ハイパフォーマンス
コンピューティング 86－7
（2001．5．25）

動的リンクライブラリを用いた実行時最適化の評価

窪田昌史† 暮町尚俊†† 津田孝夫†

数値計算プログラムでは，ループアンローリングなどのループ最適化の効果が大きい．ループ最適化に用いるアンローリング段数などのパラメータの値は，使用する計算機で最適な値が異なるため，コンパイル時に最適な値を解析することは困難である．そこで，我々は計算機に適したループの最適化を実行時に行なうことで高速化を図る手法を提案してきた．本稿では，動的リンクライブラリを利用した実行時最適化手法について述べる．

本手法を，C 言語の行列積，LU 分解のプログラムに適用し，Sun Enterprise450(CPU: UltraSPARC II 300MHz) で実行した．その結果，高速化の全てをコンパイラに任せる場合よりも計算時間の短縮されることが示された．

Evaluation of Runtime Optimizations with Dynamic Link Libraries

Atsushi Kubota,† Naotoshi Kuremachi†† and Takao Tsuda†

Loop optimization techniques such as loop unrolling have large effect on performance of numerical applications. The optimal parameter of loop optimizations such as unrolling step is different by underlying computers. Thus, it is difficult to analyze the optimal parameter at compile time. We have therefore proposed techniques to improve performance by applying loop optimizations that suit for the underlying computers at runtime. This report presents runtime optimization techniques which make use of the functionalities of dynamic link libraries.

We apply these techniques to matrix multiply and LU decomposition programs and execute them on Sun Enterprise 450(CPU: UltraSPARC II 300MHz). As a result, it is shown that the execution time of the programs optimized at runtime is shorter than the one of the programs optimized by the ordinary compiler.

1. はじめに

プログラムの実行性能の高速化手法としては様々なものが提案されている．それらの中にはコンパイラによって自動的に適用されるものもあるが，ソースプログラムの書き換えを行わなければ適用できないものも多い．

例えば，ループ実行の最適化技法として，ループ交換，ループアンローリング，ブロック化など，様々な技法が知られているが，これらの適用の可否は実行される計算機の CPU 性能，メモリやキャッシュの容量などによって異なるため容易に求めることはできない．ループアンローリングを適用する場合，最適なアンローリング段数は実行環境によって異なるため容易に求めることはできない．ブロック化を適用する場合も，最適なブロックサイズを求めることは困難である．そのため，さまざまなアンローリング段数やブロックサイズのプログラムを記述しては実行することを繰り返し，最適な段数や最適なブロックサイズを求めるなどの手法がとられているのが現状であり，これらのパラメータをコンパイラによって自動化することは困難である．

これは，コンパイラによる最適化は，プログラムの実行前にその実行過程，つまり，命令の実行順序やメモリ上のデータのアクセス順序を予測することで行われていることに起因する．

ATLAS[1] は，パラメータ探索によって最適なパラメータを求める自動チューニング機能を持つ数値計算パッケージである．ATLAS では，あらゆるパラメータで実行時間を計測するため，実行した計算機においてほぼ最適なパラメータを求めることができるが，数十秒の実行時間の行列積のパラメータ探索に数時間もの非常に長い時間がかかることもある．そのため，一度，実行する計算機に最適化されたプログラムを得て，それを同じ計算機で繰り返し実行する用途にしか適さない．

† 広島市立大学 情報科学部
Faculty of Information Sciences, Hiroshima City University
†† 広島市立大学 大学院 情報科学研究科
Graduate School of Information Sciences, Hiroshima City University

一方，ループアンローリングなどのループ最適化技法では，アンローリング段数などのパラメータが必ずしも最適値でなくとも，最速に近い性能のプログラムを得ることができる．

そこで，我々は，パラメータ探索の範囲を狭め，比較的短いパラメータ探索時間で最速に近い性能のプログラムを得る手法について検討を進めている．少数のパラメータの中から最適に近いパラメータを比較的短時間で求めることが可能であれば，実行時にパラメータ探索を含めたとしても，高速化の達成が期待できると考えられる．我々はこの方針に基づき，Javaプログラムのループ実行時最適化手法[2]を提案し，その有効性を示してきた．

本稿では，Cプログラムの一部を動的リンクライブラリとし，実行時にそのライブラリを入れ換え，プログラムを最適化する手法を示す．また，本手法をいくつかのベンチマークプログラムに適用し，その適用範囲と有効性を評価した結果について述べる．

以下，2章で動的リンクライブラリを用いた実行時最適化について，3章で本手法を数値計算プログラムへ適用し，その有効性を評価した結果について述べる．4章で総括を行い，今後の課題について述べる．

2. 実行時最適化

最適化を行うために十分な情報がコンパイル時には得られないが，入力データなどの実行時に得られる情報を利用すると，性能向上が見込めることがある．

例えば，マルチバージョンコードを生成し，ある変数の値を実行時に検査する条件文によって最適化されたコードと最適化されていないコードへと実行を振り分ける手法などが提案されている．

しかし，最適化の中にはループアンローリングなどのように，アンロール段数などのパラメータを変化させなければ最適化できないものがある．あらゆるアンロール段数についてのコードをコンパイル時に生成しておくことは不可能であるため，この場合はマルチバージョンコードでは対処できない．そこで，実行時にコードを書き換えて最適化する手法が提案されてきた．

実行にコード生成を行う言語／言語処理系としては，DyC[3]やtcc[4]などがある．村井ら[5]は，分散メモリ型並列計算機へのメモリ割り付けを実行時に最適化する手法を提案している．

我々は，最適化対象部分を動的リンクライブラリ化し，実行時にそのバイナリを生成し，ロードし直すことでプログラムの一部分の入れ換える手法を採用した．動的リンクライブラリという広く用いられている機構を利用することにより，様々なOS，実行環境での本手法の実現が容易になると考えられる．

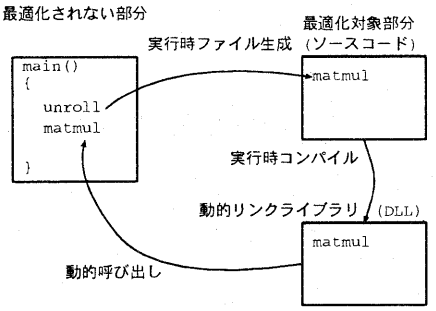

図1 動的なプログラムの呼び出し

2.1 動的なプログラムの呼び出し

実行時に最適化されるプログラムを，図1に示すように最適化の対象となる部分とそれ以外の部分に分ける．最適化されない部分には，パラメータ探索を行って最適化を制御する機能が新たに加わる．現時点では，元のプログラムをこのように変更する作業は，プログラム実行前にユーザがあらかじめ行っている．

最適化対象部分は，動的リンクライブラリ中に存在するため，C言語であれば関数としておく必要がある．この関数のソースコードを含むファイルを実行時に生成し，そのファイルをコンパイルして動的リンクライブラリにする．最適化対象部の実行は，この動的リンクライブラリ中の関数を呼び出すことで実現される．

動的リンクライブラリ中のある関数funcを再コンパイルして変更し，実行するには以下のような手順をとることになる．

(1) それ以前にリンクしていた動的リンクライブラリ中の関数funcがあれば，実行中のプロセスのアドレス空間から解放する．
(2) 最適化部を再コンパイルして動的リンクライブラリとする．
(3) 新たな動的リンクライブラリを実行中のプロセスのアドレス空間に割り付ける．
(4) 動的リンクライブラリ内の関数funcを動的に呼び出して実行する．

2.2 最適化パラメータ探索方法

行列積を例として，最適化パラメータの探索方法を示す．行列Aと行列Bの積を行列Cに格納するものとし，行列A, B, Cの各i, j要素をそれぞれa_{ij}, b_{ij}, c_{ij}，行列のサイズを$n \times n$とすると

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

となる．この行列積のプログラムの主要ループ部分をC言語で記述すると，以下のようになる．

```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

アンローリング段数を変化させて次のようなアンローリング段数探索用のプログラムを得る (ここでは，コードが繁雑にならないように，N は，4 で割り切れる値の場合の例を示す)．探索プログラムの計算量が小さくなるように，最外側ループの繰り返し回数は 1 回に変更している．

```
k=0; /* 1 回のみの繰り返し */
for (i=0; i<N; i++)
  for (j=0; j<N; j+=4) {
    c[i][j+0] += a[i][k] * b[k][j+0];
    c[i][j+1] += a[i][k] * b[k][j+1];
    c[i][j+2] += a[i][k] * b[k][j+2];
    c[i][j+3] += a[i][k] * b[k][j+3];
  }
```

ループアンローリングなどの最適化に用いるパラメータを探索するために，複数のパラメータを用いて最適化対象部分が実行され，実行した中で最も計算時間が短いパラメータを選び出される．パラメータ探索は以下のような手順で行われる．

(1) 計算量が等しくなるようなパラメータ探索用のプログラムを作成する．行列積にループアンローリングを適用する場合ならば，最外側ループの繰り返し回数のみを 1 などとして計算量を小さくし，最内側ループにアンローリング段数を変化させて適用したプログラムを複数作成しておく．

(2) パラメータ探索用のプログラムを，探索したいパラメータの個数分だけ関数として異なるファイルに出力する．

(3) 生成されたすべてのファイルを一括してコンパイルすることで動的リンクライブラリにする．

(4) 動的リンクライブラリにあるパラメータ探索用の関数のうち，1 つを呼び出して実行する．このとき，実行時間を測定しておく．

(5) (3) で計測した実行時間を，他のパラメータでの実行時間と比較する．比較したパラメータの中で最も実行時間が短いものを保持しておく．

(6) パラメータ探索後，最も実行時間が短くなるパラメータで最適化されたプログラムを作成 (ソースコードの出力，コンパイル) し直して実行する．

3. 性能評価

行列積と LU 分解のプログラムに実行時に最適化を適用し，その性能を測定した．プログラムの実行時間の計測は，Sun Enterprise 450 で行った．表 1 にその仕様を示す． コンパイラは C コンパイラ (Sun Workshop Compilers 4.2.1) を使用した．

表 1 Sun Enterprise 450 の仕様

CPU	CLOCK	1 次キャッシュ
UltraSPARC II	300MHz	I:16KB/D:16KB
2 次キャッシュ	主記憶	OS
2MB	512MB	Solaris 2.6

3.1 行 列 積

行列積のプログラムにループアンローリングとブロック化を適用した．

3.1.1 ループアンローリング

3 重ループの外側ループの外側ループ変数 k にループアンローリングを適用する．アンローリング段数を 2,4,5,10,15,20,25,30 と変化させ，外側から 2 番目の i について，繰り返し回数を 2 としたプログラムをパラメータ検索に用いる．

3.2 ブロック化

図 2 に行列積にブロック化を適用した例を示す．ブロックサイズを 2 のべき乗とその ±1 である 3, 4, 5, 7, 8, 9, 15, 16, 17, 31, 32, 33, 63, 64, 65 と変化させ，最外側の k と kk のループの繰り返し回数を 1 としたプログラムをパラメータ探索に用いる．

```
for (k=0; k<N; k+=4)
  for (i=0; i<N; i+=4)
    for (j=0; j<N; j+=4)
      for (kk=k; kk<k+4; kk++)
        for (ii=i; ii<i+4; ii++)
          for (jj=j; jj<j+4; jj++)
            c[ii][jj]+=a[ii][kk]*b[kk][jj];
```

図 2 行列積にブロック化を適用した例

3.3 ループアンローリングとブロック化の同時適用

図 2 のブロック化を適用した行列積のプログラムの kk のループにアンローリングを適用する．まず，最適に近いブロックサイズを求め，次にブロックサイズを固定してアンローリング段数を探索する．探索プログラムが複雑にならないように，ブロックサイズは 2 のべき乗の 2, 4, 8, 16, 32, 64 を探索し，最外側の k と kk のループの繰り返し回数を 1 としたプログラムをパラメータ探索に用いる．次に，ブロック化された 6 重ループの最外側ループ変数 k の繰り返し回数を 1 回とし，アンローリング段数を 2 からブロックサイズまでの 2 のべき乗として探索する．

3.3.1 実 行 時 間

C コンパイラの最適化オプションによってコンパイル時間や生成されるコードの性能に顕著な差が見られるため，最適化オプションに-xO3 と-fast(-xO4, -dalign など，性能を重視したオプションの最適な組み合わせ) を指定した場合の実行時間を示す．行列のサイズは 1024×1024 である．

表 2,3 に実行時間 (秒) を示す．動的リンクライブ

表 2　行列積 (-xO3) の実行時間 (秒)

	(1)DLL	(2) 最速	(1)-(2)
アンローリング	45.28(5)	33.09(10)	12.19
ブロック化	57.75(16)	40.71(27)	17.14
両方	37.59(8,16)	24.94(8,16)	12.65
最適化なし	74.82		

表 3　行列積 (-fast) の実行時間 (秒)

	(1)DLL	(2) 最速	(1)-(2)
アンローリング	40.18(10)	22.00(10)	18.18
ブロック化	50.05(17)	32.96(27)	17.09
両方	33.74(8,16)	17.73(8,16)	16.01
最適化なし	45.70		

ラリを用いた実行時最適化を適用したプログラムを (1)DLL, あらかじめ最適なパラメータを用いて最適化したプログラムを (2) 最速とする．実行時間の右の括弧内の数値は最適と判断されたパラメータの値である．アンローリングとブロック化の両方を適用した場合の括弧内は，順に，最適なアンローリング段数とブロックサイズの値である．また,「最適化なし」とは，アンローリング，ブロック化などの最適化を行っていない場合である．コンパイラへの最適化オプション (-xO3, -fast) は指定している．

最適化オプションに-xO3 を指定した場合 (表 2) は，実行時の「最適化なし」のプログラムに比べて，(1)DLL の最適化の効果は大きい．しかし，実行性能は，最適化オプションに-fast を指定した場合 (表 3) の方が高い．

最適化オプション-fast の場合の性能向上を表 3 から算出すると，ループアンローリングのみを適用した場合，(1) の DLL は実行時の「最適化なし」のプログラムに比べ 14%の性能向上が得られ，同様に，ループアンローリングとブロック化の両方を適用した場合は 35%の性能向上が得られることがわかる．しかし，ブロック化のみを適用した場合，9%の速度低下となった．また，ブロック化とアンローリングの両方を適用する場合，(1)DLL が実行時に求めたパラメータは (2) 最速のパラメータの値と等しい．ゆえに，ブロック化とアンローリングの両方を適用した場合の効果が大きいといえる．

3.3.2　実行時オーバヘッド

表 4　行列積 (-xO3) の実行時オーバヘッド (秒)

	(3) 実計算所要時間	(2) 最速	(3)-(2)
アンローリング	33.51 (5)	33.09 (10)	0.42
ブロック化	42.51 (16)	40.71 (27)	1.80
両方	25.10 (8,16)	24.94 (8,16)	0.16

表 4,5 は，表 2,3 の (1)DLL において実際の計算そのものに要する (3) 実計算所要時間と (2) 最速の実行時間との差を示す．(3) 実所要時間とは，(1)DLL から動的にプログラムを変更することに伴うパラメータ探索，コンパイルに要するオーバヘッドを除いた時間である．

表 5　行列積 (-fast) の実行時オーバヘッド (秒)

	(3) 実計算所要時間	(2) 最速	(3)-(2)
アンローリング	22.10 (10)	22.00 (10)	0.10
ブロック化	33.88 (17)	32.96 (27)	0.92
両方	17.71 (8,16)	17.73 (8,16)	0.02

(3) 実計算所要時間と (2) 最速のアンローリングやブロック化のパラメータの値が等しい場合の実行時間の差は，動的リンクライブラリ内に存在する関数を動的に呼び出すことによるオーバヘッドであると考えられる．

3.3.3　実行時コンパイル時間

表 6　行列積の実行時コンパイル時間 (秒)

	(1)DLL(-xO3)	(1)DLL(-fast)	差
アンローリング	6.84	12.22	5.38
ブロック化	6.38	8.02	1.64
両方	5.34	9.60	4.26

表 6 は動的最適化を行うときのコンパイルに要する時間である．表 6 から，オプション-fast のような高度な最適化を行う場合，オプション-xO3 の場合と比べて実行時のコンパイル時間が増加していることがわかる．そのため，(1)DLL において，最適化のレベルが低いオプション-xO3 などを適用した方がコンパイルに要するオーバヘッドが小さくなり，最終的に得られる性能が高くなる可能性がある．

また，アンローリングを適用するとソースコードの行数が増加するため実行時のコンパイル時間も長くなる傾向にある．

3.4　LU 分解

LU 分解は，連立一時方程式の求解法の 1 つである．LU 分解のプログラムに多段同時消去を適用することで高速化を図ることができる．これは，多段同時消去により，ループの繰り返し回数を減らすことによる分岐命令の実行回数の削減と，ロード／ストア命令の削減が行われるためである．

以下に多段同時消去の簡単な例を示す．

```
for (j=0; j<N; j++) {
  (中略)
  for (ii=j+1; ii<N; ii++) {
    for (jj=j+1; jj<N; jj++)
      A[ii][jj] -= A[ii][j] * A[j][jj];
  }
}
```

このプログラムに3段の多段同時消去を適用すると次のようなプログラムを得る (ここでは，プログラムが繁雑にならないように，Nが3で割り切れる場合を示す).

```
for (j=0; j<N; j+=3) {
  (中略)
  for (ii=j+1; ii<N; ii++) {
    for (jj=j+1; jj<N; jj++)
      A[ii][jj] = A[ii][jj]
                - A[ii][j+0] * A[j+0][jj]
                - A[ii][j+1] * A[j+1][jj]
                - A[ii][j+2] * A[j+2][jj];
  }
}
```

上記の例で示したように，多段同時消去を適用することで分岐命令の実行回数が1/3に削減され，ロード／ストア命令も1/3に削減されており，高速化が期待できる.

3.4.1 実行性能

LU分解でもコンパイル時の最適化オプションにより実行時間に顕著な差が見られるため，最適化オプションを-Oとした場合 (表7) と，-fastとした場合 (表8) のMflops値を示す．行列のサイズは1000×1000, 2000×2000としている.

動的リンクライブラリを用いた実行時最適化を適用したプログラムが (1)DLL，あらかじめ最適な段数を求めておいて実行したプログラムが (2) 最速，多段同時消去を行っていないプログラムが (3) 最適化なし である．括弧内の数値は，最適と判定された多段同時消去の段数である.

表7 LU(-O) の実行性能 (Mflops)

	(1)DLL	(2) 最速	(3) 最適化なし
1000×1000	46.63 (8)	62.89 (12)	29.90
2000×2000	54.28 (8)	60.20 (12)	27.17

表8 LU(-fast) の実行性能 (Mflops)

	(1)DLL	(2) 最速	(3) 最適化なし
1000×1000	38.69 (8)	75.33 (14)	34.76
2000×2000	58.67 (8)	71.87 (14)	30.25

表7から，オプション-Oで，1000×1000では (1)DLLと (2) 最速のMflops値にかなりの差があるが，2000×2000では (2) 最速のMflops値にかなり近い値であるといえる．また多段同時消去を行っていない (3) と比べ大幅に高速化されていることがわかる．(1)DLLは，(3) 最適化なし と比べ，1000×1000では56%，2000×2000では100%性能が向上している．表8から，同様に，オプション-fastでの (1)DLLは，(3) 最適化なし と比べ1000×1000では11%，2000×2000では94%性能が向上している.

これらから，動的リンクライブラリを用いた実行時最適化は，LU分解に多段同時消去を適用する場合において有効であると考えられる.

表7，8から，オプション-O，-fastともに，実行前に多段同時消去を適用して実行したプログラムでは2000×2000よりも1000×1000のMflops値が高くなっているのに対し，実行時に多段同時消去を適用したプログラムは1000×1000よりも2000×2000のMflops値が高くなっていることがわかる．これは，実行時の多段同時消去のオーバヘッドが大きいこと，特にオプション-fastを指定した場合に大きいことを示している.

3.4.2 実行時コンパイル時間

表9 LU(-O) の実行時コンパイル時間 (秒)

	実行時間	コンパイル (秒,%)
1000×1000	15.12	2.96 (20)
2000×2000	98.25	2.99 (3)

表10 LU(-fast) の実行時コンパイル時間 (秒)

	実行時間	コンパイル (秒,%)
1000×1000	17.23	6.51 (38)
2000×2000	90.90	6.78 (7)

実行時のコンパイラの起動によってオーバヘッドとなる時間と全体の実行時間に占める割合 (%) を表9と表10に示す．オプションが-Oの場合は約3秒であるのに対し，-fastの場合は6秒以上かかっている．行列サイズが2000×2000の場合には全体の実行時間に対するこのオーバヘッドが3%から7%と小さいが，1000×1000の場合には20%から38%にもなるため性能向上が妨げられている.

実行時にコンパイラを起動して最適化を行う場合に，起動するコンパイラが高度な最適化を行えば，それに伴って実行時の最適化のオーバヘッドも大きくなる．実行時間の短いプログラムでは，実行時のコンパイラによる最適化ではあまり高度なことは行わない方がよい場合も生じてくる.

4. おわりに

本稿では，C言語のプログラムに，動的リンクライブラリを利用して実行時に最適化を行ったときの性能向上を行列積，LU分解などのプログラムで評価した結果について述べた.

行列サイズ1024×1024の行列積では，実行時にループアンローリングとブロック化の両方を適用した場合，

35%性能が向上することを，行列サイズ2000×2000のLU分解では，実行時に多段同時消去を行うと，100%性能が向上することを確認した．

今回用いた手法では，プログラムの実行中にコンパイラを起動し実行することによるオーバヘッドが大きい．また，最適化によってコードを書き換えて動的リンクライブラリ化した関数の呼び出しのオーバヘッドも，呼び出し回数が増加するにつれて大きくなるおそれがある．

今後は，実行時の最適化の適用範囲を逐次のC言語プログラムだけではなく，並列のC, Fortran言語などのプログラムへと広げていく予定である．また，コンパイル時に，もとのプログラムから，実行に最適化すべき部分を抜きだし，それを制御する部分を自動的に生成すること，最適化のパラメータ探索アルゴリズムを改良すること，実行時のコード生成をコンパイラを起動する方法ではなくバイナリコードからバイナリコードへの変換とすることなども検討すべき課題である．

謝　辞

研究を行なう上で，有益なご助言をいただいた広島市立大学コンピュータアーキテクチャ講座の諸氏に感謝いたします．

参考文献

1) R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. SC98*, November 1998. Orlando, FL.
2) 山崎泰伯, 窪田昌史, 津田孝夫. Javaクラスファイルの実行時ループ最適化手法. 情処研報 2000-HPC-82-21, August 2000.
3) Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. A evaluation of staged run-time optimizations in DyC. In *Programming Language Design and Implementation(PLDI)*, pp. 293–304, Atlanta, GA, USA, May 1999.
4) Massimilizno Poletto, Dawson R. Engler, and M. Frans Kaashoek. `tcc`: a system for fast, flexible, and high-level dynamic code generation. In *Programming Language Design and Implementation(PLDI '97)*, pp. 109–121. ACM SIGPLAN, June 1997. Las Vegas, NV, USA.
5) 村井均, 荒木拓也, 松浦健一郎, 末広謙二, 妹尾義樹. 実行時再コンパイルによる並列プログラムのメモリ割り付け最適化. 情処研報 99-HPC-79-11, December 1999.