計算機アーキテクチャ 136－3
（2000．1．7）

# ハイパフォーマンスコンピューティングに適したメモリアーキテクチャの予備評価

大河原 英喜† 近藤 正章††
中村 宏† 朴 泰祐††

ハイパフォーマンスコンピューティングを考えていくにあたって、メモリアーキテクチャは重要な問題である。将来的に、プロセッサに十分な演算能力が与えられたとしても、下位のメモリ階層へのアクセスによる性能低下が問題となる。我々はこれまで、この問題への解決手法として、チップ上に主記憶の一部を実装する、SCIMA(Software Controlled Integrated Memory Architecture)の提案を行ってきた。本稿では、SCIMAのクロックレベルのシミュレーション環境を構築し、行列積を用いた予備評価を行ったので報告する。

# Preliminary Performance Evaluation of New Memory Architecture for High Performance Computing

HIDEKI OKAWARA,† MASAAKI KONDO ,†† HIROSHI NAKAMURA † and TAISUKE BOKU††

In high performance computing, memory hierarchy affects performance significantly. In large scale scientific application, accesses of lower memory hierarchy occur very frequently, which leads to serious performance degradation. In order to solve this problem, we have proposed a SCIMA, Software Controlled Integrated Memory Architecture, which includes software-controllable on-chip memory. We developed an environment for clock lebel simulation. In this paper, we present preliminary performance evaluation and effectiveness of SCIMA.

## 1. はじめに

近年、プロセッサの性能向上に比べ、メモリの性能向上が遅く、今後メモリアクセスがボトルネックになると考えられる[1)2)]。データセットの大きいハイパフォーマンスコンピューティング(HPC)においては、従来のメモリアーキテクチャは有効でなく、下位のメモリ階層へのアクセスが頻発し性能が大きく低下する。特に、オフチップメモリアクセスの頻発によって著しく性能が低下するため、オンチップ記憶をいかに有効に利用できるかが重要となる。

オンチップ記憶としてはキャッシュが広く用いられている。しかし、キャッシュのリプレースメントやアロケーションの制御はハードウェアで決められているため、再利用性のあるデータを最大限に活用することができない。そのため、再利用性をより有効活用できる様なキャッシュの構成法に関する研究が行なわれ[3)4)]、ブロッキング等のソフトウェア的最適化手法を用いてキャッシュ上のデータ再利用性を有効活用する手法も提案されている[5)6)]。しかし、ハードウェアで制御が決められるキャッシュにおいては、ソフトウェアで最適化を行なうには次の様な困難さがある。

- キャッシュへのマッピングを制御できないので、載せたい配列をうまく載せられない場合がある。
- 複数の配列をキャッシュに載せる場合の配列間の干渉(cross interference)の排除は相当難しい。
- どのデータをキャッシュに載せるかを指定できない（再利用性のない配列は載せたくない、等）

そこで我々は、ソフトウェアで制御可能なSCIMA(Software ControlledIntegrated Memory Architecture)の提案を行ってきた[7)]。SCIMAでは、チップ上に主記憶の一部（オンチップメモリ）を実装し、オンチップメモリ↔オフチップメモリのデータ転送の制御はソフトウェアで行う。

この様な、ソフトウェア制御可能なメモリアーキテクチャの研究として、次の様な関連研究が挙げられる。

† 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
†† 筑波大学 電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

- コンパイル時に発生するspill codeを載せるために、コンパイラが制御できるメモリ(CCM)をチップ上に載せる[8)]
- 組み込み用プロセッサでは、DRAMをチップ上に実装することで、小型化、低消費電力化を実現する。
- データセットの小さい場合には、小容量のscratch pad RAMを用いて、特定アプリケーションに最適なリプレースメント制御を行なう[9)10)]。

しかし、CCMはコンパイラがspill codeに対して用いるという用途に限られている。また、DRAM混載やscratch pad RAMは、データセットが小さく、ある特定アプリケーションを対象とする組み込み用プロセッサで見られるメモリアーキテクチャである。SCIMAの特徴は、データセットの大きいHPCに適したメモリアーキテクチャとしてオンチップメモリを採用し、アプリケーションに合わせてより柔軟にユーザがリプレースメント制御を行なえる点である。

これまで、いくつかのHPCアプリケーションにおいてSCIMAの有効性に関する初期評価を行なってきた[11)12)]。本稿では、さらに詳細にSCIMAの評価を行うために、シミュレーション環境を構築し、その予備評価を行った。

## 2. SCIMA

SCIMAのメモリアーキテクチャを図1に示す。拡張するアーキテクチャは以下の通りである。

**オンチップメモリの実装** ソフトウェア制御可能なメモリ階層として、オンチップメモリを実装する。また、オンチップメモリ↔オフチップメモリ間のデータ転送を行うpage load/page store命令を追加する。page load/page storeではブロックストライド転送も可能とする。

**cacheable/uncacheable属性** オフチップメモリ領域のデータに対しては、cacheable/ uncacheable属性を設け、uncacheable属性のデータに対するload/store命令では、オフチップメモリからレジスタへの直接転送を行う。

**レジスタ数の拡張** 演算能力の向上とオンチップメモリの高バンド幅を活かすために、レジスタ数を拡張する。

以下では、オンチップメモリの実装についてより具体的に述べる。

### 2.1 アドレス空間

アドレス空間上にオンチップメモリ領域を定義し、load/store時には、アドレスがオンチップメモリか否かを判定し、アクセス先を決定する。また、オンチップメモリ領域はuncacheable領域とし、キャッシュとの包含関係は生じないようにする。

オンチップメモリ↔オフチップメモリのデータ転送はpage load/store命令を用いる。この時、レジスタ↔オンチップメモリ間のload/storeと、オンチップメモリ↔オフチップメモリ間のpage load/page storeの同期管理をする必要がある。そのため、OCPMT(On-Chip Page Management Table)と呼ぶ機構を追加し、1kBなどの大きな粒度のOn-Chip Page単位で同期管理を行う。詳細は参考文献[7)]を参照されたい。

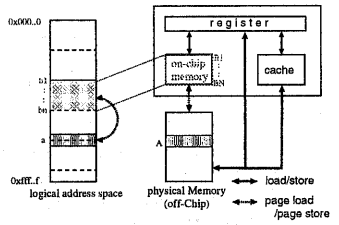


図1 SCIMA

### 2.2 オンチップメモリとキャッシュの統合

様々なHPCアプリケーションにおいて、最適なオンチップメモリとキャッシュの容量は異なる。ここでは、キャッシュとオンチップメモリでハードウェアを共用し、総容量は一定だがオンチップメモリとキャッシュの容量の割合をソフトウェアで可変にする構成について述べる。

#### 2.2.1 ハードウェア構成

以下の特殊レジスタをハードウェア的に用意し、キャッシュをオンチップメモリとして利用する。

**Way Lock register(WLR)** キャッシュのway数と同ビット数のWay Lock Registerを用意する。各Wayに対応するビットがセットされていれば、そのWayをLockしてオンチップメモリとして用いる。

**On-Chip addres start register(ASR)** オンチップメモリとしてマップされる領域の先頭論理アドレスを保持する。先頭アドレスは、オンチップメモリ領域サイズのアライメントに一致する。

**On-Chip address mask register(AMR)** オンチップメモリ領域のサイズを示すためのマスクレジスタ。オンチップメモリのサイズは、Wayサイズ(キャッシュサイズ/連想度)の$2^n$とする。

ハードウェア構成を図2に示す。図2では、Way数が4の32kBキャッシュを、16kBキャッシュと16kBオンチップメモリとして用いる場合を例として示している。

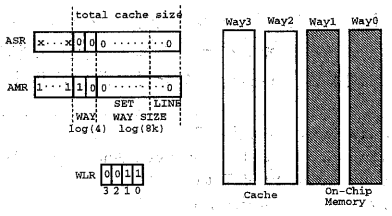


図2 キャッシュとオンチップメモリの統合

オンチップメモリ領域の割り当ては、用意されたシステムコールを介して行い、Way Lock時には該当Wayの内容はflushされる。アドレス空間上にオンチップメモリ領域を割り当てAMRとASRをセットする。AMRのWAYビット部を元にLockするWay数を、ASRのWAYビット部を元にLockするWayを決定する。ここで、WAYビット部のビット数は$\log_2$(連想度)となる。図2の例では、AMRのWAYビット部が10なのでWay 2つがオンチップメモリとして用いられ、ASRのWAYビットが00となのでWay0から2つのWayがLockされる。オンチップメモリとして用いるサイズによって、表1の様なWayの割り当て方が考えられる。

表1　オンチップメモリ領域の割り当て

| オンチップメモリサイズ | ASR WAY部 | AMR WAY部 | オンチップメモリとなるWay | WLR |
|---|---|---|---|---|
| 32kB | 00 | 00 | way0～way3 | 1111 |
| 16kB | 00 | 10 | way0～way1 | 0011 |
| | 10 | 10 | way2～way3 | 1100 |
| 8kB | 00 | 11 | way0 | 0001 |
| | 01 | 11 | way1 | 0010 |
| | 10 | 11 | way2 | 0100 |
| | 11 | 11 | way3 | 1000 |

### 2.2.2 アクセス動作

メモリアクセスが発生したら、アドレスをAMRでマスクし、ASRと比較する。その結果が一致すればオンチップメモリ領域、一致しなければオフチップメモリ領域となる。オンチップメモリへのアクセスの場合、アドレスのWAYビット部がアクセスすべきWAYを現している。図2の例の場合、アドレスのWAYビット部が00ならWay0、01ならWay1へアクセスする。Way上のどのSETをアクセスするかは、オンチップメモリ領域/オフチップメモリ領域に依らず、アドレスの下位ビットから一意に定まる。従って、データアクセスとオンチップメモリ領域か否かの判定を並行して行うことができ、高速なアクセス動作が可能である。オフチップメモリ領域へのアクセスの場合、Way2、Way3に対してキャッシュアクセス動作を行う。

この様に、一定のオンチップ記憶容量に対して、キャッシュとオンチップメモリ(scrathpad RAM)の割合を可変にしている例は、アーキテクチャの違いはあるものの既に実装されている[13)10)]。SCIMAにおいても、この様な機構を取り入れることは検討していく。

## 3. 評価環境

SCIMAでは、2節で述べた様なアーキテクチャ拡張を行うが、具体的にはMIPS4アーキテクチャを拡張したものとして評価を行う。これらの拡張に対応した最適化コンパイラを開発するのが理想的である。しかし、オンチップメモリへ置くべき配列の宣言や、page load/page storeによるデータ転送、cacheable/uncacheableの指定は、ユーザによる最適化が比較的容易かつ的確であると思われるので、ユーザがソースレベルで行う。レジスタ拡張に関しては、コンパイラで対応すべきだが、現在、SCIMAに対応したコンパイラがないため、既存のMIPSコンパイラを用いることとし、アーキテクチャ拡張への対応を行うプリプロセッサを作成した。

### 3.1 ソースコードの作成法

C言語を用いる場合、ソースコードレベルでは、page load/page storeやオンチップメモリ領域の確保等は以下に用意する疑似関数を用いて表現する。

**get_OnChipAddr()** オンチップメモリ領域を確保し、その先頭アドレスをポインタに返す。ソースレベルでは、このポインタを用いてオンチップメモリにアクセスする。

**p_load(*source_add,*dest_add,size,blocksize,stride)** page loadを行う。引数として、転送元、転送先の先頭アドレス、総転送サイズ、転送ブロックサイズ、ストライド幅を与える。転送サイズなどはB単位で指定する。page storeに関しても、同様な疑似関数p_storeを用いる。

**get_Uncache(size)** 宣言サイズのuncacheable領域を確保し、その先頭アドレスをポインタに返す。

これらの疑似関数を用いたコード例を以下に示す。

```
double *On_Chip;
On_Chip = (double *)get_OnChipAddr();

p_load(&a[0][0],&On_Chip[0],800,1,0);

for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
    On_Chip[100+N*i+j]+=On_Chip[N*i+k]*On_Chip[N*k+j];

p_store(&On_Chip[100],&c[0][0],800,1,0);
```

### 3.2 プリプロセッサ

シミュレーション時には、既存のコンパイラを用いてbinaryを生成することとし、既存のコンパイラから生成される*.sに対し、co-processor命令を用いて、アーキテクチャ拡張に伴う追加情報を挿入する。この様な、評価に必要なコード処理を行うプリプロセッサを作成した。

また、既存のコンパイラでは、レジスタ数が32で不足した場合にはスタック領域に対するspill codeが発生するため、プリプロセッサで次の様にレジスタ拡張を行いspill codeを除去する。

(1) spill codeを対象とするスタック領域で分類する
(2) スタック領域に対するspill codeを、レジスタ↔拡張レジスタ間のmov.dに置き換える。同一スタック領域に対するspill codeは、同一の拡張レジスタへ割り当てる。
(3) 置き換えられた拡張レジスタに対して、Basic Block内に限り値のlifetimeを調べ、可能な限りmov.dを除去する。

プリプロセッサでは、以下の追加情報を挿入する。

```
jal p_load          → ldc2 $0,0($0)
jal p_store         → ldc2 $0,1($0)
jal get_Uncache     → ldc2 $0,2($0)
jal get_OnChipAdrr  → ldc2 $0,3($0)
レジスタ拡張情報     → lwc2
```

シミュレーション用コードを生成する手順を整理すると、以下の様になる。

(1) 疑似関数を用いてソースコードを作成する
(2) 既存のコンパイラを用いて*.sを作成する
(3) プリプロセッサを用いて疑似関数呼び出しをco-coprocessor命令に置き換える。また、レジスタ数の拡張を行ってspill codeを除去する。
(4) 既存のコンパイラを用いてbinaryを作成する。

### 3.3 クロックレベルシミュレータ

MIPS4(R10000)をベースとしたbinary入力のクロックレベルシミュレータを作成した。アーキテクチャ拡張については、プリプロセッサで挿入されたco-processor命令から追加情報を読み取りシミュレーションを行う。シミュレータの対象となるモデルを図3に示す。シミュレータでは、out-of-orderに命令実行を行う。キャッシュはL1キャッシュのみをシミュレーションし、lock up freeキャッシュとする。シミュレーション時に設定する主なパラメータとして、以下のパラメータがある。

- レジスタ数(INT,FP)
- RESERVATION STATION SIZE(INT,FP,LS)
- 実行ユニット数(INT,FP,LS,On-Chip LS)
- キャッシュサイズ、ラインサイズ、assovciativity
- オンチップメモリサイズ、On-Chip Page SIZE
- オフチップメモリスループット、レーテンシ

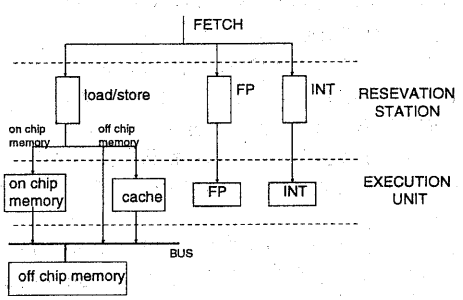


図3 シミュレータ対象のモデル

## 4. 性能評価

### 4.1 行列積プログラム

N*Nの行列積をとりあげて性能評価を行う。行列積では種々の最適化手法が提案されているので、本評価でも以下に述べる最適化を行う。

#### 4.1.1 ブロッキング(オンチップメモリ、キャッシュ)

**SCIMA用コード** 図4の様に、オンチップメモリを3つのブロックに分割し、page load/page store命令を用いてデータのリプレースメントを行う。

**キャッシュ用コード** キャッシュ上でブロッキングを行う。BLOCK*BLOCKがキャッシュに載る様にブロックサイズを決定し、再利用性の最も高いブロックが極力キャッシュに載るようにする。また、キャッシュブロッキングを行っても、同一ブロック内のデータがコンフリクトを起こしてしまう可能性がある。そこで、別の連続領域にブロックをコピーすることでコンフリクトを防ぐ。

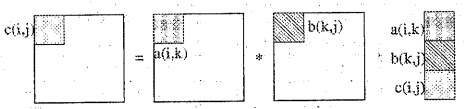


図4 オンチップメモリ上でのブロッキング

#### 4.1.2 先行page load/先行コピー

ブロッキングを行う際に、次の計算に必要となるブロックを先行して持ってくることで、レーテンシ隠蔽を図ることができる。例えば、オンチップメモリの場合、図5の様に5つに分割して、計算と並行して、次の処理に必要なブロックのpage loadを先行して行う。キャッシュの場合も同様に考えることができる。

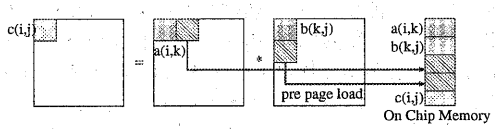


図5 先行page load

#### 4.1.3 レジスタブロッキング

三重ループ(ijk)に対して、外部ループ(ij)をそれぞれr回unrollingしてレジスタブロッキングを行う。最内ループで必要となるレジスタ数を考え、$r^2+2r <$ レジスタ数となるrを目安にunrollingを行う。

### 4.2 評価条件

以下の様なR10000相当のハードウェア構成を仮定して評価を行った。

**レジスタ数** INT=32,,FP=32
**RESERVATION STATION** INT=8,FP=8,LS=8
**実行ユニット数** INT=2,FP=1,LS=1,On-Chip LS=1
**キャッシュ** ラインサイズ32B、2way associativity
**オンチップメモリ** On-Chip Page SIZE 1kB
**オフチップメモリスループット** 4B/cycle

メモリ容量は、SCIMAの評価ではオンチップメモリ32kBとキャッシュ1kB(整数データのみに用いる)、キャッシュの評価ではキャッシュ32kBとし

た。オフチップメモリアクセスレーテンシは、60cycle,10cycle,0cycleの3通りについて評価を行った。10cycleの評価は、全データがL2キャッシュに載る場合を想定している。

評価は表3に示す4通りのコードで行われ、それぞれブロッキングサイズは異なる。また、レジスタブロッキングは外部ループ(ij)に対して4回づつ行い、その結果、最内ループでは8load,16maddを行っている。

表3　評価プログラム

| | | ブロックサイズ |
|---|---|---|
| SCIMA | 先行 page loadなし | 36*36 |
| SCIMA | 先行 page loadあり | 28*28 |
| cache | 先行コピーなし | 64*64 |
| cache | 先行コピーあり | 44*44 |

評価に用いたキャッシュ用コード（先行コピーなし）の、実機上(SGI O2,R10000,peak 360MFLOPS)で実測した性能を図6に示す。また、実機上で最適な行列積ルーチンを生成する、ATLAS (Automatically Tuned Linear Algebra Software)[14]によって生成された行列積ルーチンの性能も示す。図6より、評価に用いるコードはかなり最適化されていることがわかる。

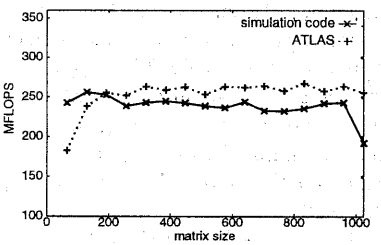


図6　評価用プログラムのR10000上での性能

## 5. 評価結果

図7に、先行page load/先行コピーを行わない場合のコードにおける、シミュレーション結果を示す。

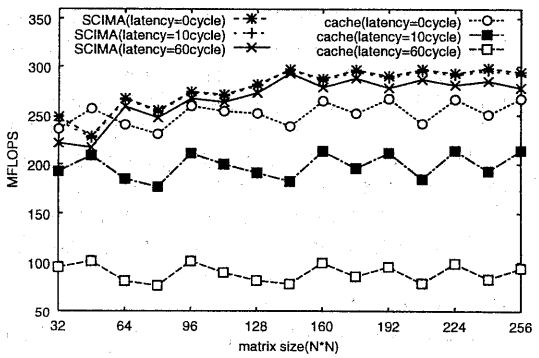


図7　オンチップメモリとキャッシュの性能

また、キャッシュにおいて先行コピーを行った結果を図8に示す。

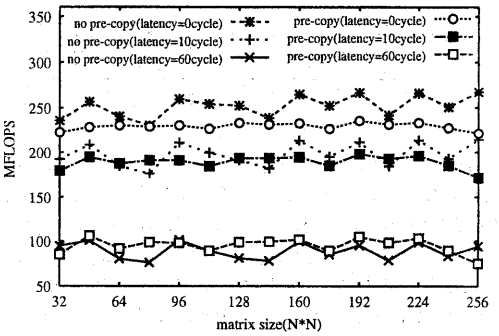


図8　キャッシュにおける先行コピーの影響

## 6. 考　　察

### 6.1 行列サイズによる性能変化

行列サイズを変化させると性能が上下するが、これは、行列サイズとブロックサイズの値に依存する。SCIMAの場合のブロック分割数と性能の関係を表2に示す。例えば、96*96の行列を36*36のブロックに分割をした場合、3*3個のブロックに分割されるが、最後のブロックは24*24になる。分割個数が同じ場合には、行列サイズが大きい方が端数が少なく、有効なブロッキングが行われるために性能が向上する。

### 6.2 SCIMAの有効性について

SCIMAでは、オフチップメモリレーテンシの値に因らず高い性能を示すが、キャッシュの場合には、オフチップメモリレーテンシにより大きく性能が低下する。これは、オフチップメモリからブロックデータを転送する際に、SCIMAでは1kBのOn-Chip Page単位でpage load/storeを行っているためにレーテンシの影響が小さいのに対し、キャッシュでは32Bラインごとにレーテンシがかかるためと考えられる。SCIMAの場合には、レーテンシが0cycleの場合と比べて60cycleの場合に数%しかMFLOPS値が低下しないのに対し、キャッシュの場合には65%低下する。

また、SCIMAとキャッシュの性能を同じレーテンシ同士で比較すると、レーテンシ0cycleの場合で9割程度の性能となっている。この違いは、SCIMAとキャッシュでのデータ転送量そのものの違いとして考えられる。理想的には、ブロックサイズの大きいキャッシュの方がデータ転送量は少ない筈だが、実際は、ラインコンフリクトによるキャッシュミスのためデータ転送量が増加する。SCIMAでは、データのリプレースメントをソフトウェア制御するため、この様なコンフリクトによるメモリトラフィックの増加は発生しない。また、キャッシュの場合は、配列をコピーする際に、データごとにレジスタを介してload/storeを行うが、SCIMAではpage loadで、オンチップメモリ↔オフチップメモリ間の転送を直接行うため、メモリトラフィックは少なくてすむ。

また、レーテンシが0cycleの場合には、キャッシュ

表2　SCIMA用コードでのブロック分割結果（ブロックサイズ 36*36）

| 行列サイズ | 32*32 | 48*48 | 64*64 | 80*80 | 96*96 | 112*112 | 128*128 | 144*144 | 160*160 |
|---|---|---|---|---|---|---|---|---|---|
| ブロック分割数 | 1*1 | 2*2 | 2*2 | 3*3 | 3*3 | 4*4 | 4*4 | 4*4 | 5*5 |
| 性能（MFLOPS） | 223 | 221 | 263 | 253 | 272 | 270 | 278 | 297 | 284 |

においても高い性能が得られたが、レーテンシ60cycleの場合、キャッシュのMFLOPS値は、SCIMAのMFLOPS値の1/3程度しか出ていない。レーテンシ10cycleの場合で7割の性能となっている。この様に、キャッシュの場合にはオフチップメモリアクセスレーテンシによって大きく性能が低下してしまう。そこで、キャッシュ構成を活用する場合は通常レーテンシ10cycle程度のL2キャッシュを載せることを想定し、SCIMA(latency=60cycle)とキャッシュ(latency=10cyce)を比較する。その場合も、キャッシュはSCIMAの7割程のMFLOPS値しか出ておらず、従来のL1キャッシュとL2キャッシュから構成されるアーキテクチャより、SCMIAの方が性能的にも実装面積的にも有利であることを示している。

### 6.3 先行page load/先行コピーの効果

6.2節で述べた様に、SCIMAでは大きい粒度でpage loadeを行うために、レーテンシの影響が少ない。そのため、先行page loadでレーテンシを隠蔽しなくても、性能にそれほど影響がないと思われる。

キャッシュの場合には、小さいラインサイズ単位でデータをオフチップメモリから転送するためレーテンシの影響が大きい。そこで、先行コピーすることによってレーテンシの隠蔽を図った。しかし、先行コピーを行うためにブロックを細く分割すると、ブロック辺りのコピー回数が増え、load/store数が増加してしまう。図8では、レーテンシが大きくキャッシュミスによる影響が大きい場合には、先行コピーによるレーテンシ隠蔽を行った方が性能が良いが、レーテンシが小さい場合にはload/store数の増加が無視できなくなり、先行コピーを行わない方が性能が良くなっている。

## 7. ま と め

本稿では、SCIMAの詳細な性能評価を行うために、命令セットレベルでのシミュレーション環境を構築した。行列積における評価では、オンチップメモリ↔オフチップメモリ間のpage load/page storeを、大きな粒度で行うことによる性能向上が得られた。また、オンチップメモリ↔オフチップメモリのデータ転送をプログラマが制御できることで、L2キャッシュなしでも高性能が得られることを示した。

今回は、予備評価ということで行列積プログラムをとりあげ、ハードウェア構成やデータセットの小さい評価を行った。今後は、ハードウェア構成を拡張し、データセットの大きい評価も行う。また、異なるアプリケーションにおける性能評価も行っていく。

## 参 考 文 献


1) D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. *Proc.of ISCA'96*, pp. 78–89, 1996.
2) A. Salsbury, F.Pont, and A. Nowatzyk. Missing the memory wall:the case for processor/memory integration. *Proc. of ISCA'96*, pp. 90–101, 1996.
3) N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associatie cache and prefetch buffers. *ISCA'90*, pp. 364–373, 1990.
4) A.Gonzalez, C.Aliagas, and M.Valero. A data cache with multiple caching strategies tuned to different types of locality. *Proc. of ICS'95*, pp. 338–347, 1995.
5) M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. *Proc. of ASPLOS-IV*, pp. 63–74, 1991.
6) P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. on Computers*, Vol. 48, No. 2, pp. 142–149, 1999.
7) 近藤正章, 坂井修一, 朴泰祐, 中村宏. HPC向けプロセッサのメモリアーキテクチャの基本構成. 情処研報,ARC-134, Vol. 99, No. 67, pp. 1–6, 1999.
8) K.Cooper and T.Harvey. Compiler-controlled memory. *ASPLOS-VIII*, pp. 2–11, 1998.
9) Sony's emotionally charged chip. *MICROPROCESSOR REPORT*, Vol. 13, No. 5, 1999.
10) Strongarm speed to triple. *MICROPROCESSOR REPORT*, Vol. 13, No. 6, 1999.
11) 近藤正章, 坂井修一, 朴泰祐, 中村宏. オンチップメモリを用いたHPCプロセッサの検討. 情処研報,ARC-132, Vol. 99, No. 21, pp. 85–90, 1999.
12) 大河原英喜, 中村宏, 吉江友照, 金谷和至. ハイパフォーマンスコンピューティングに適したメモリ階層の検討. 情処研報,ARC-133, Vol. 99, No. 41, pp. 55–60, 1999.
13) SH-4ハードウェアマニュアル. http://www.hitachi.co.jp/Sicd/Japanese/ Products/micom/shmicom.htm, 平成10年.
14) R.Clint Whaley. Atlas(automatically tuned linear algebra software). http://www.netlib.org/atlas/index.html.