TM-1190

SPARC プロセッサにおける 高速コンテキスト切替え方式

洒井　浩（東芝）

July, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

SPARCプロセッサにおける高速コンテキスト切替え方式

酒井 浩

(株) 東芝 総合研究所

本稿は，SPARCプロセッサにおける軽量プロセス間のコンテキスト切替え機構が従来遅いとされてきた理由について考察し、より高速な切替え機構を提案する。SPARCプロセッサにおける従来のコンテキスト切替えでは、実行中の軽量プロセスが使用していたレジスタウインドウの退避をシステムコールで行い、これに処理時間の大半を費やしている。本稿が提案するコンテキスト切替えは、非特権命令だけを使用することにより、従来の方法と比較して、典型的な場合において数倍、高速である。また、本コンテキスト切替え方式を、並列計算機の動作をクロックレベルでシミュレーションするソフトウェアに適用した結果、やはり数倍の高速化を達成した。

A Fast Context Switching Mechanism on SPARC Processor

Hiroshi Sakai

Toshiba Research and Development Center

This paper discusses the reason why a SPARC processor has been said to show less efficiency in context switching between light-weight processes and proposes a faster mechanism. Every conventional context switching mechanism on a SPARC processor consumes much time in executing a system call which saves the processor's register window used by the current process. The proposed mechanism, which uses only non-privileged instructions, shows several times better performance in a typical case. Moreover, it has accelerated a clock-level, software simulation of a parallel computer's behavior by several times.

1 はじめに

軽量プロセス (light weight process) は、通常のプロセスと比較して、その生成・消滅、データの共有、コンテキスト切替えのコストが小さいという特徴があり、並行動作を効率良く実現手段として注目されている。しかしながら、SPARC プロセッサ [SPARC'90] における軽量プロセスの実装に関しては、レジスタウインドウの退避と復旧の処理が重く、プロセッサ性能の割にはコンテキスト切替えが遅いとされてきた [新城 92]。本稿では、SPARC プロセッサにおける従来のコンテキスト切替え方式について考察するとともに、より高速なコンテキスト切替え方式を提案する。

1.1 研究の動機

著者は、通産省第五世代コンピュータプロジェクトの一環として、新世代コンピュータ技術開発機構（ICOT）から再委託を受け、並列推論マシンの研究開発に従事している。そして、並列推論マシンＰＩＭ／ｋの試作とともに、マルチプロセッサシステムの並列実行をシミュレートするソフトウェアシミュレータＳＳＦ（System Simulation Framework）を開発し、処理系のデバッグや計算機アーキテクチャの定量的評価に使用している。ＳＳＦは、大部分がＣ言語で書かれており、マルチプロセッサシステム内のプロセッサをはじめとする各種エージェントの動作を軽量プロセスでシミュレートする。

Ｓｕｎ３用に開発した初版ＳＳＦでは、軽量プロセス間のコンテキスト切替えを高速化するため、[多田 90] のアセンブラ版 c_sched と類似のものを使用した。その後、高速化のためＳｕｎ４に移植した際、コンテキスト切替えにはＳｕｎ ＯＳの lwp(light weight process) ライブラリ [Sun90] の lwp_yield を採用した。その結果、期待に反して、実行速度がＳｕｎ３の場合より遅くなった。この原因を調べた結果、lwp_yield の所要時間は、SPARC Station 2 (40MHz) で約３５μ秒であり、Sun3/260 におけるアセンブリ言語で記述したサブルーチンの所要時間（約７μ秒）より遅いことが分かった。そこで、Sun4 における軽量プロセス間のコンテキスト切替えを高速化しようとしたのが本研究の動機である。

1.2 ＳＳＦの概要と軽量プロセス使用の利点

ＳＳＦは、並列プログラムのデバッグや計算機アーキテクチャの定量的評価など使用目的に応じて、モデルの構成を適宜選択するようになっている。例えば、並列プログラムのデバッグに使用する場合、プロセッサ内部の構成やキャッシュやバスの動作には興味がないので、各プロセッサを一つの軽量プロセスでシミュレートし、それ以外には軽量プロセスを設けない。各軽量プロセスは、プログラムカウンタの指し示すメモリ番地に格納されている命令をフェッチし、ロード／ストアあるいはレジスタ間の演算命令等を実行する。そして、一命令の実行が終わると次のプロセッサに対応する軽量プロセスにコンテキスト切替えする。このようにして、ラウンドロビンのスケジューリングにより、すべてのプロセッサが一命令ずつ実行するのを一サイクルとして、プログラムの並列実行を行う。

また、プロセッサアーキテクチャを定量的に評価する場合は、例えば、プロセッサの命令パイプラインの各ステージに軽量プロセスを一つずつ割当てる形でプロセッサをシミュレートし、SPEC ベンチマーク等を実行させることが考えられる。

以上述べたように、ＳＳＦでは独立的に動作しうるエージェントを軽量プロセスで実現する。代替案としては、軽量プロセスを使用せず、各エージェントについて状態遷移表を作成し、Ｃ言語の switch 文で各状態に分岐して処理することも可能である。しかし、何段かにネストして呼び出されたサブルーチンの中でコンテキスト切替え（または状態遷移）を起こす場合、軽量プロセスの方が記述しやすく、デバッグも容易である。例えば、ＰＩＭ／ｋのメモリアーキテクチャの評価では、最高３段にネストしたサブルーチンの中でコンテキスト切替えを起こす場合がある。

なお、ＳＳＦのようにＣ言語でシミュレータを作成する他に、ハードウェア記述言語でシミュレーションを行うことも可能であり、例えばＶＨＤＬではＳＳＦと同様のプロセス記述が可能である [?]。

1.3 他の研究との関係

軽量プロセス間のコンテキスト切替えは、CMUのmachのようにカーネルで行うものとユーザプログラムで行うものとに大別できる。本稿で述べる方式は、後者に属する。多田らは、大域jumpの機構を用い、種々のOSやプロセッサで利用可能な移植性の高いコンテキスト切替え機構を提案した[多田90]。多田らの機構は、Sun OSのlwpライブラリより高速であるが、これは、主にスケジューリング機構等を簡略化した効果である。SPARCプロセッサ用のSun OSの大域jump用関数では後述するレジスタウインドウの退避用システムコールが使われているため、コンテキスト切替え速度の改善は数10%程度である。

新城らは、メモリ共有型マルチプロセッサに適用できる軽量プロセスを提案した[新城92]。種々のプロセッサに適用できるという意味で汎用性を重視した研究である。SPARCプロセッサに適用する場合には、レジスタウインドウの退避用システムコールを使用しており、コンテキスト切替え速度は、lwpライブラリの場合と類似している。

SPARCプロセッサ用のコンテキスト切替えに関しては、PardoがSPARCのスタック構造やレジスタウインドウの退避用システムコールを使ったコンテキスト切替え機構の作成方法を解説している[Pardo90]。本稿では、この解説に基づいて作成した方式（仮にBCS: Basic Context Switch mechanismと呼ぶことにする）を従来方式として参照する。

本稿が提案するコンテキスト切替え方式（仮にFCS: Fast Context Switch mechanismと呼ぶことにする）は、レジスタウインドウの退避を含むすべての処理を非特権命令で実現するところに特徴がある。FCSは、サブルーチン呼出しのネストが大きくない（およそ5以下）の場合に従来の方式より高速である。特にSSFにおいては効果が大きい。なお、本稿では、preemptionの方法については述べないが、u構造体の参照により実現可能であると考えている。

本稿の構成は、次のとおりである。まず、SPARCプロセッサのレジスタウインドウについて説明した後、従来のコンテキスト切替え方式について述べる。次に、FCSについて説明し、性能評価および結果の考察を行う。

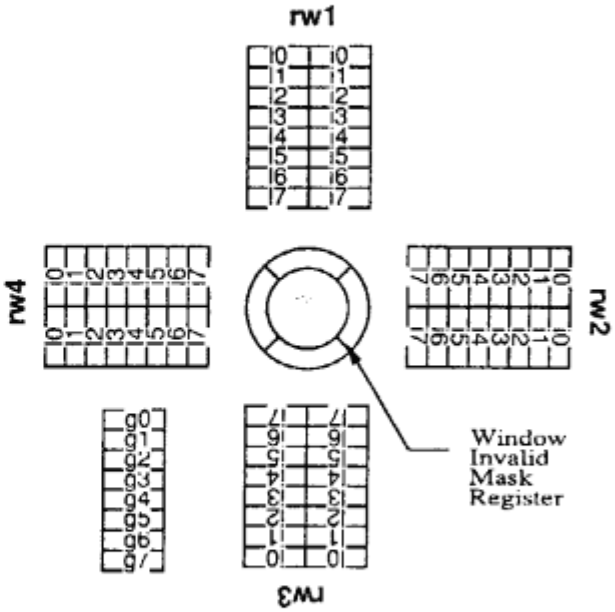
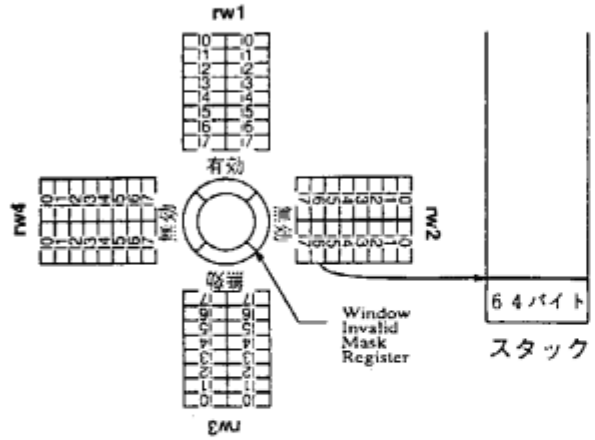


図1: SPARCプロセッサのレジスタ構成

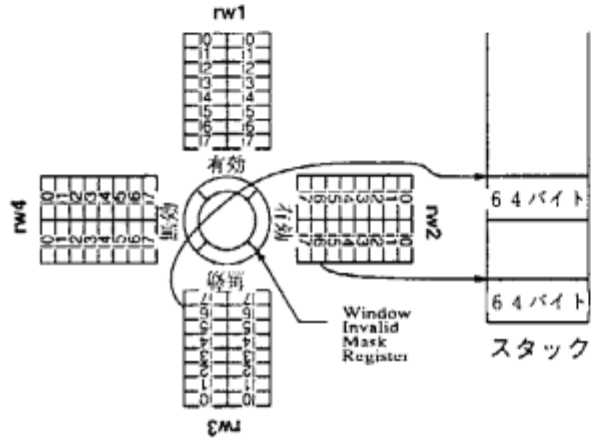
2 レジスタウインドウの概要

SPARCプロセッサのレジスタウインドウを図1に基づいて説明する。一つのレジスタウインドウは、l0～l7とi0～i7という全部で16個のレジスタの組であり、いくつかのレジスタウインドウがリングを構成している。図1では、rw1～rw4という4個のレジスタウインドウがリングを構成している。これらのレジスタウインドウに有効なデータが保持されているか否かを記憶するため、Window Invalid Mask Registerが備わっている。

SPARCプロセッサでは、この他にo0～o7とg0～g7という16個のレジスタ及び浮動小数点レジスタが使用可能である。このうち、o0～o7は、隣のレジスタウインドウのi0～i7を指す。例えば、rw1が選択されている場合にo0～o7を参照すると、rw2のi0～i7が使用され、rw4が選択されている場合にo0～o7を参照すると、rw1のi0～i7が使用され



(a) スタートルーチン実行時のスタックの状態



(b) サブルーチン実行時のスタックの状態

図 2: スタックとレジスタウインドウの関係

る。o6 は、スタックポインタとして使用される。一方、g0 ～ g7 及び浮動小数点レジスタは、レジスタウインドウとは別のレジスタ群であり、どのレジスタウインドウが選択されているかに拘らず、常に同じレジスタ群が参照される。g0 はゼロレジスタである。

プログラムの実行開始時点では、任意の一つのレジスタウインドウが選択されており、Window Invalid Mask Register には、選択されたレジスタウインドウだけが有効であると記録される。図 2(a) は、rw1 が選択されている場合を表わしており、Window Invalid Mask Register には、rw1 が有効であると記録される。

このとき、スタックポインタ o6 は、スタックの底から 64 バイト小さいアドレスを指している。この 64 バイトのスタック領域は、タイマ割込み等の発生時やその結果引き起こされるプロセス間でのコンテキスト切替え時に rw1(すなわち i0 ～ i7 と l0 ～ l7) を退避するのに使用される。

ここで、プログラムの初めに実行開始するルーチンをスタートルーチンと呼ぶことにする。スタートルーチンがサブルーチンを呼出す場合、引数を o0 ～ o5 にロードした後、call 命令が実行される。call 命令は、その call 命令の格納番地を o7 に格納する。

サブルーチンでは、初めに save 命令を実行する。この命令では、図 2 (b) に示すように rw2 が選択されるとともに、rw1 のスタックポインタ o6 から 64 バイト以上の値を減算したものを rw2 のスタックポインタ o6 に格納する。rw2 のスタックポインタ o6 から 64 バイトの領域は、タイマ割込み等の発生時に rw2 を退避するのに使用される。また、Window Invalid Mask Register には、rw1、rw2 が有効であると記録される。

サブルーチンでは、rw2 の l0 ～ l7 と i0 ～ i7 のうち、rw1 のスタックポインタとして使用される i6 と、サブルーチンへの call 命令のアドレスを格納している i7 を除き、他のレジスタを自由に使用できる。

サブルーチンからスタートルーチンへ復帰する場合、restore 命令を実行し、rw1 を選択するとともに、Window Invalid Mask Register には、rw1 だけが有効であると記録される。その後、rw1 の o6 の内容（すなわち call 命令の格納番地）＋ 8 番地にジャンプすることにより、スタートルーチンへ復帰する。

以上述べたように、SPARC プロセッサでは、レジスタウインドウを持たないプロセッサにおいて必要であったサブルーチン呼出しにともなうレジスタの退避と復旧の代わりに、save 命令と restore 命令を実行するだけでよいため、サブルーチンの呼出しの高速化が期待できる。しかし、軽量プロセス間のコンテキスト切替えでは、レジスタウインドウをその軽量プロセス用に割当てられたスタック領域に退避する必要がある。

3 従来の方式（ＢＣＳ）

ＢＣＳは、下記に示すＳＰＡＲＣアセンブリ言語で記述されたサブルーチンであり、従来のコンテキスト切替え機構のうちで最も高速であると考えられる。

- o6, o7 を現在実行中の軽量プロセスのコンテキスト格納領域に退避
- ST_FLUSH_WINDOWS システムコール[1]
- o6 ← ６４バイトの作業領域の先頭番地
- 次に実行すべき軽量プロセスのコンテキスト格納領域に格納されている o6, o7 のデータを g1, o7 に復旧
- g1 が指す番地から６４バイトの領域に格納されているデータを現在選択中のレジスタウインドウに復旧
- o6 ← g1
- o7 + 8 番地へジャンプ

ＢＣＳは、一般にサブルーチン呼出しの前後で、g1 ～ g7、o0 ～ o5 の内容は破壊されても構わないことを前提にしている。また、入口と出口で save 命令と restore 命令を省略し、退避すべきレジスタウインドウの数が増えないようにしている。

ST_FLUSH_WINDOWS システムコールは、現在選択中のレジスタウインドウから順に restore 命令で次のレジスタウインドウを選択しながら、Window Invalid Mask Register に有効と記録されているすべてのレジスタウインドウを対応するスタック領域に退避する。

o6 レジスタに作業領域の先頭番地を一時的に設定するのは、レジスタウインドウの復旧途中でタイマ割込み等の発生により、元の軽量プロセスのスタック領域が破壊されるのを防ぐためである。

次に実行する軽量プロセスについては、一つのレジスタウインドウだけを復旧するので、Window Invalid Mask Register には、そのレジスタウインドウだけが有効であると記録されている。コンテキスト切替え後、軽量プロセスがサブルーチンから呼出し側へ復帰のため restore 命令を実行すると、レジスタウインドウ・アンダフロートラップが発生し、スタックに退避されているデータを使ってレジスタウインドウの復旧が行われる。

実行を再開した軽量プロセスが save 命令や restore 命令を実行しないうちに再びコンテキスト切替えを行った場合、ST_FLUSH_WINDOWS システムコールでは現在選択中のレジスタウインドウだけを退避すればよく、コンテキスト切替えは比較的高速である。しかし、この場合においても、上記の方式によるコンテキスト切替えの所要時間は、SPARC Station 2 (40MHz) でユーザ時間 5 μ 秒、システム時間 15.5 μ 秒であり、合計で 400 命令余りの処理に相当する。Sun3/260 では、約 7 μ 秒（約３０命令に相当）で実現できることを考えると、遅いといわざるを得ない。

[1] /usr/include/machine/trap.h で定義されている

4 新しい方式（FCS）

FCS はレジスタウインドウの退避フェーズと復旧フェーズで構成され、非特権命令だけで実現しているのが特徴である。FCS の骨格部分は、次のとおりである。なお、実際使用中のサブルーチンは、SPARC のアセンブリ言語で記述するとともに、高速化のため最適化をはかっている。

- o6, o7 を現在実行中の軽量プロセスのコンテキスト格納領域に退避
- g4 ← o6

L1: g4 ← g4 - 4

- [g4] ← o6
- o6 が指す 64 バイト領域に現在選択中のレジスタウインドウを退避
- restore
- i6 とゼロを比較し不一致であれば、L1 へジャンプ
- g4 を現在実行中の軽量プロセスのコンテキスト格納領域に退避
- 次に実行すべき軽量プロセスのコンテキスト格納領域に格納されている g4 のデータを g4 に復旧
- g3 ← [g4]
- g4 ← g4 + 4

- g1 ← 64バイトの作業領域の先頭番地

L2: save %g1, %g0, %o6 [2]

- g3 ← [g4]
- g4 ← g4 + 4
- g3が指す64バイト領域に格納されているデータを使って現在選択中のレジスタウインドウを復旧
- g3とg4を比較し不一致であれば、L2へジャンプ
- o6 ← g3
- o7 + 8番地へジャンプ

前半のレジスタウインドウ退避フェーズでは、本来はWindow Invalid Mask Registerに有効と記録されているレジスタウインドウだけを退避すればよいのであるが、これを参照する命令は特権命令のため利用できず、やむを得ず現在選択中のレジスタウインドウからスタートルーチンに対応するレジスタウインドウまでのすべてを退避している。スタートルーチンに対応するレジスタウインドウであるか否かの判定は、i6の内容がゼロか否かで行い、そのレジスタウインドウについては退避および復旧を省略している。なお、このフェーズで各レジスタウインドウのo6の内容をg4が指すスタック領域に格納しているのは、後半のレジスタウインドウ復旧フェーズで退避と逆の順番に復旧するのに必要なためである。

後半のレジスタウインドウ復旧フェーズでは、スタートルーチンに対応するレジスタウインドウから最近に呼出されたサブルーチンに対応するレジスタウインドウまですべてのレジスタウインドウの復旧を行う。これは、前半のフェーズで使用中のすべてのレジスタウインドウを退避する際に、レジスタウインドウ・アンダフロートラップが起きるのを防ぐためである。o6レジスタに作業領域の先頭番地を一時的に設定するのは、レジスタウインドウの復旧途中でタイマ割込み等の発生により、元の軽量プロセスのスタック領域が破壊されるのを防ぐためである。

FCSが典型的なケースにおいてBCSより高速であるのは、下記の理由によるものと考えられる。

- システムコールは一般にその入口と出口において、必要のないレジスタまで退避、復旧することが多い。また、システムコールの要因解析にも若干の処理時間を要する。
- レジスタウインドウの退避ではo6が指す番地に内容を格納するため、システムモードでの実行ではシステム領域が破壊されないよう、o6の内容の正当性検査が必要である。それに対してユーザモードで実行では、正当性検査を省略できる。

問題点としては、使用中のレジスタウインドウをすべて退避するため、BCSより遅くなる可能性があり、適用範囲が限定されることである。ただし、SSFのようにサブルーチン呼出しのネストが深くないところでコンテキスト切替えが起こる場合には十分有効である。

5 性能評価と考察

5.1 従来方法との比較

FCSとBCSおよびlwp_yieldについてコンテキスト切替え速度を比較するため、2つの軽量プロセス間でコンテキスト切替えを合計で2,000,000回繰返すプログラムを作成し、実行時間を測定した。各軽量プロセスは、下記の処理を繰返す。

「サブルーチン呼出しがLだけネストした状態から、ネストがCに達するまでサブルーチン呼出しを繰返し、そこでコンテキスト切替えを行う。実行再開後は、ネストがLに戻るまでサブルーチンからの復帰を繰返す」

上記プログラムの実行時間には、サブルーチン呼出しが含まれるため、コンテキスト切替えの部分だけを除いたプログラムについても実行時間を測定した。

使用計算機は、SPARC Stattion 2 (40MHz)であり、プログラムの実行時間の測定には、timeコマンドを使用した。コンテキスト切替え1回当たりの所要時間は、(プログラム実行時間／コンテキスト切替え回数)で計算した。

図3は、FCSが不利なL = Cの場合について、L

[2] 新しく選択されたレジスタウインドウのo6にg1の内容がコピーされる

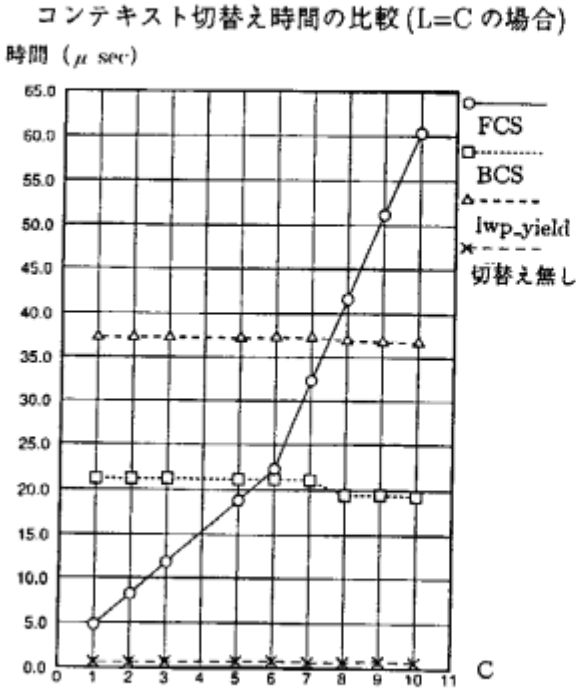


図 3: FCS と従来方式の性能比較（1）

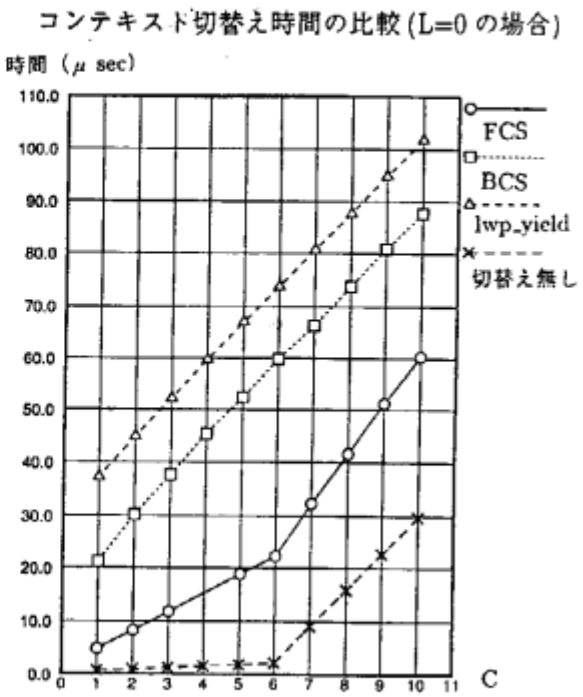


図 4: FCS と従来方式の性能比較（2）

を1から10まで変化させた場合のコンテキスト切替え時間を調べたものである。

- BCSとlwp_yieldでは、Lが変化しても処理時間はほぼ一定である。これは、レジスタウインドウの退避用システムコールにおいて、Window Invalid Mask Register に有効と記録されているレジスタウインドウの数が一定であるためと考えられる。BCSがlwp_yieldより高速であるのは、スケジューリングの簡略化とともに、サブルーチンの入口と出口でsave命令とrestore命令を省いたことにより、システムコールで退避する必要のあるレジスタウインドウ数を減らしたことが大きく寄与している。

- FCSの所要時間は、L≦6の範囲でLが増加するに従い、ほぼリニアに増加する。これは、退避と復旧を行うレジスタウインドウの数がLに等しいためである。また、L＞6の範囲では増加の傾きが急になっている。これは、使用計算機のSPARCプロセッサのレジスタウインドウ数が8で、L＞6ではレジスタウインドウのアンダーフローとオーバフロートラップが発生するためと考えられる。

- L＞5では、FCSとBCSの優劣が逆転する。

図4は、FCSが有利なL＝1の場合について、Cを1から10まで変化させた場合のコンテキスト切替え時間を調べたものである。

- サブルーチン呼出し／復帰のみの所要時間は、L=6を境として急激に増加している。これは、レジスタウインドウのオーバフロートラップとアンダフロートラップの発生によるものと考えられる。

- BCSとlwp_yieldの所要時間は、Cが増加するに従い、ほぼ直線的に増加している。ただし、所要時間の中には、サブルーチン呼出し／復帰の時間が含まれているので、それを差し引くと、C＞6でほぼ一定となる。また、BCSでC≦6で直線的に増加するのは、レジスタウインドウ退避

用システムコールで退避すべきレジスタウインドウ数がCであること、および、Cだけネストしたサブルーチンの中から毎回レジスタウインドウ・アンダフロートラップを起こしながら呼出し側に復帰することによる。

- FCSが、C≦6でCの増加にともなってリニアに増加するのは、退避および復旧すべきレジスタウインドウ数がCに等しいためである。また、C＞6で所要時間が急激に増加するのは、レジスタウインドウのアンダフロートラップ等によるものと考えられる。

5.2 SSFへの適用

FCSをSSFに適用して効果を調べた。その結果、シミュレーションモデルにより、軽量プロセスでコンテキスト切替え1回当たりに行う本来のシミュレーションに要する処理時間が異なるため、効果にばらつきはあるものの、lwp_yieldを使用した場合と比較して、実行速度が3～6倍になった。コンテキスト切替え頻度は、多いもので100,000回／秒程度であり、FCSを適用した後も全体の処理時間の半分以上がコンテキスト切替えに費やされている。SSFの場合、サブルーチンのネストレベルは高々3であり、コンテキスト切替えの高速化の効果がそのままシミュレーション速度の向上につながった。

評価結果をまとめると下記のことがいえる。

- FCSはサブルーチン呼出しのネストが5以下のところでコンテキスト切替えが起こる場合に有利である。
- コンテキスト切替え1回当たりの所要時間は、BCSと比較して最大で約15 μ秒改善されているので、1秒間に数千回以上コンテキスト切替えが起きるようなプログラムに対して適用を検討すべきである。
- FCSで最短の場合の所要時間は正味約4 μ秒であり、Sun3/260の約7 μ秒と比較すると不満が残るものの、処理内容から考えるとリーズナブルである。より高速化するには、SPARCアーキテクチャに「Window Invalid Mask Registerの参照命令を非特権命令とする」変更を加えるべきである。

6 あとがき

SPARCプロセッサにおける軽量プロセス間の高速なコンテキスト切替え方式を提案した。レジスタウインドウの退避と復旧の方法を改良することにより、従来のコンテキスト切替え方式と比較して、典型的な場合に数倍の高速化を達成した。ただし、提案した方式は、従来の方式と比較して遅くなる場合もあるので、適用範囲は限定される。この問題を本質的に解決するには、SPARCアーキテクチャの若干の仕様変更が望まれる。

謝辞

本研究は、通産省第五世代コンピュータプロジェクトの一環として、財団法人新世代コンピュータ技術開発機構の再委託を受けて行った。日頃ご指導頂く、ICOT第1研究室の渕室長と平田主任研究員に感謝いたします。

参考文献

[Sun90] SUN microsystems. Programming Utilities and Libraries, Chapter 2, Lightweight Processes.

[Pardo90] pardo@cs.washington.edu, USER-SPACE THREAD CONTEXT SWAPS ON THE Sun/SPARC.

[SPARC90] The SPARC Architecture Manual, Chapter 4.

[新城92] 新城, 清木, 並列プログラムを対象とした軽量プロセスの実現方式, 情報処理学会論文誌, Vol.33, No.1, pp.64-73.

[多田90] 多田, 寺田, 移植性・拡張性に優れたCのコルーチンライブラリー実現法, 電子情報通信学会論文誌, '90/12 Vol.J73-D-I, No.12.