ICOT Technical Memorandum: TM-1293

TM-1293

SPARC V8プロセッサにおける コルーチン間のコンテキストの 切替えの高速化（第2報）

酒井　浩、岩田　和秀（東芝）

March, 1994

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5
Institute for New Generation Computer Technology

SPARC™ V8 プロセッサにおけるコルーチン間のコンテキスト切替えの高速化（第２報）

A Faster User-space Context Switching on the V8 Sun/SPARC™

酒井 浩　岩出 和秀

sakai@isl.rdc.toshiba.co.jp　iwata@rds.rdc.toshiba.co.jp

(株) 東芝 研究開発センター

概要

コルーチンは，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シミュレーションする手段として使われてきた．しかしながら，SPARCプロセッサ [5] におけるコンテキスト切替えは，レジスタウィンドウの退避と復旧のコストが大きいため，遅いとされてきた．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はウィンドウの退避用のシステムコールを使っているために，レジスタウィンドウのメモリへの単純な格納に要する時間の２倍以上かかっていることを指摘し，非特権命令だけを用いた，より高速なコンテキスト切替えを提案する．従来方式との比較実験の結果，切替え時の手続き呼出しのネストが浅い場合は，新方式の方が常に高速で，Sun OS ™ の lwp ライブラリより最高で８．１倍，従来方式の中で最高速の方式より４．３倍程度高速であることがわかった。また，切替え時の手続き呼出しのネストが深い場合は，従来方式の方が高速な場合もあり，最悪の場合には，新方式は lwp ライブラリより４０%程度高速であるものの，従来方式の中で最高速の方式より２０%倍程度遅いことがわかった．

1　はじめに

コルーチンは，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シミュレーションする手段として使われてきた．しかしながら，SPARCプロセッサ [5] におけるコンテキスト切替えは，レジスタウィンドウの退避と復旧のコストが大きいため，遅いとされてきた．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はウィンドウの退避用のシステムコールを使っているために，レジスタウィンドウのメモリへの単純な格納に要する時間の２倍以上かかっていることを指摘し，非特権命令だけを用いた，より高速なコンテキスト切替えを提案する．

1.1　研究の動機

著者は，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開発機構からの再委託を受け，並列推論マシンＰＩＭ／ｋの研究開発に従事してきた。その中で，並列プログラムのデバッグやアーキテクチャの定量的な評価を行うツールとして，個々のプロセッサ，キャッシュメモリ等を命令実行レベルあるいはクロックレベルでシミュレートするソフトウェアシミュレータＳＳＦ（System Simulation Framework）

を開発した。ＳＳＦは，プロセッサ，キャッシュメモリ，共有バスといった各エージェントに対応するコルーチンが，それぞれ１クロック分の動作を実行するごとにラウンドロビンでコンテキスト切替えすることにより，システム全体の動作をシミュレートする．

このようなソフトウェアシミュレータでは，コルーチンを使用する代わりに，手続きの呼出しを行い，１クロック分の動作を行うと手続きから復帰するようなスタイルでプログラミング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手続きの中で１クロック分の動作が終了する場合，その続きの処理の記述やデバッグのし易さの点で，コルーチンの方が優れている．実際，ＰＩＭ／ｋ のアーキテクチャの評価用の並列キャッシュでは，最高で３段にネストした手続きの中でコンテキスト切替えを起こす場合がある．

初版ＳＳＦは Sun 3™ 用であり，コルーチン間のコンテキスト切替えを高速化するため，[8] のアセンブラ版 c_sched を使用した．その後，高速化を図るため，Sun 4™ への移植を行った。その際，コンテキスト切替えには Sun OS の lwp (light weight process) ライブラリ [3] の lwp_yield を採用した．その結果，期待に反して SPARC Station 2™ における実行速度が Sun3/260™ 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 (クロック速度 40MHz) で約３７μ秒であり，Sun3/260 におけるアセンブリ言語で記述したコンテキスト切替え手続きの所要時間約７μ秒より約５倍遅いことが分かった．そこで，Sun4 におけるコルーチン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　他の研究との関係

コルーチン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空間で行うものとに大別でき，本稿で述べる方式は後者に属する．

多田らは，大域ジャンプの機構を用い，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 [8]．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やコルーチン間のプリエンプトなコンテキスト切替のための機構を省略した効果である．SPARC プロセッサ用の Sun OS の大域ジャンプ用関数では後述するレジスタウィンドウの退避用システムコールが使われるため，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コルーチンを提案した [7]．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り，SPARC プロセッサに適用する場合には，レジスタウィンドウの退避用システムコールを使用するため，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 がスタック構造やレジスタウィ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 [4]．本稿では，この解説に基づいて作成した単純かつ最高速の方式ＢＣＳ（Basic Context Switch）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ＦＣＳ（Fast Context Switch）および，その改良版ＲＣＳ（Revised fast Context Swich）は，レジスタウィンドウの退避を含むすべての処理をシステムコールを使わず，非特権命令だけで実現するのが特徴である．

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ィンドウについて説明した後，ＢＣＳについて述べる．次に，ＦＣＳおよびＲＣＳについて説明し，性能評価および結果の考察を行う．

2　レジスタウィンドウの概要

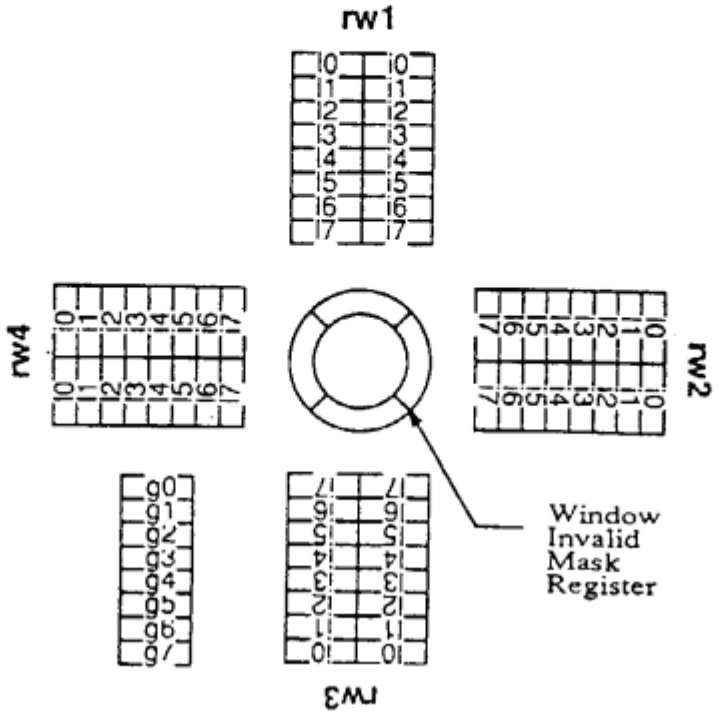


図 1: SPARC プロセッサのレジスタ構成

SPARCプロセッサのレジスタウィンドウを図1に基づいて説明する．一つのレジスタウィンドウは，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ィンドウがリングを構成している．図1では，rw1～rw4 という４個のレジスタウィンドウの場合を表している。これらのレジスタウィ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，ＷＩＭレジスタ（Window Invalid Mask Register）が備わっている．

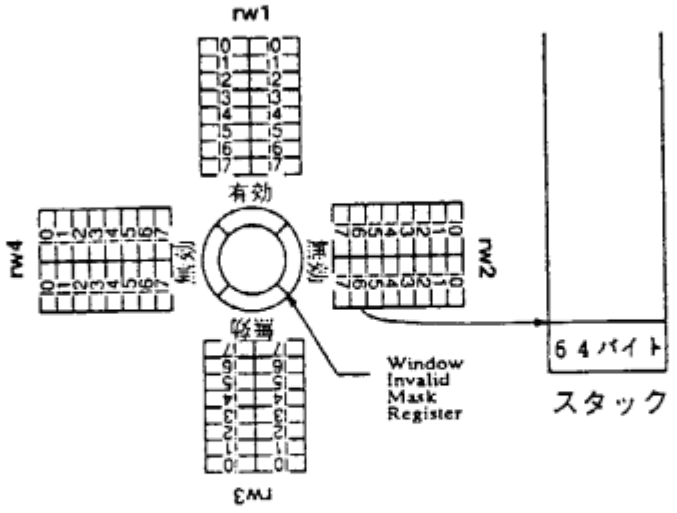
SPARCプロセッサでは，この他に o0～o7 と g0～g7 という１６個のレジスタ及び浮動小数点レジスタが使用可能である．このうち，o0～o7 は，隣のレジスタウィンドウの i0～i7 を指す．例えば，rw1 が選択されている場合に o0～o7 を参照すると，rw2 の i0～i7 が使用され，rw4 が選択されている場合に o0～o7 を参照すると，rw1 の i0～i7 が使用される．o6 は，スタックポインタとして使用される．一方，g0～g7 及び浮動小数点レジスタは，レジスタウィンドウとは別のレジスタ群であり，どのレジスタウィンドウが選択されているかに拘らず，常に同じレジスタ群が参照される．なお，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，ある一つのレジスタウィンドウが選択されており，ＷＩＭレジスタには，選択されたレジスタウィンドウが有効であると記録される．

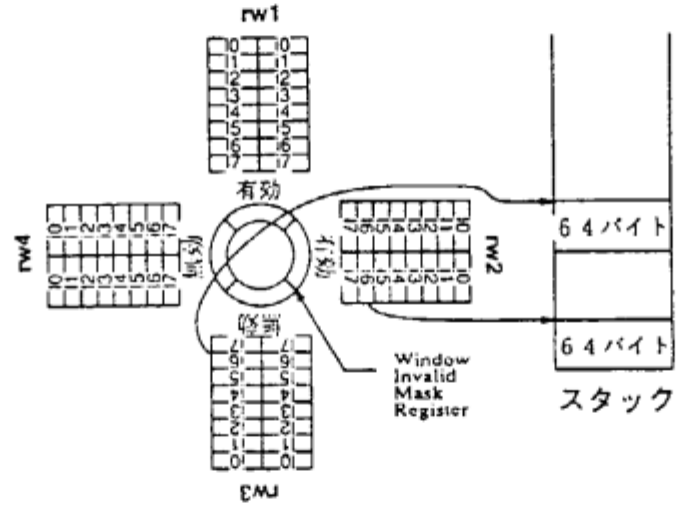
図 2(a) は，rw1 が選択されている場合を表わしており，ＷＩＭレジスタには，rw1 が有効であると記録される．[1]

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

[1] ただし，o0～o7 を保持する隣のウィンドウも使用されている



(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ィンドウの関係

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手続きを呼出す場合, 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.

手続きでは, 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, WIM レジスタには, rw1, rw2 が有効であると記録される.

手続きでは, rw2 の l0〜l7 と i0〜i7 のうち, rw1 のスタックポインタである i6 と, 手続きへの 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.

手続きからスタートルーチンへ復帰する場合, restore 命令を実行することにより, rw1 が選択されると

ともに，ＷＩＭレジスタ には rw1 が有効であると記録される．その後，rw1 の o6 の内容（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，レジスタウィンドウを持たないプロセッサにおいて必要であった手続き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，手続きの呼出しの高速化が期待できる．しかし，コルーチン間のコンテキスト切替えでは，レジスタウィンドウをそのコルーチン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，下記に示す処理であり，レジスタウィンドウの退避をシステムコールで行う従来のコンテキスト切替え機構のうちで最高速と考えられる．

ＢＣＳでは，手続き呼出しの前後で，g1〜g7，o0〜o5 の内容は破壊し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ィンドウの数が増えないようにしている．

- o6，o7 を現在実行中のコルーチンのコンテキスト格納領域に退避
- ST_FLUSH_WINDOWS システムコール[2]
- o6 ← ６４バイトの作業領域の先頭番地
- 次に実行すべきコルーチン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ィンドウに復旧
- o6 ← g1
- o7 ＋ ８番地へジャンプ

ST_FLUSH_WINDOWS システムコールは，現在選択中のレジスタウィンドウから順に restore 命令で次のレジスタウィンドウを選択しながら，ＷＩＭレジスタに有効と記録されているレジスタウィンドウの内容を対応するスタック領域に格納することにより行う．

残りの処理は，実行を再開するコルーチンについて，実行中であった手続きに対応するレジスタウィンドウだけを復旧する処理である。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，レジスタウィンドウの復旧途中で，タイマ割込み等の発生により，元のコルーチンのスタック領域が破壊されるのを防ぐためである．

この復旧処理により，実行再開の時点では，ＷＩＭレジスタには，i0〜i7 に対応するレジスタウィンドウのみが有効であると記録される．コンテキスト切替え後，コルーチンが手続きから呼出し側へ復帰のため restore 命令を実行すると，レジスタウィンドウ・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ィンドウの復旧が行われる．

また，実行を再開したコルーチンが 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う場合，ST_FLUSH_WINDOWS システムコールでは現在選択中のレジスタウィ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．

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

4　新しい方式（ＦＣＳおよびＲＣＳ）

4.1　ＦＣＳ

ＦＣＳは，レジスタウィンドウの退避と復旧を非特権命令だけで行う．ＦＣＳの骨格部分は，次のとおりである．

- o7 を現在実行中のコルーチン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ィンドウを退避
- restore
- i6 とゼロを比較し不一致であれば，L1 へジャンプ
- g4 を現在実行中のコルーチンのコンテキスト格納領域に退避
- 実行を再開すべきコルーチン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 [3]

- 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ィンドウを復旧
- g3 と g4 を比較し不一致であれば，L2 へジャンプ
- o6 ← g3
- o7 を復旧
- o7 + 8 番地へジャンプ

前半のレジスタウィンドウ退避フェーズでは，本来ＷＩＭレジスタ に有効と記録されているレジスタウィ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使用できない。そこで，現在選択中のレジスタウィンドウからスタートルーチンに対応するレジスタウィンドウまで，すべてを退避する．スタートルーチンに対応するレジスタウィ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，そのレジスタウィンドウについては退避と復旧を省略している．なお，このフェーズで各レジスタウィンドウの o6 の内容を g4 が指すスタック領域に格納しているのは，後半のレジスタウィンドウ復旧フェーズで退避と逆順に復旧するのに必要なためである．

後半のレジスタウィ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ィンドウから最近に呼出された手続きに対応するレジスタウィンドウまですべてのレジスタウィ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ィンドウを退避する際に，レジスタウィンドウ・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，レジスタウィンドウの復旧途中でタイマ割込み等の発生により，元のコルーチンのスタック領域が破壊されるのを防ぐためである．

[3] 新しく選択されたレジスタウィンドウの o6 に g1 の内容がコピーされる

4.2 ＦＣＳの長所および問題点

ＦＣＳの長所は，ＳＳＦのように手続き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ＢＣＳより高速なことである．それは，下記の理由によるものと思われる．

- レジスタウィンドウの退避では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．
- 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．

一方，問題点は，次の二つである。第一に，スタートルーチンをi6の内容がゼロか否で判定しているため，将来のＯＳのバージョンアップにより動かなくなるおそれがあること，第二に，使用中のレジスタウィ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ることである．これらを回避する方法は，２通り考えられる．

第一の方法は，SPARC アーキテクチャに「ＷＩＭレジスタの参照命令を非特権命令とする」変更を加えることであり，上記二つの問題点を同時に解決できる。

第二の方法は，レジスタウィンドウの個数を予め求めておき[4]レジスタウィンドウの退避フェーズの終了条件を「スタートルーチンに達するか，あるいは，すべてのレジスタウィンドウを退避するまで」と変更することであり，第二の問題点をある程度回避できる。例えば，レジスタウィンドウの数が８の場合，手続き呼出しのネストが７以上の場合，現在選択中のレジスタウィンドウを退避した後，restore 命令実行とレジスタウィンドウの退避の組をを６回繰返すところまでで打ち切れば良い。また，レジスタウィンドウの復旧フェーズも実行を再開するコルーチンの手続き呼出しのネストが７以上の場合には，最近の７個の手続きに対応するレジスタウィンドウの復旧を行えば良い。このアイデアは，慶応大学の萩谷先生に指摘頂いた．

この方式をＲＣＳ（Revised Fast Context Switch）と呼ぶことにする。なお，ＢＣＳ，ＦＣＳ，ＲＣＳのコンテキスト切替え部およびコンテキスト初期化のソースプログラムリストを付録に示す．

5 性能評価と考察

5.1 従来方法との比較

ＦＣＳ，ＲＣＳ，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つのコルーチン間でコンテキスト切替えを繰返すプログラムを作成し，SPARC Station 2 (40MHz) 上で実行時間を測定した．各コルーチンは，下記の処理を繰返す．

「手続き呼出しがＬだけネストした状態から，ネストがＣに達するまで手続き呼出しを繰返し，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になるまで手続きからの復帰を繰返す」

ただし，ネストがＩであるとは，コルーチン本体に対応する手続きが動作している状態を表わす。この時，その手続きは，スタートルーチンから呼び出されているので，使用中のレジスタウィンドウの数は２である。

上記プログラムの実行時間には，手続き呼出しや繰返し回数に達するまでのループが含まれるため，コンテキスト切替えの部分だけを除いたプログラム（本稿ではＮＣＳと呼ぶことにする）についても測定した．ただし，SPARCプロセッサでコンテキスト切替えの時間の評価では，コンテキスト切替えに要する時間とと

[4]後述する性能評価により，求めることができる

もに手続き呼出しに伴うレジスタウィンドウのアンダーフローおよびオーバフローのトラップ処理に要する時間もあわせて評価する必要がある。

コンテキスト切替え時間の測定方法は,10,000 回のコンテキスト切替えに要する時間を gettimeofday 関数を利用し,経過時間を精度１μ秒で求めた。タイマ割込みやイーサネットを流れるメッセージの受信処理により,測定結果に誤差が含まれる可能性をなるべく避けるため,１００回の測定を行い,その最小値を採用した。この測定結果には,手続き呼出しの時間が含まれている。

Ｌ＝１の場合について,Ｃを１から１１まで変化させた場合の手続き呼出し／復帰,およびコンテキスト切替え時間の合計を図3に示す。

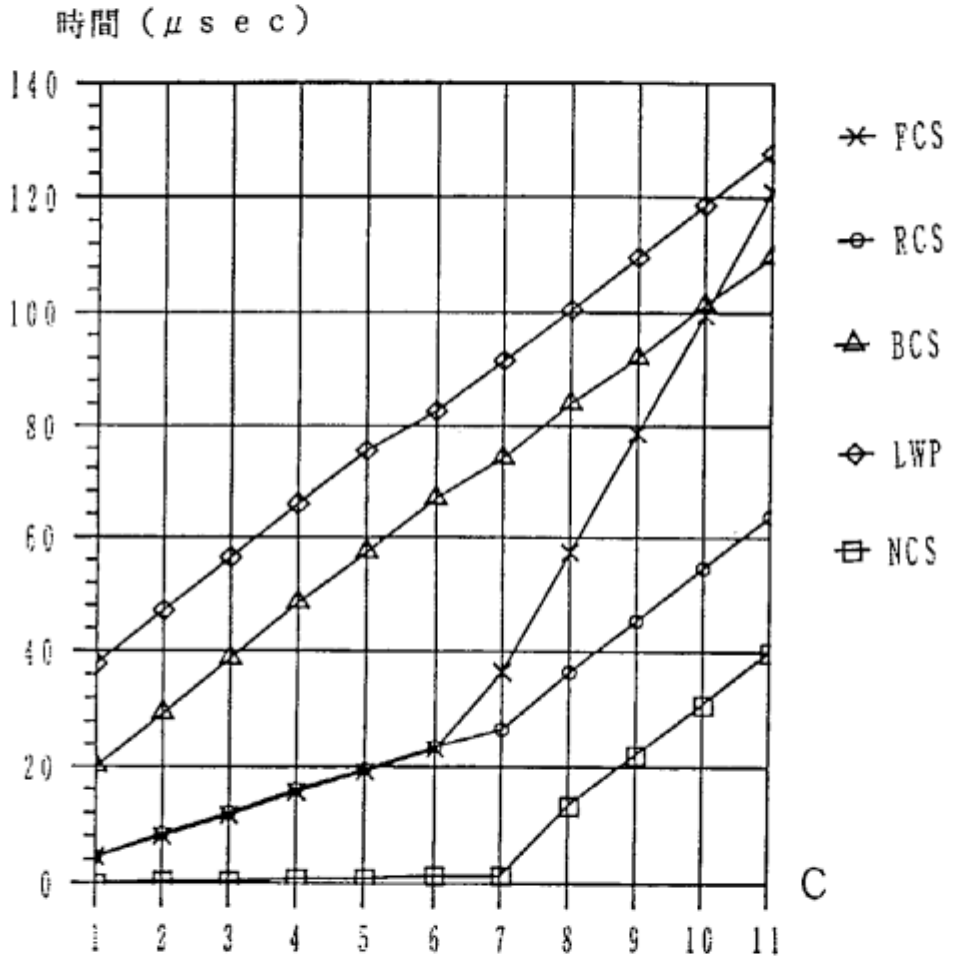


図 3: Ｌ＝１の場合の新方式と従来方式の比較

- ＮＣＳ,すなわち,コンテキスト切替えが無い場合の所要時間は,Ｃ＝７を境として急激に増加している.これは,レジスタウィ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.Ｃ＝７の場合,使用されるレジスタウィンドウの数はネストされた手続きの数（＝７）＋１であることから,本マシンのレジスタウィンドウの数は８であることがわかる。また,Ｃ≧７におけ

る直線の傾きと，C≦7における直線の傾きの差から，オーバフロートラップとアンダフロートラップの処理時間の合計は，約8．6μ秒であることがわかる．

- ＢＣＳ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．ＢＣＳの場合，退避すべきレジスタウィンドウの数は，ｍｉｎ（C，7）であり，レジスタウィンドウのオーバフロートラップの発生回数がｍａｘ（0，C－7），アンダフロートラップの発生回数が（C－1）である。従って，ほぼ直線的に増加するということは，オーバフロートラップ1回の処理時間とレジスタウィンドウの退避用システムコールにおけるレジスタウィンドウ1個分の退避処理がほぼ等しいことを示している。

- lwp_yield の所要時間は，すべてのCについて，ＢＣＳの処理時間より約17．7μ秒大きい．この差は，手続きの入口と出口で save 命令と restore 命令を実行するため，退避すべきレジスタウィンドウ数が1多いこと，および，lwp_yield のスケジューリング処理が複雑なためである。

- ＦＣＳの場合，所要時間は，C≦6でCの増加にともない直線的に増加し，6≦C≦7で傾きが急になり，C≧7では，その傾きがさらに急になっている。C≦6では，C個のレジスタウィンドウを非特権命令で退避および復旧する処理が大半を占める。レジスタウィンドウ1個の退避および復旧の所要時間は，約3．6μ秒であることがわかる。また，C＝7の場合，ネストが7の手続きを呼出しを呼出すところで，スタートルーチンに対応するレジスタウィンドウと競合するため，オーバフローが1回発生し，コンテキスト切替えにおいて，スタートルーチンに対応するレジスタウィンドウを参照する時にアンダフロートラップが1回発生する。また，C＞7では，Cが増加するごとに，オーバフローとアンダフロートラップの発生回数が2ずつ増加する。

- ＲＣＳの所要時間は，C≦7でCの増加にともない直線的に増加している。ＦＣＳと異なり，コンテキスト切替え時に，レジスタウィンドウのオーバフローとアンダフロートラップが発生しないため，C≧7における直線の傾きは，手続き呼出し／復帰の際のオーバフローとアンダフロートラップ処理に対応する。また，C≦6の範囲で所要時間をＦＣＳと比較すると，約0．3μ秒大きく，その理由は，終了条件の判定が複雑になったためである。

- C＜7におけるＢＣＳとＦＣＳの傾きの差（約5．6μ秒）は，レジスタウィンドウ1個分の退避と復旧を，システムコールとアンダフロートラップで行った場合と，相当の処理を非特権命令で行った場合の差である。従って，SPARC プロセッサにおいてコンテキスト切替えが遅いとされてきたが，これはレジスタウィンドウというアーキテクチャの特徴だけでなく，その退避と復旧の方式にも起因するものと言える。

次に，L＝Cの場合について，Cを1から11まで変化させた場合のコンテキスト切替え時間を図4に示す。

- ＢＣＳ および lwp_yield の処理時間は，Cに関係無くほぼ一定である．これは，レジスタウィンドウの退避用システムコールにおいて，ＷＩＭレジスタに有効と記録されているレジスタウィンドウの数が，ＢＣＳの場合は1，lwp_yield の場合は2であるためである．両者の処理時間の差は，L＝1の場合と同じである。

- ＦＣＳの所要時間は，C≦6の範囲でCが増加するのは，退避と復旧を行うレジスタウィンドウの数がCに等しいためである。C≧6の範囲で傾きが急になるのは，Cが1増加するごとに，コンテキスト切替え時にレジスタウィンドウのアンダーフローおよびオーバフローの発生回数が1ずつ増加するためである。

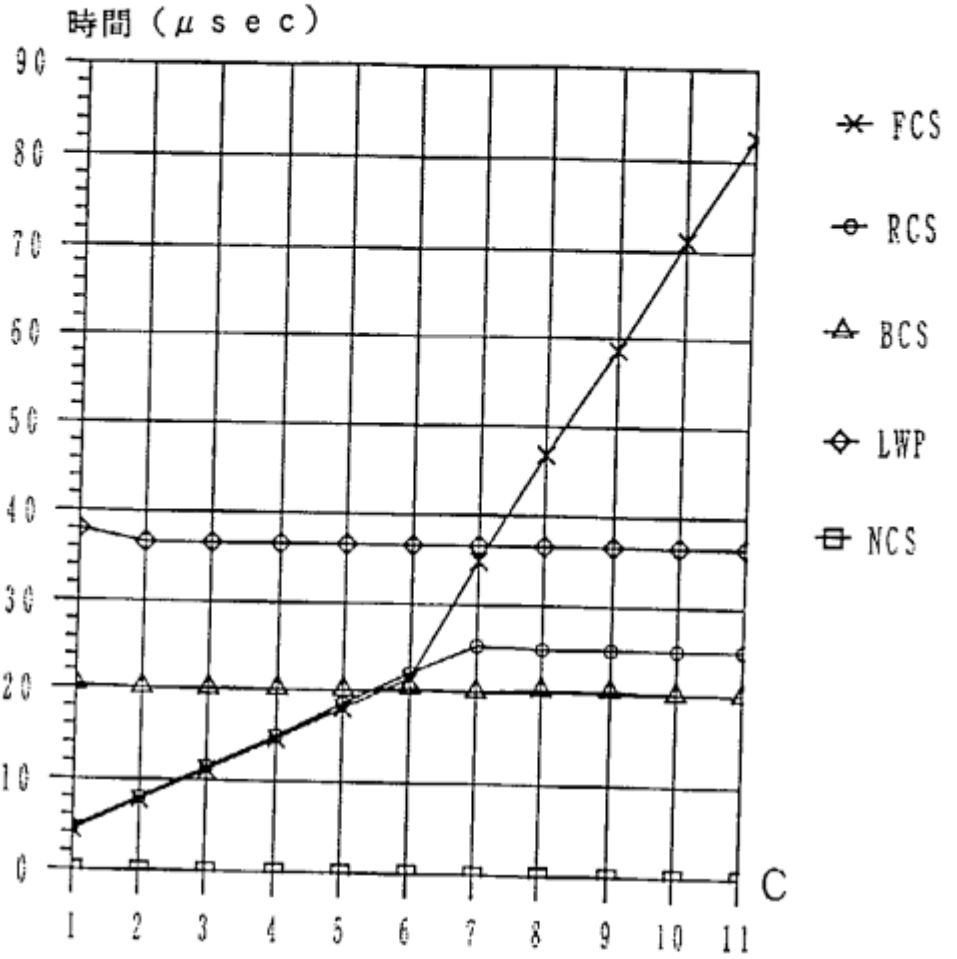


図 4: L＝Cの場合の新方式と従来方式の比較

- ＲＣＳの所要時間が，C≧７ではほぼ一定であるのは，退避と復旧を行うレジスタウィンドウの数がmin（L，7）であることによる。

L＝Cの場合は，退避および復旧の対象となるレジスタウィンドウの数が少ない従来のコンテキスト方式に有利といえる。しかし，ＦＣＳおよびＲＣＳは，C≦５の範囲においてＢＣＳより高速であり，また，C≧７の範囲でも，7個のレジスタウィンドウの退避と復旧を行うＲＣＳのコンテキスト切替えの所要時間は約２３．６μ秒であり，1個のレジスタウィンドウの退避と復旧を行うＢＣＳの所要時間，約１８．９μ秒と比較して，２０%程度の遅さでおさまっている。また，lwp ライブラリと比較すると，約４０%高速である。

5.2 SSF への適用と評価のまとめ

ＦＣＳ を SSF に適用したところ，lwp_yield を使用した場合と比較して，実行速度が３～５倍になった．効果が変動するのは，シミュレーションモデルにより，コルーチン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の処理時間が異なるためである．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４０％以上がコンテキスト切替えに費やされている．SSF の場合，手続き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およびＲＣＳは,SPARC Station 2 の場合，手続き呼出しのネストが５以下のところでコンテキスト切替えが起こる場合には，従来の方式より常に高速である．特に，ＲＣＳは，最悪の場合でも，従来の最高速の方式より，２０％程度の遅さであり，ほとんどの場合に適用できる可能性がある。
- 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１６μ秒改善されており，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効果がある．
- ＦＣＳおよびＲＣＳで最短の場合の所要時間は正味約４μ秒であり，Sun3/260 の約７μ秒と比較して不満は残るが，処理の内容から考えて，やむを得ない。

6　あとがき

SPARC プロセッサにおけるコルーチン間のコンテキスト切替えを実現する新しい方式を提案した．従来のコンテキスト切替え方式と比較して，切替え時の手続き呼出しのネストの深さが小さい場合は，新方式は,Sun OS の lwp ライブラリより最高で７．２倍，従来方式の中で最高速の方式より３．８倍程度高速であることがわかった。また，最悪の場合，新方式は lwp ライブラリより４０％程度高速であるものの，従来方式の中で最高速の方式より２０％倍程度遅いことがわかった．新しい方式を，従来方式より常に高速とするためには，ＷＩＭレジスタの参照を非特権命令とする SPARC アーキテクチャの変更が必要である。

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，新世代コンピュータ技術開発機構（ICOT）から再委託を受けて行った。研究開発のご指導頂いた,ICOT の瀧室長（現，神戸大学助教授）と平田研究員（現,NTT）に感謝いたします。また，コンテキスト切替え方式の改良案を示唆して頂いた慶応大学の荻谷先生に感謝いたします．

参考文献

[1] Taki K.,Parallel Inference Machine PIM, Proceedings of the International Conference on Fifth Generation Computer Systems 1992,pp.50-72.

[2] Sakai H.,et al,Applying Inter-cluster Shared Memory Architexture to a Parallel Inference Machine, Parallel and Distributed Computing in Engineering Systems, pp.3-8,1992.

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

[4] pardo@cs.washington.edu, User-space Thread Context Swaps on the Sun/SPARC.

[5] SPARC アーキテクチャマニュアル バージョン 8，株式会社 トッパン．

[6] 酒井，浅野，武脇，並列推論マシン，東芝レビュー 1992,Vol.47,No.7,pp.521-523.

[7] 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.

[8] 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.

付録

```
/*
    SPARC V8 用のコルーチン間のコンテキスト切替えルーチン

        programmed by Hiroshi Sakai

        このソースプログラムは，本文中のＢＣＳ，ＦＣＳ，ＲＣＳに対応する。
        アセンブルする時，次の３行のうち，いずれかひとつを有効とすることに
        より，各々に対応するオブジェクトプログラムを生成できる。

#define BCS
#define FCS
#define RCS
*/
/*
        アセンブルは，例えば，as -P -DBCS で行う。

        下記の値は，ＲＣＳの場合に必要となるレジスタウィンドウの個数である。
        ＳＳ２に関しては，８であることが確認されている。
*/
#ifdef RCS
        nrw = 8       /* ターゲットマシンのレジスタウィンドウの数 */
#endif
/*
    Ｃプログラムとのインタフェース

    データ構造

        struct env {
            long *pc_when_resuming;
                コルーチンの実行再開時のプログラムカウンタ
            long *stack_frame_address;
                レジスタウィンドウの復旧に必要なスタック上のアドレス
            long nesting_level;
                レジスタウィンドウ数ーセーブされたレジスタウィンドウ数
                ＲＣＳの場合のみ使用
        };
        名称    R構造体
        役割    各コルーチンの実行再開に必要な情報を格納する構造体
        注釈    これ以外の情報は，すべてスタック中にとられる。
```

関数

```
        void mainctxinit(struct env *envp)
        機能    メインのコルーチン用の初期化ルーチン
        引数    envp      本コルーチン用の実行再開情報格納領域
                          （Cプログラム側で確保）へのポインタ

        void ctxinit(void (*func)(), char *stack, struct env *envp)
        機能    メイン以外の各コルーチン用の初期化ルーチン
        引数    func      本コルーチンに対応する関数へのポインタ
                stack     スタック領域（Cプログラム側で確保）の
                          底のアドレス（先頭アドレス＋サイズ）
                envp      本コルーチン用の実行再開情報格納領域
                          （Cプログラム側で確保）へのポインタ

        void ctxsw(struct env *new_envp)
        機能    現在のコルーチンの実行を中断し，引数で指定された
                R構造体に対応するコルーチンの実行を再開する
        引数    new_envp 本コルーチン用の実行再開情報格納領域
                          （Cプログラム側で確保）へのポインタ
```

本サブルーチンに関する全般的な注意

- 各コルーチンのスタックの最も底のレジスタウィンドウに関しては，オールゼロであることを前提に，その退避／復旧を省略している。
- プリエンプションはできない。
- 各コルーチンに関数引数としてパラメータを与えることはできない。

```
*/
        .text

        .globl  _ctxsw
        .globl  _mainctxinit
        .globl  _ctxinit

        .seg    "bss"
        .align  8
regw:   .skip   64          ! レジスタウィンドウの復旧処理時に割込みが
                            ! 発生した場合に，カレントのレジスタウィンドウが
                            ! 破壊されることを防ぐための作業領域
```

```
self:   .skip   4               ! 現在実行中のコルーチンのR構造体へのポインタ

        .seg    "text"

_ctxsw:

#ifdef BCS
        sethi   %hi(self), %g2
        ld      [%g2 + %lo(self)], %g1   ! g1 <= self
        st      %o0, [%g2 + %lo(self)]   ! g2 <= new_envp
        st      %o7, [%g1]               ! g1->pc_when_resuming <= pc
        st      %sp, [%g1 + 4]           ! g1->stack_address <= sp
        ta      0x03                     ! ST_FLUSH_WINDOWS
        sethi   %hi(regw), %g2
        add     %g2, %lo(regw), %sp
        ld      [%o0 + 4], %g1           ! g1 <= new_envp->stack_addres
        ld      [%o0], %o7               ! o7 <= new_envp->pc_when_resuming
        ldd     [%g1 + 0x00], %l0
        ldd     [%g1 + 0x08], %l2
        ldd     [%g1 + 0x10], %l4
        ldd     [%g1 + 0x18], %l6
        ldd     [%g1 + 0x20], %i0
        ldd     [%g1 + 0x28], %i2
        ldd     [%g1 + 0x30], %i4
        ldd     [%g1 + 0x38], %i6
        jmp     %o7 + 8                  ! resume the callee
        mov     %g1, %sp

#else                                    ! in case of FCS or RCS

        sethi   %hi(self), %g4
        ld      [%g4 + %lo(self)], %g1   ! g1 <= self
        mov     %o0, %g2                 ! g2 <= new_envp
        st      %o0, [%g4 + %lo(self)]       ! self <= new_envp
        st      %o7, [%g1]                ! g1->pc_when_resuming <= pc
        sub     %sp, 4, %g4
        st      %sp, [%g4]
#ifdef RCS
        mov     nrw - 1, %g5
#endif
!
!       register window save phase
```

```
!
1:
        std     %l0, [%sp + 0x00]
        std     %l2, [%sp + 0x08]
        std     %l4, [%sp + 0x10]
        std     %l6, [%sp + 0x18]
        std     %i0, [%sp + 0x20]
        std     %i2, [%sp + 0x28]
        std     %i4, [%sp + 0x30]
#ifdef RCS
        deccc   1, %g5
        be      2f
#endif
        std     %i6, [%sp + 0x38]
        restore
        dec     4, %g4
        cmp     %fp, %g0
        bne     1b
        st      %sp, [%g4]
2:
        st      %g4, [%g1 + 4]
#ifdef RCS
        st      %g5, [%g1 + 8]
#endif
        sethi   %hi(regw), %g1
        ld      [%g2 + 4], %g4
#ifdef RCS
        ld      [%g2 + 8], %g5
#endif
        ld      [%g4], %g3
        inc     4, %g4
#ifdef RCS
        cmp     %g0, %g5
        be      5f
        add     %g1, %lo(regw), %sp
#endif
        cmp     %g3, %g4
        be,a    4f
        ld      [%g2], %o7
!
!       register window restore phase
!
```

```
        save    %g1, %lo(regw), %sp
3:
        ld      [%g4], %g3
        inc     4, %g4
#ifdef RCS
5:
#endif
        ldd     [%g3 + 0x38], %i6
        ldd     [%g3 + 0x30], %i4
        ldd     [%g3 + 0x28], %i2
        ldd     [%g3 + 0x20], %i0
        ldd     [%g3 + 0x18], %l6
        ldd     [%g3 + 0x10], %l4
        ldd     [%g3 + 0x08], %l2
        ldd     [%g3 + 0x00], %l0
        cmp     %g4, %g3
        bne,a   3b
        save    %sp, %g0, %sp       ! this delayed instruction is
                                    ! executed only if the conditional
                                    ! branch is taken.
        ld      [%g2], %o7
4:      jmp     %o7 + 8             ! resume the target coroutine
        mov     %g3, %sp
#endif

_mainctxinit:
        sethi   %hi(self), %g1
        jmp     %o7 + 8
        st      %o0, [%g1 + %lo(self)]

_ctxinit:
        dec     8, %o0
        st      %o0, [%o2]
        dec     0x60, %o1           ! alloc a register window save area
        and     %o1, -8, %o1        ! assure 8 byte alignment
        mov     %g0, %g1

        std     %g0, [%o1 + 0x00]   ! clear the stack bottom
        std     %g0, [%o1 + 0x08]
        std     %g0, [%o1 + 0x10]
        std     %g0, [%o1 + 0x18]
        std     %g0, [%o1 + 0x20]
```

```
        std     %g0, [%o1 + 0x28]
        std     %g0, [%o1 + 0x30]
        std     %g0, [%o1 + 0x38]
#ifdef BCS
        st      %o1, [%o2 + 4]
#else
        st      %o1, [%o1 - 4]
        dec     4, %o1
        st      %o1, [%o2 + 4]
#ifdef RCS
        st      %o1, [%o2 + 8]
#endif
#endif
        jmp     %o7 + 8             ! resume the callee
        nop
```