並列コンパイラ Compas の意味処理部の性能評価†

西山博泰†† 板野肯三†††

並列コンパイラ Compas の意味処理部は，動的に生成される意味処理プロセスの集合として実現されている．この意味処理の動作状況を正確に把握するために，プロセス管理に関する処理を行う部分を仮想マシンとして分離することにした．この仮想マシンは独立したプロセスを実行する環境であり，プロセスの動的な生成と共有メモリによるプロセス間でのデータの共有，および，ストリームを用いた1対1，多対1のプロセス間の通信機構を意味処理部に提供する．仮想マシンの実現は SPARC を CPU として用いた密結合型マルチプロセッサ・アーキテクチャ SMiS と，その上で仮想マシンのプリミティブを実現するスレッド・ライブラリとに分けて行った．SMiS シミュレータを作成して動作の測定を行った結果，Compas-PL/0 では意味処理を行う仮想マシンのプリミティブのうち，スケジューリングやストリームによる仮想マシン部の実行が性能面でのボトルネックとなっており，このような仮想マシン実行の大部分を占めているプロセス管理に関する詳細な情報が得られた．

1. はじめに

コンピュータのハードウェアの分野では，プロセッサの高速化，メモリの大容量化が進んでおり，これにともなってソフトウェアも大規模なものが許容されるようになってきた．このため，プログラムを開発するコストの中で，コンパイルによって占められる割合も無視できなくなりつつある．例えば，Xウィンドウ・システムや emacs などでは，システムの再構築のためのコンパイルだけでも，数時間のオーダの時間が必要である．

これまで，著者らはこのような状況を改善するための一つの方法として，コンパイラ内部の処理を並列化してコンパイルを高速化するための研究を行ってきた．コンパイラの内部処理のうち，字句解析処理や，構文解析部はハードウェアで高速化することが可能であり[1),2)]，コード生成フェーズについても構文解析と同様の手法を用いたパターンマッチによる手法が知られている[3)]．しかし，最適化処理を含む意味処理は，対象とする言語によってその扱う内容が大幅に異なることから，表駆動のように形式には向かない．したがって，意味処理に関しては，ストリームを用いた並列意味処理モデルを新たに考案し，このモデルに基づいて動的に並列性を抽出して密結合型のマルチプロセッサ上で並列処理を行う手法を考案した．

現在採用している並列意味処理のモデル[4)]では，コンパイラの構文と意味の定義を行う際に，解析木のノードに対応して意味処理を行うプロセスを定義し，同時に意味処理を行うプロセス間の通信路であるストリームの結合関係を文法に沿って与えることで，意味処理の記述を行う．このように，意味処理を並列に動作するプロセスの相互作用として定義することにより，並列に実行される意味処理の記述を容易にしている．また，意味解析プロセス間のデータの受渡しにストリームを用いることにより，データに内在する並列性の抽出を動的に行うことを可能としている．実際のコンパイル時には，意味解析部では制御プロセスと呼ばれる特殊なプロセスが，構文解析部で行われた構文解析動作に関する情報と，トークンに付随した属性値を構文解析器から受け取り，解析の終わったノードに対して意味解析処理を行うプロセスが定義されていればその生成を動的に行い，プロセス間を結ぶストリームの結合を行う．生成された各プロセスは自立的に動作し，各プロセスがストリームを介して通信を行うことにより意味解析処理を行う．

このようなモデルによって表現した PL/0[5)] の並列コンパイラ Compas-PL/0 の初版では，SUN OS 上で Light Weight Process[6)] を時分割処理することによって，仮想的なプロセッサ上で実行されるプロセスの切り替えを行い，意味解析プロセスの実行状態の変化からプロセッサ稼働率を予測した[7)]．これによって意味処理の並列実行モデルの実現可能性を検証することはできたが，コンパイラの意味処理部の高速な実現やハードウェアによるサポートなどを考えるためには実際のハードウェアシステム上での意味処理実行時の動的な特性の精密な評価が不可欠である．

そこで，この問題に対応するためのマルチプロセッ

† Evaluation of a Semantic Analyzer of a Parallel Compiler Compas by HIROYASU NISHIYAMA (The Doctorial Program of Engineering, Graduate School, University of Tsukuba) and KOZO ITANO (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学大学院工学研究科
††† 筑波大学電子・情報工学系

サアーキテクチャ SMiS の命令レベルのシミュレータを開発して，クロック単位でのプロセスの実行時間を測定することにした．さらに，並列意味処理そのものと，プロセス間通信やプロセス管理のための処理を分離して測定することを容易にするために，前者をプロセスの“実行環境”である仮想マシンとして抽象化し，この仮想マシンの実行時間と意味解析処理の時間を分離して測定した．

この測定の結果，プロセスの粒度が小さいことに起因したプロセス管理のオーバヘッドにより，単一プロセッサで逐次処理向きに開発された意味処理器に対して総合的な性能を凌ぐことは現時点ではできなかったものの，内部の細かなプロセスの動的特性を観測することができ，細粒度並列処理に関する意味のある知見が得られた．本論文では，今回設計した仮想マシンの実現と，シミュレーションの結果に基づいた並列意味処理器の動作特性について説明する．

2. 並列意味処理用仮想マシン

Compas で採用している意味処理のモデルでは，入力プログラムに対応して意味処理を行うプロセスを動的に生成し，各プロセスがストリームを介して通信を行うので，並列処理用の仮想マシンではこれに対処する機構を用意している．これらの機能を含めて，すべてをハードウェア化することは不可能ではないが，当面は，特殊なハードウェアのメカニズムは用いないで，ごく一般的なハードウェアの構成を前提とすることにしたので，特殊な機能はソフトウェアで実現することにした．

2.1 プロセス生成プリミティブ

仮想マシンは意味処理を行うプロセスに対してメモリ，CPU，プロセス間の通信機構などの実行環境を提供するものであり，1つのプロセスに対して1つの仮想マシンが対応して存在するものと考える．プロセス側からは各仮想マシンは独立して動作するように見え，意味処理部ではスケジューリングや実際のプロセッサ数などの実現の詳細に関しては考慮する必要はない．

仮想マシンにはプロセスの生成と消滅を行うために，次の2つのプリミティブがある．

- create_process (proc, arg, …)

新たにプロセスと仮想マシンの組を生成し，procで指定した手続きを実行する．

- destroy_process ( )

現在実行中のプロセスと仮想マシンを消す．

UNIX の fork などの場合とは異なり，生成するプロセスと生成されたプロセスとで環境の継承などは行わないが，生成時に引数としてストリーム（後述）を与え，これで仮想マシンの間を接続する．プロセスは仮想マシンに接続されているポートにアクセスするだけで，ストリームの接続は仮想マシン間で自動的に行われる．

2.2 メモリ管理プリミティブ

各プロセスに対応した仮想マシンは全仮想マシンで共有されるメモリと，各仮想マシンに固有なメモリを持つ．共有メモリ空間はすべての仮想マシンで同一のアドレスに存在する．メモリ管理のためのプリミティブとしては，共有メモリと局所メモリのそれぞれに対する確保と解放を行うため，次の4つのプリミティブを用意する．確保された共有メモリは，すべての仮想マシンから同一アドレスでアクセスすることが可能である．

- malloc (size)

sizeで指定された大きさの領域を局所メモリに確保し，そのアドレスを返す．

- mfree (address)

局所メモリ中に確保されている address で指定した領域を解放する．

- shmalloc (size)

sizeで指定された大きさの領域を共有メモリに確保し，そのアドレスを返す．

- shmfree (address)

共有メモリ中に確保されている address で指定した領域を解放する．

共有メモリは，意味処理を行う各プロセス間で，記号表内のデータや中間コードの受け渡しを効率良く行うために使用し，仮想マシンに固有なメモリはプログラム・テキスト，スタック，局所データなどを格納するために使用する．共有メモリをプロセス間の低レベルの通信機構とすることも可能であるが，同期やアクセスの制御が煩雑になるため，仮想マシンのレベルでは共有メモリはデータの共有機構としてのみ用い，プロセス間の通信機構としては，次に説明するストリームを用意する．

2.3 プロセス間通信プリミティブ

仮想マシンレベルでのプロセス間の通信機構としてはストリームによる同一の通信路を介した通信機能を提供する．ここでのストリームは単方向の通信を可能

とする抽象データであり，プロセス間のデータの転送や通信路の結合は仮想マシンのプリミティブより行われる．ストリームは以下に示す仮想マシンのプリミティブにより動的に生成され，生成されたストリームは，プロセス生成時に手続きの引数として渡され，プロセスに結合される．

・create_streem ()

新たなストリームを生成し，そのストリームの識別子を返す．

まず，1対1のプロセス間の通信に用いられるストリームへの送信と受信，およびストリーム中のデータの存在を確認するための操作を示す．1対1の通信は主に中間コードの受け渡しなど1方向のプロセス間通信に使用する．ただし，CSP[8]に見られるようなデータのコピーを行うような通信はデータの大きさが1ワードに収まるものだけに制限し，より大きなデータについては共有メモリ上のデータの格納アドレスのみを受け渡す．このためのプリミティブは次の3つである．

・send_stream (stream, data)

stream で指定したストリームを通して data で指定したデータを送る．

・recv_stream (stream)

stream で指定したストリームからデータを受け取る．

・poll_stream (stream)

stream で指定したストリームにデータがあるかどうかを確認する．

2つのストリームを組にして複数のプロセスで共有することにより，多対1の通信を行うことも可能である．ストリームによる多対1の通信は意味処理のレベルでクライアント・サーバ型の通信を実現するためのもので，複数のプロセスが同一のストリームを介して1つのプロセスに要求を送り，結果を受けとることを可能にする．この多対1の通信は意味処理部のレベルでは記号表の管理などに使用する．このための仮想マシンプリミティブとして，サーバへの要求の送信と結果の受信をアトミックに行う操作を提供する．

・send_and_recv_stream (stream1, stream2, data)

stream1 で指定したストリレームを通してサーバ・プロセスに data で指定した要求を送り，サーバからの結果を stream2 から受けとる．

サーバ側では，recv_stream によりクライアントからの要求を待ち，send_stream により結果を返す．クライアントは send_and_recv_stream によりサーバに要求を送り，結果を受け取る．send_and_recv_stream はアトミックなプリミティブであるため，複数の要求があった場合には排他制御が行われる．このため，send_and_recv_stream を用いた場合には，2つのストリームは実質的には大きさ0の双方向のキューと同じ働きをする．

また，通信路の構成を動的に変更することを可能とするために，ストリーム自身をストリームを介して受け渡すことを可能とする．ストリーム中のデータがストリームかどうかは次に示す操作によって確認することができる．この操作により，ストリームをストリームのデータとして受け渡すことができるようになり，通信路の構成を動的に変更することが可能になる．

・is_stream (data)

data で指定されたデータがストリームかどうかを確認する．

3. 仮想マシンの実現

2章で述べた仮想マシンをハードウェア部とソフトウェア部に分けて実現した．ここでは，仮想マシンのハードウェア部に対応するマルチプロセッサ・アーキテクチャ SMiS の構成と，ソフトウェア部に対応するスレッド・ライブラリの実現について説明する．

3.1 マルチプロセッサアーキテクチャ SMiS

2章で述べた仮想マシンのハードウェア部の実現としてマルチプロセッサのハードウェアアーキテクチャ SMiS の設計を行った．

SMiS で想定しているハードウェアは SPARC CPU[9]とローカルメモリを持った複数台のプロセッシング・エレメント (PE) が共有バスを介して共有メモリに接続されている密結合システムである（**図1**）．

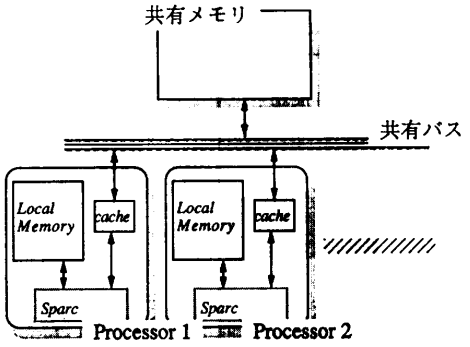


図 1 SMiS (Sparc Multi-Processor System) の構成
Fig. 1 Organization of SMiS (Sparc Multi-Processor System).

共有メモリに対するキャッシュには，プロセッサ間のメモリ・アクセスの競合を低く押えるためバス監視機構を用いたライトバック方式のキャッシュ[10]を採用し，共有バスに接続された共有メモリ以外のプロセッサ間通信機構としては特殊なハードウェア等は用意していない．

図 2 にSMiSアーキテクチャのメモリ・マップを示す．メモリの 0x00000000 から 0x00001FFF には割り込みベクタその処理ルーチンを格納する．さらに，0x00002000 から 0x7FFFFFFF まではプログラムのテキスト部，およびデータ部に割り当て，0x8F000000以降はスタックとして使用する．共有メモリには 0x80000000 から一定の大きさの領域を割り当てる．また，0xFFFFFFF0 から 0xFFFFFFFF までには，レジスタ・ウィンドウの大きさやプロセッサの台数など実行環境に関する情報を格納する．

3.2 スレッド・ライブラリ

SMiS と Compas の意味処理部との間に仮想マシンを実現するソフトウェア部をスレッド・ライブラリとして実現した．このスレッド・ライブラリは手続き呼び出しの形式で意味処理部から呼び出され，ハードウェアで直接実現することが難しい仮想マシンの機能の一部を実現する．現在のところ，スレッド・ライブラリが実現しているのは仮想マシンのプロセス生成とストリームによる通信機構である．SMiS のハードウェアのレベルではプロセッサの間の通信機構としては共有メモリしか存在しないため仮想マシンの管理情報の一部とストリームは共有メモリ上に実現している．

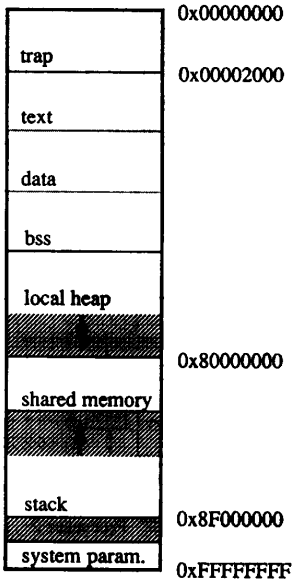


図 2 SMiS のメモリ・マップ
Fig. 2 Memory map of SMiS.

仮想マシンはプロセスと共に動的に生成されるため，実際の実現では実プロセッサ上で複数の仮想マシンを動的に切り替えながら実行を行う方式を取っている．この仮想マシンの管理にはFIFO型のスケジューリング方式を用いている．各仮想マシンはストリームによる通信待ちか，仮想マシン上で実行するプロセスの終了によって自発的に実行を放棄するまで実行を続け，時分割処理等による強制的な切り替えは行わない．新たなプロセスをつくる場合は，生成するプロセスに関する情報を共有メモリ上に書き込み，アイドル状態になったプロセッサがこれを取り出してプロセスと仮想マシンの生成を行い実行を開始する．SPARCの持つレジスタ・ウィンドウは実行中の仮想マシンがすべて使用し，仮想マシンの切替時に有効なものすべてを退避するという方式を採っている．

ストリームによる通信機構は，共有メモリ上でSPARC の不可分メモリアクセス命令 swap および ldstubを用いてプロセッサ間の排他制御を行い，固定長の環状バッファとして実現している．ストリームへの送信あるいは受信ができない場合には，仮想マシン管理モジュールを呼び出し仮想マシンの切替を行う．

4. SMiS 上での並列意味処理の評価

3章で述べたSMiSアーキテクチャのシミュレータを作成し，その上でPL/0の並列コンパイラ Compas-PL/0 の意味処理の性能評価を行った．Compas-PL/0の意味処理部ではスレッド・ライブラリの提供する仮想マシンのプリミティブを使用してプロセスの生成やプロセス間通信を行う．

4.1 実験環境

ここでは，実験に用いた並列コンパイラ Compas-PL/0 と SMiS アーキテクチャのシミュレータについて説明する．

(1) Compas-PL/0

Compas-PL/0 は Compas の並列コンパイラのモデルを PL/0 を対象言語として実現したものである．今回用いた版ではソース・プログラムのステートメントのレベルで意味処理を行うプロセスを生成する．また，各ブロックの宣言に対して記号表を管理するプロセスを設けている．

(2) SMiS シミュレータ

SMiS シミュレータは UNIX 上でコンパイル／リンクしたオブジェクトを実行可能とすることにより，

UNIX 上の開発環境を流用することを可能としている．また，プロセッサごとの手続きの呼び出し回数や，総実行クロック数などの実行プロファイルを得る機能も実現している．

このシミュレータでは命令の基本実行クロックを Cypress の CY7C600 シリーズ[9]に合わせており，大部分の命令は内部パイプラインにより実質1クロックで実行される．ただし，シミュレータでは内部パイプラインの実現は行っていない．PE のローカル／共有の各メモリは CPU との間にキャッシュを持ち，キャッシュ・ヒット時のメモリ・アクセスを1クロック，ミス時のメモリアクセスには6クロックを要すると仮定している．現在のシミュレータでは，このキャッシュのヒット率を変化させることでキャッシュの効果をシミュレートしている．SMiS シミュレータは，プログラムの実行開始時，全プロセッサに同一のメモリイメージを読み込み，各プロセッサはプロセッサ ID によってプロセッサごとの処理を選択するようになっている．

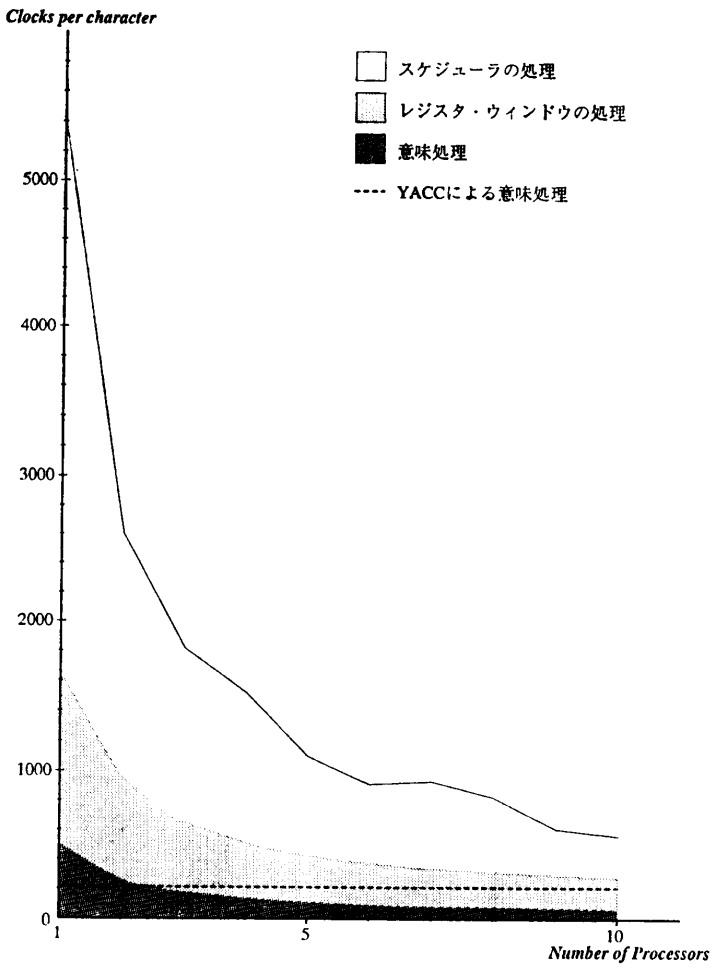


図 3 Compas-PL/0 の性能
Fig. 3 Performance of Compas-PL/0.

4.2 意味処理の並列実行の評価

テスト・プログラムをCompas-PL/0によりコンパイルした場合の意味処理部の実行と，Compas-PL/0 を等価な YACC プログラムに変換した場合の実行のグラフを**図3**に示す．このグラフは3つの部分に分かれており，上から順に，仮想マシンのスケジューリング等の管理，レジスタ・ウィンドウの管理，意味処理に要した1文字あたりのクロック数を表し，横軸はプロセッサの台数を表している．プロセッサ1台での実行に要したクロック数とプロセッサ10台での実行に要したクロック数を比較すると，5,420クロックから560クロックとなり約9.6倍の速度向上となっており，ほぼ，プロセッサの台数に反比例する結果となっている．また，実際の意味処理に使用されている時間は全体の10%から20%程度であり，その他は仮想マシンやレジスタ・ウィンドウの管理を行っている時間である．

意味処理そのものの時間だけでなく，仮想マシンでの実行時間を含めた全処理時間がプロセッサ数にほぼ反比例しているのは，仮想マシンでのストリーム関連プリミティブやレジスタウィンドウの切り替えに要する処理が，複数のプロセッサにそのまま分散されたことを意味している．また，プロセス間の同期待ち等によるオーバヘッドは，現時点では現れていないことも意味している．ただし，仮想マシンでの実行時間が大幅に減少した場合には，顕在化してくる可能性もある．

図3に示すように，YACC で生成した単一プロセッサ上で動作する逐次型の意味解析器の動作時間は，現在の並列意味解析器のプロセッサ10台の時の処理時間の約1/2程度であり，並列処理による高速化という観点からすると，まだ，最終的な目標は到達されていない．これは，仮想マシンでの実行時間が大きな割合を占めているためであり，意味処理部の実行時間だけを取り出して比較すると，並列意味処理器のほうが

3.6倍程度速い．このことから，仮想マシンの効率よい実現が不可欠であることが分かる．

4.3 意味処理プロセスのライフタイム

細粒度並列処理としての特性をより精密に観測するために，意味解析中に動的に生成されている意味処理プロセスのライフタイムを測定した結果を図4に示す．この図で縦軸はプロセス数を表し，横軸はプロセスのライフタイムに対応したクロック数を表している．ライフタイムの平均は約50,000クロックであり，約10,000クロックのところにピークがある．10,000クロックから50,000クロックの比較的短いライフタイムのプロセスは，主に式やステートメントに対応した中間コードの生成を行うプロセスで，生成後短時間で消滅する．これに対し，記号表管理を行うプロセスや構文解析器や中間コードの出力を行うプロセスは，長期間生き続ける．図4に示す意味処理の例では，入力のテストプログラムに対して生成されたプロセス数は964であり，同時に存在したプロセス数は，平均250くらいであった．

また，図5に，プロセスが実行のコントロールを得

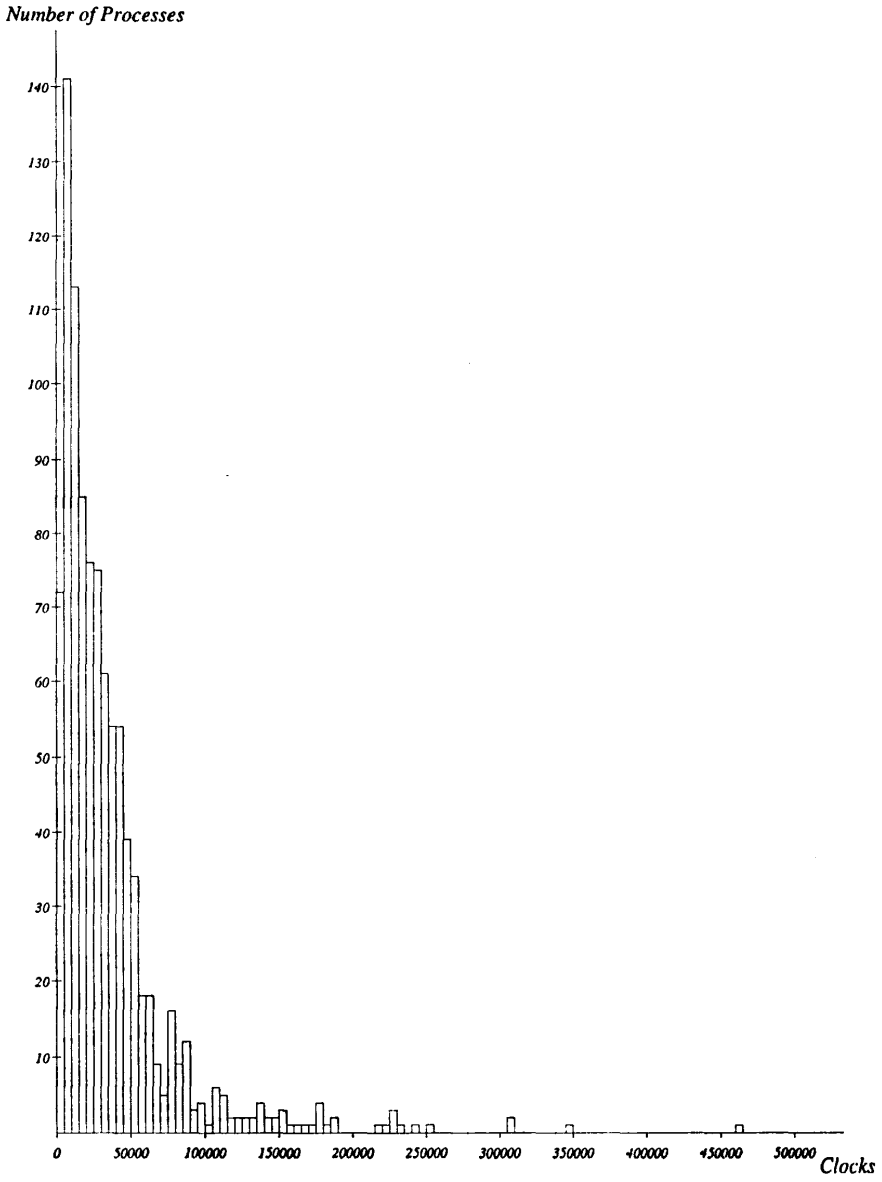


図4 意味処理プロセスのライフタイム
Fig. 4 Lifetime of semantic analysis processes.

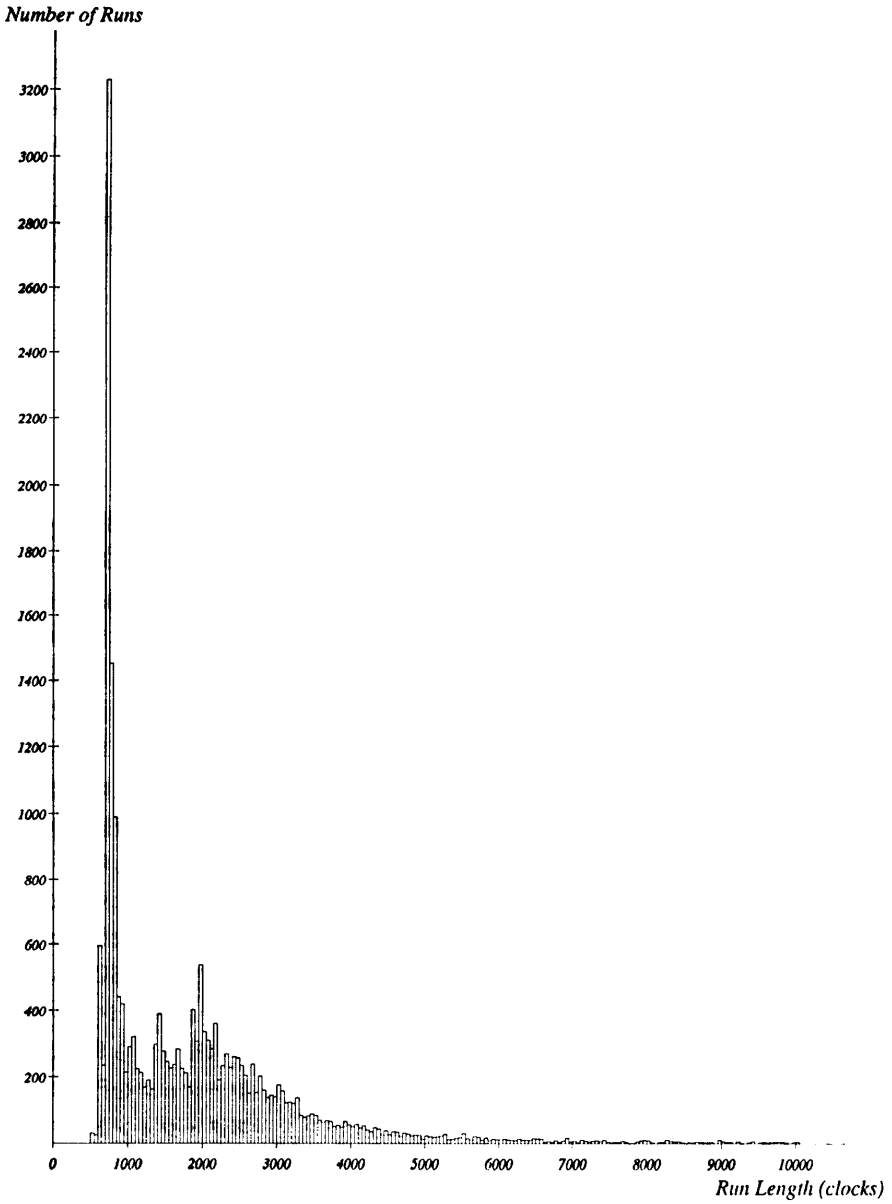


図 5 意味処理プロセスのランレングスの分布
Fig. 5 Distribution of run length of semantic analysis processes.

てから失うまでの区間の実行時間を表すランレングスを示す. 横軸は各ランレングスのクロック数, 縦軸はランの数を表している. 各プロセスの平均的なプロセス切り替えの回数は約22回であり, 平均1000クロック程度, ピークは700と2000クロックあたりに見られる.

5つのサンプルで測定したところ, ソースプログラムの1文字あたりの意味処理時間には10-20%の変動が見られたが, プロセスのライフタイムやランレングスの分布には大きな変動はなく同じ傾向が見られた.

4.4 仮想マシンプリミティブの実行時間

仮想マシンの各プリミティブの呼び出し回数と実行時間の割合を, プロセッサ5台の場合について**表1**に示す. なお, ストリームを介した通信待ちが発生した場合, プロセス管理モジュールが呼び出されるが, この実行時間はプロセス管理のためのプリミティブに含まれている.

現在の実現では仮想マシンプリミティブの1回あたりの実行時間は約3,000クロックとなっている.

表 1 仮想マシンプリミティブの内訳
Table 1 Detailes of virtual machine primitives.

	呼び出し回数	1回あたりのクロック数	仮想マシンの実行に占める割合
プロセス管理	4,733	8,160	83.4%
メモリ管理	1,980	822	3.5%
プロセス間通信	6,612	851	12.2%
その他	2,310	170	0.9%

なお，仮想マシンのプリミティブの実行時間のうち，22% はレジスタファイルの退避に，61% は，仮想マシンの切り替えに使われており，ストリームを介した通信そのものに使われる時間は12% 程度でそれほど多くはない．したがって，プロセス管理機構の実現が性能面での速度の低下の原因となっている．

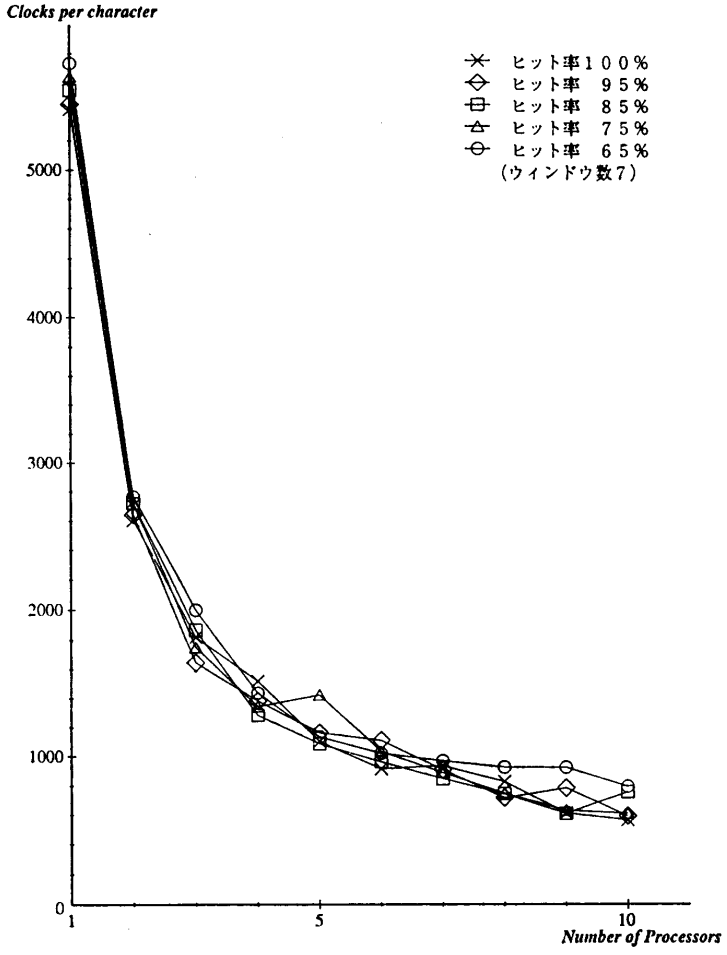


図 6 キャッシュメモリアクセスのヒット率がコンパイラの性能に与える影響
Fig. 6 Effects of hit rates of cache memory accesses to the performance of a compiler.

4.5 共有メモリアクセスの評価

以上の評価では，共有メモリと局所メモリのキャッシュのヒット率をそれぞれ100% としてシミュレーションを行っていた．しかし，実際のマルチプロセッサシステムでは共有メモリへのアクセス競合がシステムの性能を左右することが考えられる．このため，キャッシュメモリのヒット率を100% から65% まで変化させて Compas-PL/0 の実行に与える影響を測定した．この結果を図 6 に示す．この結果から，キャッシュのヒット率の低下によるスピードへの影響はプロセッサ台数が増してもそれほど大きくなっていないことが分かり，現状では，キャッシュメモリのヒット率は全体の性能にあまり影響を与えていないように思われる．

4.6 レジスタ・ウィンドウの評価

SMiS で採用した SPARC アーキテクチャの特徴として親子関係にある手続き間でレジスタのオーバラップを可能とするレジスタ・ウィンドウを持つことが挙げられる．レジスタ・ウィンドウは高速な手続き呼び出しを可能にするが，一方ウィンドウ数の増加はコンテクスト切り替え時に退避が必要となる情報を増やすことにもなる．

上で示した評価ではレジスタ・ウィンドウの段数をプログラムの開発に利用している SUN 4 と同じく 7 としてシミュレーションを行っていたが，これを 3 から 9 まで変化させてシミュレーションを行った．この場合，レジスタ・ウィンドウの段数を小さくすると，仮想マシンを切り替える際のオーバヘッドが減少するが，手続き呼び出しのオーバヘッドが増加する．反対に，レジスタ・ウィンドウの段数を大きくすると，仮想マシン切り替えのオーバヘッドは大きくなるが，手続き呼び出しのオーバヘッドが減少する．この結果を図 7 に示す．ウィンドウ数が 3 から 5 の間はレジスタ・ウィンドウの効果が現れるが，それ以上ではそれほどの効果は得られていない．

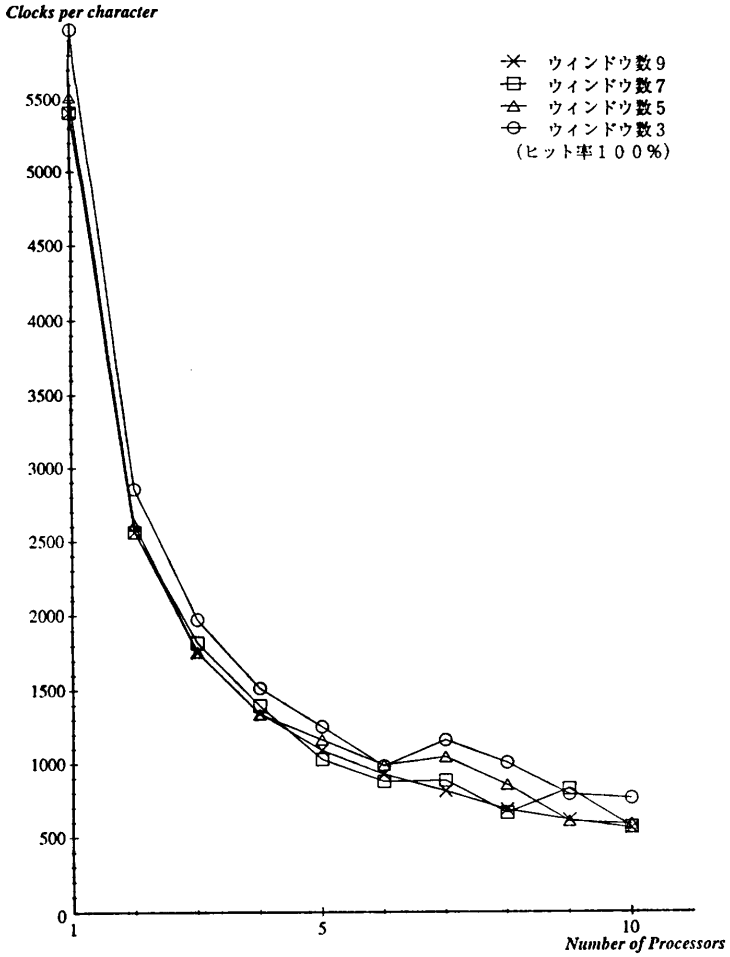


図 7 レジスタウィンドウ数がコンパイラの性能に与える影響
Fig. 7 Effects of number of register windows to the performance of a compiler.

4.7 考 察

今回のシミュレーションから意味処理を細粒度プロセスにより並列に実行した場合マルチプロセッサによる実行による速度向上は見られるが，コンパイラの意味処理の実行の多くの部分が仮想マシンの管理，それも細粒度プロセスの管理に費やされているという結果が得られた．これは，主に Compas-PL/0 の意味処理を行っているプロセスの粒度が式あるいはステートメントレベルの細粒度のプロセスとなっていることに起因するものである．このことから，意味処理の効率的な並列実行を行うためには，細粒度のプロセスの実行をサポートする効率的な機構や，意味処理を行うプロセスの粒度を適当な大きさに調節するような機構を考える必要があることが判明した．

5. おわりに

意味処理からプロセスやストリームの管理を分離するために仮想マシンを導入し，マルチプロセッサアーキテクチャ SMiS のシミュレータ上でシステムの動作状態の精密な測定を行った．この結果，Light Weight Process を用いたシミュレーションでは分からなかった問題点が明らかになった．現在のシステムでは 4 章で示したように，スケジューリングやストリームによる通信などの仮想マシンの管理に多くの時間が取られている．これは，今回シミュレーションの対象としたコンパイラが比較的単純なものであることから，意味解析プロセス各々の粒度が小さいことがその主な原因となっている．

種々の最適化を行うような複雑化したコンパイラでは意味解析プロセスの粒度は大きくなることが想定されるので，仮想マシン部で生じるオーバヘッドは相対的に減少すると考えられる．しかし，今回評価の対象としたような比較的単純な構成のコンパイラではプロセスの粒度を大きくすることは難しく，何らかの対策が必要である．この解決の方法としては，プロセスの逐次化等により逐次的に実行される意味処理の単位を大きくすることでプロセス管理に要する相対的な時間を短縮すること，仮想マシン部のソフトウェアの実現をさらにチューニングして効率よくしていくこと，あるいは，プロセス管理やストリーム処理のための特別なハードウェアを設計することなどが考えられる．これらの点に関して，現在，改良を行っているところである．

実用的なコンパイラでは最適化処理が不可欠であり，現在，最適化処理の並列化について検討を進めている．このほかに，インタプリタやデバッガなどの動的意味を扱う言語処理系や，プログラミング言語だけでなく文脈自由文法で定義可能な構造を持ったデータの処理に同様の手法を適用し評価することも今後の課題であると考えている．

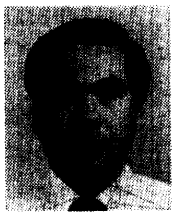
参考文献

1) Itano, K., Sato, Y., Hirai, H. and Yamagata, T.: An Incremental Pattern Matching Algorithm for the Pipelined Lexical Scanner, *Inf. Process. Lett.*, Vol. 27, No. 5, pp. 253-258 (1988).
2) Itano, K., Nishiyama, H. and Chu, Y.: A Bottom-up Parsing Coprocessor for Compilation, Technical Report TR-2280, Department of Computer Science, University of Maryland (1989).
3) Graham, S. L.: Table-driven Code Generation, *IEEE Computer*, Vol. 13, No. 8, pp. 25-34 (1980).
4) 西山博泰，板野肯三：ストリームに基づいた並列意味処理の記述，情報処理学会論文誌，Vol. 31, No. 5, pp. 731-739 (1990).
5) Wirth, N.: *Algorithms+Data Structures=Programs*, Prentice-Hall (1976).
6) Sun Micro Systems: Programming Utilities & Libraries, SunOS Reference Manual, Sun Micro Systems (1991).
7) 西山博泰，板野肯三：マルチプロセッサ・システム SMiS 上での並列コンパイラ Compas の実現と性能評価，情報処理学会プログラミング—言語・基礎・実践—研究会，3-22, pp. 195-203 (1991).
8) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 547-557 (1974).
9) Cypress Semiconductor: Sparc RISC User's Guide, ROSS Technology, Inc., Cypress Semiconductor Company (1990).
10) Archibald, J. and Bear, J. L.: Cache Coherence Protocols, *ACM Trans. Comput. Syst.*, Vol. 4, No. 4, pp. 273-298 (1986).

(平成3年12月24日受付)
(平成4年7月10日採録)

西山 博泰（正会員）

1965年生．1989年筑波大学第三学群情報学類卒業．現在同大学院博士課程工学研究科に在学中．言語処理系，並列処理方式，プログラミング環境に興味を持つ．ACM会員．



板野 肯三（正会員）

昭和23年生．昭和46年東京大学理学部物理学科卒業．昭和48年同大学大学院修士課程修了．昭和51年同博士課程単位取得後退学．理学博士．筑波大学計算機センタ準研究員，同大学電子・情報工学系助手，講師を経て，現在，同助教授．コンピュータアーキテクチャ，オペレーティングシステム，プログラミング言語処理系に興味を持つ．ソフトウェア科学会，IEEE, ACM各会員．