第5章　細粒度通信機構に基づくマルチスレッドアーキテクチャの評価

前節で述べた細粒度通信機構に基づくマルチスレッドアーキテクチャEM-X について、80 台の要素プロセッサ (PE) からならプロトタイプを用いて評価を行なう。以下では、まず通信や同期機構の基本性能を見るための小さなプログラムによるマイクロベンチマークを実行し、前節で述べた特性を確認する。次に、その個々の特性がどのように並列性能に貢献するかを見るために、マイクロベンチマークよりはやや大きいが比較的小さなカーネルベンチマークを用いて、その実行状況の詳細を評価する。最後に、実アプリケーションに近いより大きなマクロベンチマークにより、実性能について評価を行う。

5.1 マイクロベンチマーク

EM-X の通信や同期の性能を特徴づけるために、各特性ごとにその性能を評価できるような小さなプログラムを作成し、それぞれのプログラムを用いて性能評価を行う。通信性能はある単位時間の間にメッセージをどれだけ送ることができるかというスループットと、1 つのメッセージの転送を開始してから、終了するまでにどれだけの時間がかかるかというレイテンシの 2 つによって評価される。スループットは大規模データの転送時に重要となり、それに対し、レイテンシは比較的小さなメッセージを転送するときに重要となる。

5.1.1 静的レイテンシ

EM-X のネットワークトポロジーは直接網であり、通信相手先によってそのネットワーク遅延は異なる。また、本ネットワークは一方向通信を基本としており、通信相手先 PE への要求とその通信相手先 PE からの返答は別の経路を通ることになる。要求と返答の両者を合わせた往復のネットワーク遅延は、いくつかのグループに分けられる。すなわち、ネットワーク遅延をネットワーク上の転送のホップ数で測定することにすると、往復遅延のホップ数はグループ内要素プロセッサ (PE) の数の倍数、例えば80PE システムの場合グループ内 PE は 5 なので、0、5、10、15 のいづれかとなる。この場合の往復通信の平均ホップ数は 10.13 である。

ネットワーク上に他のパケットが無い場合の往復通信レイテンシを静的レイテンシあるいは基本レイテンシと呼ぶ。これは、(要求側通信セットアップオーバヘッド)+(行きのネットワーク遅延)+(相手先処理時間)+(相手先通信セットアップオーバヘッド) +(帰りのネットワーク遅延) と表わすことができる。図 5.1 は、以下に示すいくつかの通信処理を実行した場合の静的レイテンシの実測結果である。各処理は EM-C によるスレッドライブラリ関数を用いて実装されている。以下では各処理の説明とともに EM-C による記述の例を示している。

SYSRD 直接リモートメモリアクセス機構を用いたリモートメモリ読み出し。

```
int global *p; r = *p;
```

USRRD メモリ読み出しハンドラを用いるリモートメモリ読み出し。

```
int global *p; r = em_get(p, PT_USRRD);
```

RPC0 0 個の引数を持つリモート関数呼び出し

```
r = remote_call(pe, func, 0);
```

RPC6 6 個の引数を持つリモート関数呼び出し

```
r = remote_call(pe, func, 6, a0, a1, a2, a3, a4, a5);
```

通常リモートメモリ読み出しにはハードウェア支援機構を用いる SYSRD 方式を用いる。EM-C では他のプロセッサのメモリを PE 番号+局所アドレスというシステムで一意なグローバルアドレスで指し

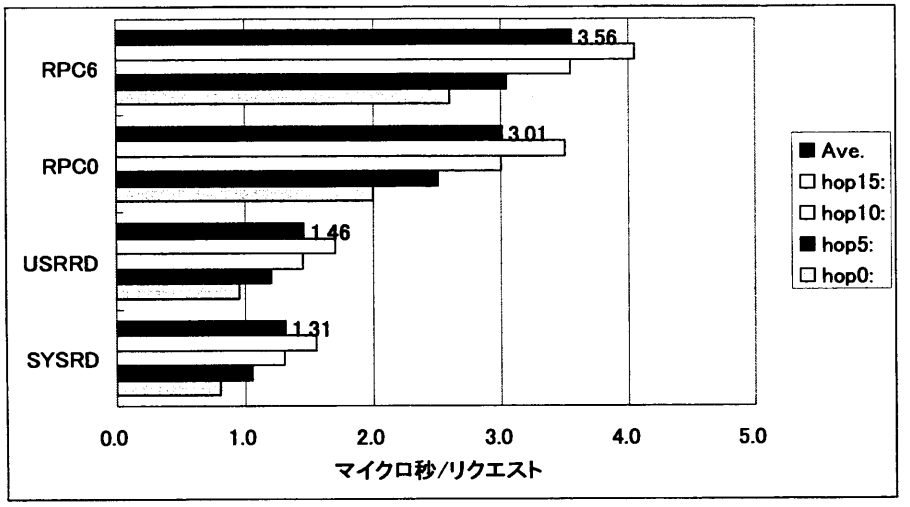


図 5.1: 各処理における静的レイテンシ

示すことが可能であり、そのようなグローバルメモリへのポインタを int global *p; というように global という修飾子で指定する。global 修飾子のついたポインタ変数の参照があると、EM-C コンパイラは自動的にリモートメモリアクセス要求を生成するとともに、スレッド切り替え時に必要となるリソース管理処理を挿入する。

この SYSRD では 2 点間の FIFO 性が保証されていない (詳細は 4.3.2 直接リモートメモリアクセスを参照)。通常のスレッド処理では 2 点間の FIFO 性は保証されるため、メモリ参照用の特殊ハンドラを起動する USRRD というリモートメモリ参照パケットが用意されている。EM-C ではこのような返り値を要求するパケットを送信する関数として em_get() がある。第一引数がグローバルアドレス、第二引数が起動するハンドラタイプである。USRRD はシステムが用意しているハンドラであるが、ユーザが記述したハンドラを起動することも可能である。また、関数呼び出しとして記述するが、コンパイラによって自動的にインライン展開され、2 命令程度の命令シーケンスとして実現される。

3 つ目の RPC0 は引数を持たないリモート関数呼び出しの場合である。5.1 の結果は、何もせずに直ちに結果を返すダミー関数を用いた場合の遅延を表わしている。EM-X ではリモート関数呼び出しを行う場合、まず、相手先 PE から関数フレームを取得し、その関数フレームを用いて引数の受け渡しを行う。そのため、関数呼び出しでは 2 往復のやり取りが必要となる。remote_call() は引数の個数が変化する varg 型の関数で、第一引数は関数呼び出しを行うプロセッサ番号、第 2 引数は読み出す関数、第 3 引数が引数の個数、あとにその個数だけの引数が並ぶ。

4 つ目の RPC6 は 6 個の引数を持つ場合のリモート関数呼び出しである。RPC0 と同様にダミー関数の遅延を表わしている。上の引数なしのときと比べると、引数処理にかかるオーバヘッドが推定できる。

図 5.1 は 80PE システムで測定した結果である。この結果は、上記処理を一定回数だけ繰り返し、それにかかった時間からループオーバヘッドを除き、それをループ回数で割った値である。この値には、各処理に必要な命令実行、スレッド切り替えに要する命令実行、リモートアクセスレイテンシが含まれている。図では各処理について、往復の通信に必要な hop 数ごとの値とともに、平均を示している。SYSRD を用いた場合で、平均 1.31 マイクロ秒である。同様に USRRD を用いた場合で平均 1.46 マイクロ秒である。静的レイテンシでは SYSRD と USRRD の差は 150ns(3 クロックサイクル) と小さいが、相手先 PE で何かスレッドが実行されているとその差は大きくなる。3 クロックサイクルの差はハンドラ起動 (1 クロックサイクル) およびハンドラ実行 (ld および send の 2 クロックサイクル) のオーバヘッドである。引数なしのリモート関数呼び出しは平均 3.02 マイクロ秒であり、6 引数のリモート関数呼び出しは平均 3.57 マイクロ秒である。これより、引数 1 個あたり約 100ns(2 クロックサイクル) のオーバヘッド

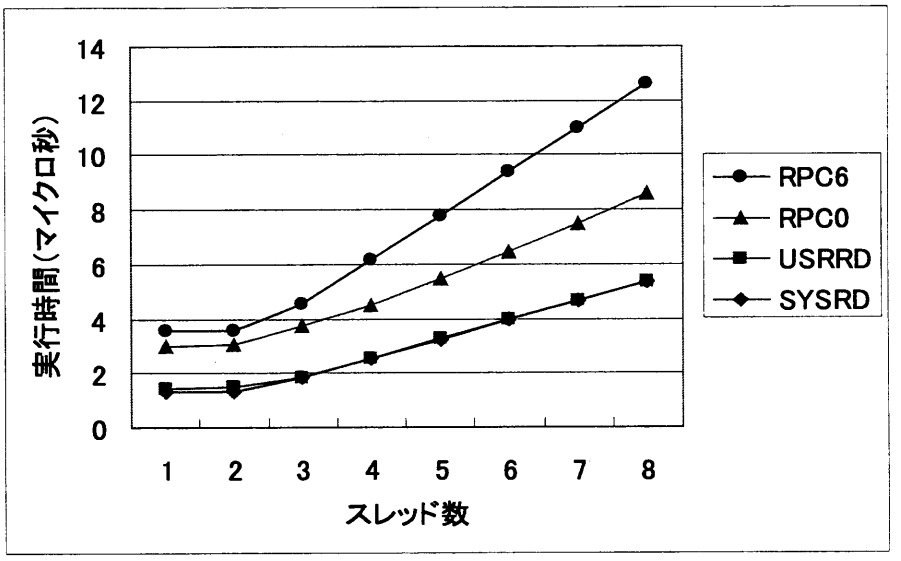


図 5.2: 静的レイテンシのマルチスレッドによる隠蔽

表 5.1: 静的レイテンシの内訳

処理	総レイテンシ	オーバラップ可能	オーバラップ不可能
SYSRD	1.31	0.86 (65.6%)	0.45 (34.4%)
USRRD	1.46	1.01 (69.1%)	0.45 (30.9%)
RPC0	3.02	2.17 (71.6%)	0.85 (28.4%)
RPC6	3.57	2.22 (62.0%)	1.35 (38.0%)

であると分かるが、これは引数転送のための ld 命令と send 命令の 2 命令が必要になっているためである。このように EM-X では 3 マイクロ秒程度でリモート関数呼び出しが行えるため、並列オーバヘッドをあまり気にせずに並列に関数呼び出しが行える。

さらに、この値はネットワーク転送等を含んだ値であり、この間、他のスレッドの処理を caller 側のプロセッサで実行することが可能である。図 5.2 は上記の静的レイテンシ測定プログラムを同一プロセッサ上で複数走らせた結果である。この図の結果はループオーバヘッドも含んだ値であるため 5.1 の値よりやや大きな値になっているが、2 スレッド走らせた場合の実行時間は 1 スレッドの実行時間とほとんど変らない。これは 1 スレッド実行中のリモートアクセスレイテンシの間にもう一つのスレッド実行がすっかり入ってしまっていることを示している。スレッド 3 より大きな場合はほぼリニアに実行時間が増加していることから、スレッド数が 2 か 3 でリモートアクセスレイテンシによるプロセッサがアイドルになっている状態が無くなっていることが分かる。スレッド数 3 以上の時のグラフの傾きから、各処理で他のスレッドとオーバラップできない時間を求めることができる。SYSRD の時には、傾きから求められる処理時間からループのオーバヘッドを除いた値は 0.45 マイクロ秒であり、1 スレッドの時のレイテンシからこの値を差し引くと、他のスレッド実行に利用可能な部分が 0.86 マイクロ秒であることが分かる。他の場合を含めて表 5.1 に示す。

5.1.2 2PE 間スループット

静的レイテンシと同様に、他のアクティビティが無い場合の 2PE 間スループットも、並列計算機の基本特性として重要である。2PE 間でブロック転送するプログラムを実行して、2PE 間のスループットを

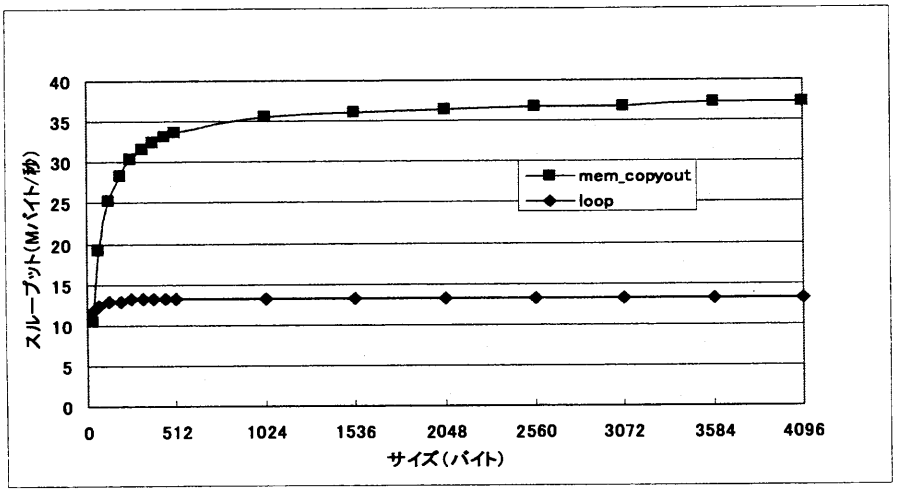


図 5.3: 2PE 間のスループット

測定した。EM-X はブロック転送をハードウェアではサポートしていないため、以下のようなループを実行することとなる。

```
int *soc;
int global *dest;
for (i=0;i<size;i++) {
    *dest = *soc;
    dest ++;
    soc ++;
}
```

現在の EM-C の生成するコードはループあたり 6 命令である。これに対し、アセンブラレベルの最適化を施すとともに、16 ワード (64 バイト) 以上のデータ転送に対して 16 ループのアンローリングを行って更なる高速化を行った mem_copyout() というライブラリ関数が作成されている。このライブラリの詳細については 4.2.2 ブロック転送を参照。この関数は 16 ワード転送あたり 34 命令で実行できる。この両者を転送サイズを変更して比較した結果が図 5.3 である。ネットワーク転送のピークが 40M バイト/秒なので、mem_copyout() ライブラリでは 4K バイトの転送でピークの 90%程度は出ていることが分かる。

また、このブロック転送には SYSWR パケットが用いられており、SYSWR パケットを受け取った PE ではその処理をスレッド実行とは独立に行える。このため、全 PE がネットワークへのパケット送信と、ネットワークからパケットを受け取ってメモリへ書き込む処理を同時に可能である。また、隣接間通信の場合パケットの衝突は起きないため、システム全体で最大で $37.2 \times 80 = 2.976$G バイト/秒の転送が実現できる。

5.1.3 実行時レイテンシ

静的レイテンシや 2PE 間スループットは、他のアクティビティのないシステム上で測定した結果である。しかし、実際のプログラム実行時にはネットワーク上での他のパケットとの衝突や、相手先プロセッサが他の処理を行っているための待ち時間など他の要素がレイテンシに加わる。そこで、システムにいくつか異なる負荷を与えた状態でのレイテンシを測定した。これを実行時レイテンシあるいは動的

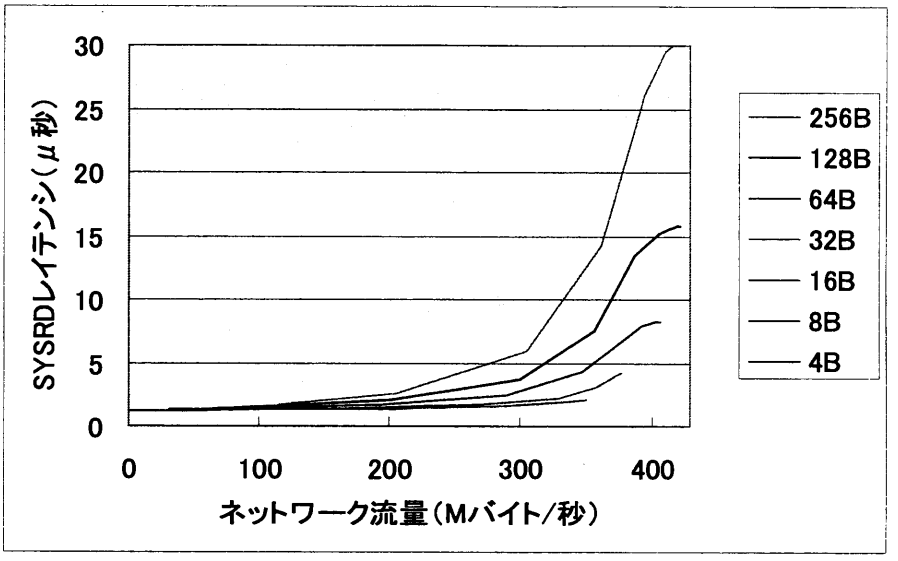


図 5.4: ネットワーク負荷によるランタイムレイテンシの変化

レイテンシと呼ぶ。

この評価では、80PE を用いて、PE0 では他の全 PE に SYSRD によるリモートメモリ呼び出しを繰り返しつつ、他の PE では、ネットワークに負荷を与えるような単純ループを繰り返す。このループは、あらかじめランダムに生成したテーブルをもとに、相手先 PE を選択し、N バイトのリモートブロック転送 (mem_copyout() ライブラリを利用) を行い、L クロックサイクルの間通信を休む (実際にはダミーループ)、ということを繰り返す。このループは通信と計算の割合をモデル化したもので、L サイクルの通信休止期間が計算処理を、N バイトのブロック転送が通信処理をそれぞれ表している。N を固定して L を大きくすると、それに比例してネットワークに対する平均通信負荷が小さくなる。上記 N と L のパラメータを変えながら、このループを一定回数実行して、その実行時間で転送バイト数を割ると、平均ネットワーク流量が測定できる。そのようなネットワーク負荷の時の、PE0 での SYSRD の平均レイテンシを表したのが図 5.4 である。各グラフで nB とは n バイトのブロック転送を用いた場合を示している。そして、各グラフの右端は L を 0 にした場合である。

この図によると、16 バイト程度の転送を繰り返したときには、連続して転送しても SYSRD の実行時レイテンシはほとんど変らない。これは、64 バイトより小さいブロックサイズでは mem_copyout() のループアンロールの効果が無く、4 クロックサイクルに 1 ワード (4 バイト) の通信頻度であることが影響していると思われる。この時の出力ポートの利用率は最大 50%である。これらはネットワーク上での衝突などを含んだ実行時間であり、ネットワークはスムーズに流れていることが分かる。一方、ネットワーク流量が 200M バイト/秒から 300M バイト/秒の範囲では短いブロック転送ではほとんどレイテンシが変らないのに対し、256 バイト程度のブロック転送では徐々に実行レイテンシが増加しているのが分かる。これは 64 バイト以上のブロックサイズでは mem_copyout() のループアンロールと各種最適化により 2 クロックサイクルに 1 ワードの通信頻度となり、ブロック転送時の出力ポートの利用率はほぼ 100%になっている。このため、パケット衝突の頻度が高くなっているためであると思われる。さらにネットワーク総量が 300M バイト/秒を越えると、実行時レイテンシは加速度的に増大する。400M バイト/秒を越えたあたりでネットワークは飽和していると考えられる。

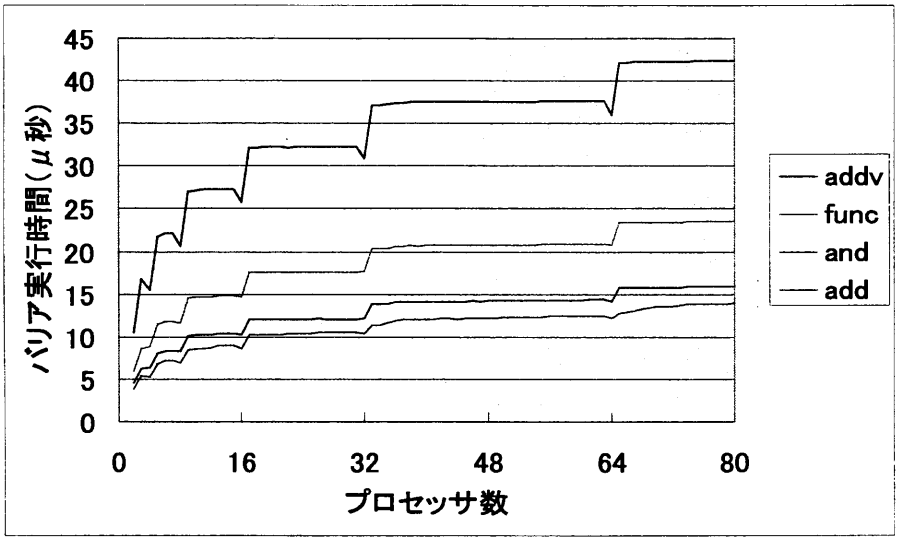


図 5.5: バリア同期の性能

5.1.4 バリア同期

複数のスレッド間の同期における EM-X の性能を評価するために、局所同期を用いたバリア同期ライブラリを用いて、性能測定を行った。このバリア同期ライブラリでは、局所同期に I-structure を用いて、バタフライネットワークをエミュレーションしている。このようにソフトウェア処理によりバリア同期を実現しているため、バリア同期と同じ処理時間で各種リダクション処理を行うことが可能である。このため、バリア処理のみを行うライブラリ関数は無く、全てリダクション処理を伴う。最も基本的なリダクション関数である各プロセッサのデータの総和を求める barrier_adds() を通常はバリア同期関数として用いる。これらのライブラリ関数に関する詳細は 4.2.2 バリア同期/リダクション処理を参照。

図 5.5 はプロセッサ数を変化させた場合のバリア同期の時間をマイクロ秒単位で示したものである。バリア同期時間としては、バリア関数を呼び出すだけのループをそれぞれのノードで実行し、その処理時間からループオーバヘッド (ループ本体が空のループの処理時間) を除き、ループ回数で割った値とした。図 5.5 で add とは各プロセッサのデータの総和をバタフライネットワークを用いて求めるライブラリ関数 barrier_adds() の結果である。

EM-X ではバリア同期に対する特別なハードウェアサポートは無いが、実装したソフトウェアルーチンは barrier_adds() で 64PE の時に約 12 マイクロ秒と十分高速であると言える。また、64PE では各 PE で 6 回の I_structure を用いた局所同期を行っており、1 局所同期あたり 2 マイクロ秒、約 32M/秒の同期が行えていることが分かる。また、ソフトウェア処理であることにより様々なスレッドの組合わせのバリアを複数同時に処理することも可能である。

図 5.5 には barrier_adds() を含めて 4 つの種類のリダクション処理を示している。この barrier_adds() はアセンブラレベルでの最適化を行っているため、80PE で 14 マイクロ秒と他の処理に比べてやや性能が良い。その他はライブラリのメンテナンス性を高めるために完全に C により記述されている。and は各プロセッサのデータの and 処理を行うライブラリ関数 barrier_ands() の結果である。func は任意の 2 入力処理を指定してリダクション処理を行うことができるライブラリ関数 barrier_funcs() を用いて、関数として 2 入力の和を求める関数を指定した場合の結果である。

addv は各プロセッサの持つ配列の各インデックス毎の総和を求めるライブラリ関数 barrier_addv() の結果であり、図 5.5 はベクタサイズが 1 の時の結果である。64PE の場合にベクタサイズを変化させたときの 1 ワードあたりのリダクション処理時間を図 5.6 に示す。この図によると 4 ワードよりも大きなベ

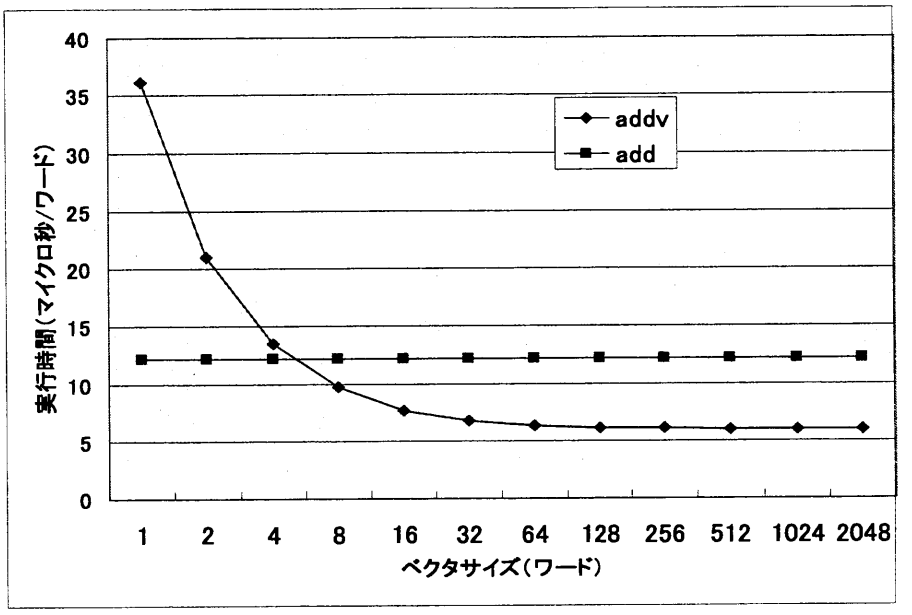


図 5.6: ベクタ型バリア同期の性能

クタの総和を求めるときには barrier_addv() を用いると良いことが分かる。

5.2 カーネルベンチマーク

マイクロベンチマークは、前記のように通信や同期に関する EM-X の性能を示すことはできるが、EM-X の実行性能全体を表してはいない。より現実的な例題における性能を示す前に、もう少し小さなカーネルプログラムを用いて、詳細な実行状況などによる性能評価を行う。ここでは、行列乗算プログラム、ナップサック問題、三角方程式の解法を実行し、その実行時間ならびに各要素プロセッサの実行状況について評価を行なった。それぞれのプログラムは、EM-C とスレッドライブラリを用いて記述されいる。

5.2.1 行列乗算

各 PE に分散配置された行列の乗算の性能評価を行う。行列の各データは単精度浮動小数点数とする。これは、EM-X の演算ユニットは倍精度浮動小数点数をサポートしていないためである。リモートメモリのアクセス方式として、利用するプロセッサが読み出す方式と、データを所有しているプロセッサが利用するプロセッサに書き込む方式の 2 種類が考えられるため、その両者を比較する。さらに、マルチスレッド化やレジスタ待避オーバヘッドの削減についても検討を行う。

まず最初に、必要な行列要素をリモートメモリ読み出しを用いて参照するプログラムを考える。本プログラムでは、要素プロセッサ (PE) 数は 64 台とし、行列のサイズを変化させて性能を評価する。行列は行サイクリックで各 PE に分配しているものとする。カーネル部分を図 5.7 matmul-rd に示す。これを配列サイズ 64 から 512 まで実行した結果が図 5.8 の rd である。縦軸は逐次処理時間を 1 としたときの性能向上率である。図によるとほぼ配列サイズに関係なく逐次処理の 10 倍という結果であった。このように 64 台で 10 倍という低い台数効果となったのは、レイテンシの隠蔽を行っていないこと、および最内ループでスレッドの切り替えが起きるため、レジスタ待避のオーバヘッドが生じるためである。レジスタ待避のオーバヘッドはコンパイラで書き換えが起きるレジスタのみを静的に選んで行うため、数

```
matmul-rd()
{
  int i,j,k;
  float s;

  for (i=0;i<rsize;i++)
    for (j=0;j<size;j++) {
      s = 0.0;
      for (k=0;k<size;k++)
        s += A[i][k] * float_read(GLOBAL_ADDR
                (petbl[k%PES], &B[k/PES][j]));
      C[i][j] = s;
    }
}
```

```
matmul-wr()
{
  int i,j,k;
  float s;

  for (j=0;j<size;j++) {
    for (i=0;i<rsize;i++)
      broadcast(&T[i*PES+id]), B[i][j]);
    barrier_add(0);
    for (i=0;i<rsize;i++) {
      s = 0.0;
      for (k=0;k<size;k++)
        s += A[i][k] * T[k];
      C[i][j] = s;
    }
    barrier_add(0);
  }
}
```

図 5.7: 行列乗算プログラム

命令 (この場合は 8 命令) 程度であるが、ループ自体が 31 命令と短いため、相対的に大きなオーバヘッドとなる。

リモートメモリ読み出しのレイテンシを隠蔽するために、マルチスレッド化したプログラムも評価した。サイズが 64 の時にもうまくマルチスレッド化できるように、図 5.7 で j に関するループを偶数と奇数の 2 つのスレッドに分割して実行した。その結果が図 5.8 の thd2 である。マルチスレッド化により逐次処理に比べて 16 倍と、シングルスレッド版に比べて 6 割ほどの性能向上となっている。同様に 3 つのスレッドに分割したプログラム thd3 を実行した結果が図 5.8 の thd3 であるが、ほとんど thd2 と同じ性能であった。これにより 2 つのスレッドですでに十分レイテンシを隠蔽できていると考えられる。最内ループの実行は 31 命令 31 クロックサイクルであり、レイテンシの平均を L とすると、rd の最内ループの実行時間は $(31 + L)$ クロックサイクルと表わすことができる。一方、thd2 の最内ループの実行時間はレイテンシが完全に隠蔽できているとすると 31 クロックサイクルとなる。thd2 が rd の 1.6 倍の性能を持つことより、$(31 + L)/31 = 1.6$ という式が導け、これよりリモート呼び出しの平均レイテンシはおよそ 19 サイクルであることが分かる。これは前節で求めた他の負荷がない場合の静的レイテンシとほぼ同じレイテンシであり、スレッド実行などに影響されずに低いレイテンシでリモートメモリアクセスが実行できていることが分かる。

次に、データを所有しているプロセッサが他のプロセッサに必要なデータを書き込むプログラムを考える。行列の分割は上と同様に行サイクリックであるとする。カーネル部分を図 5.7 matmul-wr に示す。書き込みベースのプログラムでは必要な matb の列を一時ベクタに格納する。この時、同じ列を必要とする計算をまとめて行うことにより、通信回数を減らすことが可能である。これを配列サイズを 64 から 4096 まで実行した結果を図 5.9 の wr に示す。縦軸は逐次処理に対する性能向上率である。ただし、逐次処理ではメモリサイズの制限によりサイズ 512 までしか実行できないので、それより大きなサイズは外挿により求めた値を利用した。配列サイズが大きくなるにつれて列の再利用の効果が増大し、4096 では 64PE で 62 倍の性能を達成している。

列の再利用は読み出しベースのプログラムでも可能である。先ほどの 2 スレッド版で列の再利用を用いた結果を図 5.9 の thd2 に示す。図 5.8 とは異なり配列サイズが大きくなるにつれて性能は向上しているが、wr に比べると配列サイズ 256 で 15%ほど性能が低い。これはリードベースでは各リモートメモリ読み出し毎にスレッド切り替えが生じるためオーバヘッドがやや大きいためと思われる。配列サイズ

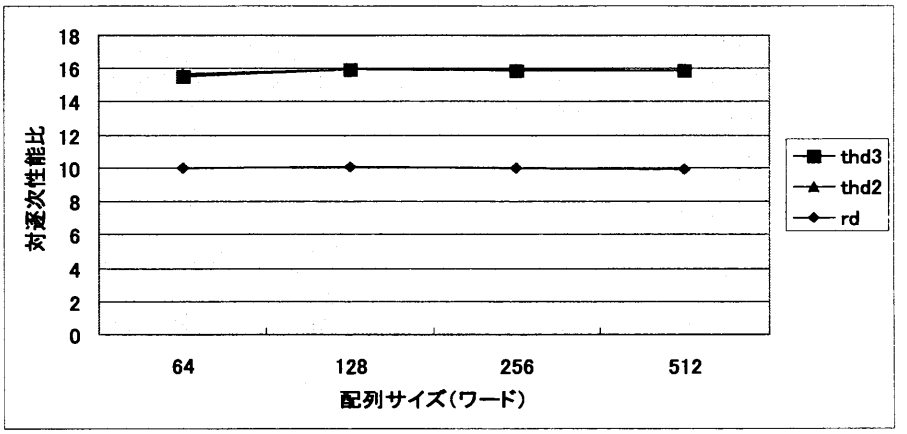


図 5.8: リモートメモリ読み出しによる行列乗算 (64PE)

が 64 の時にその差が小さいのは書き込みベースがバリア同期が必要になるためである。しかしバリア同期は一時ベクタの取得の度に 1 度で良いので配列サイズが大きくなるにつれてそのオーバヘッドは小さくなる。一方、リモートメモリ読み出しのオーバヘッドは配列サイズに比例するので、配列サイズ大きくなると一時ベクタ取得にかかる時間の差が大きくなる。しかし、配列サイズが大きくなると、全実行時間に対する一時ベクタ取得にかかる時間の割合が小さくなるため、全体としての両者の違いは小さくなる。これにより、図のように配列サイズ 256 くらいで両者の差が最大となり、それより配列サイズが大きくなるにつれての両者の差は小さくなる。

5.2.2 ナップサック問題

ナップサック問題とは、Object の個数 n、その重さのリスト W[i]、その価値のリスト P[i]、および重さの合計の上限値 Cap が与えられたときに、その制限を満たした上で価値の合計が最大となる Object の組合わせを求める探索問題である。ここでは、分岐限定法とダイナミックプログラミングという 2 つの異るアルゴリズムについてそれぞれ並列化を行い、EM-X で評価を行った。

まず、最初に分岐限定法を用いた評価を行う。分岐限定法では、ある Object を選択するかしないかで場合分けを行い、それぞれについて残りの探索を繰り返すという探索木を生成することとなる。全ての場合分けを行えば最適解が見つかるわけであるが、n 個の object があるとすると、$O(2^n)$ の手間がかかるのでそのまま実行することは現実的ではない。しかし、すでに見つかった解やその他の情報を用いて、ある枝以降の探索ではそれより良い解が見つからないことが確認できれば、その枝の探索を省くことが可能である。これを枝刈りという。探索木の生成の順序や枝刈りに使う情報等により種々の方法が考えられるが、ここでは次のようなアルゴリズムに基づいたプログラムを逐次処理プログラムとした。

- あらかじめ object を価値/重さによりソートしておく。
- 各探索木の下限として、object を順に選択していき選択できない object が出てくるまでの合計を用いる。また、上限として、その最初に置けない object を残りの重さ分だけ分割して選択した場合の値を用いる。object は価値/重さの順にソートされているので、この探索木の価値の合計がこれより大きくなることはない。
- 下限がこれまで見つかった解 (近似解) よりも大きければ、これを新たな近似解とする。
- 上限が近似解よりも小さければ、この枝は最適解とはならないので、枝刈りする。

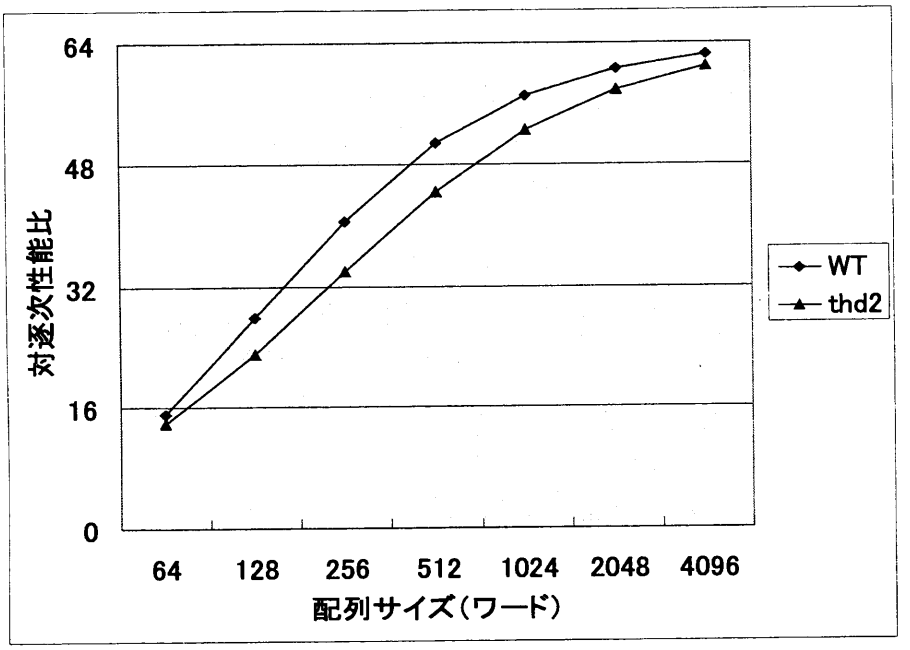


図 5.9: 列を再利用した行列乗算 (64PE)

- 連続して object を選択しているときにはその探索木の上限/下限は変らないので、ループ呼び出しに変換して再帰オーバヘッドを軽減する。
- i 番目以降の最も小さい object の重さを表にしておき、再帰の終了条件に利用する。

次に、これを並列化した。並列化の手順は以下の通り。

- PE0 をマスター、それ以外の PE をスレーブとする。
- PE0 から object を選択しない場合をスレーブに fork する。
- PE0 では object を選択した場合を継続。
- 利用可能な PE をリングバッファにより PE0 で管理。処理が終了した PE は、結果を返すとともに、自分の PE 番号をリングバッファに追加する。
- スレーブ PE では逐次的に処理を行う。
- すでに見つけた近似解 (GLow) を全 PE で共有する。具体的には、各 PE は GLow のコピーを持ち、ローカルに参照を行う。新たな Glow を見つけた場合は、全 PE へ更新要求を行う。
- 各 PE が独立に GLow の更新要求を行うため、受け取った側が本当により良い解であるかどうかをチェックする必要がある。このため、リモートメモリ書き込みではなく、リモート関数呼び出しによる更新要求が必要である。
- 通常のリモート関数呼び出しでは関数フレームの取得などオーバヘッドが大きいので、特殊パケットによるハンドラ起動を用いる。
- 更新要求を高プライオリティ処理として設定しておくと、他の検索スレッドよりも優先的に起動されるため、GLow が優先的に更新され、枝刈りが効率的に行える。

表 5.2: ナップサック (分岐限定法) の実行結果

問題番号	object数	EM-X(80PE) 逐次実行時間 (秒)	EM-X(80PE) 並列実行時間 (秒)	EM-X(80PE) 速度向上率	CS6400(16PE) 逐次実行時間 (秒)	CS6400(16PE) 並列実行時間 (秒)	CS6400(16PE) 速度向上率
1	200	0.025	0.033	0.76	0.008	0.370	0.02
2	200	0.027	0.037	0.73	0.009	0.348	0.03
3	200	0.029	0.037	0.78	0.009	0.377	0.02
4	300	0.059	0.064	0.92	0.019	0.568	0.03
5	300	0.073	0.070	1.04	0.021	0.562	0.04
6	42	0.004	0.004	1.00	0.0005	0.003	0.17
7	100	175.409	4.526	38.80	43.525	14.826	2.94
8	100	230.120	11.907	19.30	51.999	49.742	1.05
9	500	43.180	0.357	120.90	19.108	1.598	12.00

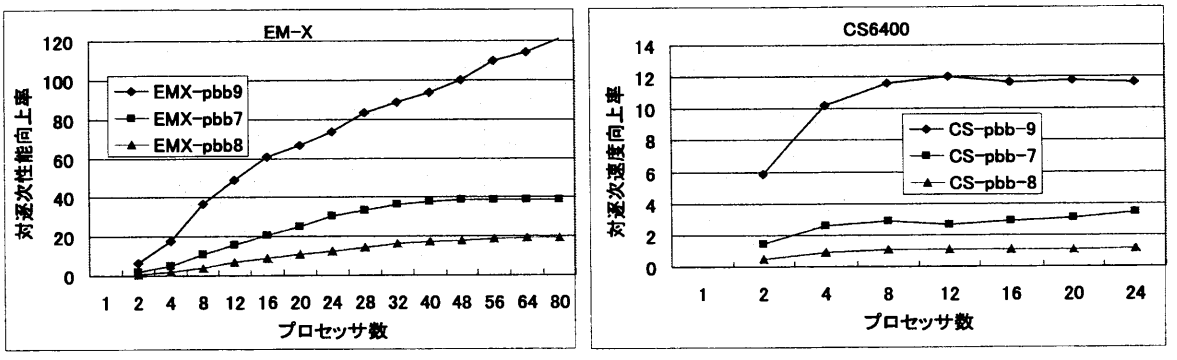


図 5.10: 分岐限定法によるナップサック問題の台数効果

この並列プログラムを 80PE の EM-X で実行した結果が表 5.2 である。問題としては、並列処理シンポジウム 1997 で開催された並列プログラムコンテストの本選の問題 9 題を用いた。また、比較として SMP 型並列計算機である CraySuperserver6400(以下 CS6400) での結果も示した。CS6400 は 60MHz の SuperSparc+をノードプロセッサとする SMP マシンである。共有メモリバスのバンド幅は 1.7G バイト/秒である。CS6400 は計算機センタの計算機であり、他のユーザと一緒に利用している。そのため、他のジョブの影響を考えて、64PE のうち 16PE のみ利用した結果を示している。object 数は 42 から 500 と問題により異るが、object 数によらず、問題により大きく実行時間が異ることが分かる。概して逐次処理で非常に時間がかかる問題が、並列処理効果が大きい。EM-X では最大 120 倍とプロセッサ台数以上の性能が出ている場合がある。これは並列探索問題では往々にしてあることであるが、並列に探索していてそのうちのどれか一つで早いうちに最適解を見つけられると、その後の枝刈り効率が向上することによる。一方、逐次処理で時間がかからない問題というのは、逐次処理で優先的に探索する木、すなわち価値/重さの高い object から選んで行く木が正解に近い場合である。そのような場合、CS6400 では逐次処理の 30 倍ほどもの時間がかかっているのに対し、EM-X ではほぼ逐次処理と同程度の時間で済んでいる。これは EM-X の並列オーバヘッドが小さいためであると言える。

また、図 5.10 は並列効果があった問題 7、8、9 に対して、プロセッサ台数を変化させてその対逐次速度向上率を示したものである。左が EM-X、右が CS6400 である。また、EMX-pbb-7 とあるのが EM-X による並列分岐限定法による問題 7 の並列効果である。EM-X でスーパリニアな結果になった問題 9 は、

80PE までプロセッサ台数に応じて性能が向上していることが分かる。一方、CS6400 では 8 台程度で性能が飽和していることが分かる。問題 7、8 では EM-X でも 40PE 程度で性能が飽和している。これは、現在スレーブ側では逐次処理をしており、並列性抽出および負荷分散が十分には行われていないことによるものであると考えられる。

次に、ダイナミックプログラミングによる解法を検討する。ある n 個の要素に関する最適解を求めるときに、i 個の要素からなる部分集合だけを用いた最適化を表にして求めておく。次に、要素を 1 つ加えたときに最適解がどう変化するかを、この表をもとに調べて、表を更新する。これを n 個の要素になるまで繰り返せば、最終的な最適化が求められる。これがダイナミックプログラミングと呼ばれる手法である。これをナップサック問題に適用すると、次のようになる。

- まず、重さの上限値 Cap をサイズとする配列を作成する。
- これに、1 個の object だけの場合の最適解を求める。すなわち、この object の重さ W[0] が cap 以下ならば、この object を選択するのが最適解であり、W[0] の欄に P[0] を記入する。
- 次に object を 1 個追加したときの最適解を求める。この object だけを選んだときには W[1] の欄に P[1] を記入し、すでにテーブルに記入されている組合わせ (例えば i の欄に p と記入されている) にこの object を追加するときには、i+W[1] の欄に p+P[1] を記入する。ただし、記入しようとした欄にすでに記入されている場合には、より高い価値を持つ場合のみ記入することとする。
- これを全ての object を追加するまで繰り返せば、最適解が求まる。
- 最小の object の重さを minw とすると値が記入されるテーブルは minw から Cap までとテーブルサイズを短くできる。
- すでに置かれているテーブルをスキャンする範囲を、記入したテーブルの最大/最小を管理することにより短くできる。
- 追加する object の順番により更新頻度が異るが、限定分岐法と同様に価値/重さの大きいの順番に object をソートして、その順番に object を追加することとする。

このテーブルの更新を各 PE に分割することにより、共有メモリでの並列化は容易に行える。EM-X では分散メモリであるため、テーブルを各 PE に分散させることになるが、他の PE に割り当てられたテーブルを参照することが必要となり、その参照方式により次の 2 つの方法を試してみた。

粗粒度並列化 各 object の処理を開始する前に、参照するテーブル (自分の持っているテーブルよりも object の重さ分だけ前のテーブルの内容) をローカルにコピーして持ってくる方式。参照のためのコピーを持つため、テーブルを直接更新して良い。

細粒度並列化 自分の持っているテーブル以外にアクセスするときにはワード単位のリモートアクセスを行う。このリモートアクセスのレイテンシを隠蔽するために、プリフェッチを適用。参照と更新が同時に起きるため、2 つのテーブルを交互に用いるダブルバッファリングを適用して、参照と更新を分離する。このため、値の更新がなくても更新側のテーブルにコピーする必要がある。

両者の並列化を実行した結果が表 5.3 である。各実行時間と逐次処理との速度比を示すとともに、比較のために限定分岐法による結果も示している。限定分岐法とは異り、ダイナミックプログラミングでは実行時間がほぼ object 個数に比例していることが分かる。また、並列効果も問題による違いは小さい。粗粒度方式で、80PE で平均 32 倍、プリフェッチ方式の細粒度方式で 24 倍の速度向上を示している。また、細粒度方式でプリフェッチなしでは平均 16.9 倍であり、プリフェッチにより 44%ほどの速度向上を示している。しかし、これでも粗粒度方式より遅い。これは最内ループ内でスレッド切り替えが

表 5.3: ナップサック (ダイナミックプログラミング) の実行結果

問題番号	ダイナミックプログラミング							限定分岐法	
		粗粒度並列		細粒度並列					
				プリフェッチなし		プリフェッチあり			
	逐次実行時間 (秒)	並列実行時間 (秒)	速度比	並列実行時間 (秒)	速度比	並列実行時間 (秒)	速度比	並列実行時間 (秒)	速度比
1	22.67	0.63	35.9	1.19	19.0	0.84	26.9	0.033	687
2	12.20	0.35	34.6	0.65	18.7	0.47	26.2	0.037	330
3	5.44	0.17	31.6	0.30	18.1	0.22	24.7	0.037	147
4	49.97	1.30	38.5	2.46	20.3	1.77	28.2	0.064	781
5	27.66	0.74	37.6	2.36	11.8	0.99	27.9	0.070	395
6	0.19	0.01	14.6	0.02	10.0	0.02	11.9	0.004	47.5
7	1.31	0.05	27.9	0.08	15.6	0.06	21.8	4.526	0.29
8	1.34	0.05	27.9	0.09	15.8	0.06	22.3	11.907	0.11
9	171.52	3.97	43.2	7.58	22.6	5.87	29.2	0.357	480
平均			32.4		16.9		24.4		

起き、レジスタ待避などのオーバヘッドが生じるためである。しかし、本プログラムでは他のスレッドは走っていないためレジスタ待避の必要はない。そのような最適化を行った場合はほぼ粗粒度並列と同程度の性能が確認できている。

また、表 5.4 は粗粒度方式の結果を、共有メモリ型並列計算機 Xfire と比較したものである。Xfire は 336MHz の UltraSparc-II をノードプロセッサとする SMP マシンである。64 プロセッサからなり、共有メモリバックプレーンは 10.4G バイト/秒のスループットを持つ。CS6400 と比べると、逐次性能は 8 倍ほどであるが、ナップサックによる並列性能は 4-5 倍くらいである。問題規模の大きな問題 9 では、Xfire では 16 プロセッサで 6.3 倍とピークの 4 割程度の並列性能を出しているが、EM-X では 80 プロセッサで 43.2 倍とピークの半分以上の性能を出している。Xfire と比べると向上率は高いが半分程度の性能にとどまっているとも言える。これは EM-X ではリモートデータをいったんローカルにコピーしてから処理行っているが、ローカルにおける処理が小さいため、データ転送のオーバヘッドが相対的に大きくなっていることによると考えられる。このデータ参照はテーブルをある範囲で連続的に参照するため、空間的局所性を有効に利用することができ、Xfire でも高い性能が出せていると考えられる。一方、問題規模の小さな問題 6 では、Xfire では逐次処理よりも 3 倍程度かかっている。これはスレッド起動のオーバヘッドが大きく、問題規模が小さいため、相対的にそのオーバヘッドが大きくなっているためと考えられる。EM-X ではこのような問題でもおよそ 14 倍の性能向上を示している。

図 5.11 は問題 6、7、5、9 についてプロセッサ台数を変化させて、その逐次性能比を示したものである。問題はそれぞれ object 数が 42、100、300、500 の場合である。ダイナミックプログラミングではほぼ object 数に比例した時間がかかって要るため、4 種類の問題規模を選択している。左が EM-X の結果、右が Xfire の結果である。他のプロセスなどの影響をさけるため、64 プロセッサ中 24 プロセッサまでを用い、また、3 回計測して最も良い値を採用している。この図によると、EM-X では問題規模が小さいと性能向上は低いが、それでも全ての場合でプロセッサ数を増加すると性能が向上している。一方、Xfire では問題サイズが最も小さい問題 6 の場合には、全く性能が向上せず、かえってプロセッサ数が増加すると性能が低下して、プロセッサ数 24 では逐次性能の 1/4 になっている。また、問題 7 ではプロセッサ数が 8 程度で飽和し、それ以上プロセッサ数が増加すると性能が低下している。問題 5 では

表 5.4: ナップサック (ダイナミックプログラミング) の比較

問題番号	EM-X			xfire		
	逐次	80 並列	速度比	逐次	16 並列	速度比
1	22.668	0.631	35.917	0.991	0.1760	5.63
2	12.197	0.353	34.561	0.510	0.1048	4.87
3	5.441	0.172	31.554	0.227	0.0653	3.85
4	49.968	1.299	38.477	2.250	0.3751	6.00
5	27.664	0.735	37.642	1.147	0.2056	5.58
6	0.194	0.013	14.460	0.011	0.0282	0.38
7	1.308	0.047	27.776	0.057	0.0355	2.05
8	1.335	0.048	28.024	0.058	0.0347	1.67
9	171.523	3.971	43.196	7.218	1.1457	6.30

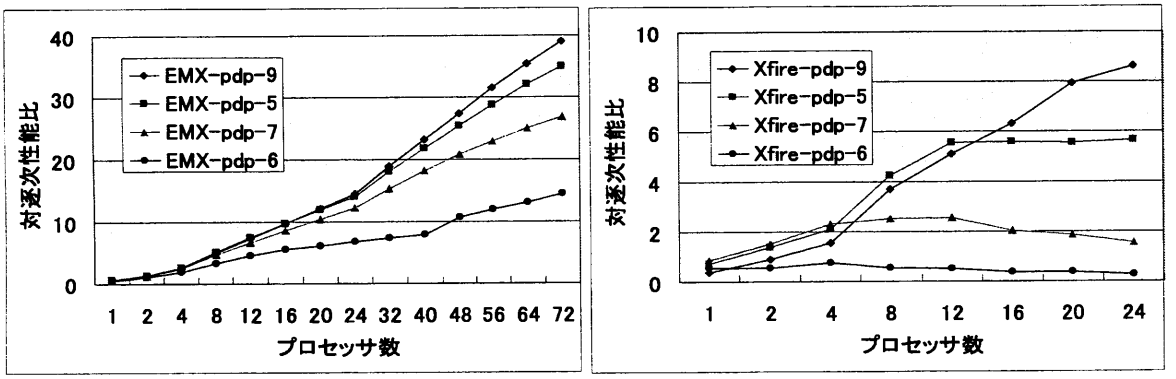


図 5.11: ダイナミックプログラミングによるナップサック問題の台数効果

12 プロセッサまで 6 倍とまずまずのスケーラビリティを示すが、その後性能が飽和してしまっている。問題 9 では 24 プロセッサまで性能が増加しているが、24 プロセッサで 9 倍程度にとどまっている。ただ、性能が飽和した問題は実行時間が 0.2 秒以下と極めて短時間であるので、あきらかに問題規模が小さい過ぎると言える。

5.2.3 三角方程式の解法

三角方程式とは、式 5.1 に示すような対角線より上 (あるいは下) の配列が全て 0 の場合の方程式であり、この方程式は式 5.2 のように単純に代入処理によって解くことができる。ここでは単純化のため、対角行列はすでに 1.0 に正規化されているものとする。LU 分解法は一般の配列をこのような三角行列に分解して方程式を解く方式である。通常 LU 分解部の計算量は配列サイズの 3 乗に比例するのに対し、三角方程式を解く計算量は配列サイズの 2 乗に比例するため、LU 分解部の高速化のみが注目され三角方程式の並列化はあまり行われていない。また、LU 分解部は並列化がしやすいのに対し、三角方程式はウェーブフロント型の依存関係があり、また負荷が不均一になっているので、効率的な並列化が難しいことも並列化されない理由である。本節では、EM-X の特徴の一つである、細粒度な同期を用いて、この三角方程式の並列化を行い、その性能を評価する。

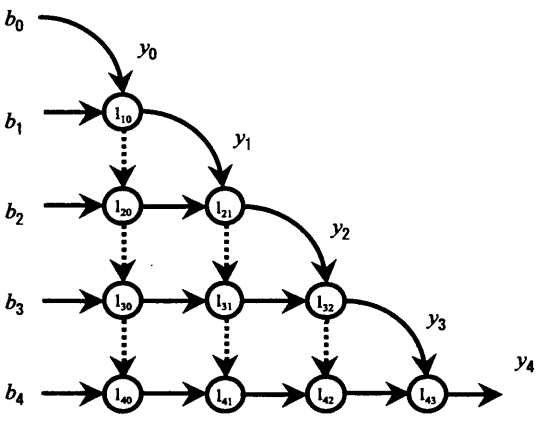


図 5.12: 三角方程式の計算の依存関係

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ l_{10} & 1.0 & 0.0 & 0.0 & 0.0 \\ l_{20} & l_{21} & 1.0 & 0.0 & 0.0 \\ l_{30} & l_{31} & l_{32} & 1.0 & 0.0 \\ l_{40} & l_{41} & l_{42} & l_{43} & 1.0 \end{pmatrix} \times \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \tag{5.1}$$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 - l_{10}y_0 \\ b_2 - l_{20}y_0 - l_{21}y_1 \\ b_3 - l_{30}y_0 - l_{31}y_1 - l_{32}y_2 \\ b_4 - l_{40}y_0 - l_{41}y_1 - l_{42}y_2 - l_{43}y_3 \end{pmatrix} \tag{5.2}$$

図 5.12 に各計算の依存関係を示す。実線矢印は各ノードの計算結果に対する依存を、点線矢印は (ノードからではなく)y_i に対する依存を表している。

行列が列サイクリックに各 PE に分散されているものとすると、y_i の通信は 1 対 1 通信となる。各 PE は y_i を受けとると、各項の計算を行い、その結果を隣の列に送る、ということを各行について繰り返す。この時、各行毎に通信/同期を行うと、wavefront 型の並列性が抽出できる。通信および同期は図の実線矢印で起き、n(n+1)/2 回必要である。この列サイクリックの方式を実装して、評価を行った。図 5.13 は 64PE を用いたときの逐次処理からの速度向上を、配列サイズを変化させて示したものである。逐次処理では、行方向に計算した方が配列への書き込み回数が少ない分高速であるので、そちらを基準とした。また、メモリ量の制限のため、逐次処理では配列サイズが 960 までしか計算できなかったため、それ以上のサイズについては、外挿により求めた値を用いている。配列サイズが小さい場合は並列呼び出しなどのオーバヘッドのため性能が低いが、配列サイズが 4096 くらいになると、逐次性能に比べて 64PE で 16 倍程度の性能を出している。プログラムの静的な解析によると、最内ループは 19 命令であった。この最内ループでは局所同期のためにスレッドを中断しており、レジスタの待避/復帰のオーバヘッドがかかっているが、本プログラムでは他にスレッドは走っていないため、このレジスタ待避/復帰を省くことが可能である。このような最適化を施したところ最内ループは 10 命令となり、性能も 1.4 倍に向上した。これにより局所同期のためのスレッド間隔 sync を推測すると、$(19 + sync)/(10 + sync) = 1.4$ より $sync = 12.3$ となる。また、合わせて 31.3 サイクルに 1 回の割合で各 PE で局所同期が行われており、41M 回/秒 (ハンドオプティマイズしたプログラムでは 57M 回/秒) の局所同期が行われていることになる。

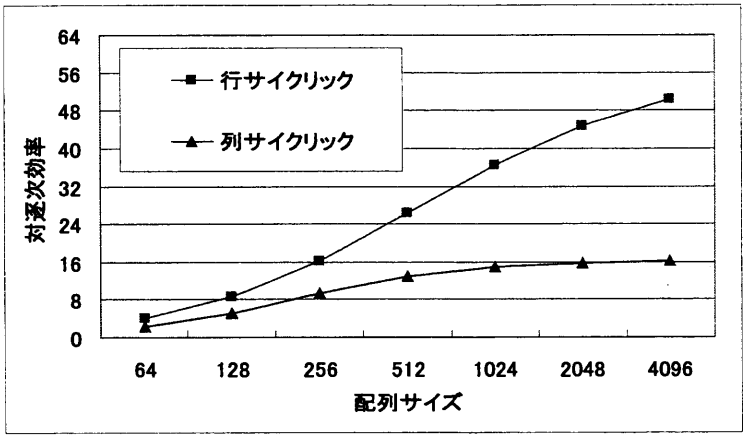


図 5.13: 三角方程式の結果 (64PE)

一方、行列が行サイクリックに各 PE に分散されているものとすると、y_i の通信は 1 対多通信となる。各 PE は y_i を受け取る度に各項の計算を行い、その行の解が求まると、それ以降の行全てに解を送るという処理となる。プロセッサ数が配列のサイズと同じとすると、通信および同期は y_i の実線および点線の矢印で起き、全体で n(n-1)/2 回必要である。ただし、プロセッサ数が配列サイズよりも小さい場合には、通信および同期の数を減らすことが可能である。すなわち、解が求まったら全プロセッサに対して 1 回だけ通信/同期を行い、プロセッサに割り当てられている他の行ではその解を再利用するだけで、通信/同期の必要はない。これにより、通信および同期の回数は $n \times pe$ となる。この行サイクリック方式を実装して、評価を行った。64 プロセッサで実行した結果が図 5.13 の行サイクリックである。行列サイズが小さいときには、列サイクリックよりもやや性能が良い程度であるが、サイズが大きくなると急激に性能が向上し、配列サイズ 4096 では逐次処理の 50 倍を達成した。これは一度プロセッサが受け取った解を再利用する割合が、サイズが大きくなるにつれて増大することによる。

5.3 マクロベンチマーク

マイクロベンチマークやカーネルベンチマークは、プログラムの一部の性能を詳細に示すことはできるが、EM-X の実行性能全体を表してはいない。より現実的な例題における性能を示すために、以下では MP3D および Radix ソートを実行し、その実行時間ならびに各要素プロセッサの実行状況について評価を行なった。MP3D はスタンフォード大学の作成した SPLASH ベンチマーク [Sin92] の 1 つである。Radix ソートも SPLASH や NAS 並列ベンチマークなどに含まれているが、radix のビットサイズやデータサイズがあまり適切な値となっていないため、独自に作成した。それぞれのプログラムは、EM-C とスレッドライブラリを用いて記述されいる。

5.3.1 MP3D

SPLASH は共有メモリプログラミング向けのベンチマーク集である。EM-X はこのような共有メモリ型のプログラムも、マルチスレッド機構と直接リモートメモリアクセス機構により効率的に実行できる。ただし、現在の EM-X のプログラミング環境では、自動並列化コンパイラは整備されていないため、共有変数を明示的に指定して global 属性をつけたり、大規模共有配列の明示的割り当て/参照など EM-X 向けの書き換えが必要である。

MP3D は SPLASH に含まれる 3 次元粒子シミュレーションを行うプログラムである。プログラムは、

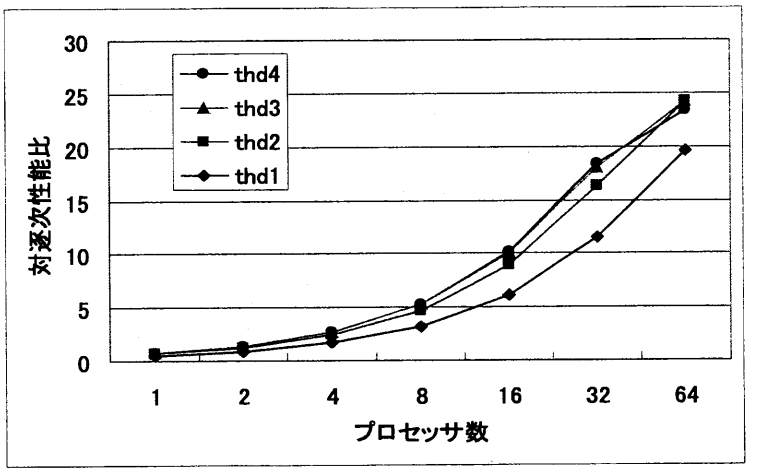


図 5.14: MP3D の実行結果

空間をセルに分割し、各粒子の速度と位置を計算し、同じ空間セルにある粒子との確率的なモデルを用いて衝突をシミュレーションする。これを時間ステップ毎に繰り返す。元の共有メモリプログラムは、粒子をプロセッサ毎に分割し、空間セルを共有メモリ上で共有するものである。EM-X では次のように並列化した。

データの割り当て 元のプログラムと同様に粒子を各プロセッサに割り当てる。空間セルの配列を全プロセッサにインターリーブして割り当てる。空間セルはそれぞれのプロセッサからリモートメモリアクセスで参照される。メインループでは、それぞれの粒子は局所的に参照されるが、衝突が検出された場合、相手の粒子をリモートに参照する。

マルチスレッドによる実行 それぞれのプロセッサ内でも、複数のスレッドを用いて、同時に複数の粒子の移動/更新の計算を行うことによって、空間セルに対するリモートメモリアクセスのレイテンシを隠蔽する。各時間ステップの処理のうち、粒子の移動/更新のみをマルチスレッド化している。すなわち、各時間ステップのループは単一スレッドで実行し、粒子の移動/更新を実行する際に、指定したスレッド数 − 1 のスレッドを新たに生成し、粒子の移動を各スレッドで計算して、全粒子の計算が終了するとスレッド間の同期をとって、以後の処理はまた単一スレッドで行う。

時間ステップの終わりにおいて、バリア同期でプロセッサの同期を行う。粒子については計算量はほぼ一定であるため、動的な負荷分散は行っていない。衝突回数等の統計情報は、バリア同期とともにリダクション計算を行うことにより、計算することができる。

図 5.14 はプロセッサ数を変化させたときの結果を示している。実行パラメータは、$24 \times 24 \times 8$ の空間中に 3000 粒子をおき、50 タイムステップだけ実行した。元のプログラムは各データは double 型であるが、EM-X では double 型をハードウェアではサポートしていないため、float 型とした。逐次プログラムの実行時間は 1.4 秒ほどであり、各曲線はそれに対して何倍の性能向上を果たしたかを示している。thd-n は各プロセッサでの処理を n 個のスレッドで実行した場合を示している。シングルスレッド実行では 1PE の時におよそ逐次処理の半分の性能、2PE で逐次処理とほぼ同じ性能、以降はプロセッサ台数の増加にほぼスケーラブルな性能向上を示している。

マルチスレッドによる効果を、5.15 に示す。これはシングルスレッド実行の性能を 1 としたときの各マルチスレッド実行の性能を示している。1PE では空間セルへのアクセスをリモートメモリアクセスによって行うため、スレッドを複数にすることによりそのレイテンシの間に他のスレッドの処理が行えることにより、3 割ほどの性能向上を示している。ただし、同一 PE へのリモートアクセスのレイテンシは

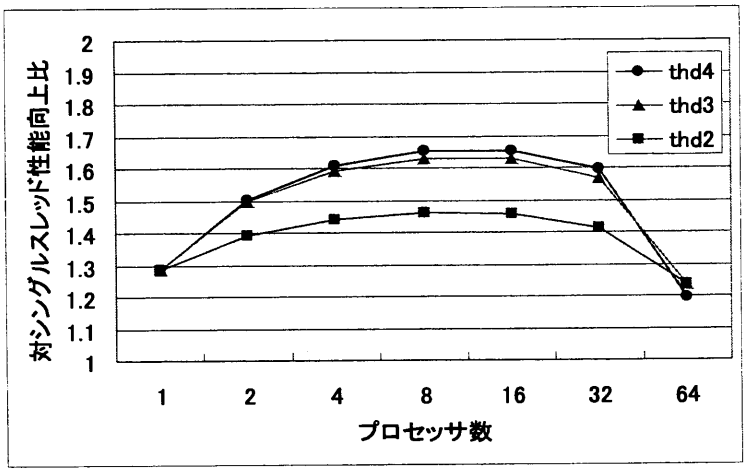


図 5.15: MP3D のマルチスレッド実行の効果

小さいため、スレッドを 2 以上にしてもほとんど性能は変らない。プロセッサ数が 2 以上では 2 スレッドで最大 46%性能向上するのに対して、3 スレッドでは 63%とさらに性能向上を示している。しかし、4 スレッドにしても 3 スレッドとほぼ同じで 3 スレッドあればほぼ全てのレイテンシを隠蔽できていると考えられる。また、64PE の場合には、3、4 スレッドの性能が 2 スレッドと同じになっている。これは粒子数が 3000 と小さいため、各 PE に割り当てられる粒子は 50 程度であり、これをさらにスレッドで分割するため各スレッドの処理量が小さくなり、相対的にスレッド生成のオーバヘッドが大きくなるためであるとも考えられる。また、処理量が小さくなると、負荷分散の影響も相対的に大きくなる。

しかし、粒子数を 24000 と大きくしてみたところ、並列性能は 64PE で最大 24.4 倍、マルチスレッドの効果も 16PE で最大 72%とやや性能は向上するが、64PE でのマルチスレッドの効果はやはり 2 スレッドで 29%、3 スレッドで 32%と頭打ちになる。このため、先に考えられた各スレッドの処理量が小さくなることにより、マルチスレッドの効果が小さくなるという考察は、主な要因ではないと推察できる。次に考えられる原因は、ネットワーク負荷による影響である。上記の実行は 80PE システムのうちの一部のプロセッサを使った結果である。16PE までは各プロセッサが間接オメガ網で接続された状況と同じであり、ネットワーク負荷の観点から見ると、かなり余裕がある使い方となっている。これに対して、64PE では各ネットワークノードがパケットを生成する状況となり、さらにマルチスレッドによりレイテンシを隠蔽してより高い頻度でパケット生成が行われるため、パケット出力バッファがいっぱいになってパイプラインがストールしている可能性がある。そこで、実際に実行トレースをとって調べてみると、32PE までは実行効率 80%程度で実行され、パイプラインストールは最大 1%程度しかなかったものが、64PE では 56%程度の効率に落ちており、パイプラインストールは最大 4.5%起きている。パイプラインストールは特に、デッドロック回避のために要求バンクをアップするプロセッサで起きており、通過するパケットを優先するためにパケット出力バッファがフルになりやすいためと思われる。ある PE の実行効率が落ちると、それにつれて全体の実行時間も延びる。これは各時間ステップで同期をとっており、他のプロセッサではもっとも遅いプロセッサが処理を終わるまでアイドルとなるためである。そのため、全体の実行効率はパイプラインストールの割合以上の低下となる。

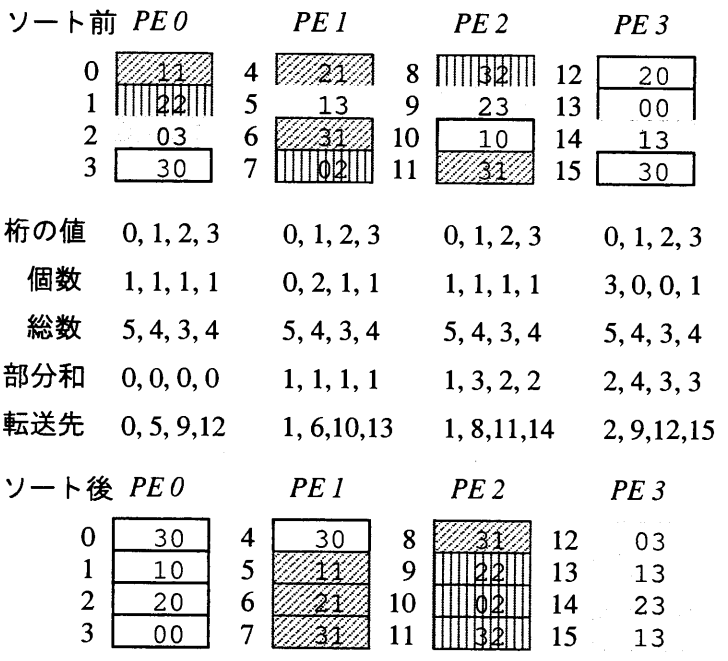


図 5.16: 並列 radix ソートアルゴリズム

5.3.2 Radix ソートの評価

Radix ソートの並列化

radix(基数) ソートとは、各データを n-進数で表し、各桁ごとのソートを繰り返すことにより全体のソートを行う方法である。効率化のために 2 の巾乗を n とすることが多い。各桁はたかだか n 通りしかないので、その各値の個数を数え上げることによりソート後の位置を計算することができる。クイックソート等のようにデータどうしの比較を行わないため実行時間の見積もりが容易であり、固定長のデータに対する効率的なソートとして知られている。ただし、radix ソートではデータと同じサイズのワークエリアを必要とし、かつランダムなワード単位の書き込みが生じるためメモリサイズやキャッシュ性能の影響を受けやすい。

データを各要素プロセッサ (PE) に分散配置しておくことにより、分散共有メモリの並列計算機で radix ソートを並列化することは容易であり、かつ、メモリアクセス競合がなく十分なデータ数があればプロセッサ台数に対して非常にスケーラブルな性能を期待できる。また、radix ソートはスタンフォード大学でまとめられている共有メモリ並列ベンチマーク SPLASH-2[Woo95] にも取り上げられている。図 5.16 にプロセッサ数を 4、データ数を 16、radix を 4 とした場合の最下位桁のソートの様子を示す。ソートの手順は以下のとおり。

1. 各 PE でローカルに最下位桁が 0、1、2、3 である要素数を数える。
2. (1) の結果の全 PE の総和を求める (総和)。
3. (1) の結果の自分より PE 番号の小さい PE までの部分和を求める (部分和)。
4. 上の 2 つの値から各値の転送開始アドレスを求める (転送先)。
5. 各データを上の転送アドレスから転送する。

たとえば、PE2 の先頭要素 (index=8) の転送開始アドレスを求めるには次のようにすればよい。

1. ソートキーとなる桁の値は2。
2. ステップ (2) で求めた各要素数の総和から最下位桁が 0、1 である要素が全体で $5+4=9$ 個あることが分かる。
3. ステップ (3) で求めた部分和から最下位桁が 2 である要素が PE0、PE1 に 2 個あることが分かる。
4. したがって PE2 の最下位桁が 2 である要素は転送先バッファの $9+2=11$ 番目の要素から転送開始すればよい。

また、データ転送時に各 PE のデータ数が同じになるように負荷分散を行うことにより、効率の良い負荷分散が可能である。

全 PE での総和は、ハードウェアのバリア同期機構で求めることのできる並列計算機もあるが、部分和を求める機構まで備えた計算機はない。そのため、バタフライネットワークなどをソフト的にエミュレートする方法が一般的であり、PE 台数の log オーダで処理が行える。

データ転送はワード単位で行われ、転送アドレスも不規則である。このため、セットアップに時間がかかるようなメッセージ通信機構の場合は非常に効率が低下する。そのような計算機で通信を効率良く行うためには、ワード単位に分かれているデータをいったんブロックデータにまとめる必要があるが、これらの処理に余分なメモリアクセスが必要となり、性能低下の原因となる。

radix ソートの処理時間は、通信の衝突などを無視した場合、データの内容には依存せず、データ数と PE 台数および radix の大きさにより定式化できる。ローカルな要素数の数え上げの 1 ワードあたりの処理時間を Tl、総和と部分和を求めるグローバルな処理に要する 1 データあたりの時間を Tg、1 ワードあたりの転送時間を Tw、総データ数を N、PE 台数を P、データの bit 長を s、radix を r とする。1PE あたりのデータ数は $n=N/P$、全体をソートするのにかかる繰り返しの数は $R=s/\log r$ であるので、全体の処理時間は次のとおり。

$$\begin{aligned} T &= R\times(nTl+rTg+nTw) \\ &= N/(P\log r)\times s(Tl+Tw) \\ &\quad +r/\log r\times(sTg) \end{aligned}$$

第 2 項は総データ数 N とは独立であるため、N を十分大きくすることにより第 2 項を無視することが可能であり、非常にスケーラブルな並列化が行えることが分かる。また、radix が大きいほど第 1 項は小さくなるが、第 2 項は大きくなるため適切な値をとることが必要となる。この値は計算機の各並列プリミティブ性能やデータサイズなどにより変化する。たとえばローカル処理や転送処理に比べてグローバルな処理が遅い場合は radix は小さな値に抑える必要があるが、グローバルな処理が速いほど radix を大きくとることが可能となり、全体の速度向上に役立つ。

並列 Radix ソートの実行と評価

上で述べた並列 radix ソートを、我々の開発した細粒度通信機構を持つ並列計算機 EM-X および他の並列計算機で実行し評価を行う。他の並列計算機としては EM-X と同様のマルチスレッド機構を持つ EM-4、リモートメモリアクセスをサポートしたメッセージ通信型並列計算機である AP1000+、強力なメモリバスを複数有する共有バス型並列計算機である CS6400 である。それぞれ同様のアルゴリズムでプログラムを作成した後、最適化を行い、実機を用いた実行を行っている。

最初に各並列計算機における逐次処理性能を逐次 radix ソートにより比較し、radix および問題サイズによる影響を述べる。次に各並列計算機での並列プログラムの生成および実行についてまとめる。そ

表 5.5: 要素プロセッサの性能諸元

	プロセッサ	動作周波数	データキャッシュ
EM-X	EMC-Y	20MHz	なし
EM-4	EMC-R	12.5MHz	なし
AP1000+	SuperSPARC	50MHz	L1:16K, L2:なし
CS6400	SuperSPARC	60MHz	L1:16K, L2:2MB

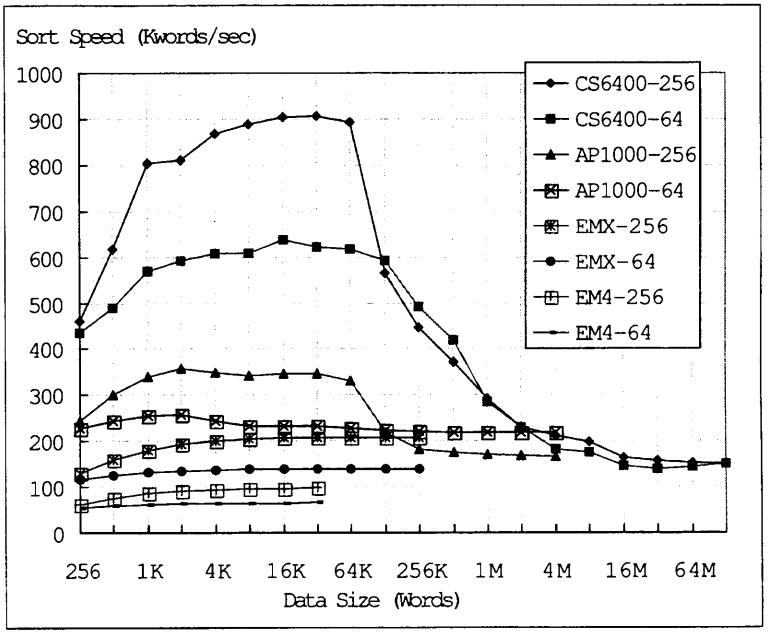


図 5.17: 逐次 radix ソート

の中で特に EM-X と EM-4 との比較を行ない、EM-X で改良された分散共有メモリ機構の評価を行う。最後に、各並列計算機において、プロセッサあたりのデータサイズを一定にして、プロセッサ台数を増やした場合のスケーラビリティについて比較を行う。

各並列計算機における逐次 Radix ソートの実行

各計算機のプロセッサ単体性能を表 5.5 に、各並列計算機の 1 プロセッサを用いた場合の逐次 radix ソートの実行結果を図 5.17 に示す。x 軸はデータ量、y 軸は 1 秒あたりのソートデータ量である。radix を 64 と 256 の場合を示した。各計算機で最大データ量が異なるのは搭載しているメモリ量が異なるためである。EM-X、EM-4 ではキャッシュがないため非常になめらかな結果となった。データは 32 ビット整数なので、radix が 64 で 6 回、256 で 4 回のステップ実行が必要であり、十分大きなデータに対しての実行結果もそれを裏付けている。

一方、SuperSPARC をプロセッサとする計算機では、キャッシュミスおよび TLB ミスの影響が現れる。radix ソートでは主に 3 種類のメモリデータを扱う。1 つ目がソートすべき元データ (soc) で、連続的に読み出されるため空間的局所性の効果が期待できる。2 つ目が各 radix に対応した書き込みポインタからなる配列 (wp) で、データサイズは radix の大きさと同じであり、他のデータに比べて十分小さく、繰り返しアクセスされるため、時間的局所性の効果が期待できる。3 つ目がソートした結果を格納するデータ (dst) で、ワード単位にランダムにアクセスされるが、各 radix ごとに見ると空間的には連続して

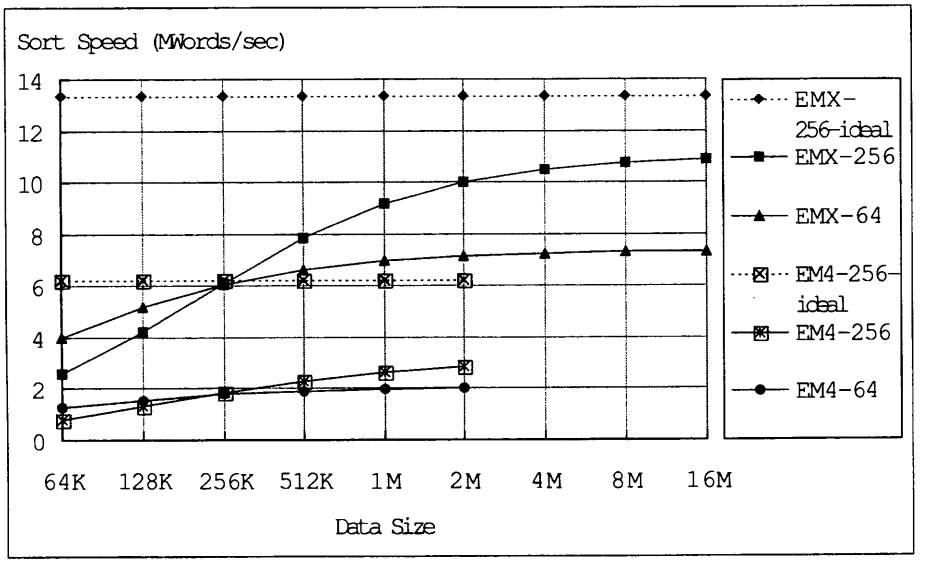


図 5.18: EM-X および EM-4 での並列 radix ソート（64PE）

書き込みが行われるため、空間的局所性の効果が期待できる。ただし、SuperSPARC では書き込み時にキャッシュミスが起きたときにキャッシュへの割り当てを行わないため、1 次キャッシュから溢れた段階でキャッシュ効果はなくなり、ライトバッファの効果のみとなる。データサイズが 4K ワードから 64K ワードまでの AP1000+と CS6400 の差が動作周波数以上に大きいのは 2 次キャッシュの効果である。

データサイズが 64K ワードを超えると AP1000+で性能が低下するのは TLB ミスのためである。SuperSPARC は 64 本の TLB を持っており、AP1000+では TLB は 4K バイトのページサイズである。radix が 256 の場合、データサイズが 64K ワードを超えると、書き込みを行うページ数が 64 を超え、TLB のスラッシングが始まり性能低下となる。データがすでにソートされていると、radix が 64 でもこの TLB のスラッシングの影響が見られることは確認したが、データがランダムであるため図 5.17 ではスラッシングの影響はあまり見られない。CS6400 ではデータサイズが 64K ワードを超えると、この TLB のスラッシングの影響が現れ、さらに 256K ワードを超えると、2 次キャッシュの溢れの影響が現れる。そのため radix が 64 のときも性能が低下する。2 次キャッシュはダイレクトマップであるためキャッシュ衝突の影響が大きい。

各並列計算機における並列 Radix ソートの実行

EM-X および EM-4

並列 radix アルゴリズムを、EM-X 上の C コンパイラである EM-C[20] を用いて実装を行った。逐次版から並列版へは、以下に示す変更が必要であった。EM-C ではリモートメモリを参照するグローバルポインタを利用できるため、カーネルループ部は 3. を除き変更が不要であった。すなわち、逐次のメモリアクセスと同様にワード単位のデータ転送をポインタ変数への代入という同一の記述で行なえる。また、共有メモリ上の index をグローバルアドレスに変換するオーバヘッドも、転送開始アドレスを示す各グローバルポインタを求めるときのみであり、データ数に対してほぼ無視できる程度に削減できる。

1. ローカルな要素数の数え上げのあとに、グローバルな操作により全体の要素数などを求める。
2. 共有メモリ上の index を分散メモリ上のアドレスに変換する。
3. データ転送アドレスの更新の際に、境界に達したかどうかのチェックを行う。境界に達していたら

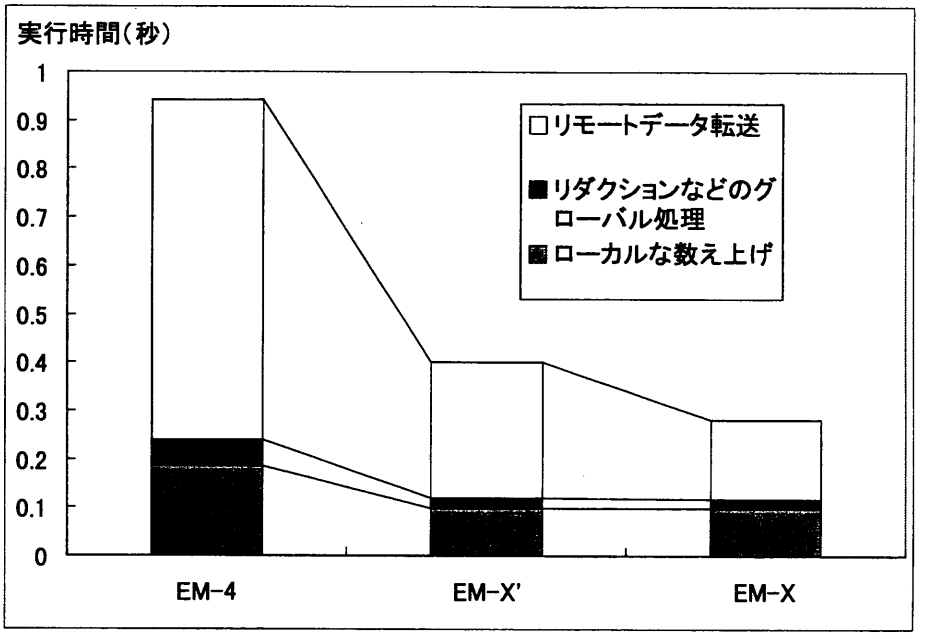


図 5.19: EM-X での並列 radix ソートの処理内容（64PE、2.5Mword）

転送アドレスを次の PE の先頭に設定する。

4. 転送終了の同期を行う。

また、EM-4 においてもほぼ同じプログラムがそのまま実行できる。ただし、搭載メモリサイズの違いなどにより最大データ量が EM-X の 1/8 と小さい点と、EM-X の特長である直接リモートメモリアクセス機構が EM-4 にはないため、パケットバッファが溢れないようにスレッドを明示的に中断することが必要である。

PE 台数 64 台で問題サイズを変化させて実行した結果が図 5.18 である。x 軸が総データ量、y 軸が 1 秒あたりのソートデータ量である。図には 64、256 の 2 種類の radix について EM-X、EM-4 の両計算機の結果を示すとともに、radix が 256 の場合の逐次処理性能から理想的台数効果を得られるとした場合の値を示している。全体のヒストグラムを生成する際のグローバル処理のオーバヘッドにより、EM-X、EM-4 ともに総データ量が 256K ワード以下では radix が 64 の場合が高速であるが、それ以上になると radix が 256 の場合が高速となる。また、EM-4 では逐次処理からの理想処理速度に対して実際の並列性能が半分程度であるのに対し、EM-X では 8 割以上の処理速度を実現している。これは EM-X の直接メモリ参照機構の効果である。この効果をより詳しく比較するため、各処理ごとの実行時間を比較する。

総データ数を 2.5M ワード、PE 台数を 64、radix を 256 としたときの EM-4 および EM-X の実行時間を図 5.19 に示す。全実行時間を各 PE 内でのローカルな数え上げ、全 PE での総和や部分和などのグローバル処理、リモートデータ転送に分けて示している。また、EM-X の持つ分散共有メモリ機構の評価のため、EM-X で EM-4 と同様にスレッド起動によるリモートメモリ書き込みを用いた場合を EM-X' として示した。EM-X' を EM-4 と比べると約 2.5 倍速い。クロックスピードの改良は 1.6 倍であるので、それ以外は命令セットアーキテクチャおよびメモリ参照機構の改良によるものである。メモリ参照機構の改良による効果としては、局所的な数え上げのループ実行があげられる。各ループは EM-4、EM-X ともに 10 命令からなるが、3 回のメモリ参照を含むため、EM-4 では 13 サイクルかかるのに対し、EM-X では 10 サイクルと 3 割ほど高速になっている。また、命令セットアーキテクチャの改良の効果としては、ブロック転送があげられる。総和を求める場合には配列に対するリダクション処理が行われており、このブロック転送性能が重要となる。EM-4、EM-X ともブロック転送機構を持っておらず、1 ワードの

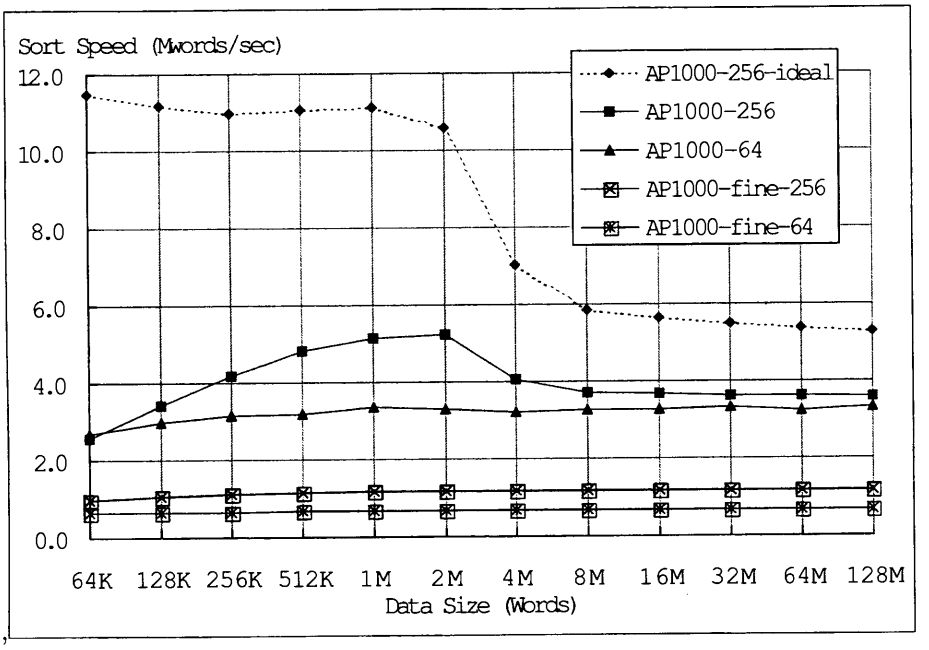


図 5.20: AP1000+での並列 radix ソート（32PE）

転送を繰り返すことが必要であり、ループアンローリングを用いて最適化を行ったライブラリを用意している。EM-4 ではメモリおよびパケットのアドレス更新を別途行うため、1 ワードにつき 5 クロック必要であるが、EM-X では自動更新型のメモリアドレッシングやレジスタ相対型のパケットアドレッシングにより 1 ワードにつき 2 クロックと 2.5 倍のスループットを実現した。これはネットワークのハードウェア転送性能と同じスループットである。

また、EM-X を EM-X' と比較すると、データ転送以外は等しく、データ転送では 1.7 倍の速度向上が見られる。これはリモートメモリ書き込みの処理がスレッド処理とオーバラップされるという直接リモートメモリ参照機構によるものである。オーバラップされるリモートメモリ書き込み自体の処理量は、高速化により削減された処理量の 2 割程度にすぎない。この直接のオーバラップ効果の他に、リモートメモリ書き込みのスケジューリングのためにスレッドを中断する必要がないため、スレッド環境待避のオーバヘッドが削減できる効果が大きい。命令コードの静的な解析によると、スレッド中断時には 4 変数の待避／復帰に 8 クロックサイクルが必要であったが、これが不要となり、カーネルループの実行が約 1.5 倍に高速化された。さらに、直接リモートメモリ参照機構の効果は性能向上ばかりではなく、パケットバッファの溢れをほぼ気にしなくてもよいことによりプログラミングの容易さにもつながっている。

AP1000+

AP1000+ではメッセージバッファを経由しない PUT/GET 機構を用いてユーザレベルで直接リモートメモリアクセスができる。そこで、EM-X と同様に、ワード単位のリモートメモリ書き込みを用いた場合 (fine) と、いったん転送データをまとめてからそのブロックをリモートメモリ書き込みで行った場合 (coarse) のそれぞれについて調べた。

ワード単位で転送する方式 (fine) では、PUT の DMA の終了をチェックすることによりコマンドキューに留まっているリクエストの最大個数の管理を行っている。これは PUT のリクエストを連続して与えるとコマンドキューが溢れて、コマンドのリストアのソフトウェア処理により遅くなってしまうためである。実験の結果では 7 個までのリクエストを先行受付けする方式が最適であった。

また、いったんデータをまとめてから転送する方式 (coarse) では、各 PE でのデータ転送がほぼいっ

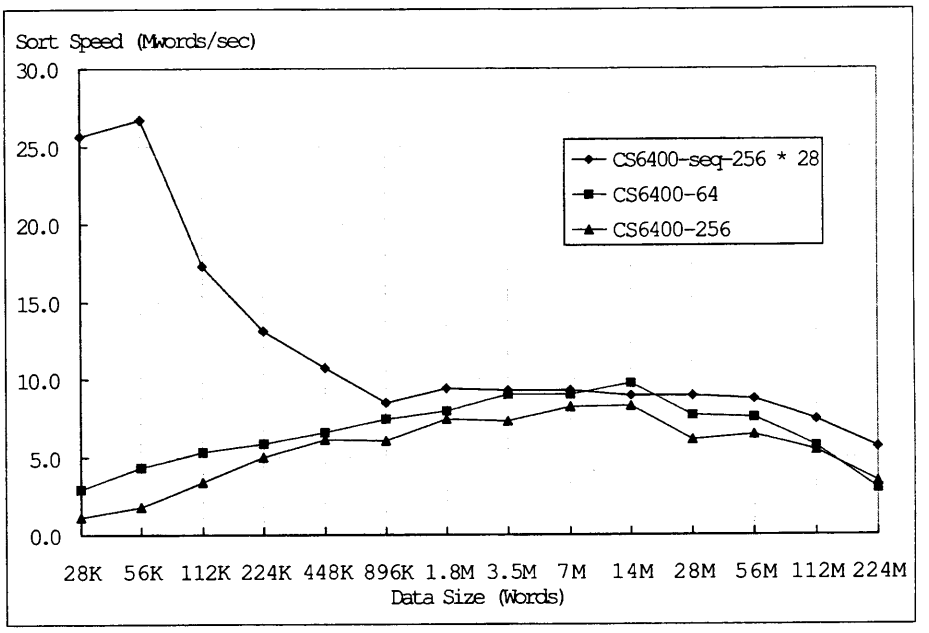


図 5.21: CS6400 での並列 radix ソート（28PE）

せいに始まるため、転送先の受信処理の衝突が起きないようデータ転送の順番をスケジューリングしている。一方、fine では元から転送先がランダムになっているのでその必要はない。

AP1000+では全PE での総和を求めるためのライブラリはサポートされているが、部分和を求めることはできず、また対象となるデータはスカラに限られてしまう。このため本プログラムでは通常のメッセージ通信を用いてバタフライネットワークを実現することにより配列データに対する総和および部分和の計算を行っている。これによりスカラデータに対する総和をデータ数分繰り返す方式に比べ radix が 256 の場合で 30 倍以上の高速化を実現した。

PE 台数 32 台で問題サイズを変化させて実行した結果が図 5.20 である。x 軸が総データ量、y 軸が 1 秒あたりのソートデータ量である。radix は 64 と 256 の場合を示すとともに、radix が 256 の場合の逐次処理性能からの理想的台数効果を得られるとした場合の値を示した。ワード単位の細粒度通信を行う方式 (fine) ではメッセージ生成処理は必要ないが、それ以上にデータ転送処理に時間がかかっている。これは AP1000+では、PUT/GET 機構を用いて直接リモートメモリアクセスが可能であるが、1 回の起動のオーバヘッドが 5.1 マイクロ秒 [Hay94] とワード単位の転送を行うには大きすぎるためである。細粒度通信を効率的に実行するには、送信側のレイテンシを削減する機構が必要となることが分かる。一方、いったんデータをまとめてから転送する方式 (coarce) では、グローバル処理のオーバヘッドを相対的に減らすためにデータ量を大きくしないといけないが、データ量を大きくすると TLB ミスの影響が現れて逆に実行時間が増大してしまう。また、逐次処理からの理想処理速度と比較して、たとえば、データサイズ 2M の場合、およそ半分の処理速度となっている。これについては、3.3 節でプロセッサ台数を変化させた場合のスケーラビリティとして詳しく述べる。

CS6400

CS6400 でのプログラムは Solaris の thread ライブラリを用いて並列化している。CS6400 の 2 次キャッシュは 2M バイトと大容量であるが、ダイレクトマップなので共有データを 2 次キャッシュ境界に合わせるとともに、2 次キャッシュでキャッシュラインの衝突が起きにくくなるよう、データ量を 2 の巾乗個より若干多くして実行している。また、グローバルヒストグラムを生成するためのリダクション処理は

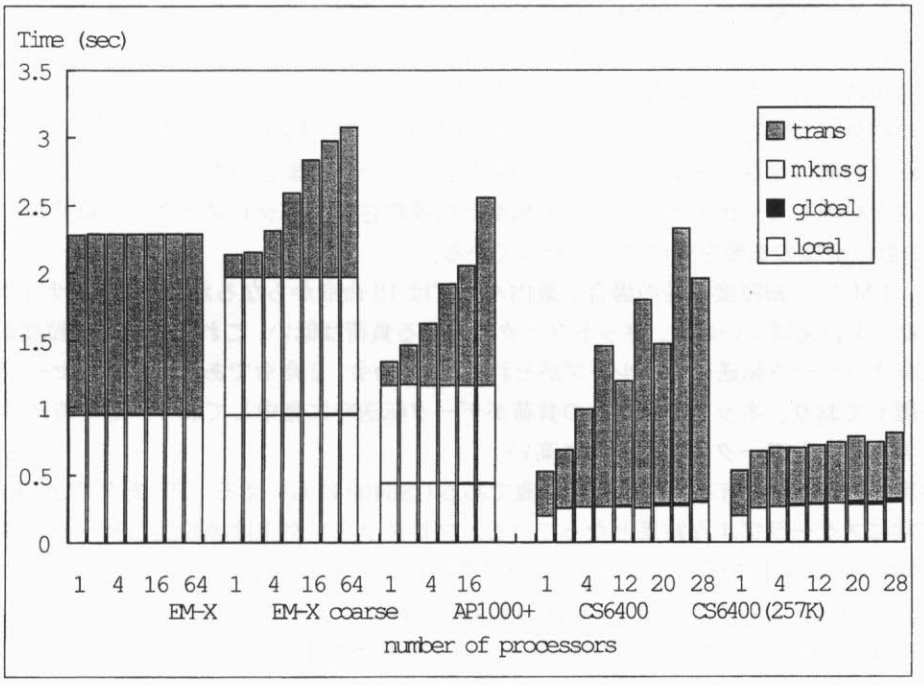


図 5.22: PE あたりのデータ量を一定（256K）にした場合

$O(N)$(N は PE 台数) のアルゴリズムを用いているためデータ数が少ない場合はその影響が見られるが、バリア処理を文献 [Cru91] に基づく共有メモリ向きの高速な処理を行うことにより、大きなデータ量に対してはグローバル処理は無視できるようにしている。

スレッド数を 28 とし、問題サイズを変化させて実行した結果を図 5.21 に示す。x 軸が総データ量、y 軸が 1 秒あたりのソートデータ量である。使用したマシンがサーバ機であり、他のユーザの影響を避けるため、32 プロセッサ中、使用するプロセッサを 28 プロセッサまでとし、10 回繰り返した中で 1 番良い結果を表示している。radix は 64 と 256 とし、比較のため radix が 256 の場合の逐次処理の結果をスレッド数 (28) 倍した値を示す。radix の違いによる差は、データ数が少ないときのグローバル処理のオーバヘッドの差を除き、小さい。また、逐次処理との比較では、プロセッサ台数が増えることにより 2 次キャッシュが増加する効果で、896K から 14M の範囲ではほぼ逐次処理と同等の実行効率を示しているが、ソートした結果を格納するデータに関しては 2 次キャッシュの衝突を引き起こしやすいため、スーパリニアの効果を示すところまではいかない。

PE 台数に対するスケーラビリティの評価

図 5.22 に PE あたりのデータ数を一定として、PE 台数を増加させた場合の実行時間とデータ転送時間を示す。TLB の影響を受けないよう radix は 64 とし、PE あたりのデータ量を 256K ワードとした。PE あたりのデータ量が一定なら、radix ソートの PE あたりの処理量およびデータ転送量はグローバル処理を除くと一定である。図 5.22 ではグローバル処理の実行時間はほとんど現れていない。また、EM-X では実行時間およびデータ転送時間ともに PE 台数によらず一定である。一方、AP1000+の場合は、データ転送以外は一定であるが、PE 台数が増加するとともにデータ転送時間が増大し、結果として実行時間が増加している。元々プロセッサどうしを直接接続する直接網ではバイセクションバンド幅がプロセッサ台数に比例しては増加しないため、全対全通信時にネットワークがボトルネックとなることが指摘されている。EM-X でも若干の性能の差はあるがほぼ同様のことがいえる。ところが AP1000+の場合のみにデータ転送時間の増大の現象が見られたのは、AP1000+の場合、データをいったんまとめてから転

送を開始するため、大量のデータ転送がほぼ同時に各 PE で開始されることにより、ネットワーク上での衝突を引き起こしているためであると考えられる。

このことを確認するため、EM-X でもいったんデータをまとめてから転送を行なう方式を実装し、実行した。その結果が EM-X coarse である。この場合 EM-X でも AP1000+ と同様に PE 台数の増加とともにデータ転送時間の増加が確認された。このように、データをまとめずに細粒度のままで転送を行う場合、データを処理しつつ出力するため出力頻度が均等化され、ネットワークへの負荷が均等化されるため、PE 台数の増加の影響を受けにくくなっている。

たとえば、EM-X の細粒度転送の場合、最内ループは 16 命令からなるが、この 16 サイクルごとに 1 パケットの転送を行えばよいので、ネットワークに対する負荷は低い。これに対し、粗粒度転送ではメッセージ生成およびデータ転送の最内ループがそれぞれ 13 命令、2 命令であるが、メッセージ生成とデータ転送が分離しており、ネットワークへの負荷がデータ転送時に集中しており、その時には 2 サイクルに 1 パケットとネットワーク負荷は非常に高い。

細粒度処理の効果は、共有バス型並列計算機である CS6400 にもいえる。PE あたり 257K ワードとした場合は非常にスケーラブルな結果となっている。プログラミング上は細粒度なデータアクセスを行う一方で、2 次キャッシュ上でデータのプリフェッチやブロックライトなどにより共有バスへの負荷が抑えられている。しかし、このようなキャッシュの効果はつねにうまくいくわけではない。CS6400 の場合、2 次キャッシュはダイレクトマップのため、キャッシュの衝突が生じる可能性が高く、図 5.22 によると、PE あたり 256K ワードとした場合にはデータ転送時間が 3 倍近くかかる場合があることが分かる。

第6章　考察

本章では、前章の細粒度通信に基づく並列計算機アーキテクチャの評価などを踏まえて、種々の観点から本アーキテクチャについて考察を行う。

6.1 通信性能と計算性能のバランス

EM-X では、メモリの読み出しと書き込みを一緒に行う xchg 命令等を除いて、単精度浮動小数点演算も含めて、全ての命令が 1 クロックサイクルで実行される。また、send 命令では 1 クロックサイクルで 32 ビットデータをネットワークへパケットとして出力することが可能である。そのため、EM-X では通信性能と計算性能がほぼ 1:1 という極めて通信性能が高い計算機となっている。実際には、ネットワーク上のパケット転送性能は 2 クロックサイクルで 1 パケットの転送が可能であるので、その意味では通信性能と計算性能は 1:2 と言えるかもしれないが、それでも依然計算性能に比べて通信性能は極めて高い。

プロセッサスピードが高速化したときに、この割合を維持できるのか、という問題がある。これは send 命令単独では可能である。send 命令は単純にレジスタオペランドと命令内埋め込みデータからパケットフォーマットを生成しているだけであり、他の演算性能と比べてクリティカルパスとなることはない。しかし、チップ間ネットワークとしては極めて困難であろう。まず第一に転送速度である。現在は EM-X はクロックスピードと同じクロックで転送を行っているが、プロセッサクロックが GHz クラスとなると、それをそのままネットワーククロックとすることは難しい。また、ネットワーク転送ビット幅の問題もある。現在 EM-X ではタグ部も含めて 39 ビット幅 (その他に要求、レディ等の制御線も含めると 44 ビット) で転送している。各ビット線ごとのスキュー調整などビット幅の広い通信路を持つことは、転送速度を高速化する上で非常にネックとなる。そのため、高速シリアルラインを用いることが基本的な流れになってきている。さらに、転送ビット幅がシリアル、あるいは数ビット程度になることを考えたとき、データのエラー検出/訂正等のために各ステップである程度のシリアル/パラレル変換が必要となる。このため、各ノードでデータをいったん全てを蓄えてから次のノードに送るストアアンドフォワード方式ではなく、バーチャルカットスルー方式により部分的に次ノードに送ることができるとしても、ネットワークの通信速度が相対的に低下する以上にレイテンシは大きくなる。

このように、今後はネットワークのスループットおよびレイテンシは計算性能に比べて相対的に低くなると考えられるが、そのような条件で、本提案のアーキテクチャは有効であろうか?現在はベンチマークによる評価のとおり、2 スレッド程度あれば多くのアプリケーションでレイテンシが隠蔽できるているが、EM-X ではより多くのスレッドを起動することを想定して、スレッドリソースのハードウェアによる確保を行っておらず、メモリ上のオペランドセグメント上に保持している。このため、アプリケーションにそれだけの並列度がありさえすれば今と同様の性能を維持できると考えられる。ここで重要なのは、スレッド切り替えおよび通信のセットアップにかかる時間が小さく保てるかということであるが、これは上でも述べた通り、命令実行のクリティカルパスとはならないため、今の性能を維持できる。ただ、命令実行性能と通信性能の差が大きくなるにつれて、パケット出力バッファのサイズを大きくすることが必要であったり、そのためにさらにレイテンシが大きくなることが予想される。

また、現在の計算/通信性能比では、少しでもプロトコル変換等の処理が通信に必要になると、それが通信性能の低下につながる。これは EM-X で MPI 通信ライブラリを実装した経験 [56] でも確かめられている。計算性能が高くなれば簡単なプロトコル程度であれば通信性能への影響は少なくなると考えられる。

6.2 粗粒度通信 vs 細粒度通信

EM-X では 1 ワードのパケット通信のみを命令レベルでサポートしている。EM-X のこの通信は、メモリ参照命令と非常によく似ている。多くの RISC プロセッサではメモリ参照はワード単位のみをサポートしている。連続的にメモリを参照する時にはソフトウェアでループを形成して実現している。このように良く使う命令のみを実装し、あとはそれを組み合わせて利用するというのが、RISC の考え方である。EM-X でもブロックデータの通信の場合は、ワードレベルの通信を組み合わせて実現している。また、サポートしている命令が他と効率よく組み合わせやすいということも重要である。例えば、命令参照の例では、ベースレジスタ相対アドレッシングという基本的な仕組みをサポートすることにより、毎回アドレスを別途計算しなくても、直接構造体の要素にアクセスしたりすることが可能である。EM-X の通信命令でもこのベースレジスタ相対アドレッシングをサポートしており、リモートメモリアクセスの効率的実現に役立っている。

一方、通常の計算機システムでも I/O とのやり取りなどでは DMA(ダイレクトメモリアクセス) 機構を別途持っていてバースト転送をハードウェアでサポートしている。これは I/O 等処理速度の大きく違う相手を待っていては全体の処理速度を低下させてしまうため、処理と通信とのオーバラップを図っているものである。これはプロセッサ間のバースト転送時にも当てはまる場合があると考えられる。計算処理と通信処理とがほぼ同程度であるような処理の場合、通信をプロセッサにより行う EM-X のような方式ではうまく計算と通信をオーバラップできない場合もある。今後は、細粒度通信だけではなく DMA を用いた粗粒度通信との融合を検討する必要もあるかもしれない。ただ、粗粒度通信のためにいったんバッファにデータをまとめ直してから通信を行う場合には、先の radix ソートの例にもあるとおり、ブロック転送の通信がネットワーク上で衝突を起こしやすくなり、速度の低下をもたらすこともある。そのような場合には、計算処理と通信処理を分けずに、計算処理中に細粒度通信を埋め込む形の方がネットワーク負荷の点からは望ましく、性能もスケーラブルなものになりやすい。全ての粗粒度通信がこのような形で融合できるわけではないので、両者のバランスが重要であろう。

6.3 局所同期のハードウェアサポート

EM-X では full/empty ビットに基づいて局所同期をサポートしている。このため、メモリのビット幅がデータサイズよりも余分に必要である。その他データタグもあるが、これは現在の EM-X プログラミングではほとんど使われていない。ただし、結果を返すパケットのパケットタイプを渡すのに、データ部のタグ部が利用されている。これに関しては、現在のパケットフォーマットではシステム全体のアドレスが 32 ビットに限られており、グローバルアドレスの拡大などパケットファーマットの改良は必要であるので、その中で検討すべきであろう。

Alewife[Aga95] では full/empty ビットをメモリ自体ではなく、共有メモリ用のディレクトリエントリに持たせている。また、*T[Nik92] ではこのような full/empty ビットに基づく局所同期ではなく、カウンタによる同期を用いている。ここで、どちらの方式にせよ、局所同期のための処理自体は数命令で可能であり、それ自体はそれほど性能に違いはないものと思われる。したがって、局所同期のためにチップ外のメモリに特別な仕様を要求する今の EM-X の方式は再検討が必要であろう。しかし、同期ミス時の処理がデータフロー計算機のように演算ユニットのバブルとなったり、Alewife のようにトラップを起こすようでは、局所同期を頻繁に利用すると性能ボトルネックとなってしまう危険性がある。EM-X や*T のように同期ミス時の処理をスレッド実行から隠蔽できるような仕組みが有効であると思われる。

6.4 スレッド切り替えオーバヘッド

EM-X ではスレッドの中断から、次のスレッドの起動までは、通常のスレッド起動で 1 サイクル、特殊パケットハンドラの場合 0 サイクルのオーバヘッドとなっている。しかし、これはレジスタの待避/復帰を含んでいない値である。EM-X ではコンパイラと協調して本当に必要なレジスタのみを待避/復帰するように最適化されており、数十から百命令を越えるようなスレッド長を持つようなリモート関数呼び出しにおいては、スレッド切り替えのオーバヘッドは十分許容できる範囲に収まっていると思われる。しかし、ループレベルでリモートメモリ読み出しを行う場合のような 10-20 命令程度のスレッドの切り替えの場合は、このレジスタ待避のオーバヘッドが逐次処理からのオーバヘッドとして残ってしまう。このため、並列処理のスケーラビリティ、すなわち並列プログラムを 1 プロセッサから n プロセッサに増加させた場合の台数効果は n に近いが、逐次プログラムからの並列効果はより低いものになってしまう。この差はデータサイズを大きくしても変化しない。

このようなオーバヘッドはレジスタセットを複数持つことにより解決できるが、何個のレジスタセットを持てばよいのか?ハードウェアコンテキストがあふれたときの処理をどうするか?スレッドとレジスタセットを結び付けることがオーバヘッドとならないか?ハードウェア規模はどれだけ大きくなるか?等、解決すべき問題点も多い。

このようなハードウェアによる改良以外にも、ソフトウェアによる改良も可能である。現在の方式でオーバヘッドが大きく見えてくるのは、上でも述べた通りループの最内周でのスレッド切り替えであることが多い。この場合、他のスレッドは走っていない (あるいは他のスレッドによってレジスタの値は変更されない) ことを何らかの方法により保証することにより、コンパイラにレジスタ待避/復帰を省くよう指定できれば、オーバヘッドを大きく削減することができる。しかし、これではマルチスレッドによるレイテンシ隠蔽が利用できない。さらに、最内ループを複数スレッドに分割し、それぞれのスレッドでレジスタを排他的に利用するようにスケジューリングできれば、レジスタ待避のオーバヘッドを削減しつつ、マルチスレッドによるレイテンシ隠蔽も可能になる。このレジスタ分割による方法では使えるレジスタ数が減るのでそれらのトレードオフを評価する必要がある。

ただし、現在のゲート規模を考えると、ハードウェアによるコンテキスト保持はそれほどハードウェア量のオーバヘッドとはならないともいえる。8 命令同時発行といった広い命令発行バンド幅を持つスーパスカラプロセッサでは、複数の演算ユニットやキャッシュ機構などと比べると、レジスタなどのハードウェアコンテキストは割合が小さい。また、同時マルチスレッド (Simulteneous multhithread, SMT)[Tul95] のように複数スレッド間で演算ユニットを共有して実行する方式の評価では、演算ユニットの利用効率が高まり処理性能が向上している。これはスレッド間で各種レイテンシを隠蔽しあっている効果と見ることもでき非常に興味深い。SMT については以下のスレッド実行で考察を行う。

6.5 メモリ階層

EM-X でクロックあたりの性能が高いのは、1 つにはメモリ性能が高いことによる。これは通常のロード命令等だけではなく、リモートメモリアクセスのような通信処理にも言える。EM-X では、ゲート規模やメモリ回路の簡素化の関係からキャッシュ機構を用いていない。また、メモリ関連の回路を簡単化するために、主メモリに SRAM を用いている。このため、メモリレイテンシ 0 でロード命令を実行している。このため、EM-X の結果は他のプロセッサではキャッシュが全てヒットしている状態と考えたほうが妥当かと思われる。

EM-X にキャッシュ機構を取り入れることは、命令フェッチやメモリ参照命令からキャッシュを参照する場合には、通常のプロセッサと同様であり、それほど問題はない。ただ、マルチスレッドによるメモリアクセスがキャッシュミス率にどのような影響を与えるかは、もう少し注意深く考察する必要はある

であろう。定性的に考えると、命令キャッシュに関してはほぼ同様のスレッドが複数実行される並列実行の時にはマルチスレッドの影響は少ないと考えられる。また、異なるプログラムを同時実行するマルチプログラム環境では、命令キャッシュの容量や way 数はプログラム数だけ必要であると考えられる。しかし、元々命令キャッシュのヒット率は高いことが想定されるので、影響は少ないと思われる。一方、データキャッシュについては、それぞれのスレッドで局所性を利用できるようにキャッシュの連想度や容量をシングルスレッドよりも大きくする必要があると考えられる。ただ、これらについてはまだ定性的な評価はあまり行われていない。

一方、SMT の評価 [Tul95] では、キャッシュをスレッド間で共有するか、占有するかの評価が行われている。8way の SMT で、命令キャッシュおよびデータキャッシュとして 8K バイトのキャッシュが 8 バンク (総計 64K バイト) あるときに、各スレッドが 8K バイトずつ占有するか全スレッドで共有するかを命令キャッシュ/データキャッシュそれぞれに適用して 4 種類の構成を比較している。ここではキャッシュを占有する方式では、8 スレッド以下のスレッドでは一部のキャッシュしか使用しないことになる。その評価では、データキャッシュでは全てのスレッド数でキャッシュを共有したほうがよいが、命令キャッシュは 8 スレッドの場合は占有したほうがよいという結果である。いずれにせよ、8 スレッドの時の各構成の差は 1%程度であった。

EM-X では、メモリアクセスはスレッド内だけではなく、パケットバッファリングやダイレクトメモリアクセス、直接待ち合わせ、スレッド起動等でも行われる。このようなアクセスでキャッシュミスが起きたときの処理ついては、注意が必要であろう。新情報処理開発機構で開発された RWC-1[Mat98] ではリモートメモリアクセスや待ち合わせ処理時のスレッド実行外のキャッシュミスを避けるため、メモリアクセスや待ち合わせ処理はスレッド実行内で行わせる方式を採用するとともに、これらの処理の実行スレッド待ちによるレイテンシ増大を避けるために、高優先度スレッドによるプリエンプティブ実行をサポートしている。しかし、最近ではノンブロッキングキャッシュが普通となってきているので、リクエストを保持しておくことにより、それと同様に実装できるのではないかとも考えられる。また、これらの特有のメモリアクセスパタンに応じたキャッシュ機構の最適化についても一部研究が行われている [81]。

6.6 スレッド実行

現在の商用プロセッサのクロック周波数速度の向上は目覚ましい。EM-X のような研究用並列計算機の場合、そのホスト計算機のプロセッサ速度を見ると、開発当時の一般的なプロセッサ速度を見積もることができる。EM-X のホスト計算機は、すでに述べた通り Sun Microsystems 社の Sparc Station 2 であり、これに使われているプロセッサは 40MHz の Sparc である。Sparc はシングル命令実行の RISC プロセッサである。これに対して、EM-X のクロック周波数は 20MHz(メモリアクセスサイクルは 40MHz) であり、十分比較対象となるものであった。もちろん、開発期間を考えて、開発終了時に十分比較対象となるよう性能を設定することが必要であるが、研究用並列計算機の開発では、開発期間が延びてしまい、その間に商用プロセッサの高速化が大きく進んでしまい、単純な評価が難しくなってしまうことがある。EM-X が開発完了した 1995 年には 200MHz の UltraSparc が発表されており、クロック周波数で 10 倍の差がついている。もちろん使用している ASIC のデザインルールの差 (1.0 ミクロン vs 0.5 ミクロン, およびゲートアレイ vs カスタムチップ) によるものが大きい。

EM-X のクロック周波数向上の可能性については、現在の構成はパイプライン段数が 2 段と少なく、また一部でメモリのリードモディファイライトを行っているため、そのままではクロック周波数を向上させることは困難であると思われる。しかし、このパイプライン構成は必須のものではなく、他の RISC プロセッサと同程度のパイプライン段数にして高速化することはそれほど困難無く可能であると思われる。特に、パケット生成部に関してはメモリアドレス部生成回路と同程度のゲート規模/遅延であり、容

易に高クロック化に対応できる。

スレッド実行のパイプラインを、スーパースカラ化することは、マルチスレッド実行とは独立した問題であり、適用することに問題はない。また、スレッド実行のコンテキスト (レジスタセットなど) をハードウェアとして持たせることも、現在のゲート規模を考えると十分可能であろう。その場合は、現在のようなシングルスレッド実行だけではなく、同時マルチスレッド (SMT) のように同時実行を行うことも検討の必要があろう。また、チップマルチプロセッサ (CMP) 化についても検討の必要がある。SMT の評価 [Tul95] によると、SMT のスレッド数と CMP のプロセッサ数を同じとすると、SMT の方が少ない演算回路数でより高い性能を示すことができると主張している。これは共有している演算回路を動的にスケジューリングできることによる利点である。設計の観点でも、設計容易性の点では CMP の方が有利であるが、設計の自由度、すなわち演算ユニット数や命令発行数、レジスタ数などをより細かく設定できるという点では SMT の方が有利である。SMT では、演算のレイテンシや分岐ミス、キャッシュミスレイテンシなどを他のスレッドの実行により隠蔽することが可能である。しかし、より大きなレイテンシを持つリモート通信のレイテンシをより効率的に隠蔽するためには、スレッドの生成/中断/同期等をうまくハードウェアでサポートすることが有効と考えられ、その際には本提案のような機構が有効ではないかと考えられる。

6.7 プロトタイプ開発/評価

提案するアーキテクチャを、実際にプロトタイプとして構築して評価することは、そのアーキテクチャの実現可能性を評価し、また、実用レベルでのアプリケーションを用いた評価を行う上で、重要なことである。しかし、実際の計算機を開発するためには、小さな研究グループで人的/金銭的/時間的リソースが限られているため、非常に難しい。特に、本提案のような、新しいプロセッサアーキテクチャを構築する場合、プロセッサの設計を行うことはもちろん、基板などシステムの開発、プログラムを動作させるための最低限の OS、プロセッサ向けコンパイラの開発、および各種ライブラリ等多種多様な開発が必要である。しかし、これらを一定の期間内に行わないと、提案したアーキテクチャで仮定した制約条件が変わってきて、完成したときに十分な評価が行えなくなってしまう。そのため、開発に当たっては、種々の制約条件のなかで、特定の部分に焦点を当てて開発を進めることが必要であった。本提案では、並列処理のサポート機能、特に通信のセットアップ時間の短縮やレイテンシの隠蔽に注力し、スレッドの逐次実行部分やメモリ機構については開発当時必要最低限と考えた、かなり簡略化したものとなっている。

ASIC 等によりプロセッサの設計を行い、プロトタイプを開発すると、そのシステムにおける評価を高速に、かつ大規模に行うことが可能になる。しかし、ハードウェアが固定化されてしまうため、そのシステムの一部を変更した場合のトレードオフを、そのシステムを用いて評価することは難しい。あらかじめ、パラメータ変更などを行えるようにハードウェア設計を行っておくことにより、ある程度は可能であるが、システム評価の間に新たなトレードオフの可能性が見つかる場合もあり、それらを全てシステム設計時に予見することは不可能である。このような場合には、プロトタイプ設計時と同様に、命令レベルやレジスタ転送レベル (RTL) 等各種シミュレータを用いた評価を行うこととなる。しかし、この種のシミュレータは動作速度が遅いため、実用的な時間でシミュレーションが終わるようにするために、問題規模を小さくしたり、システム規模を小さくしたりすることが必要になってしまい、十分な評価が行えないことが多い。

RTL あるいはゲートレベルのシミュレーションを高速化する方式として、FPGA などのプログラマブルデバイスを用いたエミュレータがある。近年の FPGA のゲート規模の拡大は目覚ましく、本稿で提案したようなシンプルなプロセッサの場合、複数のプロセッサを 1 チップの FPGA に登載できるほどになっている。このような FPGA デバイスを用いてプロトタイプを構築すると、動作周波数の点で

は、ASIC 等を用いて開発したプロトタイプには劣るが、シミュレータなどよりは格段に高速なシステムを構築することが可能である。しかし、一般に市販されている FPGA 評価用ボードでは、メモリ規模/ポート数などの点で単体プロセッサのエミュレーションとしても不十分であるのに加えて、それらを複数集めて大規模並列システムをエミュレーションすることは考えられていない。また、EDA ベンダーが出しているような汎用のエミュレータでは、ゲート規模やメモリポートなどはかなり柔軟に構成することが可能であるが、それを実現するためのコストの問題や、並列システムに特有の回路パターンの対称性、すなわち同一回路であるノードをネットワークで接続した形態、を生かしてエミュレーションの複雑度を軽減するようなことが難しいなどの点で不十分である。そのため、並列アーキテクチャ評価を目的とした新しい評価システム REX[85] を大規模 FPGA を用いて開発した。今後はこの評価システムを用いて、種々の並列アーキテクチャの評価を行っていきたい。

第7章　結論

本稿では、レイテンシの削減/隠蔽により並列性能を向上させる並列アーキテクチャを目的として研究を行い、その成果をまとめたものである。

第1章で、現在の並列処理における性能向上の問題点を挙げ、それを解決する方法として通信レイテンシの削減/隠蔽の重要性について述べた。また、これにより、並列処理の適用範囲を拡大できることを指摘した。第2章で、本研究の前後に行われたレイテンシの削減/隠蔽に関する他の研究をまとめた。特に、本研究の元となったデータ駆動計算機とフォンノイマン型計算機との初期の融合方式について述べ、両者の特徴を生かしつつハードウェアを簡略化し、メッセージ通信型および共有メモリ型の各種プログラミングモデルをサポートするアーキテクチャの必要性について述べた。第3章で、第2章で述べた要件を満たす並列アーキテクチャとして、細粒度通信機構に基づくマルチスレッドアーキテクチャを提案した。本提案アーキテクチャでは、データ駆動計算機とフォンノイマン計算機をパイプラインレベルで融合することにより高速なスレッド切り替えをサポートするとともに、システムで一意に定まるグローバルアドレスを用いた細粒度リモートメモリアクセスをハードウェアで実装し、効率的な並列処理を目指している。提案アーキテクチャに基づく実装について、実際に80プロセッサからなるプロトタイプ並列計算機の開発を行った。第4章で、本プロトタイプの実装についての詳細を述べた。第5章で、プロトタイプ並列計算機上で種々のベンチマークプログラムを実行し、その並列性能の評価を行い、本アーキテクチャが効率的な並列処理を実現していることを示した。第6章で、本アーキテクチャについて種々の側面から考察を行い、今後の課題について述べた。

提案したアーキテクチャは、send命令による固定長パケット生成機能を通信機能の基本として、それ以外はこれをソフトウェア的に組み合わせて実現している。本アーキテクチャでは、シンプルなハードウェアと、柔軟で最適化されたソフトウェアの協調により、小さい通信レイテンシと高いスループット性能の両方を実現できることを明らかにした。これは逐次処理におけるRISCの考え方と同等の考え方であり、これを並列処理に拡張したものと言える。提案アーキテクチャでは、通信機能だけではなく、スレッドスケジューリング、リモートメモリアクセス等についてもこの考え方を適用している。スレッド切り替え機能では、パケットによるスレッド起動というシンプルな動的スケジューリング機構のみをハードウェアで実現し、それ以外の各スレッドのリソース管理は、コンパイラによる静的解析に基づく最適化を施すことにより、スレッド切り替えオーバヘッドを低く抑えている。また、スレッド実行とは独立にサービスを行う直接リモートメモリアクセス機構と、full/emptyビットによる局所同期機構のみハードウェアによって実現し、それらをソフトウェアで自由に組み合わせることにより、リモートメモリアクセスの実行時レイテンシを軽減し、柔軟な共有メモリプログラミングを可能としている。

また、提案するアーキテクチャの有効性を実証するために、80プロセッサからなるEM-Xプロトタイプを実際に開発した。本プロトタイプは、1995年から現在まで稼働している。このように実際にプロトタイプを構築することは、予算や労力、開発期間など論文などの研究成果とならない部分で多大な努力を要する。しかし、実際にアーキテクチャの実用性等を評価する観点からは非常に重要なことであると言える。また、大規模なプロセッサ台数での大規模なアプリケーションにおける並列アーキテクチャの性能評価を行うためには、実機の開発は不可欠であると言える。

本論文の結論を述べる。従来、並列処理おいて性能向上を図るために通信粒度を大きくして通信スループットを向上させる手法が多かったが、本研究の成果によれば、シンプルであるが命令実行パイプラインと密接に融合した適切なハードウェアサポートにより、細粒度な通信のままでもそのレイテンシを削減/隠蔽することにより並列処理性能を向上させることが可能であることを示した。細粒度通信では、わざわざ通信を粗粒度にまとめることが不要であり、より細粒度な処理でも並列処理効果が期待されるため、並列処理の適用範囲を拡大することが可能となる。また、粗粒度通信では各プロセッサが一斉に通信を行うために通信の衝突による性能低下が引き起こされるが、細粒度通信では通信が平均的に散らばるためにネットワークの負荷が平均化され通信の衝突の影響が軽減されるという利点も見られた。

今後の課題としては、本アーキテクチャを提案/開発したときには、通信やスレッド切り替え等に焦点を当てたため、その他のメモリ回路やスレッド実行機構などをなるだけシンプルにして開発期間の短縮

を図った。そのために外付けメモリはSRAMとし、スレッド実行はシングル命令実行のRISCとしており、動作周波数も抑えたものであった。しかし、その後のプロセッサ技術の進歩は目覚ましく、クロックスピードは1GHzを越え、またスーパスカラ実行による命令レベル並列の利用もその限界近くまで達している。一方、プロセス技術の進歩はいくつかその限界を指摘されてはいるが、その困難を克服して今後もチップに登載できるゲート規模はますます拡大する方向にある。命令レベル並列を増やすことによる性能向上は限界が見えて来ているが、同時マルチスレッド (SMT) やチップマルチプロセッサ、メモリ混載等、拡大するゲート規模を利用するアーキテクチャ技術は今後も研究/開発が続きそうである。今後は、このような新しいプロセッサアーキテクチャにおいて、スレッドの生成/切り替え、同期などをサポートするために、本稿で提案したアーキテクチャが適用できないか、また、より改良するにはどうしたらよいか、等について検討を行っていきたい。

謝辞

本論文は、筆者が電子技術総合研究所 (現 (独) 産業技術総合研究所) にて行った高並列計算機アーキテクチャに関する研究をまとめたものである。

本研究は、プロトタイプ計算機を開発しての実証的研究であり、実機を稼働させるまでには多くの論文とはならないような努力の積み重ねが必要であった。その意味でも、本研究を一緒に遂行した研究グループの方々には深く感謝致します。特に、本研究をまとめていただいた山口喜教 元計算機方式研究室長 (現筑波大学教授) には感謝します。本論文で提案したアーキテクチャの基盤となる並列計算機 EM-4 の設計者でもあり、筆者が電総研に入所して以来アーキテクチャ研究についてご指導頂いた坂井修一氏 (現東京大学教授) に感謝します。本プロトタイプ上のコンパイラ環境などを構築してくださり、またソフトウェア含むトータルなシステムとしての並列計算機について示唆に富む助言を頂いた佐藤三久氏 (現筑波大学教授) に感謝します。本プロトタイプの開発において、チップのテストベクタ作成やプロセッサ基盤の調整など、実機開発の本当に泥臭い部分を担当してくれた坂根広史氏に感謝します。その他、山名早人氏 (現早稲田大学助教授)、小池汎平氏にはミーティングなどでの活発な議論に参加していただき、貴重な意見を頂きました。また、本プロトタイプ開発には三洋電機の清水雅久氏、甲村康人氏にご協力頂きました。また、本プロトタイプを実際に使用し貴重な意見を頂いた Andrew Sohn 氏 (ニュージャージ工科大学)、八杉昌宏氏 (京都大学)、建部修見他のみなさまに感謝します。

本研究の遂行には、この他多くの人のご支援を頂いた。中でも、弓場敏嗣 前情報アーキテクチャ部長 (現電気通信大学教授)、島田俊夫 前計算機方式室長 (現名古屋大学教授)、大蒔和仁 独立行政法人産業技術総合研究所情報処理研究部門長に感謝します。また、本論文を執筆するにあたりご配慮頂いた、関口智嗣 産総研グリッド研究センター長、および工藤知宏 同センタークラスタ技術チーム長に感謝します。

また、東京大学大学院工学系研究科 田中英彦教授には、本論文をまとめるにあたり貴重なご指導とご助言を頂きました。また、同研究科 近山隆教授、喜連川優教授、相田仁教授、坂井修一教授には本論文の完成に有益なご指示やご助言を頂きました。

最後に、本論文の執筆を陰ながら応援してくれた妻 久子、娘 由紀子に感謝します。

付 録A　インタフェースボード上のレジスタアドレス一覧

制御レジスタ **ctrlReg (0x80:write)** システム全体を制御する設定を行うレジスタ。現在はクロック周波数の制御のみを行う。

bit1-0 システムクロック周波数選択。0: 20MHz, 1: 10MHz, 2: 5MHz, 3: 2.5MHz となる。

bit3-2 メンテナンスクロック周波数選択。0: 20MHz, 1: 10MHz, 2: 5MHz, 3: 2.5MHz となる。

それ以外 未使用

ステータスレジスタ **statusReg (0x80:read)** システム全体を状態を示すレジスタ。インタフェースボード上の LED にも表示されている。

bit3-0: ctrlReg ctrlReg の内容が読み出される。

bit8: trap いずれかのプロセッサが TRAP 信号をアサートしているときに 1 となる。

bit10: clockHalt プロセッサへのクロック供給が停止しているときに 1 となる。

bit11: pktInBufErr ホスト計算機から EXT-PE へのパケットバッファが空で無い時に上書きした場合に 1 となる。

bit12: pktInBufFull ホスト計算機から EXT-PE へのパケットバッファにパケットが存在するときに 1 となる。

bit13: pktOutBufErr EXT-PE からホスト計算機へのパケット FIFO が空の時に読み出した場合に 1 となる。

bit14: pktOutBufFull EXT-PE からホスト計算機へのパケット FIFO が空でない場合に 1 となる。個数に関しては以下の pktOutBufCount を参照。

bit15: powerOn EXT-PE からホスト計算機へのパケット FIFO が空でない場合に 1 となる。個数に関しては以下の pktOutBufCount を参照。

それ以外 未使用

クロック連続供給トリガ **clockOn (0x00:write)** 書き込みのタイミングでプロセッサへのクロック供給を開始する。

クロック定数供給トリガ **clockStep (0x01:write)** クロック供給数レジスタ (clockNReg) の設定に従い、クロックの供給を開始する。

クロック停止トリガ **clockOff (0x02:write)** プロセッサへのクロック供給を停止する。連続供給、定数供給いずれの方法で開始してもこのトリガを叩くことでクロック供給は停止する。

クロック供給数レジスタ **clockNReg (0x03:write)** 16bit レジスタで、定数クロック供給時のクロック数を設定する。一度設定すれば、クロック定数供給トリガを叩くだけで、同じ数のクロックを何回でも供給させることができる。設定すべき値は供給したいクロック数の 2 の補数。0 を設定すると 65536 を意味する。

クロックカウンタ **clockCount (0x03:read)** 16bit カウンタ。現在のクロックカウントの値が読み出せる。上位 16bit は不定。読み出した値が 0 ならば、供給数レジスタに設定した値 (の 2 の補数) だけクロックが供給されて止まったことになる。

メンテナンスバス制御ベクトル **maintCtrlVec[0x1000** (0x10000:read/write)] maintCtrlVec[i] への読み書きは

```
GA = (i >> 8) & 0xf,
CA = (i >> 12) & 7,
MA = i & 0xff,
EXTPE = i >> 15.
```

への読み書きとなる。ここで GA は PE のグループアドレス、CA はグループ内アドレス、MA はメンテナンスアドレス。ただし EXTPE が 1 のときは EXT-PE に対するアクセスとなる。ライト時には、CA=7 のとき、グループ内へのブロードキャスト、EXT-PE=1 かつ CA=7 のとき全 PE へのブロードキャストとなる。データはメンテナンスバス上を 4bit づつ転送されるが、このとき LSB を含むニブルが最初に転送される。

パケット投入レジスタ pktInBufReg[8 (0x20:write)] パケット投入バッファには 1 パケット分 3 語のレジスタが存在する。これらを a_part, d_part, hi_part と呼ぶことにする。このレジスタより出力されるパケットは

```
packet.a_part[38:0] = hi_part[22:16] # a_part
packet.d_part[38:0] = hi_part[6:0]   # d_part
```

となる。ここで # はビット列の連結を表す演算子とする。

pktInBufReg[i] への書き込みは、

```
trig = i >> 2,
mode = i % 4.
```

とするとき、mode = 0 ならば a_part に、mode = 1 ならば d_part に、mode = 2 ならば hi_part にデータが設定される。さらに、trig = 0 ならば上記のレジスタへの値の設定を行なうだけであるが、trig = 1 ならばレジスタへの設定とともにパケット投入を指示する。

以前に出力を指示したパケットが実際に EXT-PE に渡されていない (pktInBufFull = 1 である) のに、pktInBufReg への書き込みを行なうとエラー状態となる (pktInBufErr = 1 となる)。

パケット投入ベクトル pktInBufVec[0x100000 (0x100000:write)] pktInBufVec[i] へのデータ d の書き込みは、

```
packet.a_part[38:0] = hi_part[22:16]#a_part[31:22]#i[19:0]#a_part[1:0]
packet.d_part[38:0] = hi_part[6:0]#d[31:0]
```

なるパケット投入を指示する。

以前に出力を指示したパケットが実際に EXT-PE に渡されていない (pktInBufFull = 1 である) のに、pktInBufVec への書き込みを行なうとエラー状態となる (pktInBufErr = 1 となる)。

パケット取出レジスタ pktOutBufReg[8 (0x40:read)] パケット取出バッファには 1024 パケット分の FIFO と、1 パケット分 3 語のレジスタが存在する。この 3 語のレジスタを a_part, d_part, hi_part と呼ぶことにする。このレジスタと、パケットイメージとの関係は、pktInBufReg のときと同様である。hi_part のうち、パケットの bit と対応付けられていない bit については不定となる。

pktOutBufReg[i] の読み出しは、

```
trig = i >> 2,
mode = i % 4.
```

とするとき、mode = 0 ならば a_part、mode = 1 ならば d_part、mode = 2 ならば hi_part が読み出される。さらに trig = 0 ならば上記のレジスタの値が読み出されるだけであるが、trig = 1 ならばこれらレジスタに格納されたパケットが捨てられ、FIFO に存在するパケット (がもしあれば) の先頭のものがこのレジスタに格納される。

パケット取出 FIFO カウンタ pktOutBufCount (0x43: read) 11bit のカウンタで、FIFO 及びパケット取出レジスタに存在するパケットの総数を示す。上位 21bit は不定。

付 録B　特殊パケットハンドラおよび例外ハンドラ一覧

特殊パケットハンドラ、および例外ハンドラの一覧。

HOST ptype=0x01 (スタートアドレス:0x0100) ホスト計算機からシステムコール呼び出しで、起動すると直ちにシステムコール用のトラップハンドラを起動する。データ部のタグでシステムコールの種類を指定するとともに、データ部でそれへの引数を渡す。???

NULL ptype=0x02 (スタートアドレス:0x0200) fork 等で返り値が必要ない場合の結果パケットをこのパケットタイプとすることにより、関数自体には変更が不要となる。このハンドラは何もせずに終了するだけである。

GETSEG ptype=0x03 (スタートアドレス:0x0300) リモート関数呼び出しのための新たな関数フレーム (オペランドセグメント) の取得を要求する。オペランドセグメントはフリーリスト管理されており、その先頭は汎用レジスタ ftop(r25) に格納されている。ハンドラの処理としては、以下の通り。4 クロックサイクルかかる。また、フリーリストが空 (ftop が 0) の場合は、テンプレートトップを書き込む際にメモリアドレス例外が発生し、トラップハンドラ内でオペランドセグメントのフリーリストへのアクセスであることを検出すると、動的にオペランドセグメントを確保して処理を継続する。ヒープ領域が本当に無くなると、エラーとなる。

- ftop の指すメモリには、次のオペランドセグメントへのポインタが格納されているので、それ読み出す。
- ftop の指すメモリに、リモート呼び出しを行う関数へのポインタ (テンプレートトップ, パケットアドレス部により指定) を書き込む。
- ftop をパケットデータ部で示される戻りアドレスに送信する。
- 保持していたフリーリストの先頭の内容を新たな ftop とする。

USRWR ptype=0x04 (スタートアドレス:0x0400) スレッドによるリモートメモリ書き込みを行う。パケットアドレス部でメモリアドレス、データ部で書き込むデータを指定する。ハンドラは st fp,0,pr0 の 1 命令からなる。

USRRD ptype=0x05 (スタートアドレス:0x0500) スレッドによるリモートメモリ読み出しを行う。パケットアドレス部でメモリアドレス、データ部で読み出したデータを返すアドレスを指定する。ハンドラ ld fp,0,r0; send2 r0,pr0,0,0; の 2 命令からなる。

IST ptype=0x06 (スタートアドレス:0x0600) I-structure を実行する。実際の同期は、直接待ち合わせによりハードウェアで実行され、本ハンドラでは同期完了後に I_read へ結果を返す部分のみを担当する。ハンドラは send2 pr1,pr0,0,0; の 1 命令からなる。

QST ptype=0x07 (スタートアドレス:0x0700) Q-structure を実行する。本ハンドラでは入力と出力がそれぞれ 1 づつで対応できた場合、すなわち I-structure と同様な同期が行えた場合のみ起動される。ハンドラは send2 pr1,pr0,0,0; の 1 命令からなる。同期は直接待ち合わせによりハードウェアでチェックされ、入力や出力が連続したときのキューイングはマッチングエラーハンドラでソフトウェア的に処理される。

INCR ptype=0x08 (スタートアドレス:0x0800) パケットのアドレス部で指定されるメモリアドレスに、データ部で指定されるデータを加える処理を行う。3 命令からなる。

WRB ptype=0x0a (スタートアドレス:0x0a00) パケットのアドレス部で指定されるメモリアドレスの、データタグ部で指定されるバイト位置に、データ部で指定されるバイトデータを書き込む。4 命令からなる。

MAX ptype=0x0e (スタートアドレス:0x0e00) パケットのアドレス部で指定されるメモリアドレスの内容が、データ部で指定されるデータよりも値が小さい場合のみ、データ部のデータをそのメモリに書き込む。4 命令からなる。

INCRF ptype=0x0f (スタートアドレス:0x0f00) INCR の float 型版。

H_GETSEG ptype=0x23 (スタートアドレス:0x2300) GETSEG の高優先度パケット版

SYSWR ptype=0x24 (ハードワイヤード) ダイレクトリモートメモリ書き込み。パケットのアドレス部で指定したメモリに、データ部で指定したデータを書き込む。全てハードウェアで処理されるため、このハンドラが起動されることはない。

SYSRD ptype=0x25 (ハードワイヤード, スタートアドレス:0x2500) ダイレクトリモートメモリ読み出し。パケットのアドレス部で指定したメモリを読み出し、データ部で指定したアドレスにそのデータを送信する。通常はハードウェアで処理されるため、このハンドラが起動されることはないが、OBU がフルの時には、デッドロックを回避するためにリクエストパケットが FIFO に入れられ、本ハンドラが起動される。ハンドラは USRRD と同じ 2 命令からなる。

SYSIST ptype=0x26 (スタートアドレス:0x2600) IST の高優先度版。

SYSQST ptype=0x27 (スタートアドレス:0x2700) QST の高優先度版。

SYSINC ptype=0x28 (スタートアドレス:0x2800) INC の高優先度版。

H_RD ptype=0x29 (スタートアドレス:0x2900) 常にスレッド起動を行う SYSRD, あるいは USRRD の高優先度版。

SYSWRB ptype=0x2a (スタートアドレス:0x2a00) WRB の高優先度版。

DISTI ptype=0x2b (スタートアドレス:0x2b00) パイプライン的にデータをブロードキャストするためのハンドラ。パケットのアドレス部でデータを書き込むアドレスを、データ部で書き込むデータを指定する。以下の 4 命令からなる。まず自分のメモリにデータを書き込んだあと、次のプロセッサのアドレスを計算してデータを転送する。あらかじめ次に送るプロセッサ番号と自分のプロセッサ番号との差を NEXT0diff に格納しておく。この時、パイプラインの最後に転送を行うプロセッサではこのデータのタグを SYSWR に、それ以外では DISTI にしておくことにより分岐処理を省くことができる。これにより転送スループットは 20M バイト/秒とピークの半分でブロードキャストができる。

```
DISTI:
        st      fp,0,pr0
        ld      zr,NEXT0diff,r0
        add     r0,fp,r0
        send2   pr0,r0,0,0
       .break
NEXT0diff:
.word 0
```

SYSMAX ptype=0x2e (スタートアドレス:0x2e00) MAX の高優先度版。

RH_RD ptype=0x2f (スタートアドレス:0x2f00) メモリ読み出しを行うスレッド。H_RD と同様であるが、読み出したデータを返送する際のパケットも高優先度に設定する。

ILPA ptype=0x3f (ハードワイヤード, スタートアドレス:0x3f00) SYSWR の書き込みアドレスや通常のパケットのオペランドセグメント等でアドレス例外が生じたときに起動されるスレッド。現在は単にエラーをホスト計算機に返すだけ。

MTERLFT trap=0x00 (ハードワイヤード, スタートアドレス:0x4000) 左マッチングエラー時に起動される例外ハンドラ。直接待ち合わせ時に、左オペランドがすでに待ち合わせメモリに存在するときに、左オペランドが入力されたときのエラー。パケットタイプが QST あるいは SYSQST の時には、キューイング処理を行い、それ以外の時はエラー処理 (ホスト計算機への通知) を行う。

MTERRGT trap=0x01 (ハードワイヤード, スタートアドレス:0x4200) 右マッチングエラー時に起動される例外ハンドラ。直接待ち合わせ時に、右オペランドがすでに待ち合わせメモリに存在するときに、右オペランドが入力されたときのエラー。パケットタイプが QST あるいは SYSQST の時には、キューイング処理を行い、それ以外の時はエラー処理 (ホスト計算機への通知) を行う。

IBUOVF trap=0x02 (ハードワイヤード, スタートアドレス:0x4400) IBU のメモリバッファがオーバフローしたときに起動されるエラー。ホストに通知を行う。

MAERR trap=0x03 (ハードワイヤード, スタートアドレス:0x4600) メモリアドレス例外検出時に起動される例外ハンドラ。メンテナンスアドレスの MAEON(0x62[1]) により例外検出時にトラップを起こすかどうかを指定できる。このフラグが 1 の時に、IBU のメモリバッファ領域を読み書きしたり、メンテナンスアドレス MEMPRO(0x63) で指定したアドレスよりも下位のメモリに書き込みを行おうとすると、メモリアドレス例外を起こす。

PRTERR trap=0x04 (ハードワイヤード, スタートアドレス:0x4800) パリティエラー検出時に起動される例外ハンドラ。メンテナンスアドレスの PREON(0x62[2]) により例外検出時にトラップを起こすかどうかを指定できる。このフラグが 1 の時に、メモリパリティエラーを検出すると、パリティエラー例外を起こす。

TRPIDL trap=0x05 (ハードワイヤード, スタートアドレス:0x4a00) スレッドが起動されていないときに、各種例外が発生した場合の戻りアドレス。

INTEN trap=0x06 (スタートアドレス:0x4c00) 例外が起きると、以降の例外を禁止するが、それと同時にパケットの取り込みなども中止する。そのため、システムコールルーチンでパケット取り込みを許可するためには例外を許可する必要がある。本例外を発生すると、例外を許可する。

SYSENT trap=0x06 (スタートアドレス:0x4e00) ホスト計算機からのシステムコール起動のための HOST パケットにより起動されるシステムコールのための例外ルーチン。

FDIV trap=0x0d (スタートアドレス:0x5a00) float 型の割り算を行うための例外ルーチン。被除数を pr0, 除数を pr1 に格納してトラップを起動すると、pr0 に結果が返る。まず仮数部から 64 個のテーブルを参照して初期値とし、ニュートン法を 4 回繰り返して精度を確保する。32 クロックサイクル程度。

DMPREG trap=0x0e (スタートアドレス:0x5c00) レジスタの内容をホスト計算機に返す例外ルーチン。

CMPD trap=0x0f (スタートアドレス:0x5e00) double 型の比較を行う例外ルーチン。

CVTDF trap=0x10 (スタートアドレス:0x6000) double 型を float 型に変換する例外ルーチン。

CVTFD trap=0x11 (スタートアドレス:0x6200) float 型を double 型に変換する例外ルーチン。

CVTDI trap=0x12 (スタートアドレス:0x6400) double 型を int 型に変換する例外ルーチン。

CVTID trap=0x13 (スタートアドレス:0x6600) int 型を double 型に変換する例外ルーチン。

DNEG trap=0x14 (スタートアドレス:0x6800) double 型の符合反転を行う例外ルーチン。

DADD trap=0x15 (スタートアドレス:0x6a00) double 型の加算を行う例外ルーチン。

DSUB trap=0x16 (スタートアドレス:0x6c00) double 型の減算を行う例外ルーチン。

DMUL trap=0x17 (スタートアドレス:0x6e00) double 型の乗算を行う例外ルーチン。

DDIV trap=0x18 (スタートアドレス:0x7000) double 型の除算を行う例外ルーチン。

UMOD trap=0x19 (スタートアドレス:0x7200) unsigned int 型の剰余を求める例外ルーチン。

IMOD trap=0x1a (スタートアドレス:0x7400) int 型の剰余を求める例外ルーチン。

UDIV trap=0x1b (スタートアドレス:0x7600) unsigned int 型の除算を行う例外ルーチン。

IDIV trap=0x1c (スタートアドレス:0x7800) int 型の除算を行う例外ルーチン。

BRKPT trap=0x1d (スタートアドレス:0x7a00) ブレークポイント処理を行う例外ルーチン。

SYSCALL trap=0x1e (スタートアドレス:0x7c00) システムコール呼び出しを行う例外ルーチン。

HALT trap=0x1f (スタートアドレス:0x7e00) HALT パケットをホスト計算機に返す。

付 録C　命令一覧

図 C.1 に EM-X の命令フォーマットを、図 C.2 に命令オペコードの一覧をマトリックス形式で示す。ここで、DIV, SHS, FLT は aux 領域で各個別の命令を指定する。また、空欄は未使用を表わす。各命令の一覧は以下の通り。

整数演算命令

add add integer

アセンブラ　$add\ r_{sr0}, r_{sr1}, r_{dst}$
　　$add\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x00

add 命令は、sr0 に sr1 あるいは imm17 を符号拡張した値を加え、結果を dst に格納する。結果により符合付き整数の場合のオーバフローフラグ (V) および符合なし整数の場合のキャリーフラグ (C) をそれぞれ変更する。

sub subtract integer

アセンブラ　$sub\ r_{sr0}, r_{sr1}, r_{dst}$
　　$sub\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x01

sub 命令は、sr0 から sr1 あるいは imm17 を符合拡張した値を引いて、結果を dst に格納する。結果によりフラグ (V, C) を変更する。

addc add integer with carry

アセンブラ　$addc\ r_{sr0}, r_{sr1}, r_{dst}$
　　$addc\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x02

addc 命令は、sr0 に sr1 あるいは imm17 を符合拡張した値、およびキャリーフラグを加えて、結果を dst に格納する。結果によりフラグ (V, C) を変更する。

subb subtract integer with borrow

アセンブラ　$subb\ r_{sr0}, r_{sr1}, r_{dst}$
　　$subb\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x03

subb 命令は、sr0 から sr1 あるいは imm17 を符合拡張した値、およびキャリーフラグを引いて、結果を dst に格納する。結果によりフラグ (V, C) を変更する。

and bitwise and

アセンブラ　$and\ r_{sr0}, r_{sr1}, r_{dst}$
　　$and\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x04

and 命令は、sr0 と sr1 あるいは imm17 を符号拡張した値とのビット毎の論理積をとり、結果を dst に格納する。

or bitwise or

アセンブラ　$or\ r_{sr0}, r_{sr1}, r_{dst}$
　　$or\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x05

or 命令は、sr0 と sr1 あるいは imm17 を符号拡張した値の論理和をとり、結果を dst に格納する。

xor bitwise exclusive or

アセンブラ　$xor\ r_{sr0}, r_{sr1}, r_{dst}$
　　$xor\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x06

xor 命令は、sr0 と sr1 あるいは imm17 を符号拡張した値の排他的論理和をとり、結果を dst に格納する。

xnr bitwise exclusive nor

アセンブラ　$xnr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$xnr\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x07

xnr 命令は、sr0 と sr1 あるいは imm17 を符号拡張した値の排他的論理和の否定をとり、結果を dst に格納する。片方のオペランドとしてゼロレジスタ (ZERO(R31)) を指定すればビット毎の反転を計算できる。

シフト演算命令

lsl logical shift left

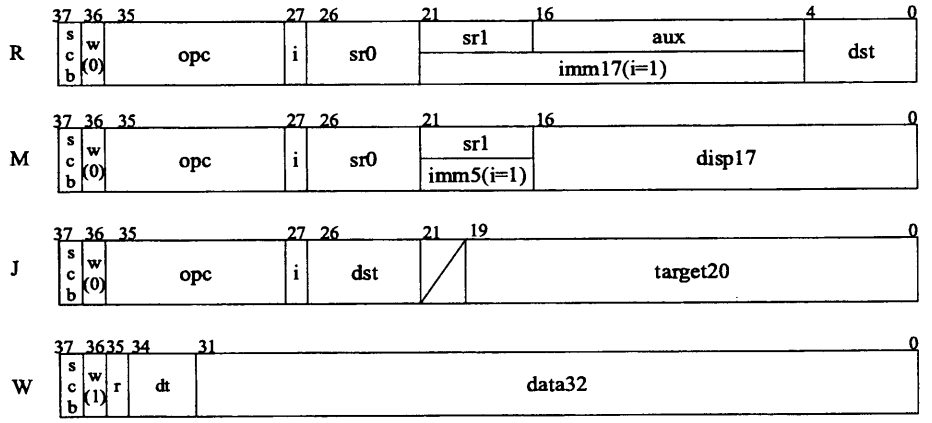


図 C.1: 命令フォーマット

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x00	add	sub	addc	subb	and	or	xor	xnr	DIV			alutst				
0x10	lsl	lsr	lsrr	lslr	asl	asr	rol	SHS			mul		FLT	mulf		maaf
0x20	ldr	ld	st	xchg	lrr	lr	sr	xchgr	ldr.a	ld.a	st.a	xchg.a	lrr.a	lr.a	sr.a	xchgr.a
0x30			enq	deq			enqr	deqr			stb				sts	
0x40	beq	beq.a	bne	bne.a	bra	bra.a	bne	bne.a	bgt	bgt.a	ble	ble.a	blt	blt.a	bge	bge a
0x50	bgtu	bgtu.a	gleu	gleu.a	bltu	bltu.a	bgeu	bgeu.a	bgtf	bgtf.a	blef	blef.a	bltf	bltf.a	bgef	bgef.a
0x60	btst	btst.a	bntst	bntst.a	btt	btt.a	bntt	bntt.a	bc	bc.a	bnc	bnc.a	bv	bv.a	bnv	bnv.a
0x70													jl	jlr	strap	reti
0x80	lpa0	lpa1	lpa2	lpa3												
0x90	lddt	anddt	stdt	chgdt	setmt	ldmt	ldsr		extsb	extb	insb		extss	exts	inss	
0xa0																
0xb0																
0xc0	send0	send1	send2	send3	senda0	senda1	senda2	senda3								
0xd0	addp	subp1	addcp	subbp	andp	orp	xorp	xnrp								
0xe0	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3
0xf0	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3	sendc0	sendc1	sendc2	sendc3

図 C.2: オペコード一覧

アセンブラ　$lsl\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lsl\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x10

lsl 命令は、sr0 を sr1 あるいは imm17 の下位 6 ビットで指定されるシフトカウンタだけ論理左シフトし、結果を dst に格納する。シフトカウンタは下位 6 ビットを符合付き整数 (ただし-31 から 31) として扱い、プラスの時には左シフト、マイナスの時には右シフトする。それぞれシフトで空になったビット位置には 0 が挿入される。

lsr logical shift right

アセンブラ　$lsr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lsr\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x11

lsr 命令は、sr0 を sr1 あるいは imm17 だけ論理右シフトし、結果を dst に格納する。その他詳細は lsl 命令を参照。

lsrr logical shift right reverse

アセンブラ　$lsrr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lsrr\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x12

lsrr 命令は、lsr 命令でシフトアウトされる部分を、dst に格納する。これは 32-r[sr1] ビットだけ lsl シフトすることと同じである。その他詳細は lsl 命令を参照。

lslr logical shift left reverse

アセンブラ　$lslr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lslr\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x13

lslr 命令は、lsl 命令でシフトアウトされる部分を、dst に格納する。これは 32-r[sr1] ビットだけ lsr シフトすることと同じである。その他詳細は lsl 命令を参照。

asl arithmetic shift left

アセンブラ　$asl\ r_{sr0}, r_{sr1}, r_{dst}$
　　$asl\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x14

asl 命令は、sr0 を sr1 あるいは imm17 だけ算術左シフトし、結果を dst に格納する。シフトカウンタが負の時には、シフトで空になったビット位置に最上位ビットの値が挿入される。その他詳細は lsl 命令を参照。

asr arithmetic shift right

アセンブラ　$asr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$asr\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x15

asr 命令は、sr0 を sr1 あるいは imm17 だけ算術右シフトし、結果を dst に格納する。シフトカウンタが正の時には、シフトで空になったビット位置に最上位ビットの値が挿入される。その他詳細は lsl 命令を参照。

rol rotate left

アセンブラ　$rol\ r_{sr0}, r_{sr1}, r_{dst}$
　　$rol\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x16

rot 命令は、sr0 を sr1 あるいは imm17 だけ左回転し、結果を dst に格納する。その他詳細は lsl 命令を参照。

shlc shift left with carry

アセンブラ　$shlc\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x17 (aux:0x00)

shlc 命令は、sr0 で指定されるレジスタの内容を 1 ビット左にシフトし、最下位ビットにキャリーフラグの値を挿入する。結果は dst で指定されるレジスタに格納される。

shrc shift right with carry

アセンブラ　$shrc\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x17 (aux:0x01)

shrc 命令は、sr0 で指定されるレジスタの内容を 1 ビット右にシフトし、最上位ビットにキャリーフラグの値を挿入する。結果は dst で指定されるレジスタに格納される。

sets shift left with carry and set LSB

アセンブラ　$sets\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x17 (aux:0x02)

sets 命令は、sr0 で指定されるレジスタの内容を 1 ビット左にシフトする。最下位ビットには、sr1 で指定されるレジスタの値が 0 以外ならば 1 を、0 ならばキャリーフラグの値を挿入する。結果は dst で指定されるレジスタに格納される。

shas shift right with carry and set LSB

アセンブラ　$shas\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x17 (aux:0x03)

shas 命令は、sr0 で指定されるレジスタの内容 (x) を 1 ビット右にシフトし、最上位ビットにはキャリーフラグの値を挿入する。さらに x の最下位ビットが 0 でなければ最下位ビットを 1 にする。結果は dst で指定されるレジスタに格納される。

shac shift right with carry and clear

アセンブラ　$shac\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x17 (aux:0x05)

shac 命令は、sr0 で指定されるレジスタの内容 (x) を 1 ビット右にシフトし、最上位ビッ

トにはキャリーフラグの値を挿入する。さらにsr1で指定されたレジスタの内容が0ならば最下位ビットを0にクリアする。結果はdstで指定されるレジスタに格納される。

整数乗除命令

mul multiply integer

アセンブラ　$mul\ r_{sr0}, r_{sr1}, r_{dst}$
　　$mul\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x1a

mul命令は、sr0とsr1あるいはimm17を符号拡張した値との乗算を行い、結果の下位32ビットをdstに格納する。

divx extend dividend to 64bit

アセンブラ　$divx\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x00)

divx命令は符合付き整数の32bit/32bitの除算を行なうために、非除数を64bitに符合拡張する。詳細はdivs命令を参照。

divi initial step for divide integer

アセンブラ　$divi\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x01)

divi命令は符合付き整数の64bit/32bitの除算のための初期化を行なう。詳細はdivs命令を参照。

divs step for divide integer

アセンブラ　$divs\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x02)

divs命令は符合付き整数の引き戻し法による除算の1ステップを行なう。sr0を非除数の上位32bit(x1)、特殊レジスタap(r27)を下位32bit(x0)、sr1を除数(y)とする。x1からyを引き、結果(r)の符号がyの符号と一致しなければ、引き過ぎたのであるから、x1を結果(r)として商を0とする。そうでなければ商を1とする。rとx0を連結して左に1ビットシフトして、LSBに商の値を挿入する。上位32ビットをdstに、下位32ビットをap(r27)に格納する。

32ビット同士の割算は、以下のように36クロックを要する。ここでx、y、q、rはそれぞれ被除数、除数、商、剰余を表すレジスタである。t0は作業レジスタを表す。

```
        div32 x y q r
=>
        divx x t0
        divi   t0 y t0
        divs   t0 y t0 (1)
        divs   t0 y t0 (2)
        ....
        divs   t0 y t0 (31)
        dive   t0 y t0
        divr   t0 y r
        mov    ap q
```

dive final step for divide integer

アセンブラ　$dive\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x03)

dive命令は符合付き整数の引き戻し法による除算の最終ステップを行なう。詳細はdivs命令を参照。

divr normaization of remainder for divide integer

アセンブラ　$divr\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x04)

divr命令は剰余の符合を非除数の符合に合わせるための補正を行なう。詳細はdivs命令を参照。

diviu initial step for divide unsigned integer

アセンブラ　$diviu\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x05)

diviu命令は符合なし整数の64bit/32bitの除算のための初期化を行なう。詳細はdivsu命令を参照

divsu step for divide unsigned integer

アセンブラ　$divsu\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x06)

divsu 命令は符合なし整数の引き戻し法による除算の1ステップを行なう。sr0 を非除数の上位 32bit(x1)、特殊レジスタ ap(r27) を下位 32bit(x0)、sr1 を除数 (y) とする。キャリーフラグが 0 で、かつ、x1 から y を引いた結果 (r) キャリーが出れば、引き過ぎたのであるから商を 0 として、x1 を結果 (r) とする。そうでなければ商を 1 とする。r と x0 を連結して MSB をキャリーフラグへ、左に 1 ビットシフトして LSB に商の値を挿入し、上位 32 ビットを dst に、下位 32 ビットを ap(r27) に格納する。

符合なし整数の 32bit 同士の割算は、以下のように 35 クロックを要する。ここで x、y、q、r はそれぞれ被除数、除数、商、剰余を表すレジスタである。t0 は作業レジスタを表す。

```
        div32u x y q r
=>
        mov    x ap
        diviu  zr y t0
        divsu  t0 y t0 (1)
        divsu  t0 y t0 (2)
        ....
        divsu  t0 y t0 (31)
        diveu  t0 y r
        mov    ap q
```

diveu final step for divide unsigned integer

アセンブラ　$diveu\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット]　R
オペコード　0x08 (aux:0x07)

diveu 命令は符合なし整数の引き戻し法による除算の最終ステップを行なう。詳細は divsu 命令を参照。

cvtdf converting double to fraction sub-operation

アセンブラ　$cvtdf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x08)

cvtdf 命令は、double から float への変換の補助を行う。

cvtfd converting fraction to double sub-operation

アセンブラ　$cvtfd\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x09)

cvtfd 命令は、float から double への変換の補助を行う。

fsign check float sign

アセンブラ　$fsign\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x0a)

fsign 命令は、sr0 と sr1 を浮動小数点数と考え、両者の符合が一致している場合は +0.0 を、異なる場合は −0.0 を dst に格納する。

swap swap registers sub-operation

アセンブラ　$swap\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x08 (aux:0x0b)

swap 命令は、sr1 を dst に格納し、それと同時に、sr0 を特殊レジスタ ap(r27) に格納する。以下の 2 命令によりレジスタの値の交換が行なえる。ただし、交換されるのはデータ部のみである。

```
swap r3,r4,r3;
mov ap,r4;
```

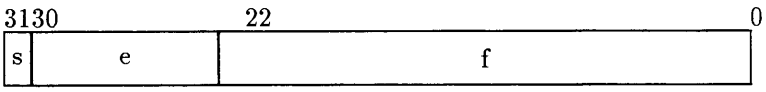
浮動小数点演算命令

addf add float

アセンブラ　$addf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x00)

addf 命令は、sr0 に sr1 を単精度浮動小数点数として加え、結果を dst に格納する。単精度浮動小数点数は図 C.3 に示す IEEE フォーマットにより表現される。ただし、DEN はサポートされておらず、0 とみなす。また、丸めは以下の 4 通りを aux 領域の下位 2 ビット (rnd) で指定する (default は 0)。

rnd = 0 Round to Nearest
rnd = 1 Round to Negative Infinity
rnd = 2 Round to Infinity



ZER	$e = 0, f = 0$	$v = (-1)^s \times (0.0) for \pm 0.0$
NORM	$0 < e < 255$	$v = (-1)^s \times 2^{e-127} \times (1.f)$
DEN	$e = 0, f \neq 0$	$v = (-1)^s \times 2^{e-126} \times (0.f)$
INF	$e = 255, f = 0$	$v = (-1)^s \times \infty$
NAN	$e = 255, f \neq 0$	$v = notanumber$

図 C.3: 単精度浮動小数点 IEEE フォーマット

rnd = 3 Round to Zero

浮動小数点演算 (fop) で以下のような場合は FOVF がセットされる。fop でそれ以外の場合は FOVF はクリアされる。fop 以外では FOVF は保持される。

- NAN が入力された場合。
- 演算結果が overflow した場合。
- $INF - INF$ を実行した場合。
- $0 \times INF$ を実行した場合。

subf subtract float

アセンブラ　$subf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x20)

subf 命令は、sr0 から sr1 を単精度浮動小数点数として減算し、結果を dst に格納する。

subrf subtract reverse float

アセンブラ　$subrf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x24)

subrf 命令は、sr1 から sr0 を単精度浮動小数点数として減算し、結果を dst に格納する。

minf minimum float

アセンブラ　$minf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x28)

minf 命令は、sr0 と sr1 のうち単精度浮動小数点数として小さい方の値を、dst に格納する。

maxf maximum float

アセンブラ　$maxf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x2c)

maxf 命令は、sr0 と sr1 のうち単精度浮動小数点数として大きい方の値を、dst に格納する。

cvtif convert integer to float

アセンブラ　$cvtif\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x30)

cvtif 命令は、sr0 を整数と解釈して単精度浮動小数点数に変換し、結果を dst に格納する。

cvtfi convert float to integer

アセンブラ　$cvtfi\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x38)

cvtfi 命令は、sr0 を単精度浮動小数点数と解釈して整数に変換し、結果を dst に格納する。

divsf divide step float

アセンブラ　$divsf\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x1c (aux:0x04)

divsf 命令は、定数 2.0 から sr0 を単精度浮動小数点数として減算し、結果を dst に格納する。

割算は除数の逆数との乗算により行なう。逆数を求めるには、ニュートン・ラプソン法に基づく以下の公式を用いて、反復法により求める。以下で a は逆数を求めたい数とする。

$$f(x) = \frac{1}{x} - a$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n(2 - ax_n)$$

初期値として仮数部上位 n ビットをキーとしてテーブルを引くことにする。テーブルサイ

ズを 64 とした時には 5 回目で計算が収束することを確認できるため、収束判定を行なわずに 4 回漸化式を繰り返した方が速い。実際の割算ルーチンは以下の通りで、クロック数は 18 から 20 クロックとなる。

```
        divf  b a c
=>
        lsr   a 17 r0
        and   r0 0x3f r0
        l     r0 inittbl r0
        and   a 0xff800000 r1
        sub   r0 r1 r0
        add   r0 0x3f800000 r0
        mulf  r0 a r1
        divsf r1 r1
        mulf  r0 r1 r0
        mulf  r0 a r1
        divsf r1 r1
        mulf  r0 r1 r0
        mulf  r0 a r1
        divsf r1 r1
        mulf  r0 r1 r0
        mulf  r0 a r1
        divsf r1 r1
        mulf  r0 r1 r0
        mulf  r0 b c
```

absf absolute float

アセンブラ　$absf\ r_{sr0}, r_{sr1}, r_{dst}$

フォーマット　R

オペコード　0x1c (aux:0x50)

absf 命令は、sr0 を単精度浮動小数点数として、その絶対値を dst に格納する。

negf negation float

アセンブラ　$negf\ r_{sr0}, r_{sr1}, r_{dst}$

フォーマット　R

オペコード　0x1c (aux:0x14)

negf 命令は、sr0 を単精度浮動小数点数として、その符号を反転した値を dst に格納する。

mulf multiply float

アセンブラ　$mulf\ r_{sr0}, r_{sr1}, r_{dst}$

フォーマット　R

オペコード　0x1d

mulf 命令は、sr0 と sr1 を単精度浮動小数点数として乗算し、結果を dst に格納する。

maaf multiply and add float

アセンブラ　$maaf\ r_{sr0}, r_{sr1}, r_{dst}$

フォーマット　R

オペコード　0x1f

maaf 命令は、sr0 と、sr1 とを単精度浮動小数点数として乗算を行ない、結果を特殊レジスタ FMR(floating multiply result resister) に格納する。同時に FMR の値と特殊レジスタ FAR(floating alu result resister) とを単精度浮動小数点数として加算を行ない、結果を FAR と dst に格納する。

n 項の内積の計算は以下の手順により n+2 命令で実現できる。

1. addf $r_{zero}, r_{zero}, r_{zero}$
   ; FMR = ??, FAR = 0
2. mulf r_{a1}, r_{b1}, r_{res}
   ; FMR = a1 b1, FAR = 0
3. maaf r_{a2}, r_{b2}, r_{res}
   ; FMR = a2 b2, FAR = a1 b1
4. maaf r_{a3}, r_{b3}, r_{res}
   ; FMR = a3 b3, FAR = a1 b1 + a2 b2

   ...

n+1. maaf r_{an}, r_{bn}, r_{res}
   ; FMR = an bn, FAR = a1 b1 + ... + an-1 bn-1

n+2. maaf $r_{zero}, r_{zero}, r_{res}$
   ; FMR = 0, FAR = a1 b1 + ... + an bn

メモリ参照命令

ldr load with register displacement

アセンブラ　$ldr\ r_{sr0}, r_{sr1}, r_{dst}$
$ldr\ r_{sr0}, imm17, r_{dst}$
$ldr.a\ r_{sr0}, r_{sr1}, r_{dst}$
$ldr.a\ r_{sr0}, imm17, r_{dst}$

フォーマット　R

オペコード　0x20,0x28(auto=1)

ldr 命令は、メモリデータを dst で指定されるレジスタに格納する。実効メモリアドレス (ea) は、sr0 と sr1 あるいは imm17 を符号拡張した値との加算により求める。加算後のメモリアドレスの下位 2 ビットは無視され、ワードアライメントで読み出しを行う。

auto=1 の時 (ldr.a 命令) は、同時に実効メモリアドレス (ea) を ap(r27) に格納する。sr0

として ap を指定して ldr.a を連続的に実行することにより、実効アドレスの更新とメモリ読み出しを並行して実行することが可能となる。

ld load

アセンブラ　$ld\ r_{sr0}, disp17, r_{dst}$
　　$ld.a\ r_{sr0}, disp17, r_{dst}$
フォーマット　M
オペコード　0x21,0x29(auto=1)

ld 命令は、メモリデータを dst に格納する。実効メモリアドレスは、ベースアドレス sr0 と、ワード変位 disp(の値を符号拡張した値を 4 倍した値) を加算した値。アセンブラではバイト変位で指定する。その他 ldr 命令を参照。

st store

アセンブラ　$st\ r_{sr0}, disp, r_{sr1}$
　　$st.a\ r_{sr0}, disp, r_{sr1}$
フォーマット　M
オペコード　0x22,0x2a(auto=1)

st 命令は、sr1 をメモリに格納する。その他は ld 命令を参照。

xchg exchange memory with register

アセンブラ　$xchg\ r_{sr0}, disp, r_{sr1}$
　　$xchg.a\ r_{sr0}, disp, r_{sr1}$
フォーマット　M
オペコード　0x23,0x2b(auto=1)

xchg 命令は、メモリデータと sr1 とを交換する。本命令実行には 2 クロック必要とする。その他は ld 命令を参照。

lrr load relative in operand segment with register displacement

アセンブラ　$lrr\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lrr\ r_{sr0}, imm17, r_{dst}$
　　$lrr.a\ r_{sr0}, r_{sr1}, r_{dst}$
　　$lrr.a\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x24,0x2c(auto=1)

lrr 命令は、メモリデータを dst に格納する。実効メモリアドレスのベースアドレスとして、sr0 の下位 9 ビットを 0 にマスクした値を用いる。その他は ldr 命令を参照。

この命令は、オペランドセグメントが 128 ワード固定長の時、オペランドセグメント内の変位によるメモリアドレッシングを可能にする命令である。

lr load relative in operand segment

アセンブラ　$lr\ r_{sr0}, disp, r_{dst}$
　　$lr.a\ r_{sr0}, disp, r_{dst}$
フォーマット　M
オペコード　0x25,0x2d(auto=1)

lr 命令は、メモリデータを dst に格納する。実効メモリアドレスのベースアドレスとして、sr0 の下位 9 ビットを 0 にマスクした値を用いる。その他は ld 命令を参照。

sr store relative in operand segment

アセンブラ　$sr\ r_{sr0}, disp, r_{sr1}$
　　$sr.a\ r_{sr0}, disp, r_{sr1}$
フォーマット　M
オペコード　0x26,0x2e(auto=1)

sr 命令は、sr1 をメモリに格納する。その他は lr 命令を参照。

xchgr exchange memory relative in operand segment with register

アセンブラ　$xchgr\ r_{sr0}, disp, r_{sr1}$
　　$xchgr.a\ r_{sr0}, disp, r_{sr1}$
フォーマット　M
オペコード　0x27,0x2f(auto=1)

xchgr 命令は、メモリデータと sr1 とを交換する。本命令実行には 2 クロック必要とする。その他は lr 命令を参照。

enq enque

アセンブラ　$enq\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x32

enq 命令は、sr1 を sr0 で指定するメモリに格納するとともに、sr0 を dst に格納する。その他は ldr 命令を参照。

この命令は主にフリーリスト管理しているキューの先頭に要素を追加するとともに、ポインタの更新を行なう命令である。

deq deque

アセンブラ　$deq\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x33

deq 命令は、sr1 を sr0 で指定するメモリに格納するとともに、そのメモリデータを dst に格納する。本命令実行には 2 クロック必要とする。その他は ldr 命令を参照。

この命令は主にフリーリスト管理しているキューの先頭から要素を取り出し、ポインタの更新を行なうとともに、その要素への値の代入を行なう命令である。

enqr enque relative in operand segment

アセンブラ　$enqr\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x36

enqr 命令は、メモリアドレスの下位 9 ビットを 0 でマスクする以外、enq 命令と同じである。

deqr deque relative in operand segment

アセンブラ　$deqr\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x37

deqr 命令は、メモリアドレスの下位 9 ビットを 0 でマスクする以外、deq 命令と同じである。本命令実行には 2 クロック必要とする。

stb store with byte insertion

アセンブラ　$stb\ r_{sr0}, r_{sr1}, r_{dst}$
　　$stb\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x3a

stb 命令は、特殊レジスタ ap(r27) で指定されるバイトデータに、sr1 あるいは imm17 の下位 8 ビットを書き込む。ただし、ap の下位 2 ビットを 0 としたワードデータを sr0 で指定する必要がある。

実際の実行は、以下に示すような ldr.a との組合せにより、2 命令で任意のバイトデータの書き込みが行なえる。ただし、r0 にバイトデータのベースアドレス、r1 にバイトデータが入っており、disp はバイトデータへの変位であるとする。

```
ldr.a r0, disp, tmp
stb tmp, r1, zr
```

sts store with short word insertion

アセンブラ　$sts\ r_{sr0}, r_{sr1}, r_{dst}$
　　$sts\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x3e

sts 命令は、特殊レジスタ ap(r27) で指定されるショートワードデータに、sr1 あるいは imm17 の下位 16 ビットを書き込む。ただし、ap の下位 2 ビットを 0 としたワードデータを sr0 で指定する必要がある。

実際の実行は、以下に示すような ldr.a との組合せにより、2 命令で任意のショートワードデータの書き込みが行なえる。ただし、r0 にショートワードデータのベースアドレス、r1 にショートワードデータが入っており、disp はショートワードへの変位であるとする。

```
ldr.a r0, disp, tmp
sts tmp, r1, zr
```

分岐命令

bcc branch on condition

アセンブラ　$bcc\ r_{sr0}, r_{sr1}, label$
　　$bcc\ r_{sr0}, imm5, label$
　　$bcc.a\ r_{sr0}, r_{sr1}, label$
　　$bcc.a\ r_{sr0}, imm5, label$
フォーマット　M
オペコード　0x40-0x6f

bcc 命令は、sr0 と、sr1 あるいは imm5 を符号拡張した値とを比較する。条件が満たされれば nPC + sign extend(disp17)× 4 で指定されるアドレス (アセンブラでは直接分岐アドレスを指定) に次次命令で分岐する。すなわち遅延スロットが 1 命令ある。もし満たされなければ分岐は起きない。各分岐条件を表 C.1 に示す。

また、opcode の再下位ビット (annulled) が 1 の時に分岐条件が満たされると、遅延スロットの命令を無効化 (実際には nop に置き換える) する。これにより annulled 領域が 1 の場

合は条件分岐命令を連続して実行することができる。

jl jump and link

アセンブラ　*jl label*, r_{dst}
フォーマット　J
オペコード　0x7c

jl 命令は、$target20 \times 4$ で指定されるアドレス (アセンブラでは直接分岐先を指定) に遅延なしで分岐する。それと同時に dst に現在の実行命令アドレスを格納する。単に分岐したい場合には dst に zr(r31) を指定する。

jlr jmp and link indirect

アセンブラ　*jlr* r_{sr1}, *disp*, r_{dst}
フォーマット　M
オペコード　0x7d

jlr 命令は、sr1 に disp で指定されるワード変位 (17bit の値を符号拡張した値を 4 倍した値) を加えたアドレスに次次命令で分岐する。すなわち遅延スロットが 1 命令ある。アセンブラではバイト変位を指定。下位 2 ビットは無視される。それと同時に dst に現在の実行命令アドレスを格納する。単に分岐したい場合には dst に zr(r31) を指定。

strap software trap

アセンブラ　*strap trapno*, *request*
フォーマット　M
オペコード　0x7e

strap 命令は、
$TRAPBASE(0xc000) + trapno \times 512$
で指定されるアドレスに遅延なしで分岐する。それと同時に特殊レジスタ IRETA0 に現在の実行命令アドレスを、特殊レジスタ IRETA1 に次実行命令アドレスを格納する。

本命令は連続して並べることはできない。また、強連結ブロックの最後に置くことはできない。

reti return from interrupt

アセンブラ　*reti aux*, *disp*
フォーマット　M
オペコード　0x7f

reti 命令は、aux のビット 0 が 0 の場合は特殊レジスタ IRETA0 に、そうでなければ特殊

表 C.1: 条件分岐命令の条件一覧

Mnemonic	Condition	Operation
beq(.a)	00000	branch on equal
bne(.a)	00001	branch on not equal
bra(.a)	00010	branch always
brn(.a)	00011	branch never
bgt(.a)	00100	branch on greater
ble(.a)	00101	branch on less or equal
blt(.a)	00110	branch on less
bge(.a)	00111	branch on greater or equal
bgtu(.a)	01000	branch on greater, unsigned
bleu(.a)	01001	branch on less or equal, unsigned
bltu(.a)	01010	branch on less, unsigned
bgeu(.a)	01011	branch on greater or equal, unsigned
bgtf(.a)	01100	branch on greater, float
blef(.a)	01101	branch on less or equal, float
bltf(.a)	01110	branch on less, float
bgef(.a)	01111	branch on greater or equal, float
btst(.a)	10000	branch on test (sr0 & sr1 $\neq$ 0)
bntst(.a)	10001	branch on test false
btt(.a)	10010	branch on tag test (sr0.dt & sr1.dt $\neq$ 0)
bntt(.a)	10011	branch on tag test false
bc(.a)	10100	branch on carry
bnc(.a)	10101	branch on not carry
bv(.a)	10110	branch on overflow
bnv(.a)	10111	branch on not overflow

レジスタ IRETA1 に disp で指定されるワード変位 (17bit の値を符号拡張した値を 4 倍した値) を加えたアドレスに次次命令で分岐する。アセンブラではバイト変位を指定。

また、トラップルーチンの最後の命令では aux のビット 1 を 1 にすることにより、各種フラグの復帰を指定する。

トラップルーチンからトラップを起こした命令に復帰するためには、

```
reti 0, 0;
reti 3, 0;
```

とする。また、トラップを起こした次の命令に復帰するためには、

```
reti 1,0;
reti 3,1;
```

とする。

パケット処理命令

send0 send packet type 0

アセンブラ $send0\ r_{sr0}, r_{sr1}, disp, wcf$
$send0.pt\ r_{sr0}, r_{sr1}, disp, wcf, pt$
フォーマット M
オペコード 0xc0

send0 命令は、sr0 を図 C.4 に示すパケットアドレスとともに出力する。send0.pt 命令では、パケットトレースフラグ trc(アセンブラでは pt の LSB から 7bit 目で指定)、パケットタイプ pt を指定する。主にオペランドセグメントが 128 ワード固定サイズの場合の関数内、および関数間の引数渡しなどに用いる。

send1 send packet type 1

アセンブラ $send1\ r_{sr0}, r_{sr1}, 0, wcf$
$send1.pt\ r_{sr0}, r_{sr1}, 0, wcf, pt$
フォーマット M
オペコード 0xc1

send1 命令は、sr0 を図 C.4 に示すパケットアドレスとともに出力する。sr1 により PE 番号を含んだメモリアドレス (グローバルアドレス) を指定する。主にリモートメモリ参照などのスペシャルパケット生成に用いる。

send2 send packet type 2

アセンブラ $send2\ r_{sr0}, r_{sr1}, 0, 0$
フォーマット M
オペコード 0xc2

send2 命令は、sr0 を図 C.4 に示すパケットアドレスとともに出力する。パケットタイプは sr1 で指定される。主に関数の結果を返すのに用いる。

send3 send packet type 3

アセンブラ $send3\ r_{sr0}, r_{sr1}, disp, wcf$
フォーマット M
オペコード 0xc3

send3 命令は、sr0 を図 C.4 に示すパケットアドレスとともに出力する。sr1 により PE 番号を含んだグローバルアドレスを指定し、命令内の即値によりセグメント内変位 disp(pt 領域と合わせて 13 ビット) を指定する。パケットタイプは sr1 で指定される。リモートメモリの構造体や配列の連続アクセスなどに用いる。

senda send packet address

アセンブラ $senda0\ r_{sr0}, r_{sr1}, disp, wcf$
$senda1\ r_{sr0}, r_{sr1}, 0, wcf$
$senda2\ r_{sr0}, r_{sr1}, 0, 0$
$senda3\ r_{sr0}, r_{sr1}, disp, wcf$
$senda0.pt\ r_{sr0}, r_{sr1}, disp, wcf, pt$
$senda1.pt\ r_{sr0}, r_{sr1}, 0, wcf, pt$
フォーマット M
オペコード 0xc4-0xc7

senda 命令は、send 命令で生成されるデータ部とアドレス部を交換したパケットを生成する。ただしパケットトレースフラグ trc(アセンブラでは pt の LSB から 7bit 目で指定) はアドレス部の trc を指定する。詳細は send 命令を参照。

sr0 で指定されるパケットアドレスが二入力待ち合わせである場合には、データ部の PT の上位 2bit はシステムでクリアされることに注意。

lpa load packet address

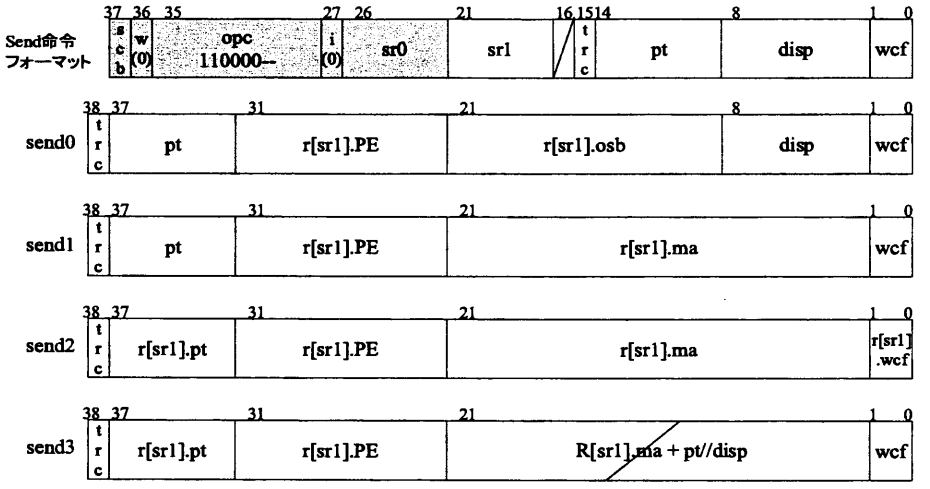


図 C.4: send 命令の生成するパケットアドレスフォーマット

アセンブラ $lpa0\ r_{sr1}, disp, wcf, r_{dst}$
$lpa1\ r_{sr1}, 0, wcf, r_{dst}$
$lpa2\ r_{sr1}, 0, 0, r_{dst}$
$lpa3\ r_{sr1}, disp, wcf, r_{dst}$
$lpa0.pt\ r_{sr1}, disp, wcf, pt, r_{dst}$
$lpa1.pt\ r_{sr1}, 0, wcf, pt, r_{dst}$

フォーマット M

オペコード 0x80-0x83

lpa 命令は、send 命令で生成されるパケットアドレス (trc ビットを除く) を dst に格納する。詳細は send 命令を参照。

alup ALU calculation and send packet

アセンブラ $addp\ r_{sr0}, r_{sr1}, disp, wcf$
$subp\ r_{sr0}, r_{sr1}, disp, wcf$
$addcp\ r_{sr0}, r_{sr1}, disp, wcf$
$subbp\ r_{sr0}, r_{sr1}, disp, wcf$
$andp\ r_{sr0}, r_{sr1}, disp, wcf$
$orp\ r_{sr0}, r_{sr1}, disp, wcf$
$eorp\ r_{sr0}, r_{sr1}, disp, wcf$
$ornp\ r_{sr0}, r_{sr1}, disp, wcf$

フォーマット M

オペコード 0xd0-0xd7

alup 命令は、sr0 と sr1 で指定された演算を行ない、結果を send0 と同様のパケットアドレスに出力する。ただし PE 番号およびオペランドセグメントベースとしては特殊レジスタ fp の内容を用いる。詳細は各演算命令および send0 命令を参照。

sendc send packet with ca address

アセンブラ $sendc0\ r_{sr0}, r_{sr1}, ca, disp, wcf$
$sendc1\ r_{sr0}, r_{sr1}, ca, 0, wcf$
$sendc2\ r_{sr0}, r_{sr1}, ca, 0, 0$
$sendc3\ r_{sr0}, r_{sr1}, ca, disp, wcf$

フォーマット M

オペコード 0xe0-0xff

sendc 命令は、send 命令で生成されるパケットのうち、宛先 PE のメンバー番号をオペコード内の ca で指定する以外、send 命令と同じ。詳細は send 命令を参照。

その他

lddt load data tag

アセンブラ $lddt\ r_{sr0}, r_{sr1}, r_{dst}$

フォーマット R

オペコード 0x90

lddt 命令は、sr0 のタグ部 (37-32 ビット部) を、dst の下位 6 ビットに格納する。それ以外の部分は 0 となる。

anddt and data tag

アセンブラ $anddt\ r_{sr0}, r_{sr1}, r_{dst}$
$anddt\ r_{sr0}, imm17, r_{dst}$

フォーマット R

オペコード 0x91

anddt 命令は、sr0 のタグ部と、sr1 あるいは imm17 の下位 6 ビットとのビット毎の論理積

を計算し、結果をdstの下位6ビットに格納する。それ以外の部分は0となる。

stdt store data tag

アセンブラ　$stdt\ r_{sr0}, r_{sr1}, r_{dst}$
　　$stdt\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x92

stdt命令は、sr0のタグ部をsr1あるいはimm17の下位6ビットで置き換え、結果をdstに格納する。

chgdt change data tag

アセンブラ　$chgdt\ r_{sr0}, r_{sr1}, r_{dst}$
フォーマット　R
オペコード　0x93

chgdt命令は、sr0のタグ部を、sr1のタグ部で置き換え、結果をdstに格納する。

stmt set maintenance address

アセンブラ　$stmt\ r_{sr0}, r_{sr1}, zr$
　　$stmt\ r_{sr0}, imm17, zr$
フォーマット　R
オペコード　0x94

stmt命令は、sr1あるいはimm17の下位8ビットで指定されるメンテナンスレジスタにアクセスする準備を行なう。メンテナンスアドレスについては付録Dを参照。dstレジスタとしてはzr(r31)を指定しなければならない。

8bitのメンテナンスアドレスのMSBが0の場合はメンテナンスレジスタの参照の準備で、アドレスがメンテナンスユニットに設定される。その後で以下のldmt命令によりメンテナンスレジスタの値をレジスタに格納する。

MSBが1の場合はメンテナンスレジスタの書き込み準備で、アドレスがメンテナンスユニットに設定されると同時に、sr0がメンテナンスユニットに設定され、書き込みサイクルが始まる。自動的に次のクロックで書き込みが行なわれる。書き込みの効果が現われるのは、2クロック後になる。

メンテナンスサイクルはシステムサイクルの2倍の周期を持つため、stmt命令を2回繰り返すことにより、メンテナンスユニットへの設定を行なう必要がある。また、メンテナンスユニットが外部からのアクセス要求に応答中で、EXUからのアクセス要求に答えることができない場合は、キャリーフラグ(C)に1が設定される。アクセスできた場合はキャリーフラグは保持される。これは連続するstmtの両方に対処するためである。このため、あらかじめキャリーをクリアしておき、stmt命令あるいはldmt命令実行後、正常に実行できたかどうかbcを用いて確かめなければいけない。また、競合の可能性が低い場合は、複数のメンテナンスアクセスを連続しておこない、まとめてキャリーをチェックしても良い。

ldmt load maintenance

アセンブラ　$ldmt\ zr, zr, r_{dst}$
フォーマット　R
オペコード　0x95

ldmt命令は、stmt命令で指定したメンテナンスアドレスからデータを読み出し、dstに格納する。メンテナンスアドレスについては付録Dを参照。

外部からのメンテナンスへの要求が優先されるため、ldmt命令はstmt命令の直後に行なわなければ、動作は保証されない。また、ldmt命令実行後に、キャリーフラグが1である場合は、外部からのメンテナンス要求のため、命令が実行されなかったことを示す。オペランドとしては必ずzrを指定すること。

ldsr load status register

アセンブラ　$ldsr\ zr, zr, r_{dst}$
フォーマット　R
オペコード　0x96

ldsr命令は、ステータスレジスタの値をdstに格納する。ステータスレジスタの内容は以下の通り。オペランドとしては必ずzrを指定すること。

bit28:INTOVF トラップ時に保持されたOVF

bit27:INTCARRY トラップ時に保持されたCARRY

bit26:INTENABLE トラップ許可フラグ。トラップが起きると自動的に0になり、trapi(aux[1]=1)により1になる。

bit25:STOPIBU ネットワークからのパケット受取を中断するフラグ。トラップが起きると自動的に1になり、trapi(aux[1]=1)により0になる。

bit24:FOVF 浮動小数点演算の結果によるoverflowフラグ。

bit23:OVF 整数演算などによるoverflowフラグ。

bit22:CARRY 整数演算などによるcarryフラグ。

bit21-0:EIA 現在実行中の命令アドレス。

extsb extract sign extended byte

アセンブラ　$extsb\ r_{sr0}, r_{sr1}, r_{dst}$
　　$extsb\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x98

extsb命令は、sr0から、sr1あるいはimm17の下位2ビットで指定されるバイトデータを抽出し、結果を符号拡張してdstに格納する。

extb extract byte

アセンブラ　$extb\ r_{sr0}, r_{sr1}, r_{dst}$
　　$extb\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x99

extb命令は、sr0から、sr1あるいはimm17の下位2ビットで指定されるバイトデータを抽出し、結果を上位に0を加えてdstに格納する。

insb insert byte

アセンブラ　$insb\ r_{sr0}, r_{sr1}, r_{dst}$
　　$insb\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x9a

insb命令は、sr0のうち、特殊レジスタap(r27)の下位2ビットで指定されるバイトアドレスのバイトデータに、sr1あるいはimm17の下位8ビットを挿入し、結果をdstに格納する。

extss extract sign extended short word

アセンブラ　$extss\ r_{sr0}, r_{sr1}, r_{dst}$
　　$extss\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x9c

extss命令は、sr0から、sr1あるいはimm17の下位2ビットで指定されるショートワードデータを抽出し、結果を符号拡張してdstに格納する。

exts extract short word

アセンブラ　$exts\ r_{sr0}, r_{sr1}, r_{dst}$
　　$exts\ r_{sr0}, imm17, r_{dst}$
フォーマット　R
オペコード　0x9d

exts命令は、sr0から、sr1あるいはimm17の下位2ビットで指定されるショートワードデータを抽出し、結果を上位に0を加えてdstに格納する。

inss insert short word

アセンブラ　$inss\ r_{sr0}, r_{sr1}, r_{dst}$
　　$inss\ r_{sr0}, imm, r_{dst}$
フォーマット　R
オペコード　0x9e

insb命令は、sr0のうち、特殊レジスタap(r27)の下位2ビットで指定されるショートワードデータに、sr1あるいはimm17の下位16ビットを挿入し、結果をdstに格納する。

alutst test operation for alu

アセンブラ　$alutst\ r_{sr0}, r_{sr1}, imm, r_{dst}$
フォーマット　R
オペコード　0x0b

alutst命令は、aluの全機能のテストを行なうための命令で、aluの制御信号であるfunc, cin, mを命令中のimmで任意に指定できる。演算データとしてはsr0で指定されるレジスタの内容と、sr1で指定されるレジスタの内容を使用する。結果はdstで指定されるレジスタに格納される。結果によりキャリーフラグ(C)が変更される。aluの各機能はTTL74181同等であり、表C.2の通り。

ldi load immediate

アセンブラ　$ldi\ imm_{tag}, imm_{data}, imr0$
　　$ldi\ imm_{tag}, imm_{data}, imr1$
フォーマット　W
オペコード　-

ldi命令は、imm_{tag}およびimm_{data}で指定される即値の値を、もしdirが0ならば特殊レ

表 C.2: ALUTST の機能一覧

function	m=1 (logic)	m=0 (arithmetic) cin=1	m=0 (arithmetic) cin=0
0 0 0 0	$\overline{A}$	A	A + 1
0 0 0 1	$\overline{A \vee B}$	$A \vee B$	$(A \vee B) + 1$
0 0 1 0	$\overline{A} \wedge B$	$A \vee \overline{B}$	$(A \vee \overline{B}) + 1$
0 0 1 1	0	-1	0
0 1 0 0	$\overline{A \wedge B}$	$A + A \wedge \overline{B}$	$A + A \wedge \overline{B} + 1$
0 1 0 1	$\overline{B}$	$(A \vee B) + A \wedge \overline{B}$	$(A \vee B) + A \wedge \overline{B} + 1$
0 1 1 0	$A \oplus B$	A - B - 1	A - B
0 1 1 1	$A \wedge \overline{B}$	$A \wedge \overline{B} - 1$	$A \wedge \overline{B}$
1 0 0 0	$\overline{A} \vee B$	$A + A \wedge B$	$A + A \wedge B + 1$
1 0 0 1	$\overline{A \oplus B}$	A + B	A + B + 1
1 0 1 0	B	$(A \vee \overline{B}) + A \wedge B$	$(A \vee \overline{B}) + A \wedge B + 1$
1 0 1 1	$A \wedge B$	$A \wedge B - 1$	$A \wedge B$
1 1 0 0	1	A + A	A + A + 1
1 1 0 1	$A \vee \overline{B}$	$(A \vee B) + A$	$(A \vee B) + A + 1$
1 1 1 0	$A \vee B$	$(A \vee \overline{B}) + A$	$(A \vee \overline{B}) + A + 1$
1 1 1 1	A	A - 1	A

ジスタ imr0 へ、1 ならば特殊レジスタ imr1
へ格納する。

付 録D　メンテナンスアドレス一覧

メンテナンスアドレスは 8bit で表されており、最上位ビットを除く上位 3bit により以下に示すように機能ユニット毎に分けられている。また、最上位ビットはメンテナンスアドレスに対する処理を表しており、0 の場合は読み出し、1 の場合は書き込みとなる。以下で特に述べない限り、各メンテナンスアドレスは読み書き可能な 32bit Flipflop を示しており、初期化により 0 クリアされる。各部の詳解で [] 内は bit 幅を示す。

0x00-0x0f メンテナンス回路および EMC-Y 全般

0x10-0x1f ネットワーク部 (Switching Unit)

0x20-0x2f 入力バッファ部 (Input Buffer Unit)

0x30-0x3f 待ち合わせ処理部 (Matching Unit)

0x40-0x5f 命令実行部 (EXecution Unit)

0x60-0x6f メモリ制御部 (Memory Control Unit)

0x70-0x7f 出力バッファ部 (Output Buffer Unit)

メンテナンスアドレスの bit 詳細で、メンテナンスアドレスの後に指示されている記号は以下のことを表す。

C 初期化時に 0 クリアされる

I 初期化時にある値にセットされる。値は bit 詳細の後に記述

R 読み出し専用

r 一部 bit が読み出し専用である。詳細は各 bit の説明を参照

メンテナンス

メンテナンス部のメンテナンスアドレスを以下に示す。

- RESET を on/off することによりフリップフロップの初期化を行なう。
- SELRn は OUTPORTn で選択されている入力 (1:IP0, 2:IP1, 4:OBU) を示す。
- RTn は INPORTn で選択されている宛先 (1:OP0, 2:OP1, 4:IBU) を示す。
- Smn は INPORTn の BANKm の状態 (0:E, 1:A, 2:D, 3:AD) を示す。変更の際は 0x1f の SELn も参照のこと。ただし、S02 は Flipflop ではなく OBU の状態からの組合せ回路であり、$0 \rightarrow 3 \rightarrow 2 \rightarrow 0 or 3$ という状態遷移をする。
- EMPmn はバッファが空であることを示す (F: 内蔵 FIFO, M: 外部メモリ, 0: low priority, 1: high priority)。本アドレスは Flipflop ではなく、各バッファのサイズ Flipflop からの組み合わせ回路である。初期値は 1。
- EX は待ち合わせフラグを示す。
- STAT は各ユニットの状態を示す (IBU 0:IDLE, 1:A, 2:D, 3:AD, 4:WR, 5:RD, 6:MM0, 7:MM, MU: 0:IDLE, 1:MM, 2:TT, 3:IF, EXU: 0:IDLE, 1:EXE, 2:WAIT, 3:LS, OBU: 0:E or A, 1:D)

- RSTn は restore を示す (0: low priority, 1: high priority)。本アドレスは Flipflop ではなく、IBU の status などからの組み合わせ回路である。
- BS は OBU のバッファ usage を示す。
- SELn は MCU におけるメモリアクセス調停結果を示す (0: first phase(1:EXU, 2:MU), 1: second phase(1:EXU, 2:MU, 4:IBU))。本アドレスは Flipflop ではなく、MCU の状態などからの組み合わせ回路である。
- HostCommunicationData はメンテナンスを用いたホストとの通信用のデータ。初期化されない。
- HostCommunicationControl はメンテナンスを用いたホストとの通信用の制御用。

```
MAINT
0x00      0[31]//RESET
0x01   Cr Network
          SELR2[3]//SELR1[3]//SELR0[3]//RT2[3]//S02[2]//
                RT1[3]//S21[2]//S11[2]//S01[2]//RT0[3]//S20[2]//S10[2]//S00[2]
0x02   I  Processor Status  (0x00c0c000)
          0[8]//IBU(EMPF1//EMPF0//EX//STAT[3]//RST1//RST0//EMPM1//EMPM0)[10]//
                MU(STAT[2])[2]//
                EXU(STAT[2])[2]//
                OBU(BS[4]//STAT)[5]//
                MCU(SEL1[3]//SEL0[2])[5]
0x03      HostCommunicationData
          HCD[32]
0x04   C  HostCommuniactionControl
          0[28]//HCC[4]
```

SU

- IPn.BNKm.A.D は INPORTn の BANKm のパケットバッファのアドレス部下位 32bit(グローバルアドレス部) を示す。ただし、初期化されず、IP0.BANK0 以外は読み出し専用。
- IPn.BNKm.A.T は INPORTn の BANKm のパケットバッファのアドレス部上位 7bit(パケットタイプ部) を示す。ただし、初期化されず、IP0.BANK0 以外は読み出し専用。
- IPn.BNKm.D.D は INPORTn の BANKm のパケットバッファのデータ部下位 32bit(データ部) を示す。ただし、初期化されず、IP0.BANK0 以外は読み出し専用。
- IPn.BNKm.D.T は INPORTn の BANKm のパケットバッファのデータ部上位 7bit(データタイプ部) を示す。ただし、初期化されず、IP0.BANK0 以外は読み出し専用。
- IORT は IO ノード向けルーティング (GA が異なれば IP1 へ) を指定する。
- HSTRT はホスト宛パケット (TRCbit が 1) のルーティングを指定する。(0: OP0, 1:OP1)
- LRn は OUTPORTn で直前に出力した入力先を示す。(1:IP0, 2:IP1, 0:OBU)
- MLPEn は INPORTn で現在出力中のパケットが MLPE パケットであることを示す。

- SELn は INPORTn で選択されている BANK を示す。(1:BANK0, 2:BANK1, 4:BANK2)
- DPARTn は INPORTn で現在データパートを出力中であることを示す。

```
SU
0x10      IP0.BNK0.A.D[32]
0x11      IP0.BNK0.D.D[32]
0x12   R  IP0.BNK1.A.D[32]
0x13   R  IP0.BNK1.D.D[32]
0x14   R  IP0.BNK2.A.D[32]
0x15   R  IP0.BNK2.D.D[32]
0x16   R  IP1.BNK0.A.D[32]
0x17   R  IP1.BNK0.D.D[32]
0x18   R  IP1.BNK1.A.D[32]
0x19   R  IP1.BNK1.D.D[32]
0x1a   R  IP1.BNK2.A.D[32]
0x1b   R  IP1.BNK2.D.D[32]
0x1c   r  0//IP0.BNK0.A.T[7]//0//IP0.BNK0.D.T[7]//
                0//IP0.BNK1.A.T[7]//0//IP0.BNK1.D.T[7]
0x1d   R  0//IP0.BNK2.A.T[7]//0//IP0.BNK2.D.T[7]//
                0//IP1.BNK0.A.T[7]//0//IP1.BNK0.D.T[7]
0x1e   R  0//IP1.BNK1.A.T[7]//0//IP1.BNK1.D.T[7]//
                0//IP1.BNK2.A.T[7]//0//IP1.BNK2.D.T[7]
0x1f   C  0[13]//IORT//HSTRT//LR2[2]//LR1[2]//LR0[2]
                //MLPEL2//SEL1[3]//MLPEL1//DPART1//SEL0[3]//MLPEL0//DPART0
```

IBU

- PA.t は SU からのパケット受信バッファのアドレス部を示す (D:グローバルアドレス部, T:パケットタイプ部)。初期化されない。
- PD.t は SU からのパケット受信バッファのデータ部を示す (D:データ部, T:データタイプ部)。初期化されない。
- FIFOA.t は MU へのパケット送信 FIFO のアドレス部を示す (D:グローバルアドレス部, T:パケットタイプ部)。本アドレスは読み出し専用であり、初期化されない。FIFO のサイズは両 priority 合わせて 16 パケットであるが、読み出される FIFO アドレスは以下の RDAn および high priority バッファの状態による (FIFO および MIB に high priority のパケットが存在すれば (BS1!=0 or SZ1!=0) 上位 8 パケットを RDA1 で、そうでなければ下位 8 パケットを RDA0 で示す)。
- FIFOD.t は MU へのパケット送信 FIFO のアドレス部を示す (D:データ部, T:データタイプ部)。本アドレスは読み出し専用であり、初期化されない。読み出される FIFO アドレスは FIFOA の項を参照。
- RDAn は MU への送信 FIFO の読み出しアドレスを示す (0:low priority, 1:high priority)。
- WRAn は MU への送信 FIFO の書き込みアドレスを示す (0:low priority, 1:high priority)。

- SZn は MU への送信 FIFO の使用サイズを示す (0:low priority, 1:high priority)。
- BASE0TOP は low priority 用 MIB(Memory Input Buffer) の開始アドレスを示す。アドレスとしては上位 nbit が 1 で, 残り (22-n)bit が 0 である値のみ指定可能 (n=1..12)。初期値は 0x3f8000。
- BSn は MIB の使用サイズを示す (0:low priority, 1:high priority)。
- BRAn は MIB の読み出し offset を示す (0:low priority, 1:high priority)。MIB の base は、low priority の場合は BASE0TOP で指定され、high priority の場合は 0 に固定。
- BWAn は MIB の書き込み offset を示す (0:low priority, 1:high priority)。base については BRA の項を参照。
- AUTOLOAD は MLPE などで用いる負荷の値を BS0 から自動的に生成することを示す ($0 < BS0 < 0x10 : LOAD = 1, 0x10 < BS0 < 0x1000 : LOAD = BS0/0x10, BS0 >= 0x1000 : LOAD = 0xff$)。0 の場合はプログラムにより以下の LOAD をプログラムにより設定する。
- LOAD は MLPE などで用いられる負荷の値を保持。AUTOLOAD=1 の場合については AUTOLOAD の項を参照。

```
IBU
0x20      PA.D[32]
0x21      PD.D[32]
0x22   R  FIFOA.D[32]
0x23   R  FIFOD.D[32]
0x24   r  00//PA.T[6]//00//PD.T[6]//00//FIFOA.T[6]//00//FIFOD.T[6]
0x25   C  0[9]//RDA0[3]//0//RDA1[3]//0//WRA0[3]//0//WRA1[3]//SZ0[4]//SZ1[4];
0x26
0x27
0x28   I  0[10]//BASE0TOP[12]//0[10] (0x003f8000)
0x29   C  0[13]//BS0[19]
0x2a   C  0[10]//BRA0[20]//00
0x2b   C  0[10]//BWA0[20]//00
0x2c   C  0[19]//BS1[13]
0x2d   C  0[17]//BRA1[13]//00
0x2e   C  0[17]//BWA1[13]//00
0x2f   C  0[23]//AUTOLOAD//LOAD[8]
```

MU

- PD0.t はパケットデータ部を示す (D:データ部, T:データタイプ部)。初期化されない。
- PD1.t は待ち合わせデータ部を示す (D:データ部, T:データタイプ部)。初期化されない。
- PAR.t はパケットアドレス部を示す (D:グローバルアドレス部, T:パケットタイプ部)。初期化されない。
- SCBTOP は命令開始アドレスを示す。初期化されない。

```
MU
0x30      PD0.D[32]
0x31      PD1.D[32]
0x32      PAR.D[32]
0x33      00//PD0.T[6]//0[4]//PD1.T[4]//00//PAR.T[6]//0[8]
0x34      0[10]//SCBTOP[20]//00
```

EXU

- OP0.t は第 0 オペランドを示す (D:データ部, T:データタイプ部)。初期化されない。
- OP1.t は第 1 オペランドを示す (D:データ部, T:データタイプ部)。初期化されない。
- IR.t は命令ワードを示す (D:下位 32bit, T:上位 6bit)。初期化されない。
- RR0.t はレジスタファイルのデータを示す (D:データ部, T:データタイプ部)。本アドレスは読み出し専用であり、初期化されない。本アドレスで読み出すことのできるのは r0 から r26 までの汎用レジスタであり、r27 から r30 までの特殊レジスタは以下のメンテナンスアドレスで直接参照することに注意。レジスタの指定は以下の MSOCEN, MSOC0 を用いる。
- AP.t は特殊レジスタ AP(r27) を示す (D:データ部, T:データタイプ部)。初期化されない。
- PR0.t は特殊レジスタ PR0(r28) を示す (D:データ部, T:データタイプ部)。初期化されない。
- PR1.t は特殊レジスタ PR1(r29) を示す (D:データ部, T:データタイプ部)。初期化されない。
- FP.t は特殊レジスタ FP(r30) を示す (D:データ部, T:データタイプ部)。FP.D の上位 10bit は PE 番号を示すが、この bit はメンテナンスよりのみ変更可能。初期化の際は 0 となるため、必ず PE 番号の設定が必要。
- IAR は現在フェッチ中の命令アドレスを示す。初期化されない。
- IRETA0 は割り込み発生時の実行中の命令アドレスを示す。初期化されない。
- IRETA1 は割り込み発生時のフェッチ中の命令アドレスを示す。初期化されない。
- INTOVF は割り込み発生時の OVF フラグの値を示す。
- INTCARRY は割り込み発生時の CARRY フラグの値を示す。
- INTENABLE は割り込みを許可するかどうかを示す。初期化時は 0 であり、1 にセットされるまで、割り込みは行なわれない。
- FOVF は浮動小数点演算で起きた overflow などを示す。
- OVF は整数演算などで起きた overflow を示す。
- CARRY は整数演算などで起きた carry を示す。
- EIA は現在実行中の命令アドレスを示す。
- FMPY は浮動小数点乗算回路の結果を保持する。初期化されない。
- FALU は浮動小数点 ALU 回路の結果を保持する。初期化されない。

- MSOCEN は上記の RR0 として読み出すレジスタ番号として以下の MSOC0 を使用することを示す。MSOCEN が 0 の場合は、メモリデータ (MCU の RD の SOC0 領域) の値が用いられる。
- MSOC0 は上記の RR0 として読み出すレジスタ番号を指定する。0 から 26 が有効。
- FMAA は fma 命令のオペランドの選択信号を先行制御するためのレジスタ。初期化されない。
- DCDALU は alu 命令 (add, sub など) の先行制御のためのレジスタ。初期化されない。

```
EXU
0x40      OP0.D[32]
0x41      OP1.D[32]
0x42      IR.D[32]
0x43   R  RR0.D[32]
0x44      AP.D[32]
0x45      PR0.D[32]
0x46      PR1.D[32]
0x47   C  FP.D[32]
0x48   r  00//OP0.T[6]//00//OP1.T[6]//00//IR.T[6]//00//RR0.T[6]
0x49      00//AP.T[6]//00//PR0.T[6]//00//PR1.T[6]//00//FP.T[6]
0x4a      0[10]//IAR[20]//00
0x4b      0[10]//IRETA0[20]//00
0x4c      0[10]//IRETA1[20]//00
0x4d   C  000//INTOVF//INTCARRY//INTENABLE//STOPIBU//
                FOVF//OVF//CARRY//EIA[20]//00
0x4e      FMPY[32]
0x4f      FALU[32]
0x50   C  0[25:0]//MSOCEN//MSOC0[5]
0x51      0[23:0]//FMAA[2]//DCDALU(FUNC[4]//CIN//M)[6]//
```

MCU

- DR.t はメモリアクセス 1st フェーズでの読み出しデータを保持する (D: データ部, T:データタイプ部)。本アドレスは読みだし専用であり、初期化されない。また、本アドレスは Flipflop ではなく、LATCH 信号によるラッチである。
- PR はメモリアクセス 1st フェーズでの読み出しパリティデータを保持する。本アドレスは読みだし専用であり、初期化されない。また、本アドレスは Flipflop ではなく、LATCH 信号によるラッチである。
- DRP は DR より計算されるパリティを保持する。
- PR1 は PR を保持する。1st フェーズのパリティチェックは DRP および PR1 を用いて次クロックで行なわれる。
- PREON はパリティエラーの検出を指定する。初期化時は 0 であり、1 にセットされるまで、検出は行なわれない。

- MAEON はアドレス例外の検出を指定する。初期化時は 0 であり、1 にセットされるまで、検出は行なわれない。
- ACAFL は MU によるメモリアクセスが行なわれていることを示す。
- MEMPRO はメモリ書き込みを禁止するアドレスの上限値を指定する。メモリアドレス上位 12bit のみ指定可能。初期化されない。
- A0EN は以下の A0S が有効であることを示す。初期化されない。
- A0S は 1st フェーズのメモリアドレスを保持する。1st フェーズのアドレス例外は A0EN が 1 の時に A0S の値を用いて行なわれる。初期化されない。

```
MCU
0x60  R  DR.D[32]
0x61  R  0[25]//PR[2]//DR.T[6]
0x62  C  0[25]//DRP[2]//PR1[2]//PREON//MAEON//ACAFL
0x63     0[10]//MEMPRO[12]//0[10]
0x64     0[9]//A0EN//A0S[12]//0[10]
```

OBU

- DPA.t は OBU の出力 FIFO のアドレス部を示す (D:グローバルアドレス部, T:パケットタイプ部)。本アドレスは読み出し専用であり、初期化されない。読み出される FIFO アドレスは以下の RP により指定される。
- DPD.t は OBU の出力 FIFO のデータ部を示す (D:データ部, T:データタイプ部)。本アドレスは読み出し専用であり、初期化されない。読み出される FIFO アドレスは以下の RP により指定される。
- RP は OBU の出力 FIFO の読み出しアドレスを示す。
- WP は OBU の出力 FIFO の書き込みアドレスを示す。

```
OBU
0x70  R  DPA.D[32]
0x71  R  DPD.D[32]
0x72  R  0//DPA.T//0//DPD.T//0[16]
0x73  C  0[21]//RP[3]//0[5]//WP[3]
```

発表文献

[1] K. Kodama, S. Sakai and Y. Yamaguti, "A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation," Proc. International Conference organized by the IPSJ to Commemorate the 30th Anniversary (InfoJapan'90), Vol.I, pp.291–298, Oct. 1990.

[2] Y. Kodama, S. Sakai and Y. Yamaguchi, "Load Balancing by Function Distribution on the EM-4 Prototype," Proc. 1991 International Conference on Supercomputing (SC'91), pp.522–531, Nov. 1991.

[3] Y. Kodama, S. Sakai and Y. Yamaguchi, "A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation," Future Generation Computer Systems, Vol.7, pp.199–209, 1991/92.

[4] Y. Kodama, S. Sakai and Y. Yamaguchi, "Evaluation of the EM-4 Highly Parallel Computer using a Game Tree Searching Problem," Proc. International Conference on Fifth Generation Computer Systems 1992 (FGCS'92), Vol.2, pp.731–738, June 1992.

[5] Y. Kodama, Y. Koumura, M. Sato, H. Sakane, S. Sakai and Y. Yamaguchi, "EMC-Y: Parallel Processing Element Optimizing Communication and Computation," Proc. 7th international conference on Supercomputing (ICS'93), pp.167–174, July 1993.

[6] Y. Kodama, S. Sakai and Y. Yamaguchi, "Evaluation of Parallel Execution Performance by Highly Parallel Computer EM-4," Systems and Computers in Japan, Vol.24, No.9, pp.607–614, 1993.

[7] Y. Kodama, H. Sakane, M. Sato, S. Sakai and Y. Yamaguchi, "A case study of optimizing parallel computation and remote memory access," RWC Technical Report, TR-94001, pp.173–174, June 1994.

[8] Y. Kodama, H. Sakane, M. Sato, S. Sakai and Y. Yamaguchi, "Message-based Efficient Remote Memory Access on a Highly Parallel Computer EM-X," Proc. International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'94), pp.135–142, Dec. 1994.

[9] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai and Y. Yamaguchi, "The EM-X Parallel Computer: Architecture and Basic Performance," Proc. the 22nd Annual International Symposium on Computer Architecture (ISCA'95), pp.14–23, June 1995.

[10] Y. Kodama, H. Sakane, M. Sato, S. Sakai and Y. Yamaguchi, "Message-based Efficient Remote Memory Access on a Highly Parallel Computer EM-X," IEICE Transactions on Information and Systems, Vol.E79-D, No.8, pp.1065–1071, Aug. 1996.

[11] Y. Kodama, H. Sakane, H. Koike, M. Sato, S. Sakai and Y. Yamaguchi, "Parallel Radix Sort Program Using Fine-Grain Communication," Proc. the 1997 International Conference on Parallel Architecture and Compilation Techniques (PACT'97), pp.136–145, Nov. 1997.

[12] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama and T. Yuba, "An Architecture of a Dataflow Single Chip Processor," Proc. 16th International Symposium on Computer Architecture (ISCA'89), pp.46–53, May 1989.

[13] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama and T. Yuba, "An Architectural Design of a Highly Parallel Dataflow Machine," Proc. of IFIP 89, pp.1155–1160, Aug. 1989.

[14] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama and T. Yuba, "Design of the Dataflow Single-Chip Processor EMC-R," Journal of Information Processing, Vol.13, No.2, pp.165–173, 1990.

[15] T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, S. Sakai, S. Sekiguchi and Y. Kodama, "Dataflow Computer Development at the Electrotechnical Laboratory," Proc. InfoJapan '90, Vol.I, pp.271–278, Oct. 1990.

[16] Y. Yamaguchi, S. Sakai and Y. Kodama, "Synchronization Mechanisms of a highly parallel dataflow machine EM-4," IEICE Trans. on Information and Systems, Vol.E74, No.1, pp.204–213, Jan. 1991.

[17] S. Sakai, Y. Kodama and Y. Yamaguchi, "Prototype Implementation of a Highly Parallel Dataflow Mahcine EM-4," Proc. 5th International Paralle Processing Symposium (IPPS'91), pp.278–286, Apr. 1991.

[18] S. Sakai, Y. Kodama and Y. Yamaguchi, "Design and Implementation of a Versatile Interconnection Network in the EM-4," Proc. International Conference on Parallel Processing 1991 (ICPP'91), Vol.1, pp.426–430, Aug. 1991.

[19] K. Okamoto, Y. Kodama, S. Sakai and Y. Yamaguchi, "Methodologies in development and testing of the dataflow machine EM-4," Parallel Computing, Vol.18, pp.901–912, 1992.

[20] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi and Y. Koumura, "Thread-based Programming for the EM-4 Hybrid Dataflow Machine," Proc. 19th International Symposium on Computer Architecture (ISCA'92), pp.146–155, May 1992.

[21] A. Shaw, Y. Kodama, M. Sato, S. Sakai and Y. Yamaguchi, "Performance of Data-Parallel Primitives on the EM-4 Dataflow Parallel Supercomputer," Proc. Frontiers'92, pp.302–309, Oct. 1992.

[22] S. Sakai, Y. Kodama and Y. Yamaguchi, "Design and implementation of a circular omega network in the EM-4," Parallel Computing, Vol.19, No.2, pp.125-142, 1993.

[23] M. Sato, Y. Kodama, S. Sakai and Y. Yamaguchi, "EM-C: Efficient Dynamic Multi-threading in Distributed Memory Space on the EM-4 Multiprocessor," Proc. International Conference on Parallel And Distributed Systems (ICPADS'93), pp.162–169, Dec. 1993.

[24] M. Sato, Y. Kodama, S. Sakai and Y. Yamaguchi, "Experience with Executing Shared Memory Programs using Fine-Grain Communication and Multithreading in EM-4," Proc. 8th International Parallel Processing Symposium (IPPS'94), pp.630–636, Apr. 1994.

[25] M. Sato, Y. Kodama, S. Sakai and Y. Yamaguchi, "EM-C: Programming with Explicit Parallelism and Locality for EM-4 Multiprocessor," Proc. International Conference on Parallel Architecture and Compilation Techiques (PACT'94), pp.3–14, Aug. 1994.

[26] A. Sohn, M. Sato, S. Sakai, Y. Kodama and Y. Yamaguchi, "Parallel Bidirectional Heuristic Search on the EM-4 Multithreaded Multiprocessor," Proc. Sixth IEEE Symposium on Parallel and Distributed Processing (SPDP'94), pp.100–107, Oct. 1994.

[27] M. Sato, Y. Kodama, H. Sakane, Y. Yamaguchi, S. Sekiguchi and S. Sakai, "Programming with Distributed Data Structure for EM-X Multiprocessor," International Workshop on Theory and Practice of Paralell Programming (TPPP'94), pp.453–464, Nov. 1994.

[28] A.Sohn, M. Sato, S. Sakai, Y. Kodama and Y. Yamaguchi, "Nonnumeric Search Results on the EM-4 Distributed-Memory Multiprocessor," Proc. Supercomputing'94, pp.301–310, Nov. 1994.

[29] H. Yamana, M. Sato, Y. Kodama, H. Sakane, S. Sakai and Y. Yamaguchi, "A Macrotask-level Unlimited Speculative Execution on Multiprocessors," Proc. International Conference on Supercomputing 1995 (ICS'95), pp.328–337, July 1995.

[30] H. Sakane, M. Sato, Y. Kodama, H. Yamana, S. Sakai and Y. Yamaguchi, "Dynamic Characteristics of Multithreaded Execution in the EM-X Multiprocessor," Proc. International Workshop on Computer Performance Measurement and Analysis 1995 (PERMEAN'95), pp.14–22, Aug. 1995.

[31] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai and Y. Yamaguchi, "Identifying the Capability of Overlapping Computation with Communication," Proc. International Conference on Parallel Architectures and Compilation Techiniques (PACT'96), pp.133–139, Oct. 1996.

[32] M. Sato, Y. Kodama, H. Sakane, H. Yamana, S. Sakai and Y. Yamaguchi, "Experience with Fine-Grain Communication in EM-X Multiprocessor for Parallel Sparse Matrix Computation," Proc. International Parallel Processing Symposium (IPPS'97), pp.242–248, Apr. 1997.

[33] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai and Y. Yamaguchi, "Fine-Grain Multithreading with the EM-X Multiprocessor," Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), pp.189–198, June 1997.

[34] O. Tatebe, Y. Kodama, S. Sekiguti and Y. Yamaguchi, "Highly Efficient Implementation of MPI Point-to-point Communication Using Remote Memory Operations," Proc. 12th ACM International Conference on Supercomputing (ICS'98), pp.267–273, July 1998.

[35] A. Sohn and Y. Kodama, "Load Balanced Parallel Radix Sort," Proc. 12th ACM International Conference on Supercomputing (ICS'98), pp.305–312, July 1998.

[36] A. Sohn, P. Ku, Y. Kodama and Y. Yamaguchi, "Communication Studies of Three Distributed-Memory Multiprocessors," Proc. of IEEE Symposium on High Performance Computer Architecture (HPCA'99), pp.310–314, Jan. 1999.

[37] H. Sakane, H. Honda, T. Yuba, Y. Kodama and Y. Yamaguchi, "Efficient Execution Techniques of Shared Memory Programs on the EM-X Distributed Memory Multiprocessor," Proc. of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS2000), pp.695–704, Nov. 2000.

[38] 児玉祐悦, 坂井修一, 山口喜教, "データ駆動型シングルチッププロセッサ EMC-R の動作原理と実装," 情報処理学会論文誌, Vol.32, No.7, pp.849–858, July 1991.

[39] 児玉祐悦，坂井修一，山口喜教, “EM-4 における並列性能評価,” 並列処理シンポシウム JSPP'91 論文集, pp.141–148, May 1991.

[40] 児玉祐悦, 甲村康人, 佐藤三久, 坂井修一, 山口喜教, “高並列処理向け要素プロセッサ EMC-Y の設計,” 並列処理シンポシウム JSPP'92 論文集, pp.329–336, June 1992.

[41] 児玉祐悦, 坂井修一, 山口喜教, “高並列計算機 EM-4 とその並列性能評価,” 電子情報通信学会論文誌, D-I, Vol.J75-D-I, No.8, pp.607–614, Aug. 1992.

[42] 児玉祐悦, 坂根広史, 佐藤三久, 坂井修一, 山口喜教, “高並列計算機 EM-X のリモートメモリ参照機構の評価,” 情報処理学会論文誌, Vol.36, No.7, pp.1691–1699, July 1995.

[43] 児玉祐悦, 坂根広史, 佐藤三久, 山名早人, 坂井修一, 山口喜教, “細粒度通信機構を用いた Radix ソートの実行,” 情報処理学会論文誌, Vol.38, No.9, pp.1726–1735, Sep. 1997.

[44] 坂井修一，平木敬，山口喜教，児玉祐悦, 弓場敏嗣, “データ駆動計算機のアーキテクチャ最適化に関する考察,” 情報処理学会論文誌, Vol.30, No.12, pp.1562–1572, Dec. 1989.

[45] 山口喜教，坂井修一，児玉祐悦, “新世代データフロー計算機 EM-4 プロトタイプ,” 電子技術総合研究所彙報, Vol.55, No.6, June 1991.

[46] 坂井修一，児玉祐悦, 山口喜教，“Architectural Design of a Parallel Supercomputer EM-5,” 並列処理シンポジウム JSPP'91 論文集, pp.149–156, May 1991.

[47] 坂井修一，児玉祐悦, 佐藤三久, 山口喜教，“超並列計算機における粒度最適化機構の検討,” 並列処理シンポジウム JSPP'92 論文集, pp.235–240, June 1992.

[48] A. Shaw, 児玉祐悦, 佐藤三久, 坂井修一，山口喜教, “Data-parallel Programming on the EM-4 Dataflow Parallel Supuercomputer,” 並列処理シンポジウム JSPP'92 論文集, pp.179–186, June 1992.

[49] 山口喜教, 佐藤三久, 児玉祐悦, 坂根広史, 平野聡, 田沼均, 須崎有康, 一杉裕志, 塚本亨治, “超並列システムの研究,” 電子技術総合研究所彙報 Vol.57, No.12, 1993, pp.71–84, Dec. 1993.

[50] 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教, “並列計算機 EM-X の並列プログラム言語 EM-C,” 情報処理学会論文誌, Vol.35, No.4, 1994, pp.551–560, Apr. 1994.

[51] 山口喜教, 坂井修一, 児玉祐悦, 佐藤三久, 坂根広史, “高並列計算機の開発と評価, 電子技術総合研究所彙報, Vol.58, No.10, 1994, pp.11–30, Oct. 1994.

[52] 山名早人, 佐藤三久, 児玉祐悦, 坂根広史, 坂井修一, 山口喜教, “並列計算機 EM-4 におけるマクロタスク間投機的実行の分散制御方式,” 情報処理学会論文誌, Vol.36, No.7, pp.1578–1588, July 1995.

[53] 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教, “並列計算機 EM-4 の細粒度通信による共有メモリの実現とマルチスレッドによるレーテンシ隠蔽,” 情報処理学会論文誌, Vol.36, No.7, pp.1669–1679, July 1995.

[54] 佐藤三久, 児玉祐悦, 坂根広史, 山名早人, 坂井修一, 山口喜教, “細粒度通信機構をもつ並列計算機 EM-X による疎行列問題の並列処理,” 情報処理学会論文誌, Vol.38, No.9, pp.1761–1770, June 1997.

[55] 坂根広史, 児玉祐悦, 建部修見, 小池汎平, 山名早人, 山口喜教, 弓場敏嗣, "ウェーブフロント型並列処理における分散メモリ型並列計算機の通信機構の評価," 情報処理学会論文誌, Vol.40, No.5, pp.2281–2292, May 1999.

[56] 建部修見, 児玉祐悦, 関口智嗣, 山口喜教, "ユーザレベルリモートメモリ書き込みを用いたMPIの効率的実装," 情報処理学会論文誌, Vol.40, No.5, pp.2246–2255, May 1999.

[57] 坂根広史, 本多弘樹, 弓場敏嗣, 児玉祐悦, 山口喜教, "細粒度通信機構を持つ並列計算機EM-Xにおける共有メモリプログラムの効率的実行," 情報処理学会論文誌, Vol.41, No.SIG 8, Nov. 2000.

[58] 児玉祐悦, 坂井修一, 山口喜教, 平木敬, "データ駆動計算機EM-4の負荷分散," 第38回情報処理学会全国大会, 2T-6, Mar. 1989.

[59] 児玉祐悦, 坂井修一, 山口喜教, "データ駆動型シングルチッププロセッサにおけるメンテナンスアーキテクチャ," 第39回情報処理学会全国大会, 5W-6, Oct. 1989.

[60] 児玉祐悦, 坂井修一, 山口喜教, "データ駆動計算機EM-4における資源管理の実現," 第40回情報処理学会全国大会, 3L-9, Mar. 1990.

[61] 児玉祐悦, 坂井修一, 山口喜教, "データ駆動計算機EM-4プロトタイプの性能評価," 第40回情報処理学会全国大会, 7P-2, Sep. 1990.

[62] 児玉祐悦, 坂井修一, 山口喜教, "データ駆動計算機EM-4の関数分散方式," 情報処理学会研究報告 ARC-85-9, pp.63–70, Nov. 1990.

[63] 児玉祐悦, 坂井修一, 山口喜教, "EM-4における動的関数分散方式の評価," 第42回情報処理学会全国大会, 3H-2, Mar. 1991.

[64] 児玉祐悦, 坂井修一, 山口喜教, "EM-4によるゲーム木探索問題の実行と評価," 電子情報通信学会技術研究報告 CPSY91-28, pp.189–196, July 1991.

[65] 児玉祐悦, 坂井修一, 山口喜教, "高並列計算機EM-5の待ち合わせ機構に関する検討," 第43回情報処理学会全国大会, 7Q-8, Oct. 1991.

[66] 児玉祐悦, 佐藤三久, 坂井修一, 山口喜教, "EM-Cによるアクティビティ分散方式の検討," 第46回情報処理学会全国大会, 6M-02, Mar. 1993.

[67] 児玉祐悦, 佐藤三久, 坂根広史, 坂井修一, 山口喜教, "高並列計算機EM-Xのアーキテクチャ," 情報処理学会研究報告 ARC-101-7, pp.49–56, Aug. 1993.

[68] 児玉祐悦, 坂根広史, 山口喜教, "論理回路エミュレータを用いた細粒度並列計算機の評価," 電子情報通信学会技術研究報告 CPSY97-44, pp.63–68, Aug. 1997.

[69] 坂井修一, 山口喜教, 平木敬, 児玉祐悦, 弓場敏嗣, "データ駆動型シングルチッププロセッサEMC-Rのアーキテクチャ," 電子情報通信学会 CPSY88-9, pp.17–24, July 1988.

[70] 山口喜教, 坂井修一, 平木敬, 児玉祐悦, 弓場敏嗣, " データ駆動型計算機EM-4における待ち合せ機構," 第37回情報処理学会全国大会論文集, IN-7, Sep. 1988.

[71] 坂井修一, 山口喜教, 児玉祐悦, " プロセッサ結合型オメガ網を用いた並列計算機の構成," 電子情報通信学会技術研究報告書 CPSY89-31, pp.1–6, Aug. 1989.

[72] 岡本一晃，児玉祐悦，坂井修一，山口喜教, “データ駆動計算機ＥＭ－４プロトタイプの開発・動作環境,” 情報処理学会研究報告, ARC-85-8, pp.55–62, Nov. 1990.

[73] 内野高志，児玉祐悦，坂井修一, 山口喜教, “データフロー計算機における論理合成の並列化について,” 第 43 回情報処理学会全国大会論文集, 7Q-6, Oct. 1991.

[74] 甲村康人, 児玉祐悦, 山口喜教, “データ駆動計算機 EM-4 における共有２分決定グラフの並列処理について,” 第 44 回情報処理学会全国大会論文集, 3D-5, Mar. 1992.

[75] 小中裕喜, 佐藤三久, 児玉祐悦, 坂井修一, 山口喜教, “EM-C によるニューラルネットワークの実現, 第 46 回情報処理学会全国大会論文集, 6M-05, Mar. 1993.

[76] 坂根広史, 児玉祐悦, 佐藤三久, 山名早人, 坂井修一, 山口喜教, “並列計算機 EM-X のプロセッサ・ネットワークインターフェースの最適化の検討,” 情報処理学会研究報告 ARC-104-14, pp.105–112, Jan. 1994.

[77] 坂根広史, 児玉祐悦, 佐藤三久, 山名早人, 坂井修一, 山口喜教, “並列計算機用要素プロセッサ EMC-Y の基本性能評価,” 情報処理学会研究報告 ARC-107-9, pp.65–72, July 1994.

[78] 佐藤三久, 児玉祐悦, 坂根広史, 山名早人, 坂井修一, 山口喜教, “細粒度通信機構を持つ並列計算機 EM-X による疎行列問題の並列処理,” 情報処理学会研究報告, ARC-113-27, pp.209–216, Aug. 1995.

[79] 坂根広史, 児玉祐悦, 佐藤三久, 山名早人, 坂井修一, 山口喜教, “行列演算ベンチマークを用いた並列計算機 EM-X の評価,” 情報処理学会研究報告, ARC-119-41, pp.239–244, Aug. 1996.

[80] 君島有紀, 坂根広史, 児玉祐悦, 中西正和, “高並列計算機 EM-X の性能モニタリングツール,” 情報処理学会研究報告 PRO-16-4, pp.19–24, Nov. 1997.

[81] 坂根広史, 本多弘樹, 弓場敏嗣, 児玉祐悦, 山口喜教, “細粒度並列処理向け要素プロセッサにおけるキャッシュメモリアーキテクチャ,” 並列処理シンポジウム JSPP’99(ポスター発表), pp.212, June 1999.

[82] 山名早人, 小池汎平, 児玉祐悦, 坂根広史, 山口喜教, “並列計算機を用いた臨界投機型情報検索サービスの予備評価, 並列処理シンポシウム JSPP’1999 (ポスター発表), pp.216, June 1999.

[83] 坂根広史, 本多弘樹, 弓場敏嗣, 児玉祐悦, 山口喜教, “並列計算機 EM-X の細粒度通信機構を持ちいた共有メモリベンチマークの実行,” 情報処理学会研究報告 ARC-137-1, pp.1–6, Mar. 2000.

[84] 高田亮, 清水昭皓, 児玉祐悦, 坂根広史, 佐谷野健二, 本多弘樹, 弓場敏嗣, “EM-X と MD One を統合化した粒子シミュレーション用並列計算機プロトタイプの構築,” 情報処理学会計算機研究報告 2000-ARC-139, pp.85–90, Aug. 2000.

[85] 佐谷野健二, 片下敏宏, 小池汎平, 児玉祐悦, 坂根広史, 甲村康人, “大容量 FPGA の応用によるマルチプロセッサエミュレーションシステムの評価” 情報処理学会研究報告, ARC-144-5, pp.25–30, July 2001.

[86] 佐谷野健二, 片下敏宏, 児玉祐悦, “オンキャッシュ通信機構を持つ分散メモリ型マルチプロセッサの FPGA による実装,” 並列処理シンポシウム JSPP’2002 (ポスター発表), pp.161–162, May 2002.

参考文献

[Aga90] A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz. "APRIL: A Processor Architecture for Multiprocessing," Proc. 17th International Symposium on Computer Architecture, pp.104–114, May 1990.

[Aga95] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," Proc. 22nd International Symposium on Computer Architecture, pp.2–13, June 1995.

[Alv90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith, "The tera computer system," Proc. International Conference on Supercomputing, pp.1–6, June 1990.

[Ang98] B.S. Ang, D. Chiou, L. Rudolph and Arvind, "The StarT-Voyager Parallel System," Proc. International Conference on Parallel Architecture and Compilation Techniques, pp.185–195, Oct. 1998.

[Arv88] Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," Proc. PARLE Conference, June 1987.

[Arv89] Arvind, R.S. Nikhil and K.K. Pingali, "I-structures: Data structure for parallel computing," ACM Trans. on Programming Languages and Systems, Vol.11, No.4, pp.598–632, Oct. 1989.

[Ble91] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith and M. Zagha, "A comparison of sorting algorithms for the Connection Machine CM-2," Proc. Symposium on Parallel Algorithms and Architectures, pp.3-16, July 1991.

[Boi98] J. Boisseau, L. Carter, K. Gtlin, A. Majumdar and A. Snavely, "NAS Benchmarks on the Tera MTA," Workshop on Multi-Threaded Execution Architecture and Compilers, Feb. 1998.

[Bri98] P. Briggs, "Tuning the BLAS for the Tera," Workshop on Multi-Threaded Execution Architecture and Compilers, Feb. 1998.

[Bur02] D. Burns, "Pre-Silicon Validation of Hyper-Threading Technology," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Car99] L. Carter, J. Feo and A. Snavely, "Performance and Programming Experience on the Tera MTA," Proc. SIAM Conference on Parallel Processing, Mar. 1999.

[Cas02] C. Cascaval, J.G. Gastanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J.E. Moreira, K. Strauss and H.S. Warren,Jr., "Evaluation of a Multithreaded Architecture for Cellular Computing," Proc. 8th International Symposium on High-Performance Computer Architecture, Feb. 2002.

[Che02] Y. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belenov and I. Santos, "Media Application on Hyper-Threading Technology," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Chi95] D. Chiou, B.S. Ang, Arvind, M.J. Beckerle, A. Boughton, R. Greiner, J.E. Hicks and J.C. Hoe, "START-NG: Delivering Seamless Parallel Computing," Proc. EURO-PAR'95, Lecture Notes in Computer Science, No. 966, Springer-Verlag, pp.101–116, Aug. 1995.

[Cru91] J.M. Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," ACM Trans. on Computer Systems, Vol.9, No.1, pp.21–65, Jan. 1991.

[Cul91] D.E. Culler, A. Sah, K. Shauser, T.von Eicken and J. Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler Controlled Threaded Abstract Machine," Proc. 4th International Conference on Architectural Support for Programing Languages and Operating Systems, pp.164–175, Oct. 1991.

[Dal87] W.J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty and S. Wills, "Architecture of a Message-Driven Processor," Proc. 14th Annual International Symposium on Computer Architecture, pp.337–344, June 1987.

[Dal90] W.J. Dally, "Virtual-Channel Flow Control," Proc. 17th Annual International Symposium on Computer Architecture, pp.60–68, June 1990.

[Den75] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor," Proc. 2nd Annual International Symposium on Computer Architecture, pp.125–131, Jan. 1975.

[Den83] J.B. Dennis, W.Y.P. Lim and W.B. Ackerman, "The MIT dataflow engineering model," Proc. IFIP 83, pp.553–560, 1983.

[Eic92] T.von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," Proc. 19th Annual International Symposium on Computer Architecture (ISCA'92), pp.256–266, May 1992.

[Gur85] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Prototype Dataflow Computer," Communications of the ACM, Vol.28, No.1, pp.34–52, Jan. 1985.

[Hal88] R.H. Halstead, Jr. and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," Proc. 15th Annual International Symposium on Computer Architecture, pp.443–451, June 1988.

[Hay94] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Ishihara and T. Shindo, "AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler," Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.196-207, 1994.

[Hei94] J. Heinlein, K. Gharachorloo, S. Dresser and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH multiprocessor," Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.38–50, 1994.

[Hir87] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada and T.Yuba, "The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems," Journal of Information Processing, IPS Japan, Vol.10, No.4, pp.219–226, 1987.

[Hir88] K. Hiraki, T. Shimada and K. Nishida, "A Hardware Design of the SIGMA-1, A Dataflow Computer for Scientific Computations," Proc. International Conference on Parallel Processing, pp.851-855, 1984.

[Hrt92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," Proc. 19th Annual International Symposium on Computer Architecture (ISCA'92), pp.136–145, May 1992.

[Hoc88] R.W. Hockney and C.R. Jesshope, "Parallel Computers 2: architecture, programming and algorithms, 2nd ed.," Adam Hilger, Bristol and Philadelphia, 1988.

[Hoe98] J.C. Hoe, "StarT-X: A One-Man-Year Exercise in Network Interface Engineering," Proc. Hot Interconnects VI, Aug. 1998.

[Ian88] R.A. Iannucci, "Toward a Dataflow / Von Neumann Hybrid Architecture," Proc. 15th International Symposium on Computer Architecture, pp.131–140, 1988.

[Len90] D. Lenoski, J. Laudon, K. Gharachoroloo, A. Gupta and J.Hennessy, "The Directory-Based Cache Coherence Protocol for DASH Multiprocessor," Proc. 17th International Symposium on Computer Architecture, pp.148–159, June 1990.

[Luk01] C.K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," Proc. 28th Annual International Symposium on Computer Architecture (ISCA2001), pp.40–51, June 2001.

[Mag02] W. Magro, P. Petersen and S. Shah, "Hyper-Threading Technology: Impact on Compute-Intensive Workloads," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Mar02] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Mat98] H. Matsuoka, K. Okamoto, H. Hirano, M. Sato, T. Yokota and S. Sakai, "Processor Pipeline Design for Fast Network Message Handling in RWC-1 Multiprocessor," IEICE Trans. Electronics, Vol.E81-C, No.9, pp.1391–1397, Sep. 1998.

[Nik89] R.S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?," Proc 16th International Symposium on Computer Architecture, pp.262–272, May 1989.

[Nik92] R.S. Nikhil, G.M. Papadopoulos and Arvind, "*T: A Multithreded Massively Parallel Architecture," Proc 19th International Symposium on Computer Architecture, pp.156–167, May 1992.

[Noa93] M.D. Noakes, D.A. Wallach and W.J. Dally, "The J-Machine Multiprocessor: An Architectural Evaluation," Proc. 20th International Symposium on Computer Architecture, pp.224–235, May 1993.

[Pre02] R.P. Preston, R.W. Badeau, D.W. Bailey, S.L. Bell, L.L. Biro, W.J. Bowhill, D.E. Dever, S. Felix, R. Gammack, V. Germini, M.K. Gowan, P. Gronowski, D.B. Jackson, S. Mehta, S.V. Morton, J.D. Pickholtz, M.H. Reilly and M.J. Smith, "Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading," Proc. 2002 IEEE International Solid-State Circuits Conference, pp.334-335, Feb. 2002.

[Shi86] T. Shimada, K. Hiraki, K. Nishida and S. Sekiguchi, "Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computation," Proc. 13th International Symposium on Computer Architecture, pp.226–234, June 1986.

[Sin92] J.P. Singh, W.D. Weber and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared memory," Computer Architecture News, ACM SIGARCH, Vol.20, No.1, pp.5–44, Mar. 1992.

[Sla71] J.R. Slagle, "Artificial Intelligence: The Heuristic Programming Approach," McGraw-Hill, 1971.

[Smi78] B. Smith, "A pipelined, shared resource MIMD computer," Proc. 1978 International Conference on Parallel Processing, pp.6–8, Aug. 1978.

[Sna98] A. Snavely, L. Carter, J. Boisseau, A. Majumar, K.S. Gatlin, N. Mitchell, J. Feo and B. Koblenz, "Multi-processor Performance on the Tera MTA," Supercomputing'98, Nov. 1998.

[Tia02] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito and E. Su, "Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Tul95] D.M. Tullsen, S.J. Eggers and H.M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," Proc. 22nd International Symposium on Computer Architecture, pp.392–403, June 1995.

[Tul96] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," Proc. 23rd International Symposium on Computer Architecture (ISCA'96), pp.191–202, May 1996.

[Tul99] D.M. Tullsen, J.L. Lo, S.J. Eggers and H.M. Levy, "Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor," Proc. 5th IEEE Symposium on High Performance Computer Architecture (HPCA'99), pp.54–58, Jan. 1999.

[Tul01] D.M. Tullsen and J.A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," Proc. 34th International Symposium on Microarchitecture, pp.318–327, Dec. 2001.

[Pap90] G.M. Papadopoulos and D.E.Culler, "Monsoon: an Explicit Token-Store Architecture," Proc. 17th International Symposium on Computer Architecture, pp.82–91, June 1990.

[Pap91] G.M. Papadopoulos and D.E.Culler, "Multithreading: A Revisionist View of Dataflow Architecture," Proc. 18th International Symposium on Computer Architecture, pp.342–351, May 1991.

[Pap93] G.M. Papadopoulos, G.A. Boughton, R.Greiner and M.J. Beckerle, "*T: Integrated Building Blocks for Parallel Computing," Proc. Supercomputing'93, pp.624-635, Nov. 1993.

[Pat85] D.A. Patterson, "Reduced Instruction Set Computers," Communications of the ACM, vol.28, No.1, pp.8–21, Jan. 1985.

[Pre02] R. Preston, R. Badeau, D. Bailey, S. Bell, L. Biro, W. Bowhill, D. Dever, S. Felix, R. Gammack, V. Germini, M. Gowan, P. Gronowski, D. Jackson, S. Mehta, S. Morton, J. Pickholtz, M. Reilly and M. Smith, "An 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading," Proc. IEEE International Solid State Circuits Conference, Feb. 2002.

[Wan02] H. Wang, P.H. Wang, R.D. Weldon, S.M. Ettinger, H. Saito, M. Girkar, S.S. Liao and J.P. Shen, "Speculative Precomputation: Exploring the Use of Multithreading for Latency," Intel Technology Journal, Vol.6, No.1, Feb. 2002.

[Woo95] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," Proc. 22nd International Symposium on Computer Architecture, pp.24–36, June 1995.

[Zag91] M. Zagha and G.E. Blelloch, "Radix Sort for Vector Multiprocessor," Proc. Supercomputing'91, pp.712–721, Nov. 1991.

[沖 89] 沖廣明, 瀧和男, 清慎一, 古市昌一, "マルチ PSI における並列詰め碁プログラムの実現と評価," 並列処理シンポジウム JSPP'89 論文集, pp.351–357, Feb. 1989.

[坂井 87] 坂井修一, 山口喜教, 平木敬, 弓場敏嗣, " データ駆動型シングルチッププロセッサＥＭＣ－Ｒにおける強連結枝モデルの導入," 電子情報通信学会データフローワークショップ予稿集, pp.231–238, Oct. 1987.

[島田 88] 島田俊夫, 関口智嗣, 平木敬, "データフロー言語 DFC の設計と実現," 電子通信学会論文誌, Vol.J71-D, No.3, pp.501–508, Mar. 1988.

[島田 93] 島田俊夫, 平木敬, 関口智嗣, "データフロー計算機 SIGMA-1 の基本性能評価," 情報処理学会論文誌, Vol.34, No.4, pp.690–698, Apr. 1993.

[白木 95] 白木長武, 小柳洋一, 今村信貴, 林憲一, 清水俊幸, 堀江健志, 石畑宏明, "高並列計算機 AP1000+ のメッセージハンドリング機構," 並列処理シンポジウム JSPP'95 論文集, pp.233–240, May 1995.

[戸田 90] 戸田賢二, 西田健次, 内堀義信, 島田俊夫, " マクロデータフロー計算機ＣＯＤＡ－アーキテクチャー," 並列処理シンポジウム JSPP'90 論文集, pp.185–192, May 1990.

[平木 95] 平木敬, 島田俊夫, 関口智嗣, "科学技術計算用データ駆動計算機 SIGMA-1 における最適化技法の評価," 情報処理学会研究会報告 ARC-95-15, pp.113-118, Aug. 1995.

[平木 93] 平木敬, 天野英晴, 久我守弘, 末吉敏則, 工藤知宏, 中島浩, 中條拓伯, 松田秀雄, 松本尚, 森眞一郎, "超並列プロトタイプ計算機 JUMP-1 の構想," 情報処理学会研究報告, ARC-102-10, pp.73–84, Oct. 1993.

[平木 02] 平木敬, "SIGMA-1: データフロースーパーコンピュータ," 情報処理学会学会誌, Vol.43, No.2, pp.127–129, Feb. 2002.