ICOT Technical Memorandum: TM-0888

TM-0888

並列論理型言語KL1の
コンパイル方式の改良

後藤厚宏，平野喜芳

May, 1990

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列論理型言語 KL1 のコンパイル方式の改良

半野 喜芳[1], 後藤 厚宏[2]

1: (株) 富士通ソーシアルサイエンスラボラトリ
2: (財) 新世代コンピュータ技術開発機構

現在ICOTでは並列推論マシンPIMの研究開発を行なっている。このPIMのために並列論理型言語KL1のコンパイル方式を改良したので報告する。今回の改良では、ゴールの実行方式として、引数を必要になるまでメモリ上に置いたままとし、レジスタを全て作業用として利用するメモリベース方式を採用した。また、同時に幾つかのコンパイル時の最適化も導入した。この改良の目的には、ゴールの引数個数やユニフィケーションの複雑さに対するハードウエアのレジスタ数による制限を無くすこと、レジスタの使用効率を良くし少ないレジスタ数で実行可能なこと、コンテキストスイッチの処理を軽くすることの3つがあるが、実際にコンパイラを作成し、実行、評価した結果、目的を達成することができた。しかし、ゴール環境を常にメモリ上に作るので、述語の実行に十分な数のレジスタが用意されている場合にはメモリのアクセス回数が増加することが分かった。そこで、この対策として、レジスタの使用量が少ない、一部の述語の再帰呼び出しをレジスタ上で行なう最適化を提案する。

1 はじめに

ICOTでは第5世代コンピュータプロジェクトの一環として、並列推論マシンPIMの研究開発を進めている。このPIMのために並列論理型言語KL1のコンパイル方式を改良したので報告する。

KL1はGHC [1] に基づいてICOTで設計されたAND並列の論理型言語である。PIMではこのKL1を、一旦抽象機械語KL1-B [2] にコンパイルし、さらに、これをPIMハードウエアで用意された機械語に変換してから実行することにしている。この時、中間言語として使われるKL1-Bは、KL1を実行するための仮想的なハードウエアの機械語として定義したものであり、PrologにおけるWAM [3] コードに相当するものである。従って、KL1-Bは言語とハードウエアの接点となるものであり、この言語仕様、即ち、KL1をどのようにコンパイルするかにより、KL1の処理効率に大きな影響を与える。

KL1-Bの仕様に関しては、今までにも、MRB (Multiple Reference Bit) によるインクリメンタルGCのサポート[4]や、クローズインデキシングによる高速化[5]等の改良が行なわれてきたが、今回、PIM用として全面的な見直しを行ない、新しい仕様をまとめた。この新しいコンパイル方式では、ゴールの実行方式としてメモリベース方式を採用しており、同時に幾つかのコンパイル時の最適化も導入した。このメモリベース方式は、ゴール引数を必要になるまでメモリ上に置いたままとし、レジスタを全て作業用として利用する方式である。

2 従来のコンパイル方式の問題点

Multi-PSI等で採用されている従来のコンパイル方式では、ゴール引数や作業用データを全てレジスタ上に置いて実行する処理方式 — レジスタベース方式を採用している。この方式は図1のような流れでゴール実行を行なっているが、以下に示すように幾つかの問題点があると考えられる。

2.1 レジスタ数による制限の問題

従来の方式では、ゴール引数や作業用データを全てレジスタ上に置いて実行している。このレジスタはハードウエアで提供されるものであり、当然、数が限られている。この限られた資源だけを使ってKL1を実行しようとするので、ゴールの引数個数やユニフィケーションの複雑さが制限を受けることになる。通常、ハードウエアで提供されるレジスタ個数は数十個であり、これによる制限値は、KL1プログラマにとって十分な値と言うことはできない。また、レジスタの個数はハードウエアの種類によって異なる場合があり、KL1プログラマを混乱させる可能性が大きい。さらに、KL1の上位言語や暗黙の引数[6]等の高機能マクロを使って記述されたプログラムを機械変換した場合には、ゴール引数の数が多くなる傾向があり、そのようなプログラムでは、レジスタ数の制限によりコンパイル不可能になる場合が多くなっている。

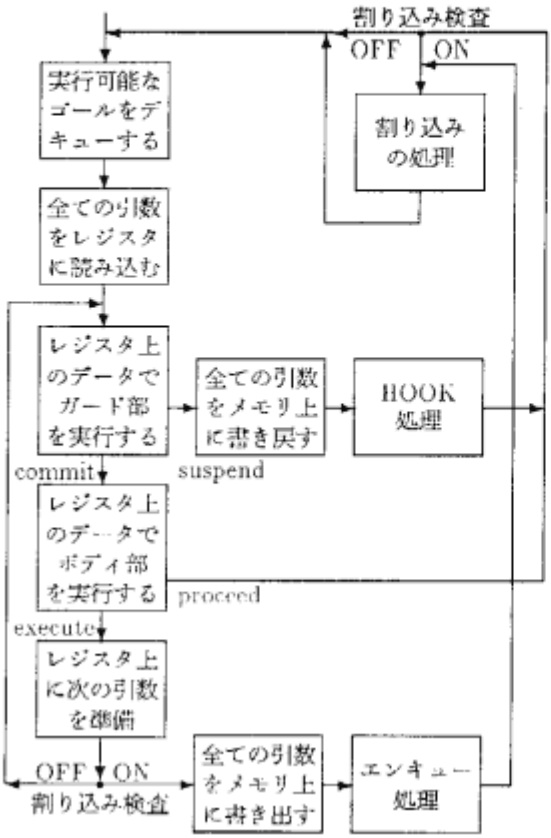

図 1: 従来のゴール実行方式

2.2 レジスタの使用効率の問題

従来の方式は、レジスタの使用効率が悪いと言うこともできる。即ち、この処理方式では、たとえ操作しない引数であっても、ゴール引数は全て無条件にレジスタ上に置いている。ところが、KL1プログラムでは、引数に対して何の操作もせずに、そのまま子ゴールに渡すだけの場合が多くある。

例えば、以下のプログラムはウィンドウドライバの一部であるが、window/8 の引数のうち、OP, P, U, St, の4個は何の操作も行なわれずに、子ゴール wait/9 に渡される。

```
window([get(X)|Rq],Dev,OP,P,U,St,Buf,C):-
  true |
    flush(Dev,[get(Y)|Dev2],Buf,Buf2,C),
    wait(Rq,Dev2,OP,P,U,St,Buf2,X,Y).
```

このような、当面使われない引数でもレジスタを占有してしまうため、ユニフィケーション等の作業用としてレジスタを有効に使用することができなくなっていると考えられる。

2.3 コンテキストスイッチの問題

従来のKL1-Bでは、現在実行中のゴールの環境(引数等)は全てレジスタ上に、それ以外のゴールの環境は全てゴールレコードと呼ばれるメモリ上の構造体に置くという処理方式になっている。そのため、図1のように、サスペンド処理等のコンテキストスイッチの時には、現在の環境を全てメモリ上に書き出し、次に実行すべきゴールの環境を全てレジスタに読み込んでくる操作が必要になっている。

例えば、先ほどのウインドウドライバのプログラムの場合には、実行可能であるか、サスペンドする必要があるかは、第1引数のみをチェックすれば分かる。しかし、従来の方式では、実行に先立ち、全ての引数を一旦レジスタ上に用意することにしており、もしサスペンドしなければならない場合には、それら全てを、メモリ上に書き戻すという処理が必要になっている。

マルチプロセッサ上で実行する並列プログラムでは、かなりの頻度でサスペンド等のコンテキストスイッチが発生すると考えられ、このコストが重い従来方式には問題があると言うことができる。

3 コンパイル方式の改良

以上のような従来のコンパイル方式の問題点を解決するために、ゴールの実行方式をメモリベース方式に変更した。また、さらに処理効率を上げるために最適化を進めた。

3.1 メモリベース方式

新しいコンパイル方式では、処理方式の基本として、メモリベース方式を採用した。これは、ゴールの引数を必要になるまでメモリ上に置いたままとし、全てのレジスタを作業用として利用する処理方式である。これにより、ゴール引数の数がレジスタ数による制限を受けなくなった。そして、レジスタを有効に利用し少ないレジスタ数で効率良く実行できるようになり、コンテキストスイッチの高速化も実現できた。

また、メモリ上にも作業用の領域を用意することにより、複雑なユニフィケーションでも実行できるようにした。この領域は、どうしてもレジスタが不足する場合に使うもので、当面必要でないデータをレジスタ上からここに退避し、空いたレジスタを今必要なデータのために利用できるようにするものである。

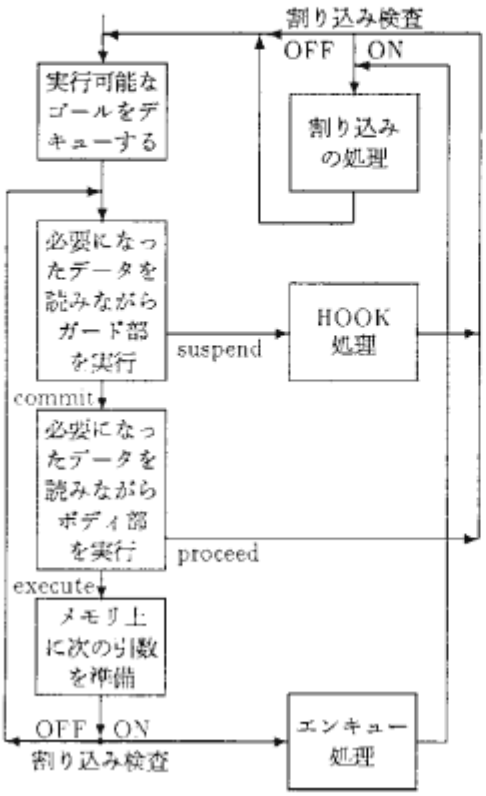

図 2: 新しいゴール実行方式

3.1.1 メモリベース方式による実行

従来の方式では、実行待ちやサスペンドしている時には、ゴールの環境をメモリ上の構造体、ゴールレコードに保持しているが、実行に先立ち、ゴール環境をレジスタ上に読み込み、ゴールレコードは棄ててしまっていた。新しい方式では、実行中のゴールでもこのゴールレコードをそのまま利用するようにし、ゴール環境をレジスタ上になるべく読み込まないようにした。これにより、全てのレジスタを作業用として利用できるようにした。

図 2 に示すように、引数は必要なものだけを、必要になった時に、命令によりレジスタ上に読み込むようにしたので、ゴール実行に先立った準備の手間を最小限に押えることができ、サスペンドする場合にも、保存されているゴールレコードをそのまま使えるので、引数を書き戻す必要がなく、コンテキストスイッチを非常に軽くすることができた。

しかし、直後に (execute命令で) 実行するゴールの場合には、従来は、レジスタ上で引数準備を行なえばよかったのが、一旦メモリ上に書かなければならなくなった。この欠点に関しては、現在はまだ対応していないが、将来、最適化によりある程度補うことができると考えられる。この最適化については、最後で説明する。

3.1.2 メモリベース方式のための命令

メモリベースの処理方式を実現するために引数をゴールレコードからレジスタ上に読み込む命令を用意した。メモリベース方式は、ゴールレコードを普通の構造体と同じように扱う方式ということができる。即ち、ゴールレコードは引数と制御情報を要素とする構造体であり、この構造体を唯一の (物理的な) 引数としてゴールを実行しているのだと考えることができる。従って、構造体の要素を操作する命令とほぼ同じ機能のゴールレコード操作命令を用意した。そして、コンパイラ内部では、ゴールを構造体と同様に扱っている。なお、子ゴールの引数を書き込む命令は従来から用意されていた。

機能	構造体用	ゴールレコード用
read + wait	read_wait	load_wait
read	read	load
write	write	store

新しい方式では、メモリ上にも作業用領域を用意するので、その操作の命令も用意した。

機能	命令
退避 (レジスタ→メモリ)	save
復帰 (メモリ→レジスタ)	restore

3.2 コンパイル時の最適化

新しいコンパイラでは、メモリベース方式を採用するとともに、さらに処理効率を上げるために、いくつかの最適化を導入した。なお、この中には従来のコンパイル方式に適用できるものもある。

3.2.1 ゴールレコード上の引数の利用

メモリベース方式では、メモリ上のゴールレコードを保存しているので、子ゴールを呼び出す場合には、親の引数環境を積極的に利用する最適化を行なうことにした。

従来の方式でも、親の引数環境(レジスタ上にある)を子ゴールにそのまま渡せる場合があったが、それは、すぐ次に (execute命令で) 実行するゴールに限られていた。

新しい方式では、複数の子ゴールを呼び出す場合には、親のゴールレコードをそれらのうちの任意のものに再利用することが可能である。そこで、コン

バイル時の解析により、再利用先として、親の引数環境を最も有効に活用できる子ゴールを選ぶことにした。これにより、引数環境をゴールレコードに書き込む操作を減らすことができるようになった。

例えば、次のようなプログラムでは、foo/5 → foo2/5 と bar/5 → bar2/5 の組合せで、引数の多くが渡されている。このような場合、従来の方式では、親の引数環境を子ゴールにそのまま渡せるのは、foo/5 → foo2/5 の場合だけで、bar/5 → bar2/5 の場合には、新たに割り付けたゴールレコードに引数を書き込まなければならなかった。しかし、新しい方式では、コンパイラが最も適した組合せを選ぶので、どちらでも親の引数環境を利用できるようになる。

```
foo(A,B,C,D,E) :- true |
      foo2(A,B,C,D,X), xxx(E,X).

foo/5:                                  親
  commit                                |
  load              r0,4,r1            -+
  alloc_goal        2,r2         ---+  |
  store             r1,r2,0        -+  |
  alloc_variable    r3              |  |
  store             r3,r2,1        -+  |
  enqueue           r2,xxx/2   <---+  |
  store             r3,r0,4            -+
  execute           foo2/5     <------+

bar(A,B,C,D,E) :- true |
      xxx(E,X), bar2(A,B,C,D,X).

bar/5:                                  親
  commit                                |
  load              r0,4,r1            -+
  alloc_variable    r2                  |
  store             r2,r0,4            -+
  enqueue           r0,bar2/5  <-----+
  alloc_goal        2,r0         ---+
  store             r1,r0,0        -+
  store             r2,r0,1        -+
  execute           xxx/2      <---+
```

3.2.2 不要構造体の要素の利用

3.2.1 で述べたゴールレコード上の引数の利用と同じように、構造体データについても同様の解析を行ない、不要構造体の要素を積極的に利用するようにした。即ち、不要となった構造体の要素データを新たに割り付ける構造体の同じ要素位置にそのまま渡している場合には、もし、その組合せで構造体を再利用することができれば、構造体要素上のデータをそのまま利用することができ、要素をコピーするためのRead/Writeの操作が不要になる。

ただし、構造体ではMRBがONの場合があるので、その時には、不要構造体の再利用が不可能、即ち、このゴールでは不要であるが、ほかのゴールで使用しているかも知れないので、新しい構造体を割り付ける必要がある。そこで、このような時には、そのまま利用するつもりだった不要構造体上の要素を新たに割り付けた構造体にコピーする操作が必要になる。このために、MRBがONのときに、どの要素をコピーすればよいかという情報を持たせた **reuse_with_elements** 命令を用意した。

この最適化は、次のような場合に有効である。

```
foo(abc(A,B)) :- true | bar(xyz(A,B)).
```

もし、この最適化をしない場合には、KL1-Bは、

```
   ...
load_wait    r0,0,r1,susp
is_vector    r1,susp
test_arity   3,r1,susp
read_wait    r1,0,r2,susp
is_atom      r2,susp
test_atom    abc,r2,susp
commit
read         r1,1,r3          ★
read         r1,2,r4          ★
reuse        r1,vector(3),vector(3)
put_atom     xyz,r5
write        r5,r1,0
write        r3,r1,1          ★
write        r4,r1,2          ★
store        r1,r0,0
   ...
```

のようになるが、この最適化により、

```
   ...
load_wait    r0,0,r1,susp
is_vector    r1,susp
test_arity   3,r1,susp
read_wait    r1,0,r2,susp
is_atom      r2,susp
test_atom    abc,r2,susp
commit
reuse_with_elements
     r1,vector(3),vector(3),{0,1,1}
put_atom     xyz,r5
write        r5,r1,0
store        r1,r0,0
   ...
```

のように、★印の付いていたRead/Write命令を削除することができた。

新命令 **reuse_with_elements** は、不要構造体のMRBがOFFの時には、reuse命令と全く同じなので、この場合には削除したRead/Write命令の分が得になる。MRBがONの時には、要素をコピーする必要があるが、このコストは、削除した命令と同じであり、最適化をしない場合に比べて同等であるといえる。

3.2.3 単一待ちの最適化

通常のKL1プログラムは、多くの場合、単一待ちになっている。そこで、このような場合には、メモリ上にあるサスペンションスタックを使用しないで、変数を読み込んだレジスタを指定することで、

サスペンド処理を行なうことにした。このために、次のような命令を用意した。

機能	通常用	単一待ち最適化用
read+wait	read_wait	**read_wait_single**
load+wait	load_wait	**load_wait_single**
suspend	suspend	**suspend_single**

この命令を使うことができた場合には、サスペンションスタックを読み書きする操作や、単一待ちか、多重待ちかの判断が不要になり、サスペンドの処理を軽くできる。また、サスペンションスタックポインタを使わない場合が多くなるので、これを汎用レジスタと兼用にして、作業用にも使うことができるようにした。

4 実験および評価

4.1 実験環境

コンパイル方式を評価するために、新方式に対応したコンパイラでベンチマークプログラムをコンパイルし、UNIXマシン上に構築された2つのKL1処理系、PDSS [6] と VPIM [7] を使って実験を行なった。

新しいコンパイラは、全てKL1自身で記述されており、PIMのセルフコンパイラとして作成したものである。現在は、PDSS上で動作している。なお、このコンパイラは、25K行の大きさがあり、これ自身もベンチマークプログラムとして使用した。

PDSSは単一プロセッサ用の処理系であり、コンパイラが出力するKL1-Bをバイトコードに変換し、それをインタープリタが実行する方式になっている。これには、従来方式と新方式に対応する2つの版がある。

VPIMは、PIMのKL1-Bレベルのシミュレータであり、並列UNIXマシンであるSymmetry上で並列実行できる。今回は、1PEと4PEの構成を用いた。

4.2 ベンチマークプログラム

今回の実験にあたって、表1に挙げた5種類のベンチマークプログラムを使用した。

Queen, Bup, Primeは従来からベンチマークとして用いられてきたものである。ただし、サスペンドが起こらないので、コンテキストスイッチの性能を含めた評価には向かない。

PrimeDDは、要求駆動方式で記述された素数生成プログラムであり、非常に多くのサスペンドが起こる。これは、コンテキストスイッチの性能の評価のために用意した。

表 1: ベンチマークプログラム

名前	リダクション数	内容
Queen	38889	8クイーン問題
Bup	34857	ボトムアップパーザ
Prime	16961	1000までの素数生成
PrimeDD	11896	要求駆動方式による100個の素数生成
KL1comp	約 22 M	KL1コンパイラ

KL1compは新しいコンパイル方式を採用したKL1コンパイラであり、ソースプログラムで約25K行、コンパイル結果のKL1-Bで184K命令の大きさがある。今回は、これを実用プログラムの1つとして実験に加えた。内容的には、コンパイラ自身の一部(6モジュール)をコンパイルする処理をPDSS上で実行した。この時、ファイル入出力などはPDSSで用意されているKL1で記述されたライブラリを用いており、リダクション数にはそれも含んでいる。なお、この実行には約22Mリダクションを要するが、コンパイラに非決定的な部分があるため、実行環境により多少の変動がある。

4.3 レジスタの使用個数

レジスタの使用個数の評価は、KL1comp(述語定義数約1230個)を対象に、従来方式と、新方式の2つのコンパイラによりコンパイルした結果のKL1-Bを静的に調べることにより行なった。なお、コンパイル時の最適化は全て行なっている。集計したのは、各述語を実行するために必要なレジスタの個数であり、図3のような結果になった。このグラフは各述語が必要としているレジスタ数の分布を表すものであり、新方式では、従来方式に比べて少ないレジスタで実行可能なことが分かる。これは、不必要なゴール引数をレジスタ上に読み込むのを止めたことの効果だと考えられる。

同じグラフから、ハードウエアで提供されるレジスタ数が少ない場合の実行性能を推定することもできる。例えば、レジスタ数が16個しかない場合を考えてみると、新方式では、ほとんどの述語が16個以下のレジスタしか要求していない。そのため、メモリ上に用意した作業用領域を使用する回数はわずかであり、KL1compを実行する場合には、ほとんど性能が低下しないと期待できる。さらに少なく、レジスタ数10個の場合を考えてみると、新方式で11個以上のレジスタを要求しているのは約15%であり、レジスタとメモリ上の作業用領域との間で入れ換えを行ないながら実行したとしても十分な性能が得られると考えられる。

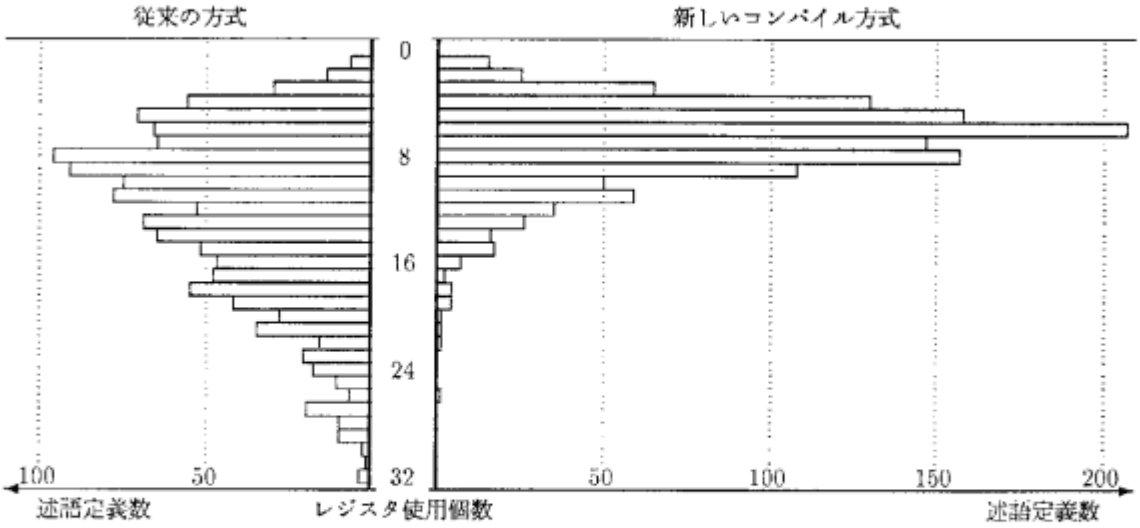

図 3: コンパイルされた述語定義が必要とするレジスタ個数の分布

4.4 メモリアクセス回数

新しいコンパイル方式では、コンテキストスイッチ処理におけるメモリアクセス回数が減少する反面、ゴールを必ずメモリ上に作成するので、メモリアクセス回数が増加する可能性もある。そこで、PDSSとVPIM(1PEと4PE)上で実際にベンチマークプログラムを実行して測定を行なった。

なお、PDSSにはメモリアクセス回数を集計する機能がないので、ゴール引数を操作する回数を測定した。旧方式と新方式の違いは、基本的に、ゴール引数を操作する部分だけなので、この差を求めれば、メモリアクセス回数の増減を知ることができる。また、旧方式の値は新方式に対応した処理系に特別なプローブを入れ、もし旧方式で実行していたらゴール引数の操作が何回になるかを集計した。

測定結果は、表2のようになった。この表の値は1リダクションあたりの回数をまとめたものである。なお、サスペンド率は1リダクションあたりのサスペンド回数である。

これを見ると、特別にサスペンドが多くなるように作られたPrimeDD以外は、ゴール引数の操作回数が、読み込み、書き出しとも1リダクションあたり数回増加しており、ゴールを必ずメモリ上に作成することの影響が大きいと考えられる。読み込みと書き出しの増減を比較してみると、書き込みの増加の方が少なく抑えられている。これは、ゴール引数再利用の最適化の効果であると考えられる。

サスペンドの多いPrimeDDでは、ゴール引数の書き込み回数が少なくなったが、これは、新方式でコンテキストスイッチの処理が軽くなったためだと考えられる。実用プログラムであるKL1compは、サスペンド率が2割ほどあり、全くサスペンドしないベンチマークプログラムより、ゴール引数操作回数の増加は少なめになっている。

VPIMでPE4台の場合には、PE1台の場合に比べてゴール引数操作回数の増加が少なくなっている。これは、負荷分散のための割り込み処理が発生するために、コンテキストスイッチの処理が多くなり、新方式に有利になったためだと考えられる。

なお、ここでの評価はレジスタ数が32個の場合であり、述語の実行に十分な数のレジスタが用意されていることを意味する。レジスタ数が少ない場合には、前節で示したように新方式のほうがレジスタ使用量が少ないので、有利になると考えられる。

4.5 最適化の効果

新しいコンパイル方式では、今回、3つの最適化を導入したので、PDSS上で実際にベンチマークプログラムを実行して測定を行なった。

測定結果は、表3のようになった。測定したのは、ゴール引数の積極的な再利用の最適化に対しては、ゴール引数操作の命令実行回数を、構造体要素の最適化に対しては、構造体要素操作の命令実行回数を、単一待ちの最適化に対しては、サスペンションスタックのpush/pop回数であり、それらを、1リダクションあたりの回数に換算してある。

最も効果が大きかったのは、構造体要素の最適化で、特に実用プログラムのKL1compでは、構造体要

表 2: リダクションあたりのゴール引数操作回数とメモリアクセス回数

プログラム	実行環境	サスペンド率	従来方式 ゴール引数操作[1]		新しいコンパイル方式 ゴール引数操作[2]		増減 = 2 − 1		メモリアクセス	
			read	write	read	write	read	write	read	write
Queen	PDSS	0.00	1.70	1.70	3.69	3.29	+1.99	+1.59	—	—
	VPIM.1	0.00	1.70	1.70	3.69	3.29	+1.99	+1.59	9.19	8.77
	VPIM.4	0.00	1.75	1.75	3.69	3.29	+1.94	+1.55	9.94	9.20
Bup	PDSS	0.00	2.54	2.54	4.88	4.60	+2.34	+2.06	—	—
	VPIM.1	0.00	2.54	2.54	4.88	4.60	+2.34	+2.05	12.60	13.96
	VPIM.4	0.01	2.76	2.76	4.91	4.59	+2.15	+1.83	15.54	15.65
Prime	PDSS	0.00	0.02	0.02	2.94	1.96	+2.92	+1.94	—	—
PrimeDD	PDSS	1.00	3.94	3.94	4.40	2.92	+0.46	−1.02	—	—
KL1comp	PDSS	0.23	3.51	3.51	5.33	4.43	+1.81	+0.91	—	—

表 3: コンパイル時の最適化の効果

プログラム	最適化 G	E	S	ゴール引数に対する load/store 命令数	構造体要素に対する read/write 命令数	サスペンションスタックの push/pop 回数
Queen	×	×	×	7.26	2.34	0.000
	○	×	×	6.98 (−0.28 −3.9%)	2.34 (0.00 0.0%)	0.000 (0.000 0.0%)
	×	○	×	7.26 (0.00 0.0%)	2.00 (−0.34 −14.5%)	0.000 (0.000 0.0%)
	×	×	○	7.26 (0.00 0.0%)	2.34 (0.00 0.0%)	0.000 (0.000 0.0%)
Bup	×	×	×	9.77	2.01	0.000
	○	×	×	9.48 (−0.29 −3.0%)	2.01 (0.00 0.0%)	0.000 (0.000 0.0%)
	×	○	×	9.77 (0.00 0.0%)	1.55 (−0.46 −22.9%)	0.000 (0.000 0.0%)
	×	×	○	9.77 (0.00 0.0%)	2.01 (0.00 0.0%)	0.000 (0.000 0.0%)
KL1 comp	×	×	×	9.79	6.98	0.706
	○	×	×	9.10 (−0.69 −7.0%)	6.99 (+0.01 +0.0%)	0.645 (−0.061 −8.6%)
	×	○	×	9.79 (0.00 0.0%)	5.85 (−1.13 −16.2%)	0.706 (0.000 0.0%)
	×	×	○	9.79 (0.00 0.0%)	6.98 (0.00 0.0%)	0.434 (−0.272 −38.5%)

命令数等は1リダクションあたりの回数。 括弧内は最適化なしの場合と比べた増減数と割合。
最適化の欄のGはゴール引数の再利用、Eは構造体要素の再利用、Sは単一待ちの最適化。

素操作の命令実行回数が1リダクションあたり1回以上減少した。

単一待ちの最適化は割合では大幅に減少したが、元々の操作回数が少ないのでメモリアクセス回数に対する効果はそれほど大きくないようである。

5 メモリアクセスを減らす最適化

新方式では、ゴールを必ずメモリ上に作成するので、述語の実行に十分な数のレジスタが用意された場合にはメモリアクセス回数が多くなることが分かった。この増加回数は、表2からメモリアクセス回数の値の10〜25%に相当し、この分、実行速度が遅くなると考えられる。KL1をより効率良く実行するためには、この問題を解決する必要がある。

メモリアクセス回数の増加は、途中でサスペンドが起きないような最下位サブルーチンで大きくなると考えられる。最下位サブルーチンは、リスト等の各要素について同じ処理を繰り返すことが多く、これは、再帰呼び出しにより実現されている。従来のKL1-Bでは再帰呼び出しによる実行を全てレジスタ上で行なうことができた。しかし、新KL1-Bでは、再帰呼び出しの場合にも一旦メモリ上に引数を書き出すので実行速度が遅くなる。

このようなサブルーチンは、何回も呼び出される場合が多く、これについて対策を立てれば、メモリ

アクセス回数を大幅に減らすことができると考えられる。そこで、このようなサブルーチンは実行に際してあまり多くのレジスタを必要としないのに注目し、これらをコンパイラが自動検出するか、ユーザーがソースコード上に特別な指示を記述することにより、レジスタ上で再帰呼び出しを行なうコードを生成することを考えている。即ち、このような場合に限り、従来のようにレジスタに引数を置いて実行するようなコードを生成するわけである。

例えば、appendの場合には以下のようなコードを生成する。

```
app([]    ,Y,Z):-true|Z=Y.
app([H|X],Y,Z):-true|Z=[H|L],app(X,Y,L).

app/3:
  load                  r0,0,r1       ★
  load                  r0,2,r2       ★
loop:
  wait_single           r1,susp <---+
  is_list               r1,next     |
  commit                            |
  read                  r1,cdr,r3   |
  reuse_with_elements   r1,list,list,[1|0]
  alloc_variable        r4          |
  write                 r4,r1,cdr   |
  unify_bounded         r1,r2       |
  move                  r3,r1       |
  move                  r4,r2       |
  execute_quick         loop    ----+
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  enqueue               r0,app,3
  proceed
next:
  is_atom               r1,fail
  test_atom             [],r1,fail
  commit
  load                  r0,1,r1
  collect_goal          r0
  unify                 r1,r2
  proceed
fail:
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  fail                  app,3
susp:
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  suspend_single        r1,app,3
```

ここでは、実行に先立ち、★印の命令により、当面必要となる第1、第3引数をレジスタ上に読み込んでいる。また、コンテキストスイッチの部分では、☆印の命令によりメモリ上に書き戻している。

実行のメインの部分は全て、レジスタ上で行なわれており、子ゴールを呼び出す部分では、move命令によりレジスタ上で引数を準備している。execute_quick命令では、レジスタ上の環境で子ゴールを呼び出すが、割り込み処理が必要な場合には、この命令はなにも行なわずに、次命令からのstoreとenqueueが実行される。

6 おわりに

新しいコンパイル方式にすることで、ゴールの引数個数やユニフィケーションの複雑さに対するハードウエアのレジスタ数による制限を無くすこと、レジスタの使用効率を良くし少ないレジスタ数で実行可能なこと、コンテキストスイッチの処理を軽くすることの3つの目的を達成することができた。しかし、述語の実行に十分な数のレジスタが用意されている場合には、メモリアクセス回数が増加し、実行速度が低下することが分かった。この対策として、レジスタの使用量が少ない、一部の述語の再帰呼び出しをレジスタ上で行なう最適化を考えており、今後、このような述語をどの程度自動検出できるか、また、この最適化を行なったとしてどの程度メモリアクセス回数を減らすことができるか評価する予定である。

謝辞

日頃ご指導頂いている内田俊一室長をはじめとするICOT第4研究室の研究員の方々に感謝します。また、数々の貴重な助言をいただいた、ICOTならびに各社PIMグループの方々に感謝します。

参考文献

[1] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. TR-208, ICOT, 1986.

[2] Y.Kimura, T.Chikayama. An Abstract KL1 Machine and its Instruction Set. In Proceedings of SLP '87, Sep. 1987.

[3] D.H.D.Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[4] 木村康則, 近山隆. 並列論理型言語KL1の多重参照管理によるガーベジコレクション. 情報処理学会論文誌, pp316-327, Feb. 1990.

[5] 木村康則, 西崎慎一郎, 中越靖行, 平野喜芳. KL1のクローズインデキシング方式. JSPP '89, pp187-194, Feb. 1989.

[6] ICOT 第4研究室. PDSS －言語仕様と仕様手引－. ICOT TM-437, 1988.

[7] 山本礼己, 今井明, 中川貴之, 河合英夫, 中越靖行, 宮崎芳枝, 堂前慶之. 並列推論マシンPIMにおける抽象機械語KL1-Bの実装 －高級機械語を実装するための道具立－. 信学技報 Vol.89 No.168, pp51-56, 1989.