マルチプロセッサ・システムSMiS上での並列コンパイラCompasの性能評価

西山博泰　　板野肯三
筑波大学工学 研究科　筑波大学電子情報工学系

コンパイラの並列実行モデルCompasを，PL/0を対象言語として実現したシステムCompas-PL/0の性能評価を行った．この評価のために，Compas-PL/0にプロセスの逐次化や記号表キャッシュなどの改良を加えた．また，コンパイラの精密な評価を行うために，SPARCを使用したマルチプロセッサ・アーキテクチャの設計を行い，コンパイラを構成するプロセスの効率的な実行を可能とした．シミュレーションを行った結果，意味処理の性能向上が見られ，ここで採用した手法が並列コンパイラの実現に有効であることが確認された．

Evaluation of the Parallel Compiler Compas on the Multi-Processor System SMiS

Hiroyasu Nishiyama
Doctoral Program in Engineering,
University of Tsukuba
and
Kozo Itano
Institute of Information Sciences and Electronics,
University of Tsukuba
Tsukuba, Ibaraki 305, Japan

We evaluated the performance of a parallel compiler Compas-PL/0 which was derived from Compas incorporating a parallel-compilation model; for this evaluation, we improved the implementation of the previous version of Compas-PL0 adopting process sequentialization and symbol-table caching. In order to evaluate the compiler precisely, a SPARC-based multiprocessor architecture simulator was also implemented for the effective execution of fine-grained multiprocesses used in the compiler. According to the simulation result, considerable speed up of the semantic processing was observed, and the approach was confirmed effective for the implementation of a parallel compiler.

1.はじめに

高度なプログラミング環境の構築には，プログラミングに携わるプログラマの思考を阻害しない応答性を擁する言語処理系と，統一的な操作環境を提供するプログラミング・システムが不可欠である．

筆者らは，実際的な並列処理アルゴリズムの検討と高度なプログラミング環境の構築を目的として，コンパイラの並列処理の研究を行っている．現在作成している並列コンパイラCompasでは，コンパイラを構成するフェーズをパイプライン的に結合する方式を基本として，ハードウェアによるサポートや並列処理を個々のフェーズに取り入れることでコンパイラ全体のスピードを向上するという方向で研究を進めている．具体的には，コンパイラのフェーズのうち字句解析部と構文解析部に関してコプロセッサによるサポートを提案した[1-2]．また，意味解析処理はハードウェア化に向かないとの認識から，意味処理についてはマルチプロセッサ・システム上での並列処理方式を考案した[3]．

以前の研究で行った並列意味処理の評価では，UNIXシステム上の１つのプロセス内でLight Weight Process[4]を用いてマルチプロセッサ・システムをシミュレートしていたことから，共有メモリへのアクセス競合等の条件を考慮したタイミングでの正確な評価を行うことができないという問題点があった．このため，性能面や意味解析部を実行するためのハードウェア・システムに関する精密な評価を行うことを目的として，マルチプロセッサ・アーキテクチャSMiSの設計を行い，そのシミュレータ上で，コンパイラの意味処理を中心に，コンパイラの並列実行モデルCompasをPL/0[5]を対象言語として実現したシステムCompas-PL/0のシミュレーションを行った．

以下，２章ではCompasの実現と改良点について，３章ではマルチプロセッサ・アーキテクチャSMiSとそのシミュレータについて説明する．４章ではSMiSによる実行結果をもとにCompas-PL/0の評価と問題点に関する考察を行う．

2.パラレル・コンパイラCompasの構成

並列コンパイラCompasでは，コンパイラのフェーズを構成するモジュールをパイプライン的に結合し，さらに各々のフェーズにハードウェア化や並列処理を取り入れることで，コンパイラを高速化するという方針を採っている．

以下では，まずCompasの全体的な構成について説明し，続いて意味処理部の並列化の方式の概要と改良点について説明する．

2.1 コンパイラの並列化

コンパイラの処理は字句解析，構文解析，意味解析，コード生成といった論理的独立性の高い複数のフェーズから成り立っている．コンパイラを並列化することを考えた場合，この論理的なフェーズに沿ってモジュールに分割し，各々をパイプライン結合した構成にすることは自然であり，各モジュールの設計や保守が容易になる．Compasでは，記号表のように一般のコンパイラでは複数のフェーズ間で共有されることの多いデータを分割し，必要なデータをローカルな情報として各フェーズに持たせることにより各フェーズを構成するモジュールが独立に動作するようにし，パイプラインによる動作を可能としている．

パイプライン構成による並列処理では，モジュール数以上のスピードアップは期待できず，最も遅いフェーズがボトルネックとなってしまう．ボトルネックとなるフェーズは，複雑な最適化を行わない比較的単純なコンパイラでは字句解析部であり[6]，最適化等により意味処理の内容が複雑化するにつれ意味解析部の処理がボトルネックとなる．

コンパイラを構成するフェーズのうち字句解

析，及び構文解析処理は単純な処理ではあるが頻繁に呼び出されることから，単純なコンパイラではこれらのフェーズの実行時間が処理の大半を占めている．この字句解析部や構文解析部を高速化する手法の１つとして，字句解析処理や構文解析を専用に行なうコプロセッサ方式が存在する．これらの専用ハードウェアを用いることで，字句解析部の処理については１トークンあたり数クロック，構文解析部については数十クロックで解析を行なうことが可能になる．これはソフトウェアで処理した場合の数十倍の性能であり，単純なコンパイラの意味解析部と比較しても無視できる程度のスピードである．

また，コード生成フェーズについては構文解析と同様の手法を用いたパターンマッチによる方法が知られており，構文解析コプロセッサと同様の手法を用いることで処理を高速化することができる．

いずれにしても意味解析を除くこれらのフェーズは単調な処理であるため表駆動によるハードウェアに向いており，ハードウェアサポートを行うことにより，高速な処理を行なうことが可能である．また，制御表を交換することで複数の言語に対応することも可能である．

2.2 意味処理の並列化

字句解析部や構文解析部と比較して，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も大幅に異なることから，表駆動のような形式には向かない．このため，意味処理を高速化することに関しては汎用のマルチプロセッサ上で並列処理を行なうことで処理の高速化を図ることとした．

意味処理を並列に行なう手法としては幾つかの方法が考えられるが，Compasでは既存の属性文法[7]による手法や，YACC[8]等で採用されている動作ルーチンによる手法を拡張するのではなく，並列に動作する意味処理の記述が容易であることや，動的な並列性の抽出が容易であることを考慮して，ストリームに基づいた意味処理の並列化を並列意味処理のモデルとして採用した．

このモデルでは，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，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これと共に与えることで，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なうことを可能としている．さらに，意味解析プロセスが内部状態を持つことが可能なことから，記号表等の永続的なデータの表現や履歴に依存するようなデータを効率良く記述できるようにしている．

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取り，構文解析の終わったノードに対して定義された意味解析処理を行うプロセスがあればその生成を行い，プロセス間を結ぶストリームの結合を行う．生成されたプロセスは各々自立的に動作し，プロセス間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なう．

ここで用いているストリームはCSP[9]におけるチャネルと同様な，プロセス間の同期機構を内在したデータ構造であり，ここでは有限の大きさのバッファからなるキューとして実現している．空のストリームからの読みだし，または，バッファーがいっぱい状態のストリームへの書き込みを行った場合，操作が可能になるまでプロセスの実行は中断される．ストリームを通して受け渡されるデータは，ポインタあるいはポインタに収まる大きさのデータに制限されており，文字列等のより大きなデータは共有メモリ

中に格納し，ポインタを介して受渡しを行なう．

2.3 Compasの改良

Compasでは意味処理を並列に実行していることから，性能改善のために逐次的な意味処理の場合とは異なった方式をとることが可能である．以下ではCompas-PL/0で以前提案した方式に対して行った改良点について説明する．

1) プロセスの逐次化

Compasでは意味処理を行なうモデルとして複数のプロセスを独立したものとして扱う．しかし，複数のプロセスを生成しても，処理の対象となるデータ間の依存関係により，実際の処理は逐次的に評価を行なった場合と変わらない場合が存在する．このような場合，個々の評価にプロセスを生成してもプロセスの制御に要する時間が無駄になるだけである．このため，Compasでは，構文と意味処理の記述を行う際に，プロセスに対して逐次化を行うことを指定することを可能にしている．この逐次化を利用することで，実行時に生成されるプロセスの粒度をコンパイラの記述を行う際に制御することが可能となる．逐次化を指定されたプロセスは，プロセスの生成時に，解析木上で連続したものがまとめられ，実際には１つのプロセスとして実行される．ただし，逐次化の対象となるプロセス間のデータの受け渡しにはストリームを用いず，スタックを介してデータの受け渡しを行う．逐次化されたプロセスから他の意味解析プロセス間のデータの参照は通常のストリームを介して行われる．逐次化されたプロセスのデータをアクセスする際にはプロセスの終了を待ち，データを受け取った後，実行を再開する．

2) 記号表参照キャッシュ

Compas-PL/0の最初の版では各々の宣言部に対応して記号表を関するプロセスを設けている．このため，PL/0やPASCALのように宣言のネストを認める言語では記号表の検索を行う場合，最も内側の記号表管理プロセスから始めて，構文に沿って外側の記号表を順次検索する必要がある．

一般にプログラムの特定のブロック中に現れる名前は限られており，しかも，同じ名前が何度も使用されることが多い．このことから，記号表を管理するプロセス間の通信量を減少させるために，現在のCompas-PL/0では外側の記号表への検索結果をローカルな記号表管理プロセス内にキャッシュするように改良を行なった．

3) ストリームの要素としてのストリームの導入

以前のCompas-PL/0では，ステートメント・リストやwhile文の中間コードの生成処理を行なう際に，解析木上で下位のノードに対応した中間コード生成処理プロセスから受けとった中間コードを適当な命令を挟み込みながらマージし，上位のノードを処理しているプロセスにコピーする処理を行なっていた．これはCompas-PL/0でストリームの要素としてストリームを送ることを認めていなかったためである．このため，下位のノードから受け取ったコードをそのまま上位ノードに渡すという処理が必要となっていた．最新のCompas-PL/0ではストリームの要素にストリームを許す事で，ストリームのコピーを行うだけの処理を不要としている．

3. マルチプロセッサ・アーキテクチャSMiS

以前，並列意味処理の評価を行った際には，SUN OSのLight Weight Processを用いて実際のプロセッサをシミュレートしていた．このシミュレーションでは，Light Weight Processを時分割処理することによって，仮想的なプロセッサ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

サ稼働率を予測していた．これにより，プロセッサ台数と意味解析プロセスの実行との間の相関関係のおおまかな評価を行うことができたが，シミュレーションの単位が意味解析プロセス・レベルであったことから，共有メモリへのアクセス競合等，実際のハードウェアの実現によって生じる種々の条件を設定できなかった．このため並列化による速度向上やハードウェアの構成等に関する詳細なデータを得ることができなかった．

これを解決するためマルチプロセッサ・システムSMiSの設計を行い，そのシミュレータの開発を行った．これにより，実際のマルチプロセッサ・システムと同様に実行を行い，実行時の詳細なデータの収集を行うことが可能な環境を構築した．

3.1 SMiSの構成

SMiSで想定しているハードウェアはSPARC CPU[10]，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形態である(図1)．共有メモリに対するキャッシュにはバス監視機構を用いたライトバック方式のキャッシュ[11]を仮定し，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想定していない．プロセッサ間の低レベルの同期は，共有メモリ上でSPARCの不可分メモリアクセス命令swapまたは，ldstubを用いて行い，高レベルの同期はこれを用いて実現する．

SMiSのCPUとしてSPARCを採用したのは，1)RISC CPUであることから命令数が少なく単純なため，シミュレータの開発が容易であり，2)マルチプロセッサ・システムに対応した命令を備えていること，3)現在使用しているワークステーションの開発環境が流用できること，の３点による．1)はシミュレータを作成する上では利点となるが，SPARCはレジスタ・ウィンドウを持っていることからコンテクスト切替時のオーバーヘッドが大きくなることが予想される．これらのハードウェアに関する評価は次章で行なう．

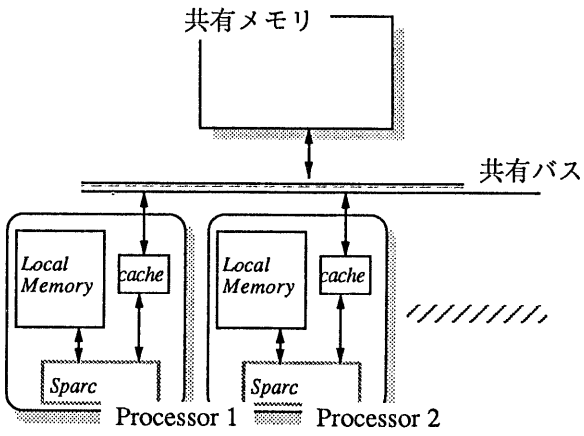

図１．SMiS(Sparc Multi-Processor System)の構成

3.2 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，上で述べたようにUNIX上の開発環境を流用することを可能としている．また，プロセッサ毎に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社のCY7C600シリーズに合わせており，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つことを仮定しており，キャッシュ・ヒット時のメモリ・アクセスは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．将来

的にはシミュレータ上で幾つかの方式のキャッシュをインプリメントすることで，より詳細なシミュレーションを可能とする予定である．

図2にシミュレータ上で動作するプログラムのメモリ・マップを示す．メモリの0x00000000から0x00001FFFには割り込みベクタと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分，及びデータ部に割り当てられ，0x8F000000以降はスタックとして使用される．共有メモリには0x80000000からシミュレータに指定した大きさの領域が割り当てられる．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．

SMiSシミュレータでは，プログラムの実行時，全プロセッサに同一のメモリイメージが良み込まれ，各プロセッサはプロセッサIDによってプロセッサ毎の処理を選択する．

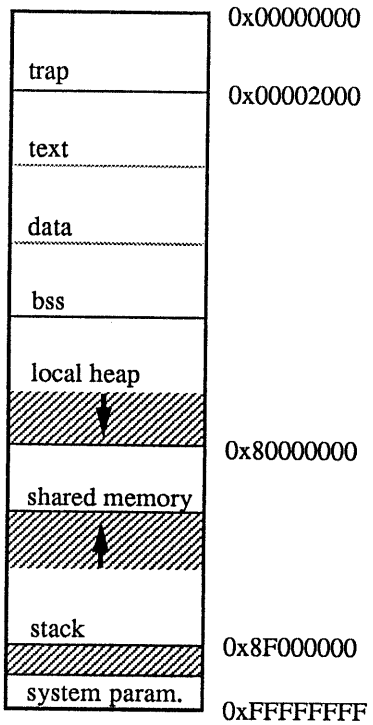

図２．SMiSのメモリ・マップ

3.3 スレッド・ライブラリ

SMiSのマルチプロセッサ環境下でプロセスのスケジューリング等の管理を行うことを目的としてスレッド・ライブラリを作成した．このライブラリではFIFO型のスケジューリング方式を用い，プロセス間の同期機構としてはモニタ[12]を用意している．プロセスの切り替えは，実行しているプロセスがモニタ等で待ち状態になるか，実行を終了した時点で行われ，時分割処理等による強制的なプロセスの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の生成を行い，実行を開始する．

4. SMiS上でのCompas-PL/0の評価

PL/0のパラレル・コンパイラCompas-PL/0の意味解析部を記述し，SMiSシミュレータを用いてその性能評価を行なった．意味解析プロセスの制御には，3.2節で述べたスレッド・ライブラリ上にストリーム通信用の手続きを定義しこれを使用した．

(1) Compas-PL/0の実行結果

Compas-PL/0により500行程度の大きさのPL/0プログラムをコンパイルした結果のグラフを図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，スケジューリング等のプロセス管理，レジスタ・ウィンドウの管理，意味処理に要した時間を表している．プロセッサ10台での実行により実行に要したクロック数が約2,100万クロックから約286万クロックとなり7.6倍程度の速度向上となっている．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2

0%程度であり，その他はプロセスやレジスタ・ウィンドウの管理を行っている時間となっている．

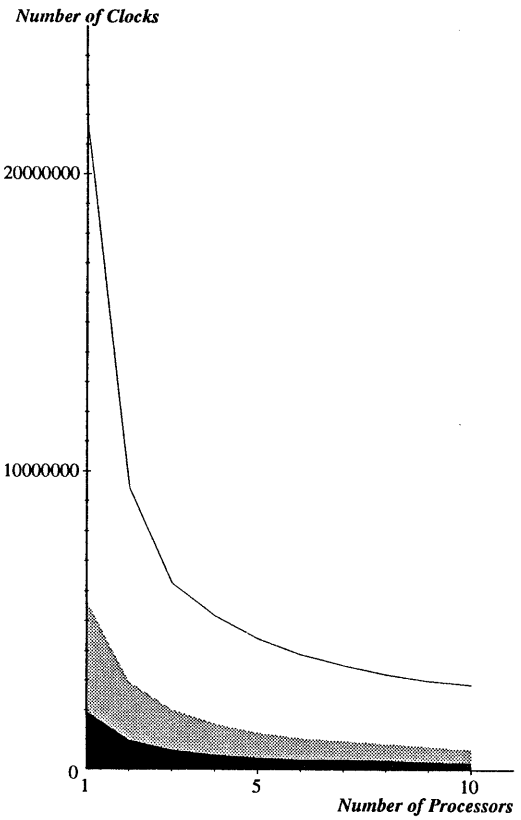

図3．Ｃｏｍｐａｓ－ＰＬ／０の実行結果

(2)Compas-PL/0の改良点の評価

次に，2.3節で述べたCompas-PL/0の改良点のプロセッサ5台での実験結果を表4に示す．ここで，プロセスの逐次化の評価には，式の解析処理を独立したプロセスで行った場合と，式の解析プロセスを逐次化した場合を用いた．プロセスの逐次化により，生成されるプロセス数は21%程減少し，実行時間も12%短縮されている．同様に記号表キャッシュにより14%の実行時間が短縮された．また，逐次化を行った場合に対してさらに，ストリームの要素としてストリームを導入することによりプロセス数は37%減少し，32%の実行時間短縮となった．これらの結果からCompas-PL/0で採用した改良が有効であることが確認された．

	プロセス数	クロック数
改良前	163	1,034,141
プロセスの逐次化	128	910,495
記号表キャッシュ		890,759
ストリーム	102	706,434

表４．Compas-PL/0の改良結果

(3) 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ー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．上に述べ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SUN4と同じく7としてシミュレーションを行なっていたが，これを3から11まで変化させてシミュレーションを行なった．この場合，レジスタ・ウィンドウの段数を小さくすると，プロセス切替えのオーバーヘッドが減少するが，手続き呼び出しのオーバーヘッドが増加する．反対に，レジスタ・ウィンドウの段数を大きくすると，プロセス切替のオーバーヘッドは大きくなるが，手続き呼び出しのオーバーヘッドが増加する．この結果を図5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著しいが，それ以上ではそれほどの効果は得られていない．図3の結

果ではレジスタ・ウィンドウとプロセスの管理が実行のかなりの部分を占めており，レジスタ・ウィンドウの段数を少なくし，代わりに複数のコンテクストに対応したレジスタ・ウィンドウの組を用意することで，プロセスの切り替えを高速に行うことが可能であると思われる．

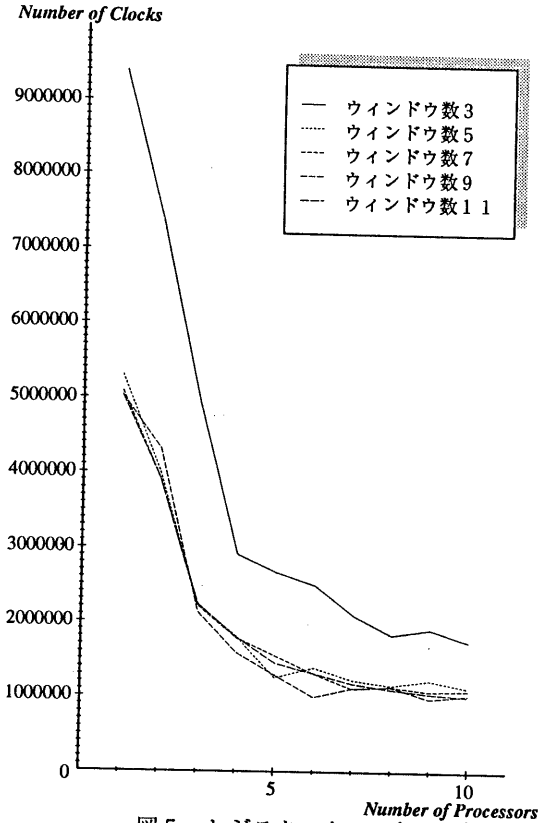

図5．レジスタ・ウィンドウの評価

(4) 共有メモリアクセスの評価

以上の評価では，共有メモリ，局所メモリそれぞれのキャッシュのヒット率を100%としてシミュレーションを行っていた．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．このため，キャッシュ・メモリのヒット率がCompas-PL/0の実行に与える影響の評価を行った．評価は共有メモリのキャッシュのヒット率を90%から60%まで変化させて，実行速度の変化を得た．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る．これは、共有メモリをプロセス間の通信にのみ用い，スタックや命令等をローカル・メモリに配置するアーキテクチャが有効であることを示している．

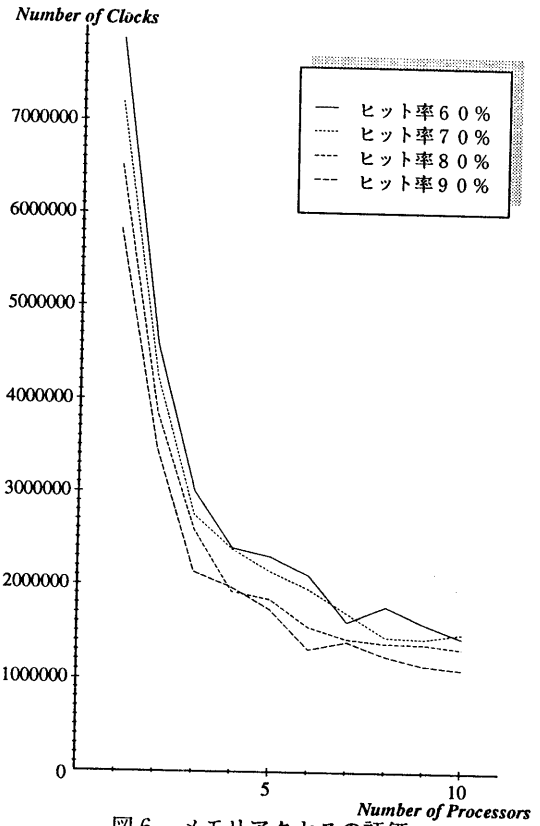

図6．メモリアクセスの評価

(5) 考察

今回のシミュレーションでは，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，コンパイラ意味処理の実行の多くの部分が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小さなためであり，

最適化処理などを採り入れることで，意味処理の複雑度が増し，意味処理が実行時間に占める相対的な割合も増加すると思われる．また，粒度の小さな並列処理に向くよう，SMiSのアーキテクチャに対する改良を行うことも考慮する必要がある．

6.おわりに

現在のシステムでは4章で述べたように，スケジューリング等，意味処理を行うプロセス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くなものになったという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が，今回評価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事でプロセス管理に要する相対的な時間を短縮する事が可能になる．ハードウェア環境に関しては，細粒度の並列処理に向くようSMiSアーキテクチャに改良を検討することも必要であろう．

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．この他に，インタプリタやデバッガなど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] 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).

[4] Sun Micro Systems: SunOS Reference Manual, Sun Micro Systems (1988).

[5] Wirth, N.: Algorithms+Data Structures=Programs, Prentice-Hall (1976).

[6] Wait, M.E.: The Cost of Lexical Analysis, Soft. Prac. Exp., Vol.16, No.5 (1986).

[7] Knuth, D.E.: Semantics of Context-free Languages, Mathematical Systems Theory, Vol.2, No.2, pp.127-145 (1968).

[8] Johnson, S.C.: Yacc - Yet Another Compiler Compiler, Computing Science Technical Report 32, AT&T Bell Laboratories (1975).

[9] Hoare, C.A.R.: Communicating Sequential Processes, Comm. ACM, Vol.21, No.8, pp.547-557 (1974).

[10] Cypress Semiconductor: Sparc RISC User's Guide, ROSS Technology, Inc., Cypress Semiconductor (1990).

[11] Archibald, J. and Bear, J.L.: Cache Coherence Protocols, ACM Trans. on Computer Systems, Vol.4, No.4, pp.273-298 (1986).

[12] Brinch Hansen, P.: The Architecture of Concurrent Program, Prentice-Hall (1977).