マルチプロセサー用キャッシュメモリーの設計

DESIGN OF CACHE MEMORY SYSTEM FOR MULTI-PROCESSORS

鈴木則久　　多田好克　　梅村恭司

Norihisa SUZUKI,　Yoshikatsu TADA,　Kyoji UMEMURA

東京大学大学院　工学系研究科　情報工学専門課程

Information Engineering Course, Graduate School, University of Tokyo

[1.] はじめに

我々はVLSI・CADとか、プログラム作成支援システムに使う高性能ワークステーション用ソフトとハードの研究を行なっている。このプロジェクトは「三四郎プロジェクト」と呼ばれている。(1)ソフトとハードの研究は同時に行なっており、ソフトは既存のスーパー・パソコン「SUNワークステーション」(2)を使って開発している。将来はこれらのソフトを現在開発中のマルチ・マイクロプロセサーによるワークステーション「けやき」の上に移して動かす予定である。「けやき」は記憶共有型MIMD汎用マルチプロセサーを目指しているが、その最もユニークな特徴は、キャッシュメモリーを使ってバス・トラフィックを減らし、多くのプロセサーを付けられる様にした点である。

現在外部プロトコールの論理仕様を終え、性能評価と論理シミュレーションを行ない、論理仕様が正しそうである事を確認した。現在は内部構造とタイミングまで入れた外部仕様を決めている。

ここでは、第2節では「けやき」の特徴をあげ、第3節ではキャッシュメモリーの外部プロトコールとその機能を述べ、第4節ではキャッシュ内部の設計についてまだ中間段階ではあるが簡単に触れる。第5節では性能評価の方法とその結果を述べ、第6節では論理シミュレーションの方法と結果について考察する。

[2.] マルチプロセサー「けやき」の特徴

現在「三四郎プロジェクト」ではM68000のシングルプロセサーを使ってソフトを開発しているが、「けやき」はM68000のマルチ・マイクロプロセサーである。この為、「けやき」への要求の第一はソフトの移植が楽に行く事である。

2－1　記憶共有型MIMDマルチプロセサー

ソフトの移植を容易にする為に「けやき」では記憶共有型（shared memory）マルチプロセサーにした。現在市販されている大型汎用計算機は大部分記憶共有型マルチプロセサーである。IBMの3081プロセサーとかFACOM380などはこの型である。この方式ではモニターの様に共通変数を通信手段としている高級言語の同期方式の実現に適している。

一方、最近はOCCAMやUNIX／Cの様にパイプでデータストリームを送って通信をし同期を取る高級言語が出て来ている。これらはネットワーク上の疎結合マルチプロセサー上に実現するのに適当であるが、当然密結合な「けやき」の上でも実現可能である。しかしながら逆は出来ない。マルチプロセサーの実験をするには記憶共有型が良い訳である。

記憶共有型を実際に作る段に成ると色々問題がある。一番問題なのは主記憶へのトラフィックをいかに減らすかである。直接プロセサをメモリーバスにつないでいたのでは、各プロセサーがメモリーを参照する度に主記憶をアクセス

するので、せいぜい2～3台しかつなげない。（図2－1参照）

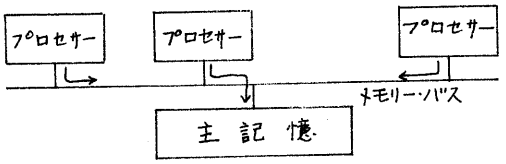


図2-1　直接結合方式。これではプロセサーがメモリーを参照する度に、メモリー・バスのトラフィックが生じる。

主記憶へのトラフィックを減らす方法は種々考えられているが、一番簡単で広く使われているのはローカル・メモリーを付ける方法である。ローカル・メモリーは各プロセサーに所属しており、ここをアクセスしている限りは他に全然影響を与えない。（図2－2）

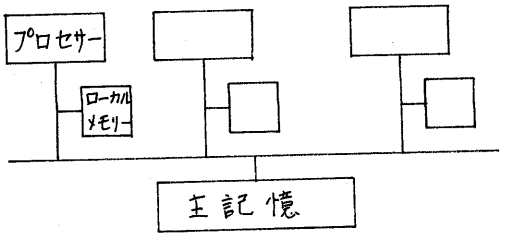


図2-2　ローカル・メモリー方式　プロセサーがローカルメモリーをアクセスする限り性能は上がる。

カーネギーメロン大学で開発されたCM*はローカル・メモリーを持ったマルチプロセサーである。ローカル・メモリーには命令を入れて成功している。(6)

ローカル・メモリー方式で問題が無い訳ではない。プログラムでアクセス可能な記憶領域のうち一部が速く、能率が良くなる。またある番地は遅いが、他のプロセサーからもアクセス出来る番地である。この様に対称性が崩れる事は一般的に良くない。能率の良いプログラムを作ろうとする為に、コンパイラーを書く時に複雑さが増す。また、良くアクセスするものはローカル・メモリーに置きたくなるので、最良のパーフォーマンスを得る為にはローカル・メモリーのスワッピングを行なう様になる。これはハードで本質すべき機能がソフトの中にはいり込んだと見るべきだろう。

我々はキャッシュを置く事にし、最も良く使う所が最も能率良くアクセスでき、他にも影響を与えないようにする事にした。（図2－3）

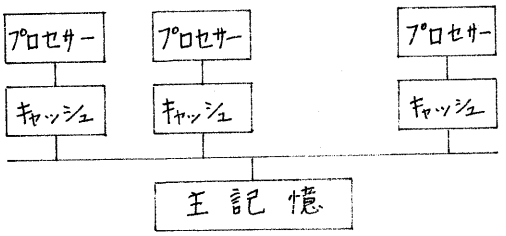


図2-3　キャッシュメモリー方式

2－2　不均質なマルチプロセサー

「ケやき」マルチプロセサーはワークステーションを作る為に設計されているので、走らせるプログラムも従来のマルチプロセサーに走らせるプログラムとは違う。特に「ケやき」は汎用なので使い方はILLIAC Ⅳ やCM*よりもIBM3081と似ている。

特に我々が重要と考えているのは、いろいろなプログラム言語を使ってシステムを書き、それらを同時に走らせて仕事をするやり方である。具体的にVLSI・CADを例に考えてみよう。今ある回路を設計し、それが論理的に正しく動作するかテストする仕事を考える。まずレイアウトからトランジスターの網が取り出される。これにいろいろなテスト用入力を入れてシミュレーションを行ない、出力を見て正しく動作しているかどうか調べる。ここで論理シミュレーションは良く定式化された問題で、出来るだけ速く動くのが良い。そこで、これらはFORTRANとかCの様に低レベルで能率よく書ける言語を使って書くのが良い。一方テスト入力を作る部分はフレキシブルに書ける事が能率の良さより重要なので、LISPの様にインタープリター向き言語が適している。

多くのプロセスが協力して一つの問題を解決するのは重要であるが、さらに違った言語で書かれたプロセスが協力し合うのが良い。この為各言語が別々のプロセサーの上で動き高速に通信できるのみならず、言語用特殊プロセサーを持った不均質な(heterogeneous)なシステムが望ましい。種々なプロセサーをつなげるには、バスのプロトコールを統一せねばならず問題が

多いが、「けやき」ではキャッシュを一段置いてあるので、メモリーバスのプロトコールを統一するのは簡単である。

2－3 高多重度マルチプロセサー

しかしなんといってもキャッシュメモリーの最大効果は、メモリーバスのトラフィックを減らし多くのプロセサーを付けられる事である。第5節で詳しく述べるが、Write-through のキャッシュを付けたマルチプロセサーでは4～8台の接続が限度であるが、「けやき」では100台までの接続が可能である。バス結合マルチプロセサーの電気的限界が100台[5]なので、この方式は目的を完全に満足する。

[3] キャッシュの外部プロトコール

「けやき」は完全にトップダウンで設計されている。キャッシュは2つのバスにつながっていて、1つはプロセサーバスで、もう1つはメモリーバスである。

3－1 プロセサーバス・プロトコール

キャッシュはプロセサーから次の命令を受け結果を返す。以下ではプロトコールを

func(arg) returns (result) exception (fault)

の様に書くが、この意味は、func というプロトコールがプロセサーからキャッシュへ送られ、同時に arg がデートして送られる。キャッシュからプロセサーへ結果として result が返って来る。異常例外として fault が返る事もある。プロトコールは

Fetch (va) returns (word) exception
(PageFault, ParityError)
Store(va, word) exception (PageFault, ParityError)
FetchBus (va) returns (word) exception
(PageFault, ParityError)
FetchMap(vp) returns (rp, flags) exception (ParityError)
StoreMap(vp, rp, flags) returns (oldRp, oldFlags)
exception (PageFault, ParityError)

(a) Fetch
仮想番地 va を貰って、そこの語を返す。ページフォールトかパリティーエラーが検出されると命令は実行されずに例外が行る。

(b) Store
仮想番地 va へ語 word を格納する。

(c) FetchBus
まずメモリーバスを獲得してから仮想番地 va にある語を返す。メモリーバスはこの命令が完了しても手放されず、次に Fetch か Store が実行されてはじめて手放す。

(d) FetchMap
仮想記憶用の vp、rp 対称表にもとずいて、vp を与えると、対応する rp と flags を返す。

(e) StoreMap
仮想記憶用 vp、rp 対称表を書き変える。

3－2 不分割操作 (atomic operation)

多プロセス間の同期等を実現する為には、リード・モディファイ・ライトを分割不可能な命令として実現できる事が十分条件である。「けやき」マルチプロセサーでは、メモリーバスが単一のリソースなので、word ← FetchBus(va); Op(word); Store(va, word) のシーケンスを実行する事により、リード・モディファイ・ライトが実現できる。ここで注意しなければならないのは、他のすべてのプロセサーもこの番地へ書き込む時は、FetchBus を使わなければならない事である。

3－3 ページ・スワッピング

仮想記憶を扱う命令として FetchMap と StoreMap が与えられたが、これらの命令でページのスワッピングが出来る事を証明しなければならない。

まず、スワッパーだけが使う仮想ページ vp_0 をもうける。他の誰もこの仮想ページをアクセスしない事はソフトで保障する。今仮想番地 vp_1 を追い出し、vp_2 を取り入れる。これを実現する命令の順序は、

```
( rp1 , flags1 ) ← FetchMap(vp1)
(  ,  ) ← StoreMap (vp0, rp1, flags1)
-- rp1 ページをディスクの vp1 ページに対応
-- する所に書き込む。次に rp1 ページに
-- vp2 ページの中味をディスクから読み
-- 込む。
```

(,) ← StoreMap(vp_2, rp_1, newFlags)

3-4 メモリーバス・プロトコール

メモリーバスには次の命令が送られる。ここで func(arg) returns(result) と書かれたプロトコールの意味は、マスターキャッシュ（バスの制御をアービターから貰ったキャッシュ）が func というプロトコールとデータ arg をメモリーバスに送ると、result が返って来るという意味である。

```
ReadBlock(ra) returns(block, duplicate)
ReadSmallBlock(ra) returns
                (smallBlock, duplicate)
WriteBlock(ra, block)
Update(ra, word)
ReadMap(vp) returns (rp, flags)
WriteMap(vp, rp, flags) returns(oldRp, oldFlags)
IORead(ra) returns(word)
IOWrite(ra, word)
```

(a) ReadBlock

実記憶番地 ra がメモリーバスに送られ、対応する block（4ワード）が返ってくる。他の cache にコピーがあれば、そこから block が送られて来て、duplicate は真である。どこにも無ければ主記憶から送られてきて、duplicate は偽である。

(b) ReadSmallBlock

実記憶 ra に対応する block のうち、ra 番地以外の3語が返ってくる。block の送り主と duplicate との関係は ReadBlock の場合と同じ。

(c) WriteBlock

実記憶番地 ra で示される block を主記憶に書き込む。

(d) Update

実記憶番地 ra に書かれる新しいデータを送る。他のキャッシュでこの番地を含んでいればデータを読み込んで block を書きかえる。同時に dirty flag を reset する。

(e) ReadMap

vp に対応する rp と flag を返す。

(f) WriteMap

vp に対応する値を rp, flags とする。

(g) IORead

ここでは記憶写影型入出力を使っているので、Fetch は入力をする。

(h) IOWrite

Store で出力をする。

[4] 外部プロトコールの機能

外部プロトコールは前節であげたので、ここではそれらの機能の仕様をあげる。この仕様はALGOL系のプログラム言語で書かれており、これをほとんどそのままPASCALに記述し直して、第6節の論理シミュレーションを行なった。

4-1 プロセサーバス・プロトコール

```
• Fetch(va) returns(word);
    (rp, fault, IO) ← TLBLookup(va);
    if fault then (rp, IO) ← GetTLB(va);
    if IO then word ← IORead(ra)
    else {
      (hit, baddr, duplicate) ← CacheALookup(ra);
      if ¬hit then {
        baddr ← NextVictim();
        if CacheA[baddr].occupied then {
          vra ← CacheA[baddr].ra;
          WriteBlock(vra, CacheD[baddr])}
        else CacheA[baddr].occupied ← TRUE;
        (CacheD[baddr], CacheA[baddr].duplicate)
            ← ReadBlock(ra);
        CacheA[baddr].ra ← ra;
        CacheA[baddr].dirty ← FALSE };
      word ← CacheD[baddr][Twobits(ra)] };
```

Fetch のうち代表的な4つのケースは図4-1、図4-2、図4-3、図4-4にあげる。

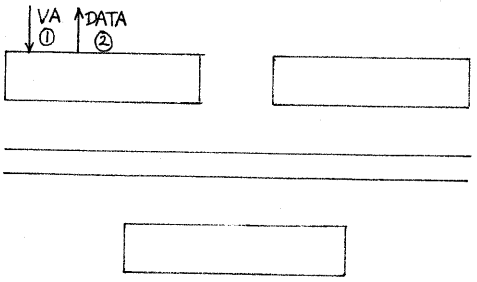


図4－1　Fetchがhitした場合.
①、②は時刻の順序を表わす。

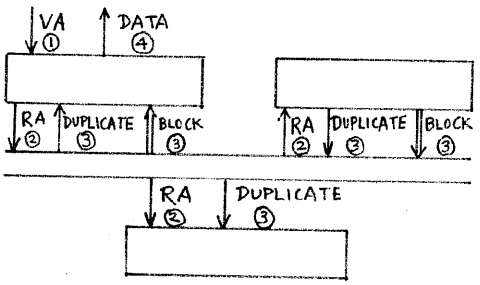


図4－2　Fetch miss/victim clean/
他のキャッシュ上にデータがある場合

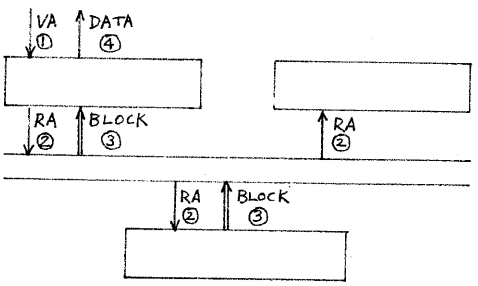


図4－3　Fetch miss/victim clean/
他のキャッシュ上にデータが無い場合

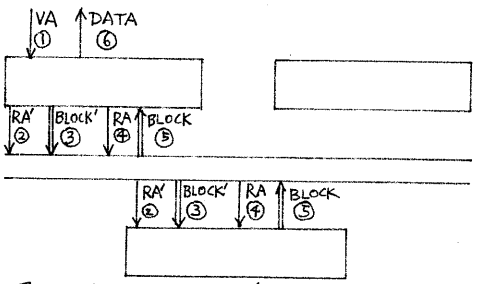


図4－4　Fetch miss/victim dirty/
他のキャッシュ上にデータが無い場合

```
• Store(va,word);
    (rp,fault,IO) ← TLBLookup(va);
    if fault then (rp,IO) ← GetTLB(va);
    if IO then IOWrite(ra,word)
    else {
      (hit,baddr,duplicate) ← CacheALookup(ra);
      if ¬hit then (baddr, duplicate) ←
            LoadSmallBlock(ra);
      if duplicate then Update(ra,word);
      CacheA[baddr].dirty ← TRUE;
      CacheD[baddr][Twobits(ra)] ← word };

  LoadSmallBlock(ra) returns (baddr, duplicate);
    baddr ← NextVictim();
    if CacheA[baddr].occupied then {
      vra ← CacheA[baddr].ra;
      WriteBlock(vra, CacheD[baddr])}
    else CacheA[baddr].occupied ← TRUE;
    (CacheD[baddr], CacheA[baddr].duplicate)
          ← ReadSmallBlock(ra);
    CacheA[baddr].ra ← ra;
    CacheA[baddr].dirty ← FALSE;
```

Storeのうち1つの場合を図4－5にあげる。

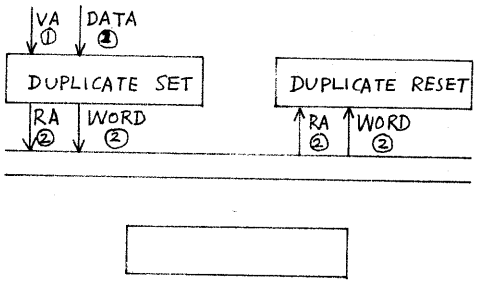


図4－5　Write hit/他のキャッシュ上にコピーが有る場合

```
• FetchBus(va) returns (word);
```
Fetchと同じだが、まず最初にメモリーバスを獲得し、実行が終わっても手放さない。

```
• FetchMap(vp) returns (rp,flags);
    (fault, rp, flags) ← FetchMapFromTLB(vp);
    if fault then {
      (rp, flags) ← ReadMap(vp);
      LoadTLB(vp,rp,flags)};
```

```
• StoreMap(vp, rp, flags) returns (oldRp, oldFlags);
    (fault, rp, flags) ← FetchMapFromTLB(vp);
    if ¬fault then StoreTLB(vp, rp, flags);
    (oldRp, oldFlags) ← WriteMap(vp, rp, flags);
```

4-2 メモリーバス・プロトコール

```
• ReadBlock(ra) returns(block, duplicate);
    (hit, baddr, duplicate) ← CacheALookup(ra);
    if hit then Return(CacheD[baddr], TRUE)
    else Return(NEUTRAL, FALSE);
```

ここでNEUTRALは、三状態出力ドライバーの中間値を示す。

```
• ReadSmallBlock(ra) returns (smallBlock,
                              duplicate);
```

ReadBlockと同じだが、4語のかわりにraを除いた3語をかえす。

```
• WriteBlock (ra, block);
```

何もしない。

```
• Update(ra, word);
    (hit, baddr, duplicate) ←
          CacheALookup(ra);
    if hit then {
      CacheD[baddr][Twobits(ra)] ← word;
      CacheA[baddr].dirty ← FALSE};
```

```
• ReadMap(vp) returns(rp, flags);
    (fault, rp, flags) ←
          FetchMapFromTLB(vp);
    if fault then Return(NEUTRAL, NEUTRAL)
    else Return(rp, flags);
```

```
• WriteMap(vp, rp, flags) returns
                    (oldRp, oldFlags);
    (fault, oldRp, oldFlags) ←
          FetchMapFromTLB(vp);
    if fault then Return(NEUTRAL, NEUTRAL)
    else {
      StoreTLB(vp, rp, flags);
      Return(oldRp, oldFlags)};
```

[5] 性能シミュレーション

性能シミュレーションの意味は，
- Cacheの大きさの決定
- 性能を左右する要因の明確化
- 性能の限界点の算出

などがある。これらは詳細な設計にはいるまえに必要なものである。

性能シミュレーションでの着眼点は，
- バスの混み率とバスの応答時間
- Cache動作モードとバスへの負荷
- Cache動作モードの起動確率

の3点である。

Cacheの大きさには，コストを度外視しても最適値がある。かりに，全メモリ空間と同じだけCacheがあったとすると，1つのCacheが，Storeオペレーションをするたびに，バス通信が起きてしまう。このときバスへの負荷は1/5程度になるにとどまる。

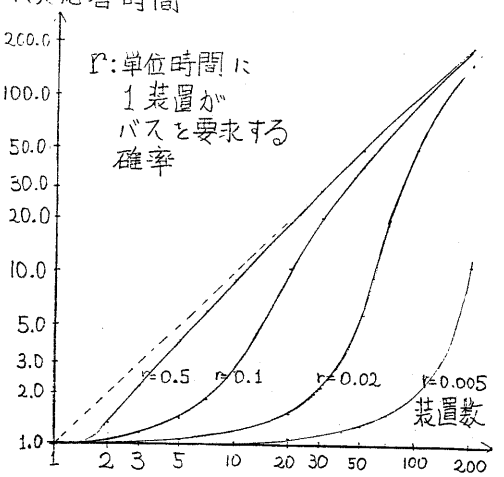
5-1 バス応答時間

問題は「各装置が1単位時間にバスを要求する確率がrであった。装置がN台あるとき，バスを要求して操作が終るまでの時間T(r,N)を求めよ」である。

装置のバスへの要求は独立に起き，バスは単位時間に1台だけ操作を完了するとする。これは，標準的な待ち行列の問題である。

定常状態で待ち行列の長さがnである確率をa_nとして $T(r,n)=\sum_{m} a_m \cdot n+1$ と求る。

バス応答時間



5-2 Cache動作モード

表5-1にまとめる。

	WRITE	HIT	DUPLICATE	DIRTY-V
1	○	○	×	
2	○	○	○	
3	○	×		×
4	○	×		○
5	×	○	×	
6	×	○	○	
7	×	×		×
8	×	×		○

(注)空白はDon't Care

(表5-1 動作モード)

- Writeとは処理装置がCacheに対し書きこみをする場合のこと
- Hitとは，処理すべきデータがCache内部にある場合のこと
- Duplicateとは，処理すべきデータが，他のCacheにもとりこまれている場合のこと
- Dirty-Vとは，Hitしない場合であって，データのとりこむべき場所に，変更をうけたものがある場合のこと。

それぞれの場合を組み合わせて，合計8通りの動作モードを考える。

5-3 各モードの時間

Cacheの各操作ごとに要する時間t_mは

内部処理時間＋バス使用回数×T(r,n)

と仮定することにする。

	Update		Write-through		Without	
	IN	BUS	IN	BUS	IN	BUS'
1	1.0	0	0.0	1	0.0	1
2	1.0	1	0.0	1	0.0	1
3	1.0	1	0.0	1	0.0	1
4	1.0	2	0.0	1	0.0	1
5	1.0	0	1.0	0	0.0	1
6	1.0	0	1.0	0	0.0	1
7	1.0	1	1.0	1	0.0	1
8	1.0	2	1.0	1	0.0	1

(表5-2 動作時間)

今回は「内部処理時間＝バスサイクル」と仮定し，Cache単独での性能向上はないものとした。

- Updateとは，今回行う方法である。
- Write-throughとは，書きこみのとき，常に全体にデータを放送する方式のことである。
- Withoutとは，Cacheを用いずに，処理装置をバスに直結する方式のことである。
- Imとは，内部処理時間をバスサイクル＝1.0として測った時間である。
- Busとは，バスの使用回数である。

5－4 各モードの確率

モードmがおきる確率をP_mとする。Write，Hit，Duplicate，Dirty-Vの場合が独立に起きると仮定し，P_mはこれらの積で与えられるとする。

(例) $P_i = Wr \times Hi \times (1 - Du)$

Write率，Hit率，Dirty率は文献より典型値を求めた。(表5－3，表5－4) これらの典型値は次のとうりである。

$Wr = 0.2$
$Hi = 0.99$ } ただしCache＝8K Byte
$Di = 0.5$

Prog. \ Size	2K	4K	10K	20K	40K	80K
FFT	93.0%	98.0%	99.8%	99.89%	99.90%	?
APL	91.0%	94.7%	99.2%	99.4%	99.65%	99.78%

32 Byte BLOCK, 64SET, SET ASSOCIATIVE CACHE

Cache Size vs. Hit Ratio

(表5－3 Hit率 文献3)

Prog.	Write	Dirty-V	Hit	note
Beads	10.5%	16.3%	99.27%	VLSI Design
Placer	18.7%	65.5%	99.82%	VLSI Design
TEX	15.2%	34.9%	99.55%	Text

4K×16 bit = 8K Byte Language = Mesa

(表5－4 Write率，Dirty-V率 文献4)

Duplicate率は下のように近似する。

$$1 - Du = \left(1 - \frac{\text{Cache 空間}}{\text{Memory 空間}}\right)^{N-1}$$

N：装置数

このモデルを説明する。まずN＝1のとき，すべて Not Duplicate である。Nが1つ増えるたびに Not Duplicate は ある比率で減少する。その比率が指数の内部である。

5－5 系の性能

Acが処理装置の外部アクセスの確率とすると，

$r = \sum_m P_m \times \{\text{バス使用回数}\} \times Ac$ である。

これによりT(r,N)を求め，

性能$= N \cdot \sum_m P_m \times t_m$ を計算する。

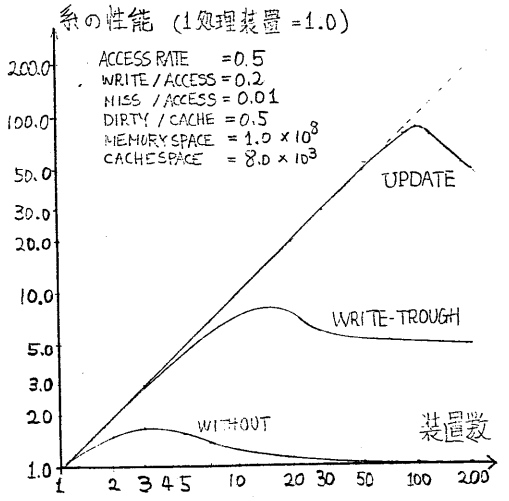
	P_m	r	T(r,N)
1	0.198	0.0087	1.71
2	0.001		
3	0.001		
4	0.001		
5	0.785		
6	0.005		
7	0.004		
8	0.004		N=50

(表5-5 P_m, r, T(r,N))

表5－5は，それぞれの典型値である。図5－6に性能のグラフを示す。

5－6 性能の限界点

この系が最高の性能をあげるのは，Cacheの大きさ4K～16Kバイトのときで，50～100台の処理装置が，各々の能力の90%以上を発揮で



(図5－6)

きる。Cacheが小さいと，Hit率の減少により台数が制限され，大きいと，Duplicateの増加により，Update 操作が増え，台数が制限される。

電気特性による台数の上限は100 台ほどという研究報告（文献５）もあり，この方式はそれとほぼ同じところに上限がある。

Cacheの動作において高速にすべきモードは１と５であり，この動作速度が２倍になれば，系の性能は１．６倍ほどになる。

[6] 論理シミュレーション

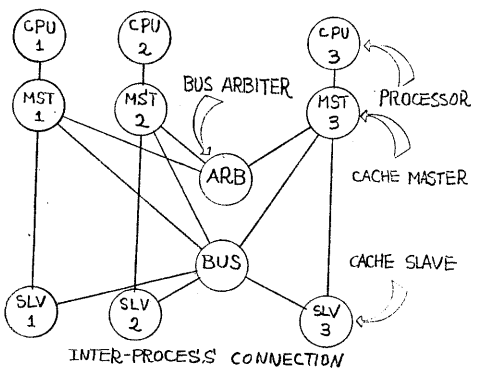
論理シミュレーションを実現するには，

- 多プロセスのプログラム
  - Shake - Hand による同期
  - 共通変数による通信

程度のレベルの道具を用いる。モニタ等の高レベルの同期手段は，今回の目的にそぐわない。このレベルで記述することで，ソフトとハードの対応がつき，より正確な記述ができる。論理シミュレーションの意味は，

- 正しい動作をすることの確認
- バス－Cache間の信号線の決定
- Cache－処理装置間の信号線の決定

などがある。特にテスト＆セット命令が実現可能であることを保障しておかないと，この系の制御が不可能である。事実，初期のCache設計において，この動作に誤りがあり，それがこのシミュレーションで発見され訂正された。



（図６－１） プロセスの割りつけ

６－１ プロセスの割りつけ

今回は（図６－１）のように割りつけた。用語を説明する。

- Processorは，処理装置に対応し，データを読み，加工し，書き出す。
- Masterは，バスの権利をとり，バスのコマンドを発行する。
- Slaveは，バスを監視し，バスのコマンドに応対する。
- Arbiterは，バスの要求を整理し，一度に一つだけにバスの権利を発行する。
- Busは メモリとバス線に対応し，バスのコマンド全体に伝達し，メモリに要求があればそれを実行する。

Cache装置１台にはプロセス２個が対応する。処理装置１台を追加するとプロセスは３個追加することになる。

６－２ 道具

以下の３つの道具を考えた。

(a) UNIX + C
(b) Concurrent Lisp
(c) Pascal（Standard）

実際に使用したのは(c)である。(a)は同期の手段が今回の目的にそぐわない。(b)は記憶素子の記述力が弱い。(c)はマルチプロセスのサポートがない。それぞれ大きな欠点がある。試行の結果，Pascal のもつ Case 文，With 文などが役にたち，(c)が最も素直な記述ができた。

６－３ 結果と課題

テスト＆セットの誤りが発見できたことで，シミュレーションの目的は達成されている。しかし今回のモデルは単純なものであり，実際の操作のうち省略したものがある。これから，それらをとりいれたシミュレーションを行う必要がある。

省略されたものとは。仮想記憶方式。入出力命令である。これらについて初期設計は終了しているので，これを今回と同じ手法を用いて，誤りの訂正と動作の保障をしなければならない。

[7] 結論

記憶共有型MIMDマルチプロセサーのメモリーバス効率を大幅に上げる方式を考案し、その論理的仕様を説明した。解析的に性能評価を行い、100台程度の並列接続が可能な事を示した。現在市販されているLSIドライバーを使ってバス結合で接続できるプロセサー数の限界とも一致するので、これ以上の論理的な工夫は必要ないと思われる。

また、複雑な動作特性を持つ計算機の高級仕様言語による設計方法を確立した。まず、高級仕様言語で外部プロトコールを定義し、その機能を内部構造とは無関係に記述し、これを走らせることによって論理シミュレーションを行ない、プロトコールのディバッグを行なった。

[8] 文献

(1) 鈴木則久、戸村哲、和田英一：“三四郎プロジェクト”，情報処理学会第25回大会予稿集，1N-11，1982年 10月.

(2) SUN Microsystems, Inc. The SUN Workstation Architecture, July 1982.

(3) Smith, A.J. "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through," Journal of the ACM, Vol. 26, No. 1, January 1979, pp. 6-27.

(4) Clark, D., et al. "The Memory System of a High-Performance Personal Computer," The Dorado: A High-Performance Personal Computer, CSL-81-1, Xerox Palo Alto Research Center, January 1981.

(5) 小林信裕、離散系シミュレータKDSS-Iの実装に関する研究、卒論、慶応大学理工学部電子工学科、1979.

(6) Swan, R.J., et al. "Cm* — a Modular Multi-microprocessor," Proc. AFIPS 1977 NCC, pp. 637-644, Vol. 46, AFIPS Press, Arlington Virginia, 1977.