計算機アーキテクチャ 146－6
（２００２．２．１）

ソフトウエア制御オンチップメモリのための最適化コンパイラの構想

藤田元信[†] 近藤正章[†] 中村宏[†]
千葉滋[††] 佐藤三久[†††]

近年プロセッサとメモリの性能差はますます拡大する方向にあり、特にハイパフォーマンスコンピューティング分野においては、それが性能に与える影響が大きい。そこでこの問題に対する一つの解決策としてソフトウエア制御可能なオンチップメモリを採用するメモリアーキテクチャが提案されている。本稿ではそのようなアーキテクチャの一つである SCIMA を対象として、これまでに作成したディレクティブベースのコンパイラの紹介を行い、さらに現在検討している自動最適化コンパイラの構想についても述べる。

A framework of the Optimize Compiler for Software-Controlled On-Chip Memory

Motonobu FUJITA,[†] Masaaki KONDO,[†]
Hiroshi NAKAMURA,[†] Shigeru CHIBA[††]
and Mitsuhisa SATO[†††]

Recently, performance disparity between processor and main memory has increased. Serious performance degradation has often occurred especially in high-performance computing due to the disparity. To solve this problem, a new VLSI architecture called SCIMA has been proposed. SCIMA integrates processor and a part of main memory into a single chip. In this paper, we propose a directive based SCIMA compiler, and present a framework of the optimize compiler for SCIMA to realize automatic optimization using software-controlled On-Chip memory.

1. はじめに

近年、クロック周波数の向上、命令レベル並列性の活用などによりプロセッサの高性能化が達成されてきた。一方、プロセッサと主記憶の性能差はますます拡大する方向にあり、今後もさらに拡大していくと考えられる。このため、プロセッサの性能はメモリの性能に制限されてしまっている。特に、科学技術計算に代表されるハイパフォーマンスコンピューティング分野においては、下位のメモリ階層へのアクセス（オフチップアクセス）が頻発し、性能が大きく低下する。

† 東京大学 先端科学技術研究センター
Research Center for Advanced Science and Technology, The University of Tokyo
†† 東京工業大学 情報理工学研究科 数理・計算科学専攻
Department of Mathematical and Computing Sciences, Tokyo Institute of Technology
††† 筑波大学 電子・情報工学系
Institute of Infomation Sciences and Electronics, The University of Tsukuba

従来、この問題に対処するためにキャッシュメモリが用いられてきた。しかしハードウエアによりリプレースメントやアロケーションの制御を行うキャッシュでは、再利用性のあるデータを最大限に活用することはできない。そのため再利用性をより活用するソフトウエア手法に関する研究が多くなされてきたが、ハードウエアで制御されるキャッシュをソフトウエアにより最適化するには次のような問題点がある。

- 配列の大きさによっては配列内の干渉が発生する場合がある
- 複数の配列をキャッシュに載せる場合に配列間の干渉が発生する場合がある
- 再利用性の有無に関わらずキャッシュに載せるデータを選択できない

そこで、従来のキャッシュに加えて、ソフトウエアによりアドレス指定可能な主記憶の一部をチップ上に載せることでデータの再利用性を最大限活用するメモリアーキテクチャである SCIMA (Software Controlled Integated Memory Architecture) が提案されている[1)]。SCIMA ではチップ上に実装した主記憶の

一部 (オンチップメモリ) とオフチップメモリ間のデータ転送の制御をソフトウエアで行う。

これまで、いくつかのアプリケーションを対象にSCIMA の有効性に関する評価が行われてきた。評価に際しては、ソフトウェア制御のオンチップメモリを用いるために、オンチップメモリ・オフチップメモリ間のデータ転送を、関数 (サブルーチン) を用いてソースコード上で書き表し、既存のコンパイラでコンパイルすることで、評価コードを作成していた。

このように、オンチップメモリの制御を関数を用いて表現することで、詳細にデータ転送などの動作を指定することが可能であったが、その表現の自由度と引き替えに、既存のアプリケーションのソースコードを大きく変更しなければならず、ユーザの負担が大きいという問題があった。そこで我々は、既存のソースコードに指示文を挿入することで、簡単に SCIMA のオンチップメモリを用いた最適化を可能とすることを目的に、ディレクティブベースのコンパイラを作成している。

ディレクティブベースのコンパイラを用いることで、SCIMA 用の最適化プログラミングは、以前に比べ比較的容易に行なえるようになると予想される。しかし、本コンパイラを用いることで最適化が容易になるとはいえ、SCIMA についてある程度の知識を持つユーザでなければ、最適化をすることは難しい。そのため、最終的には既存のキャッシュベースのアーキテクチャ用に作成したアプリケーションを、再コンパイルするだけで、SCIMA 用の最適化を行うコンパイラを作成することが望ましい。そこで我々は、自動で SCIMA のオンチップを用いた最適化を行なうべく、自動最適化コンパイラの検討を行なっている。

本稿ではまず、これまでに作成したディレクティブベースのコンパイラについての紹介を行なう。また現在検討中の自動最適化コンパイラの構想についても述べる。

2. SCIMA

2.1 SCIMA のアーキテクチャ

SCIMA のアーキテクチャを**図 1** に示す。SCIMA では、チップ上のメモリとして、キャッシュに加えオンチップメモリを搭載する。キャッシュはハードウェア制御によりデータ配置・置き換えが行われるのに対し、オンチップメモリは、ソフトウェアでデータ配置・置き換えの指定が可能である。

2.2 拡張命令

SCIMA では、page-load/page-store と呼ぶオンチップメモリ–主記憶間のデータ転送命令を ISA (命令セットアーキテクチャ) 上に備える。この命令により、オンチップメモリのデータ配置・置き換えをソフトウェアで行うことが可能となる。本命令は、データ転送元の開始番地、データ転送先の開始番地、転送サイズ、ブロック幅、ストライド幅、の 5 オペランドをとる。また、本命令によるデータ転送は *page* という大きな粒度で行われる。さらに本命令はブロックストライド転送機能を備える。この機能により、不連続なデータをオンチップメモリ上の連続領域に転送させることができるため、無駄なデータ転送を省き、チップ内の記憶領域を有効に利用可能である。

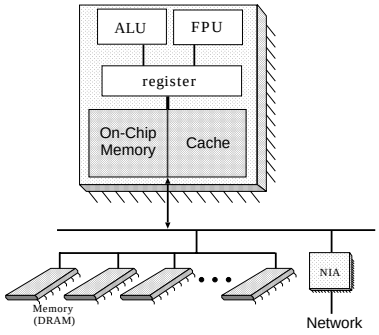

図 1 SCIMA の構成図

オンチップメモリ領域は *page* に分割され、この *page* を単位としてメモリアクセス順序保証などの管理を行う[2)]。*page* のサイズは 2 のべき乗である。page-load/page-store 命令で転送できる最大データサイズはこの *page* のサイズであり、*page* を跨いでの転送はできない。

3. SCIMA 用コンパイラ

3.1 オンチップメモリの利用

SCIMA では、ソフトウエア制御のオンチップメモリを用いて性能向上を図る。そのため、ハードウエアにより制御される従来のキャッシュとは異なり、オンチップメモリを利用するためにはユーザが明示的にデータ転送の指示を行う必要がある。

これまで、オンチップメモリへのデータ転送を指示するためにソースプログラム上では page-load/page-store 命令を関数を用いて表現していた。それら関数はこれまでに行われた SCIMA の有効性に関する評価にも用いられているが、転送先オンチップアドレスを直接指定できる点をはじめとして高い自由度を持つ反面、オンチップメモリ領域の管理をすべてユーザ責任で行う必要があり、ユーザにとって大きな負担であった。この、page-load/page-store 命令を関数を用いて表現したソースプログラムの例を**図 2** に示す。以下に、SCIMA 関数に用いられている部分について簡単に説明を行う。

call set_OnChipAddr(On_Chip) SCIMA では、オンチップメモリ領域は論理アドレス空間上に確保される。このオンチップメモリ領域をアクセス

```
      integer N
      double precision On_Chip(OCM_SIZE)
      double precision x(N), y(N), sum
      integer i,j

c  *****On_Chip(1)を物理的なOCM開始アドレスに一致させる *****
      call set_OnChipAddr(On_Chip)

      tmp=0.0d0

c  *****ユーザ指定要素をユーザ指定アドレスに転送 *****
      call load(x(i),On_Chip(1)  ,8*N,8*N,0)
      call load(y(i),On_Chip(N+1),8*N,8*N,0)

      do i=1, N
         sum=sum+On_Chip(i)*On_Chip(N+i)
      enddo
```

図 2 ベクトル内積計算 (SCIMA 関数を用いた例)

するために、特別な配列 (図 2 の On_Chip) を用意する。本関数呼び出しを用いることで、その配列の先頭アドレスのポインタを、オンチップメモリ領域を示す論理アドレス空間の先頭のアドレスに一致させる。従って、この配列をアクセスすることで、オンチップメモリへのアクセスが可能となる。

call load(x(i),On_Chip(1),8*BL,8*BL,0) 配列 x の i 番目の要素から始まる BL 個の要素を、オンチップメモリ領域用の配列 On_Chip の 1 要素目から始まる領域へ転送する。本関数の第 3 引数が総転送サイズを表している (バイト単位にするために 8 を乗じている)。この例では、連続な領域を転送しているが、ブロックストライド転送を行なう場合は、第 4、第 5 引数にそれぞれブロックサイズ、ストライド幅を指定する。

なお、最内ループにおける実際の演算は、On_Chip(i) のように、オンチップメモリ領域用の配列を用いた演算に置き換える必要がある。

3.2 SCIMA 用ディレクティブベースコンパイラ

オンチップメモリを利用すれば性能向上が見込めるとはいえ、以上の例から分かるようにそれを実現するにあたっては既存のソースコードに大幅な変更を加える必要があり、関数を用いた表現はユーザ負担が大きいものであった。

そのため、我々は SCIMA 用の最適化プログラミングを容易にする目的で、SCIMA 用ディレクティブベースコンパイラを作成した。本コンパイラは、既にディレクティブ解釈の枠組みを持つ *Omni OpemMP Compiler*[3] をベースにしている。

3.2.1 ディレクティブの仕様

今回実装したディレクティブの仕様のうち、主要なものについて**図 3** に示す。 なお、図 3 の他にも、オンチップメモリ上に 2 つの領域を確保し、交互にその領域にデータ転送をしながら演算を行なうことで、演算とデータ転送時間をオーバラップさせるダブルバッファリング用のディレクティブも用意されている。

関数ベースのものに比べ、本ディレクティブを用いて最適化を行なうことで、以下のような利点がある。

- オンチップメモリ領域を表す配列を用いることなく、オンチップメモリを用いることができる
- オンチップメモリの確保や、page-load/page-store 命令の動作などを、指示文により簡単に表現可能
- ループ中の演算部分に手を加える必要がない

さらに、ディレクティブで SCIMA 専用の動作を表現することで、既存のソースコードのセマンティクスに影響を与えないことから、ソースプログラムの可搬性が高くなる。これも、本コンパイラを用いることの利点として考えている。

```
      integer N
      double precision x(N), y(N), sum
      integer i, j

c  ***** x, y用のオンチップメモリ領域をサイズN確保する *****
!$scm begin (x, N, 0)
!$scm begin (y, N, 0)

      tmp = 0.0d0

c  ***** この反復で使う要素をオンチップに転送 *****
!$scm load (x, i, N)
!$scm load (y, i, N)

      do i = 1, N
         sum = sum + x(i) * y(i)
      enddo

c  ***** y,xの順で begin を閉じる *****
!$scm end (y)
!$scm end (x)
```

図 4 ベクトル内積計算 (SCIMA ディレクティブを用いた例)

3.2.2 ディレクティブ利用例

図 2 に示したベクトル内積計算の例に対応するものとして、SCIMA 用ディレクティブを用いて最適化した例を**図 4** に示す。用いられているディレクティブについて以下に簡単に説明する。

!$scm begin (x, N, 0) 配列 x 用のオンチップメモリ領域をサイズ BL だけ確保する。この領域は対応する! $scm end まで有効となる。配列 x は連続的に参照されるため、第 3 引数のストライド幅は 0 となる。

!$scm load (x, i, BL) x(i) を基点として、BL 個の要素を配列 x 用に確保されているオンチップメモリ領域に転送する。

最内ループにおける演算は、コンパイラにより自動的にオンチップメモリ領域への参照に変換される。このため、ユーザはインデックスの計算などを行なう必要がなくなる。

3.3 オンチップメモリ利用の自動化

ディレクティブベースの SCIMA 用コンパイラにより、SCIMA のオンチップを用いた最適化プログラミングは以前と比較して容易になると期待できる。しかしそれでもなお、SCIMA 用最適化は SCIMA に関してある程度の知識を持つユーザでなければ難しいと考えられる。

!$scm begin (<配列名>,<size_1>,<size_2>,...,<size_n>,<ストライド幅_1>,<ストライド幅_2>,...,<ストライド幅_n>)

意味：指定された配列用のオンチップメモリ領域を確保する。対応する**!$scm end**との間にある指定された配列への参照はオフチップメモリ 参照からオンチップメモリ参照に自動的に置き換えられる。<size_i> は各次元ごとの大きさを表し、<ストライド幅_i>は各次元のストライド幅を表す。ストライド幅が0のときはストライド機能を使わない。

!$scm end (<配列名>)

意味：対応する**!$scm begin**との間で指定された配列のオンチップ領域を解放する。

!$scm load (<配列名>,<index_1>,<index_2>,...,<index_n>,<size_1>,<size_2>,...,<size_n>)

意味：指定された配列の<index_1>,<index_2>,...,<index_n>を基点とし、<size_1>,<size_2>,...,<size_n>の大きさの領域をオンチップメモリに転送する。このとき転送サイズはbeginで確保した大きさを越えてはならない。

!$scm store (<配列名>)

意味： 指定された配列のオンチップメモリの領域をオフチップメモリ領域へ転送する。基点や大きさは直前のloadで指定された値が用いられる。

図 3 SCIMA ディレクティブの仕様 (一部)

表 1 配列のアクセスの特徴の分類

アクセスの特徴	分類
再利用性	あり / なし
連続性	連続 / ストライド / 不規則

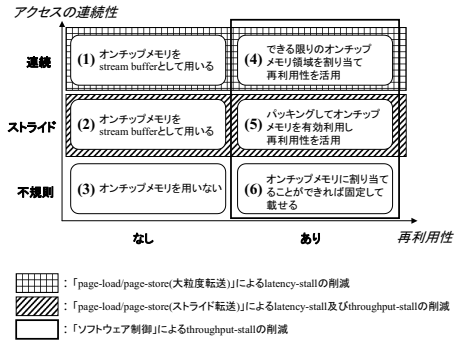

図 5 配列の特徴に対するオンチップメモリの利用法

多くのユーザの立場を考えると、既存のソースコード資産をコンパイルし直すだけで、オンチップメモリを用いた性能最適化を行なえるコンパイラがあることが望ましい。そこで現在、プログラムをコンパイルする際に自動的にオンチップメモリを利用するための最適化を行い、性能向上を図る自動最適化コンパイラについて検討を行っている。次章では、その自動最適化の構想について述べる。

4. SCIMA 用自動最適化コンパイラの構想

本章ではまず、SCIMA 用の最適化を行う上でオンチップメモリをどのように用いれば性能向上が達成できるかの指針を述べる。次に、それを基にした現在検討中の自動最適化コンパイラの構想について述べる。

4.1 オンチップメモリの利用法

我々はアクセスの連続性とデータの再利用性の観点から配列の特徴を整理し (**表 1**)、その分類に基づいたオンチップメモリの利用法を提案している[4]。具体的には**図 5** のようにオンチップメモリを用いることで性能向上を図る。

(1) 再利用性はないが連続アクセスされる配列： page-load/page-store 命令による大粒度転送を行い、転送回数を減らすことでオフチップメモリレーテンシの影響を減らすことができる。この場合はオンチップメモリ上に it page サイズのバッファを確保し、その領域をストリームバッファとして用いながらアクセスする。

(2) 再利用性はないがストライドアクセスされる配列： page-load/page-store のもつストライド転送機能により無駄なデータ転送を省き効率的なデータ転送を行うことが可能である。(1) と同様オンチップメモリをストリームバッファとして用いる。

(3) 再利用性がなくアクセスが不規則な配列： このような配列に関しては SCIMA のオンチップメモリを用いても効果が期待できない。したがってオンチップメモリを用いない。

(4) 再利用性があり連続アクセスされる配列： ブロッキング手法を用いるなどオンチップメモリサイズに分割してアクセスすることで他のデータとの干渉を防ぎながら再利用性を最大限活用する。オンチップメモリ上にループ中でアクセスされるワーキングセット分のオンチップメモリ領域を割り当てる。

(5) 再利用性がありストライドアクセスされる配列： page-load/page-store 命令のストライド転送機能により不連続なデータをパッキングしてオンチップメモリに載せ、チップ内記憶領域の有効活用を図ると同時に、他のデータとの干渉を防ぎながら再利用性を最大限活用する。(4) と同様にループ中でアクセスされるワーキングセット分のオンチップメモリ領域を割り当てる。

(6) 再利用性はあるがアクセスが不規則な配列： アクセスされる範囲があらかじめわかり、かつその範囲がオンチップメモリに載る程度の大きさであればオンチップメモリ上に固定してデータを載せ再利用性を活用する。オンチップメモリに載らない場合はキャッシュを用いてアクセスする。

(1)〜(6) は独立な最適化であるが、まず、(1),(2) に分

```
double precision a(N, N),p(N),q(N)
integer i,j,jj

do jj=1,N,NB
  do i=1,N
    do j=jj, jj+NB-1
      q(i) = q(i)+a(i,j)*p(j)
    enddo
  enddo
enddo
```

図 6 最適化前のコード

類される配列に対する最適化が重要である。これは、再利用性のないデータに対してキャッシュは無力であるため、ストリームバッファを提供してレーテンシの影響を減らすことに効果があると考えられるためである。次に、データの再利用性がある (4),(5),(6) に分類される配列に対する最適化を行い、さらなる性能向上を行う。これらの配列は従来のキャッシュベースのアーキテクチャにおいて、ソフトウエア的な手法でデータの再利用性を向上させた場合にも性能向上が期待できるものの、キャッシュ上でのデータの干渉の抑制などの観点からもオンチップメモリを用いてアクセスすることは重要である。

4.2 自動最適化の指針

我々は、以上の分類をもとに各配列に対して適切にオンチップメモリを用いた最適化を施す自動最適化コンパイラを検討している。

最適化を行う上で、プログラム中の配列が図 5 のどの分類に当てはまるかを判断することが大変重要である。現状では、それらの情報はユーザが何らかの形でコンパイラに与えることを想定している。

入力として与えるソースプログラムは、ユーザにより既にキャッシュブロッキングが行なわれたコードとし、本コンパイラによる自動ブロッキングは検討の対象外とする。これは、キャッシュブロッキングは既存のキャッシュベースのアーキテクチャでも広く用いらおり、妥当な条件であると考えている。なお、この際に、最適なブロックサイズは、オンチップメモリ容量によって変化するはずであり、本コンパイラで自動的にパラメータサーチを行なうべく、変数の形で表現しておく必要がある。

また、本コンパイラによる自動最適化後の出力は、SCIMA 用のディレクティブが挿入されたコードを仮定している。

4.2.1 ストリームバッファを用いた最適化

まず、自動最適化の第 1 段階として、図 5 の (1)(2) の特徴を持つ配列について、オンチップメモリ上に確保したストリームバッファ経由のアクセスとなるように、最適化を行なう。

この最適化の手順は次のようになる。

(i) (1)(2) の各配列用にオンチップメモリ領域に *page* サイズ分のバッファを確保

(ii) 最内ループでストリームバッファ単位のアクセスになるようにループ分割

(iii) データ転送用のディレクティブ挿入

```
c ***** ストリームバッファの要素数を宣言 *****
      integer strmbuf
      parameter(strmbuf=STRMBUF_SIZE)

!$scm begin (a,1,strmbuf,0,N-1)  ..... (A)

      do jj=1,N,NB
        do i=1,N
          do j=jj,jj+NB-1,strmbuf
!$scm load (a,i,jj,1,strmbuf)  ..... (C)
            do k=1,strmbuf
              q(i)=q(i)+a(i,j+k-1)*p(j+k-1)   } (B)
            enddo

          enddo
        enddo
      enddo
!$scm end (a)
```

図 7 ストリームバッファの導入を行ったコード

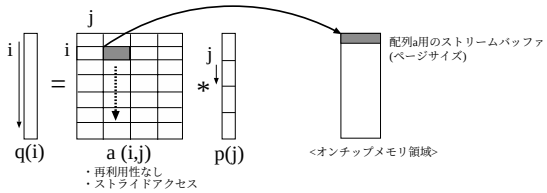

図 8 ストリームバッファ導入時のオンチップメモリ利用パターン

この第 1 段階における最適化の説明のために、**図 6** の行列ベクトル積を例として考える。行列ベクトル積では、配列 a のアクセスについては再利用はないが、ベクトル p については再利用性がある。この、ベクトル p の再利用性活用のため、図 6 の例では、p を分割してアクセスするようにブロッキングがなされている。

第 1 段階では、図 5 の (2) に該当する配列 a をストリームバッファを用いてのアクセスとなるように最適化する。上記の最適化手順に従い、自動最適化を行なった結果を**図 7** に示す。図 7 ではまず、配列 a のために *page* サイズ分のオンチップメモリ領域を確保している (図 7-(A))。そして、最内ループにおける配列 a のアクセスがストリームバッファ単位となるようにループ分割を行い (図 7-(B))、a の当該領域をオンチップメモリに転送するためのディレクティブを挿入する (図 7-(C))。

この時のデータアクセス、及びオンチップメモリの利用状況は**図 8** のようになる。

4.2.2 再利用性のある配列への最適化

次に、第二段階の最適化として、再利用性のない配列に対するストリームバッファの導入に加えて、図 5 の (4)(5) に対応する再利用性のある配列に対して可能な限りのオンチップメモリ容量を割り当てて再利用性を最大限活用するための最適化を行う。

この最適化の手順は次のようになる。

(iv) ブロッキングにおけるエレメントループ中における配列のワーキングセットサイズがオンチッ

```
      integer strmbuf
      parameter(strmbuf=512)

!$scm begin (a,1,strmbuf,0,N-1)
!$scm begin (p,NB,0)  ..... (D)
      do jj=1,N,NB
!$scm load (p,jj,NB)  ..... (E)
        do i=1,N
          do j=jj,jj+NB-1,strmbuf
!$scm load (a,i,jj,1,strmbuf)
            do k=1,strmbuf
              q(i)=q(i)+a(i,j+k-1)*p(j+k-1)
            enddo
          enddo
        enddo
      enddo
!$scm end (a)
!$scm end (p)
```

図 9 再利用性のある配列に対する最適化後のコード

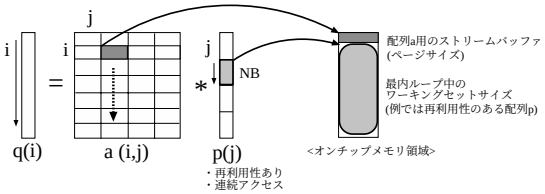

図 10 配列の再利用性も考慮したオンチップメモリ利用パターン

プメモリの残りの領域☆以下でオンチップメモリを最も効率的に用いるブロックサイズ (以降 NB) をプロファイリング等により探索

(v) 各配列用にブロックサイズ分のオンチップメモリ領域を確保

(vi) データ転送用のディレクティブの挿入

具体例として、先程ストリームバッファが設置された図 7 のコードに対して、再利用性のある配列に対する最適化を施す場合を考える。

まず、ブロッキングにおけるエレメントループのワーキングセットが残りのオンチップメモリ容量以下であり NB を探索により求める。オンチップメモリに領域を確保する (図 9-(D))。次に、データ転送用のディレクティブを挿入する (図 9-(E))。

このときのデータアクセス、およびオンチップメモリ領域の利用状況は**図 10** のようになる。

4.2.3 検 討 課 題

以上の最適化を行うため、現状ではユーザが配列の分類情報を与え、またブロッキング後のソースプログラムを入力とすることを想定しているが、最終的にはこれらについても自動的に行われることが望ましい。この点についても、今後検討していきたい。

4.3 最適化コンパイラの枠組み

本節で述べた SCIMA 用自動最適化コンパイラを実装するにあたり、我々は**図 11** に示すように Omni の枠組みを利用することを考えている。まず、fortran で書かれたソースプログラムを Omni で用いられている中間言語 Xobject に変換する。この Xobject 上でループ変換や、ディレクティブ挿入を行い、SCIMA 用に最適化された Xobject を作成する。その後は、既存のディレクティブベースコンパイラを用いることで、最終的にオブジェクトを作成する。

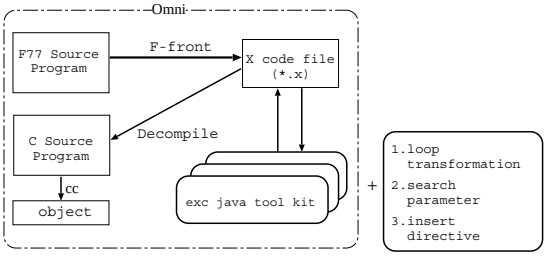

図 11 SCIMA 用自動最適化コンパイラの枠組み

5. まとめと今後の課題

本稿では、SCIMA 向けディレクティブベースコンパイラの紹介を行い、現在検討中の SCIMA 用自動最適化に向けての構想を述べた。今後、ディレクティブベースコンパイラについては現在のディレクティブに拡張の余地があるならば仕様を改良したいと考えている。また SCIMA 用自動最適化についても最適化のアルゴリズム化を進め、将来的にはオンチップメモリに最適なデータ配置やループ順[5] についても考慮に入れつつ、具体的な作成に入っていきたいと考えている。

謝辞 本研究を行なうにあたり、御助言、御討論頂いた筑波大学計算物理学研究センター関係者の皆様、ならびに東京大学南谷崇教授に感謝致します。なお、本研究の一部は日本学術振興会未来開拓学術研究推進事業「計算科学」(Project No. JSPS-RFTF 97P01102) によるものである。

参 考 文 献

1) 中村宏, 近藤正章, 大河原英喜, 朴泰祐.“ ハイパフォーマンスコンピューティング向けアーキテクチャSCIMA ”. 情報処理学会論文誌, Vol.41, No.SIG5(HPS1), pp.15-27,2000 .

2) 大根田拓, 近藤正章, 中村宏.“ SCIMA におけるメモリアクセス機構の検討 ”. 情処研報,ARC-144, Vol.99, No.76, pp.165-170, 2001

3) M.Sato, S.Satoh, K.Kusano, and Y.Tanaka. “ Design of OpenMP Compiler for an SMP Cluster ”, Proc. European Workshop on OpenMP EWOMP '99, pp.32-39, 1999.

4) 近藤正章, 中村宏, 朴泰祐.“ SCIMA における性能最適化手法の検討 ”. 情報処理学会論文誌,Vol.42, No.SIG12(HPS4), pp.37-48, 2001.

5) M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnan, I.Kadayif, and A.Parikh.“ Dynamic Management of Scratch-Pad Memory Space ”, Proc. 38th Design Automation Conference, pp.657-661, June 2001.

☆ 与えられたオンチップメモリ容量から stream buffer として確保された分の容量を除いた容量