に倣っているが、異なる点もある。ここではそれらについて述べる。

6.2.1 Active/Expired Queue の数

ソフトウェア実装では runqueue は CPU ごとにあるため、Active/Ready Queue は複数ある。一方、SSH 実装ではシステム全体で一つの Active/Expired Queue を SSH-m の中に備える。

6.2.2 Active Expired Queue のエントリ数

Linux ではプロセスの優先度は 140 段階ある。ソフトウェア実装ではこれに対応して、Active/Ready Queue は 140 のエントリを持つ。これに対し、SSH は 16 エントリしか持たないため、140 の優先度を 16 段階に区切り、異なる優先度のプロセスを一つのキューに入れる。このため、優先度の近いプロセス間で優先度逆転(より優先度の高いプロセスが実行可能であるにもかかわらず、優先度の低いプロセスが CPU を獲得してしまう)が発生するが、これによる致命的な問題は発生しない。これは次の理由による。

Linux では各プロセスの動作の統計をとり、各プロセスが I/O バウンダリなものか、CPU バウンダリなものか推定する。そしてその結果により I/O バウンダリなプロセスの優先度を自動的に上げ、CPU バウンダリなプロセスの優先度を下げる。これにより、I/O 処理や端末プログラムの応答性を上げ、また、CPU バウンダリなプロセスが CPU を独占してしまうことを防ぐ。この様に、Linux は積極的に優先度逆転を積極的に利用しており、優先度逆転の発生は問題とはならない。

6.2.3 保持可能なプロセス情報の数

SSH の Active/Expired Queue が 1 本につき 4096 プロセスの情報を保持できる。SSH は全てのスレッドがタイムスライスを使い切った場合に、Active Queue と Expired Queue を入れ替えるが、このためには、Active Queue が空となる瞬間が発生する必要がある。したがって、Expired Queue で保持できるプロセス情報の上限⁵が SSH で扱えるプロセス情報の数の上限となる。

一方、Linux が扱えるプロセス情報の上限はメモリ容量 (MB) \times 64 である⁶ため、512MB 以上のメモリを搭載している場合は SSH による上限

⁵Expired Queue は 16 本のキューを持ち、それぞれ 4096 スレッドの情報を保持するため 32768 となる。

⁶ページサイズ 4KB の場合

が最大プロセス数の制約となる。

6.2.4 プロセス情報を取得した際の動作

SSH 実装では次に実行すべきプロセスの情報を SSH-s から読み出すと⁷、そのプロセス情報は Active Queue から削除される。これに対しソフトウェア実装の runqueue では明示的に Active Queue から削除しない限りはプロセス情報は削除されない。このため、ソフトウェア実装では現在実行中のプロセスも Active Queue に入っているのに対し、SSH 実装では入っていない。また、コンテキストスイッチを行う際に、これまで実行していたスレッドがまだ実行可能状態である場合は再び Active Queue に格納する必要がある

6.2.5 プロセス情報の削除

プロセスが待ち状態に遷移するため、sleep_on() 等が呼ばれるとそのプロセス情報は Active/Expired Queue から削除される。しかし、SSH は Active/Expired Queue に入っているプロセス情報を削除することができない。このため、プロセスのデータ構造である task_struct 構造体の flags メンバに新たに「削除されたこと」を表すフラグを導入し、削除の動作をエミュレートする。

⁷正確には SSH-m から SSH-s にプロセス情報が渡された時

表 6.3: 新たに導入したフラグ

値	意味
PF_RUNNING	CPU が割り当てられ，現在実行中である．
PF_ENQUEUED	Active/Expired Queue に登録されており，スケジューリング対象である．
PF_DEQUEUED	Active/Expired Queue に登録されているが，スケジューリング対象ではない． または現在，CPU が割り当てられているが，次回，スケジューリング時に，スケジューリング対象から外される． PF_ENQUEUED とは排他的に設定される．

6.3 SSH を利用するスケジューラの設計

以上の違いをふまえ，SSH 実装では `task_struct` の `flags` メンバに新たに図 6.3 に示す 3 つのフラグを導入した．`state` は同時に一つの値しか取らず，プロセスの大まかな状態を表すのに対し，`flags` はビットフィールドを用いて複数の値を同時に取り，プロセスの細かな状態を表す．

PF_ENQUEUED は既に Active Queue または Exipred Queue にエンキュー (以下，単にエンキューという) されているプロセスを二重にエンキューしてしまわないため，現在エンキューされているか否かを判定するために導入した．

Linux のカーネルは，現在あるプロセスがエンキューされているか否かの判定を多く行っている．ところが，6.2.4 節で述べたように，SSH 実装では

現在実行中のプロセスは Active Queue に入っていないため、PF_ENQUEUED フラグは設定されない。従って、この判定を行うためには PF_ENQUEUED フラグを調べるだけではソフトウェア実装と結果が異なってしまう場合がある。そこで、現在実行中のプロセスには PF_RUNNING フラグを設定し、両方のフラグをあわせて検査することで、ソフトウェア実装の場合と同じ結果を得る。

また、6.2.5 節で述べたように、SSH の Active/Expired Queue はエンキューされているプロセス情報を削除することができない。このために導入した「削除されたこと」表すフラグが PF_DEQUEUED である。ソフトウェア実装では dequeue_task() 関数が引数で指定されたプロセス情報を、それがエンキューされているキューから削除していたが、SSH 実装では単に PF_DEQUEUED フラグを設定する。このため、SSH 実装の Active/Expired Queue には既に「削除された」スレッド情報が存在することになるが、SSH 実装のスケジューラは次に実行するスレッド情報を SSH-s から取得した際、そのスレッドに PF_DEQUEUED フラグが設定されている場合はもう一度取得し直すことで、既に「削除された」プロセスが実行されてしまうことを防ぐ。また、このため、待機状態であったプロセスを実行待ち状態に変更し、Active Queue にエンキューしようとする

際、既にそのプロセスが「削除された」状態で、Active Queue もしくは Expired Queue に入っている場合がある。この場合は、PF_DEQUEUED フラグをクリアし、PF_ENQUEUED フラグをセットする。

また、現在実行中のプロセスが待ち状態に移動する場合、`dequeue_task()` 関数が呼ばれ、その結果、ソフトウェア実装では Active Queue から削除されていたが、SSH 実装では PF_DEQUEUED フラグがセットされる。そして、SSH 実装のスケジューラではプロセスを切り替える際、これまで実行していたプロセスに PF_DEQUEUED フラグがセットされている場合は Active Queue へのエンキューを行わない⁸ことで、待ち状態に移動したプロセスが CPU を割り当てられてしまうことを防ぐ。

これらのフラグを利用して、SSH 実装のスケジューラを作成した。作成したスケジューラでの状態遷移を図 6.7 に示す。この図の 1 から 8 の状態遷移について説明する。

1. 実行中のプロセスが `interruptible_sleep_on()` または `sleep_on()` で待ち状態に遷移する場合。ソフトウェア実装ではこの過程で呼ばれる `deactivate_task()` の中で Activate Queue からデキューしていたが、SSH 実装では PF_DEQUEUED フラグをセットする。

⁸6.2.4 節で述べたように通常は再エンキューを行うことに注意。

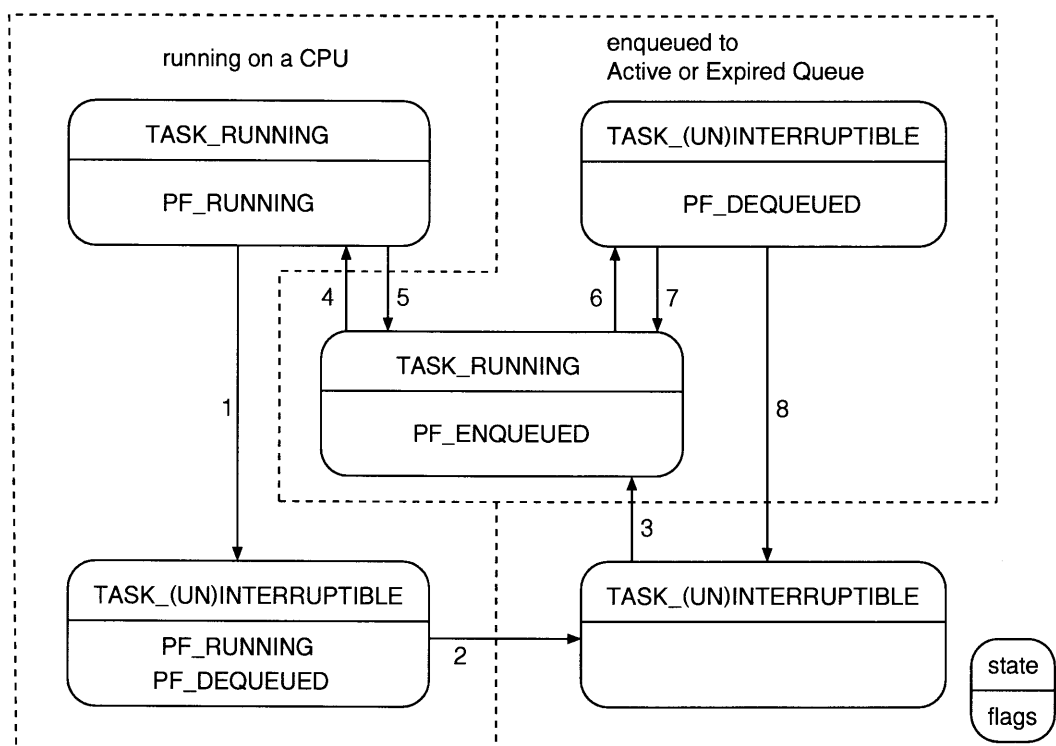


図 6.7: SSH 実装でのプロセスの状態遷移

2. この遷移は1に続いて発生するスケジューリング処理時に起こる。スケジューリング関数は今まで実行してたプロセスにPF_DEQUEUEDフラグが設定されている場合は、Active Queue へのエンキューを行わない。1,2によりこのプロセスはCPUを剥奪され、待ち状態に遷移する。
3. 待ち状態のプロセスがwake_up_interruptible() または wake_up() で起床された場合、プロセスは実行可能状態になり、Active Queue へエンキューされ、CPU 割り当て待ちとなる。
4. 実行可能状態でCPUが割り当てられるのを待っていたプロセスにCPUが割り当てられる場合。
5. 実行中のプロセスがタイムスライスを使い切った場合、プロセスは一端、CPUを剥奪され、実行可能状態となり、再度CPU割り当てを待つ。これにより、特定のプロセスがCPUを占有してしまうことを防ぐ。
6. CPU 割り当て待ちのプロセスに対し interruptible_sleep_on() または sleep_on() が実行された場合、ソフトウェア実装ではここで、Active/Expired Queue から削除されるが、SSHのActive Queue/Expired

Queue には保持している特定の要素を削除する機能がないため、PF_DEQUEUED フラグをセットするのみである。

7. 待ち状態のプロセスを起床しようとした場合、そのプロセスに PF_DEQUEUED フラグがセットされている場合がある。その場合、そのプロセスは既に SSH の Active/Expired Queue に入っていることになるため、新たに Active Queue にエンキューすることはせず、PF_DEQUEUED フラグをクリアし、PF_ENQUEUED フラグを設定する。
8. 次に実行するプロセスとして SSH-s から取得したプロセスに PF_DEQUEUED フラグが設定されている場合がある。これはすでにそのプロセスは実行可能状態では無くなっていることを示す。この場合はそのプロセスは実行せずに PF_DEQUEUED フラグをクリアする。

ソフトウェア実装のスケジューラのソースコード行数は 4700 行であったのに対し、SSH 実装のスケジューラは 2367 行になった。ただし、これはソフトウェアで行っていた処理を SSH で行うようになったことによる削減よりも、SSH 実装では利用できないソフトウェア実装のスケジューラの機能を削除したことによるものが大きい。

7 性能評価

7.1 評価環境

3.1 節で述べたように，SSH-s は CPU とオンチップでの実装を想定している．このため，PC を用いて評価を行うことはできず，専用の評価用計算機を Verilog HDL を用いて作成してその上へ Linux を移植し，Verilog シミュレータ上で動作させ評価を行った．

CPU には佐々木が作成した MIPS R3000 互換プロセッサ [2] を使用した．オリジナルの R3000 は CPU 間での同期命令を備えていないためマルチプロセッサ構成に対応しないが，本研究で使用した互換プロセッサは test and set 型の同期命令を備え，マルチプロセッサに対応する．

また Linux は R3000 をサポートしているが，前述の理由のため，Linux の R3000 固有コード部分もマルチプロセッサに対応した記述にはなっていない．このため，R3000 固有コードをマルチプロセッサに対応させる必要があった．

評価環境の諸元を表 7.1 に示す．

評価プログラムにはマルチスレッドで数列の和を計算する簡単なプログラムを使用した．このプログラムはメインスレッドが計算スレッドを生成し，それぞれに n 個の整数の和を計算させる．メインスレッドは各

表 7.4: 評価環境の諸元

CPU	MIPS R3000 互換プロセッサ
キャッシュ	64KB 命令 / 64KB データ
動作周波数	200MHz
CPU 間インターコネクト	共有バス結合
主記憶	64MB
OS	Linux 2.6.10
コンパイラ	GCC 3.4.4(Linux コンパイル用) GCC 3.4.2(評価プログラムコンパイル用)
C ライブラリ	uClibc 0.9.28
Verilog シミュレータ	VCS 7.0

スレッドが計算を終了するまで待機した後、全ての計算スレッドの結果を回収し、それらの総和を求める。計算スレッドはループ処理で和を求める部分が実行時間のほとんどを占め、その実行にかかるサイクル数は $5n$ になる。プログラムのソースコードを付録に掲載する。

7.2 スレッド数を変化させた場合の変化

プロセッサ数を 4 に、スレッドの粒度 n の値を 1000 に固定し、生成するスレッドの数を変化させ、性能評価を行った。

図 7.8 にスケジューリングに要するサイクル数 (schedule() 関数に入ってから抜けるまでに要するサイクル数) の平均を示す。ソフトウェア実装では 25 スレッドの場合に最も少ないサイクル数でスケジューリング可能でその後は増加してゆく。SSH 実装でもその傾向が見られるが増加は穏

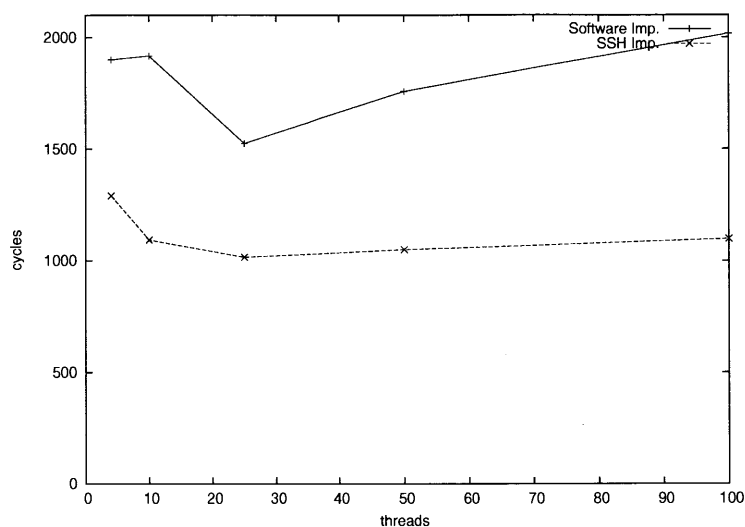


図 7.8: スレッド数を変化させた場合の平均スケジューリングサイクル数の変化

やかである．またスレッド数に依らず SSH 実装ではソフトウェア実装の 6～7 割のサイクル数でスケジューリング可能であることがわかる．

次にプログラムの実行に要したサイクル数を図 7.9 に示す．図 7.10 は SSH 実装でプログラムの実行に要したサイクル数をソフトウェア実装でのもので割って正規化したものである．

これらを見ると 50 スレッドまでは SSH 実装の方が速いものの，それよりもスレッドが多くなるとソフトウェア実装の方が速いことがわかる．スケジューリングに要するサイクル数対は 100 スレッドの場合でも SSH 実装の方が少ないにもかかわらずこのような結果が出るのは，図 7.11 に示すように，SSH 実装の方がスケジューリングが多発していることに依

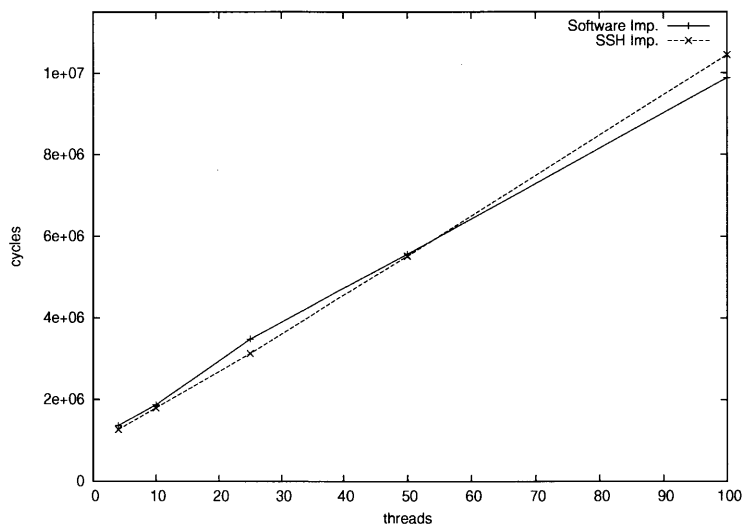


図 7.9: スレッド数を変化させた場合の評価プログラムの実行に要するサイクル数の変化

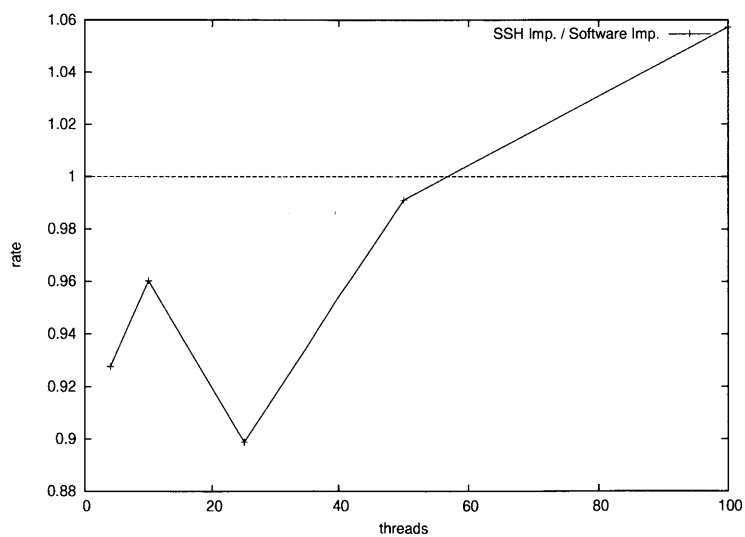


図 7.10: スレッド数を変化させた場合の評価プログラムの実行にサイクル数の変化 (正規化したもの)

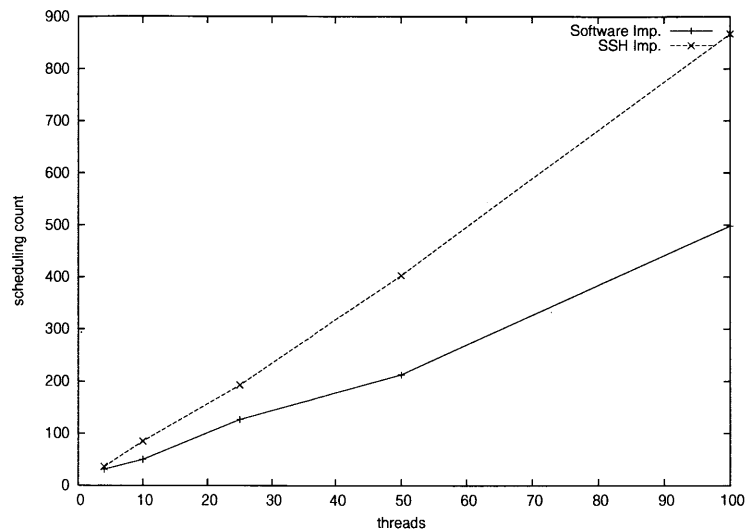


図 7.11: スレッド数を変化させた場合のスケジューリング回数の変化

るものと考えられる。

7.3 スレッドの粒度を変化させた場合

CPU 数を 4 に、スレッド数を 25 に固定し、スレッドの粒度 n を変化させ性能評価を行った。

まず、スケジューリングサイクル数の平均値の変化を図 7.12 に示す。

ソフトウェア実装では粒度の増加に比例し、緩やかにスケジューリングサイクル数の増加が見られるが、SSH 実装ではその傾向は見られない。

次にプログラムの実行に要したサイクル数を図 7.13 に、SSH 実装の実行サイクル数をソフトウェア実装のもので割って正規化したものを図 7.14 に示す。

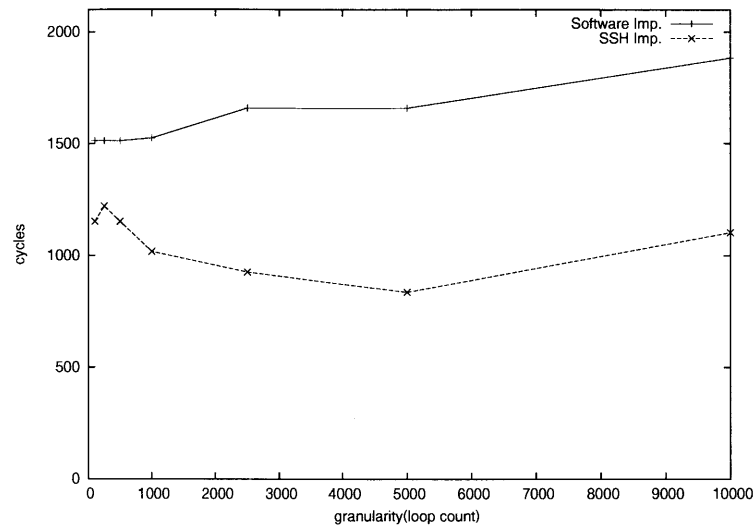


図 7.12: 粒度を変化させた場合の平均スケジューリングサイクル数の変化

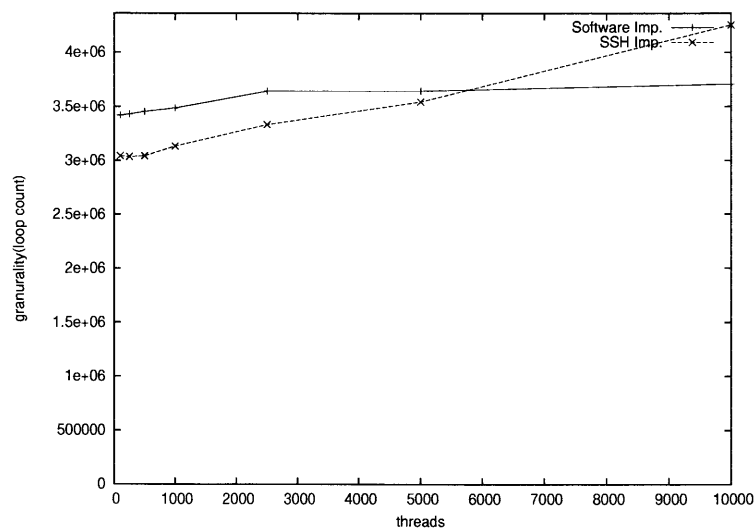


図 7.13: 粒度を変化させた場合の評価プログラムの実行に要するサイクル数の変化

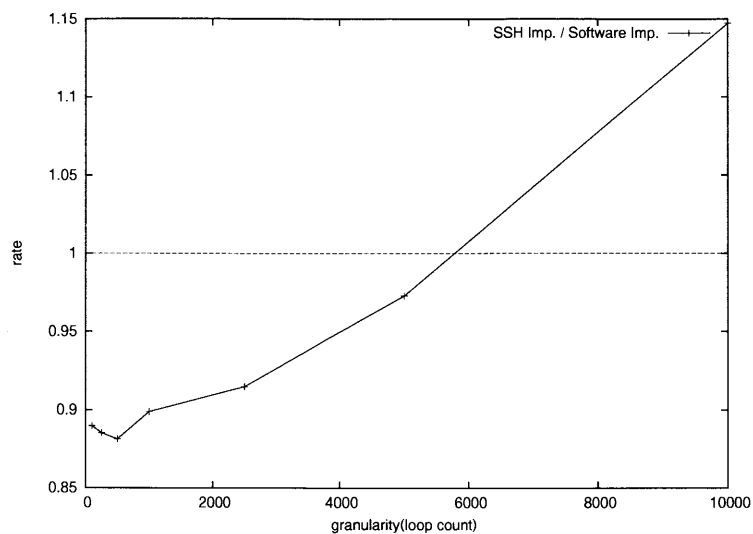


図 7.14: 粒度を変化させた場合の評価プログラムの実行に要するサイクル数の変化 (正規化したもの)

スレッド数を変化させた場合と同様に，実行時間が短い場合は SSH 実装の方が速いものの，実行時間が長くなると性能が落ちてしまう．図 7.15 に粒度を変化させた場合のスケジューリング回数の変化を示す．この図より，スレッドを変化させた場合と同様に，この傾向はスケジューリングが多発していることによるものと考えられる．

8 関連研究

マルチプロセッサ用のタスク・スケジューリング方式に関する研究 [3, 4, 5, 7, 8] は広く行われているが，その多くは OS やマルチスレッド・ライブラリ等，ソフトウェアでタスクのスケジューリングを行うものであ

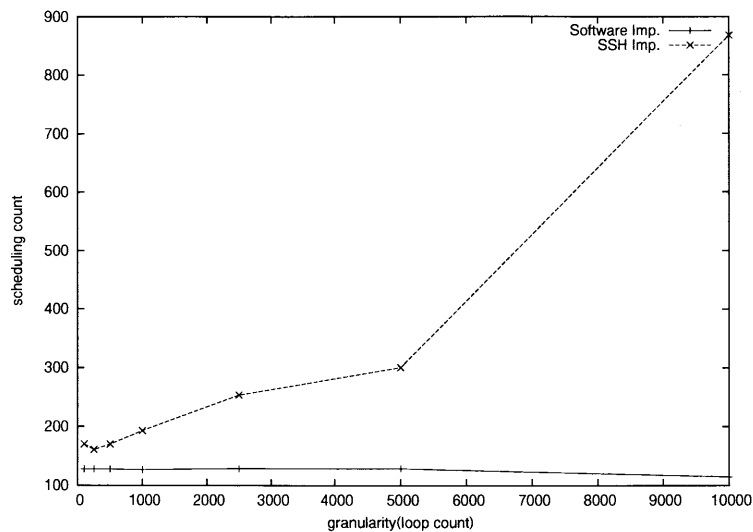


図 7.15: 粒度を変化させた場合のスケジューリング回数の変化

る。また、タスクスケジューリング、あるいは OS 全体のハードウェア化による高速化の研究も行われているが、その多くはユニプロセッサを対象としている。

仲野らによる STRON[9, 10] は、リアルタイム OS である μ ITRON の一部をハードウェア化することで高速化を目指したものである。STRON では、OS の支援を行うハードウェアを導入することで高速化を行っている。しかしながら、導入しているハードウェアはハードウェアで実現したサブルーチンであり、CPU と協調処理を行うわけではない。一方、本稿で提案する手法では、CPU で実行されるソフトウェア部分と専用ハードウェアで実行される処理が並列に動作する点異なる。これにより、ハー

ドウェアによる高速化だけでなく，処理時間の隠蔽も可能にしている．

本研究の様に，Linux のマルチプロセッサ環境でスケジューリングをハードウェア化するものとしては Boston Circuits 社の gCORE[11] がある．gCORE はネットワーク環境を模して相互接続した複数のプロセッサコアを内蔵するマイクロプロセッサアーキテクチャでワンチップでグリッドコンピューティングを実現する．gCORE は CPU へのスレッド割り当てを専用回路，Time Machine で行う．Time Machine ではスレッド情報の送受信を CPU 間ネットワークを介して行うが，この CPU 間ネットワークはメモリアクセストランザクション，I/O アクセストランザクションの送受信にも使用するため，スケジューリングが多発する近細粒度並列処理ではメモリアクセス性能低下を招く恐れがある．一方，SSH は専用のスケジューラバスを備えるためスケジューリングの多発によりメモリアクセス性能が低下することはない．

9 おわりに

本研究では，SSH を用いて Linux のスケジューリングを高速化する手法を示した．そして，SSH を組み込んだ評価用計算機を設計し，Linux を移植して提案手法を実装，評価を行った．評価ではスケジューリング処

理自体は高速になることは確認できたものの、スケジューリング処理が頻発し、結果としてプログラムの実行にかかるサイクル数が多くなってしまう場合があることがわかった。

今後はまず、スケジューリング処理が頻発する理由を明らかにし、無駄なスケジューリング処理を行わないようにする。

その後の課題は以下の通りである。

- 佐々木らが提案するコンテキストスイッチ支援ハードウェア (CSS) を Linux から利用するようにし、コンテキストスイッチも高速化する。
- 同じプロセスはできるだけ同じ CPU で実行されるようにスケジューリングを行うよう SSH を改良する。あるプロセスが過去に実行されたことのある CPU のキャッシュにはそのプロセスが利用するデータが残っている可能性が高いため、性能向上が期待できる。
- 本研究で作成した評価用計算機は実際の計算機として作成できるように設計されている。そこでこれを実際に作成し、シミュレーションでは難しい実行時間の長い評価プログラムを使った評価を行う。

謝辞

本研究を進めるにあたり，日頃からご指導，ご助言を頂きました佐々木敬泰助手，近藤利夫教授，大野和彦講師に深く感謝いたします。また，暖かい心配りを頂きました田中みゆき事務官に心より感謝いたします。

最後に，計算機アーキテクチャ研究室の諸氏には本研究の様々な局面にて多大なご協力を頂きました。ここに感謝の意を表します。

参考文献

- [1] 佐々木 敬泰, 西村 直己, 弘中 哲夫, 吉田 典可: マルチプロセッサ用スケジューリング支援ハードウェアの提案とシミュレーション評価, 電子情報通信学会論文誌, Vol.J84-D-I, No.11, pp.1515-1531 (2001).
- [2] Naoki Nishimura, Takahiro Sasaki and Tetsuo Hironaka: Prototype Microprocessor LSI with Scheduling Support Hardware for Operating System on Multiprocessor System, Asia and South Pacific Design Automation Conference 2000 (ASP-DAC2000), pp.29-30 (2000).
- [3] D. Stein, D. Shah: Implementing Lightweight Threads, USENIX Conference Proceedings, pp.1-10 (1992).

- [4] K. Taura and A. Yonezawa: Fine-grain multithreading with minimal compiler support—a cost effective approach to implementing efficient multithreading languages, Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.320-333 (1997).
- [5] M.L. Powell *et al.*: SunOS Multi-thread Architecture, USENIX Conference Proceedings, pp.1-13 (1991).
- [6] 飯田全広, 久我守弘, 末吉敏則: マルチスレッド制御ライブラリのハードウェア化によるオンチップ・マルチプロセッサの構成, 並列処理シンポジウム (JSPP'97 IPSJ Symposium Series), pp.337-344 (1997).
- [7] 坂本 力, 宮崎 輝樹, 桑山 雅行, 最所 圭三, 福田 晃: 並列性と移植性をもつユーザレベルスレッドライブラリー PPL の設計および実装, 電子情報通信学会論文誌, Vol.J80-D-I, No1, pp.42-49 (1997).
- [8] 小熊 寿, 海江田 章裕, 森本 浩通, 田村 友彦, 鈴木 貢, 中山 泰一: SMP 型計算機を活用する軽量プロセス・ライブラリ, 情報処理学会論文誌, Vol.39, No.9, pp.2718-2725 (1998).

- [9] 仲野巧, 小松平良樹, 塩見彰睦, 今井正治リアルタイム OS チップの性能評価と分散 OS への拡張, 電子情報通信学会技術報告書, Vol. CPSY-51, pp.23-30 (1998).
- [10] Nakano, T., Komatsudaira, Y., Shiomi, A. and Imai, M.: Performance Evaluation of STRON: A Hardware Implementation of a Real-Time OS, IEICE Transactions, Vol. E82-A, No. 11, pp.2375-2382 (1999).
- [11] Aron Kurland, 片岡 裕之, 梅村 誠: ネットワークで CPU コアをつなぐマルチコア型マイクロプロセッサを開発, 日経 BP 日経エレクトロニクス 9 月 26 日号 pp.179-186 (2006).
- [12] Daniel P. Bovet, Marco Cesati: 詳細 Linux カーネル 第2 判, オライリー・ジャパン (2003).
- [13] Josh Aas: Understanding the Linux 2.6.8.1 CPU Scheduler, <http://josh.trancesoftware.com/linux/> (2005).
- [14] 高橋 浩和: Linux カーネル 2.6 読解室 第1 回 プロセススケジューリング, ソフトバンク パブリッシング UNIX USER 2004 年 6 月号 pp.89-106 (2004).

- [15] Harlan McGchan: NIAGARA 2 OPENS THE FLOODGATES,
http://www.sun.com/processors/niagara/M45_MPFNiagara2_reprint.pdf
(2006).
- [16] INTEL RESEARCH ADVANCES ‘ERA OF TERA’,
Intel Corporation News Release (2007)
ftp://download.intel.com/research/platform/terascale/terasflops/Teraflops_Research_Chip_Press_Release_Feb%2011%2007.pdf.
- [17] J. A. Hahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer,
and D. Shippy:
Introduction to the Cell multiprocessor, IBM Journal of Research
and Development (2005)
<http://researchweb.watson.ibm.com/journal/rd/494/kahle.pdf>.
- [18] John L. Hennessy and David A. Patterson: Computer Architecture
A Quantitative Approach Fourth Edition, MORGAN KAUFMANN
PUBLISHERS (2006).
- [19] 高橋 浩和, 小田 逸郎, 山幡 為佐久: Linux カーネル 2.6 解説室, ソフトバンク パブリッシング (2006).

- [20] 大原 一輝, 佐々木 敬泰, 大野 和彦, 近藤 利夫: ハードウェアスケ
ジューラによる Linux 上での近細粒度並列処理の高速化, 情報処理学
会研究報告 2006-ARC-170, pp.25-30 (2006).