計算機アーキテクチャ 89−25
（1991. 7. 19）

バス結合型並列計算機のプロセッサ間交信評価システム

寺沢 卓也† 天野英晴†
†慶應義塾大学 理工学部

プロセッサ間交信評価システム MILL(Multiprocessor system Instruction Level simuLator) は CPU の動作を命令レベルでシミュレートすることにより並列計算機の交信機構の詳細を評価することができる。並列計算機テストベッド ATTEMPT 上で稼働したプログラムはそのまま、MILL 上で様々なハードウェア仕様の下でシミュレーショトすることができる。MILL はプロセッサ部、キャッシュ部、同期機構部、バス/共有メモリ/時刻管理部の 4 つの独立したビルディングブロックから構成され、ブロックの交換により全く構成の異なるアーキテクチャのシミュレーションが可能である。

An Evaluation System of Inter-processor Communication for Bus-connected Multiprocessors

Takuya Terasawa† Hideharu Amano†
† Faculty of Science and Technology, Keio University

MILL(Multiprocessor system Instruction Level simuLator) is an execution-driven multiprocessor simulator for evaluating inter-processor communications. Through the instruction level simulation, details of the communication system can be analyzed. The program which is working on the multiprocessor test-bed ATTEMPT can be simulated directly on MILL under various types of hardware specifications. MILL consists of four building-blocks: Processor, Cache, Synchronization mechanism, and Bus/Shared-memory/Time-manager. By replacing the above blocks, different types of multiprocessor can be treated.

1 はじめに

並列計算機の開発時にはアーキテクチャ構成上の様々なトレードオフを考慮する必要がある。このためには並列計算機上で実際的なアプリケーションプログラムを走らせた場合の統計データが必要になる。我々はこのような統計データの採集を目的としたテストベッド ATTEMPT(A Typical Testing Environment of MultiProcessor sysTems) の開発を進めている。現在、プロトタイプ第一号機（以後ATTEMPT0と呼ぶ）が稼働しており、様々な興味深いデータが得られている。

しかし、並列処理においてはプロセッサ数やハードウェアの実行速度が変わるとプログラムの挙動が変化するため、テストベットを用いた評価結果はそのテストベットとプロセッサ数、ハードウェアの実行速度がある程度似ているシステムにしか適用できない。

そこで我々はCPU の命令レベル、交信機構の詳細をシミュレーションするシステム MILL(Multiprocessor system Instruction Level simuLator) を開発している。MILL は本質的にはあらゆる並列計算機のシミュレーションが可能であるが、現状では ATTEMPT0 の仕様に完全に合わせてある。すなわち、ATTEMPT0 上のプログラムはほとんどそのままの形で稼働する。MILL 自体の実行速度は極端に遅い (シミュレーション対象の ATTEMPT0 が約 10000 倍速い) ため、MILL と ATTEMPT0 は通常セットで用いられる。すなわち、ATTEMPT0 上でプログラムの挙動を十分解析し、可能ならば問題となるプログラムの核となる部分を取り出してから MILL を用いて、プロセッサ数の増加後、ハードウェアの変更後のシミュレーションを行ない結果を比較する。

また、MILL には時刻管理を行なわない高速版 MILL/D があり、ATTEMPT0 上のプログラムのデバッグに用いることができる。

本報告では MILL と ATTEMPT0 から成る並列計算機設計環境について述べる。

2 並列計算機シミュレータ MILL

並列計算機の評価法としては以下の手法が知られている。

- 理論解析、評価モデルによる解析
- アドレストレース [1][2][3][4]
- 命令レベルシミュレーション [5]
  - 対象とする並列計算機と同じ種類の CPU を持つマシン上で、共有資源アクセス時にトラップをかける。
  - CPU の命令レベルでシミュレートする。

理論解析、評価モデルによる解析は、アーキテクチャ全体の設計には重要ではあるが、細部のトレードオフを比較する場合、有効ではない。アドレストレースはプロセッサ数を変えることができない点が問題であり、トラップに基づく方法は、CPU の種類が制限される問題点がある。

このため、MILL ではインストラクションの実行のレベルでシミュレートを行ない、インストラクションのフェッチや、オペランドアクセスによるキャッシュアクセスなどを実際に起こさせる構造を採用した。この方法の最大の問題点は実行の遅さにあり、対象とするプログラムの挙動が全くつかめていないと、有効なデータをとることが難しい。しかし、MILL の場合、ユーザは ATTEMPT0 によって対象を絞ってシミュレーションすることができるため、この問題点はある程度補われている。

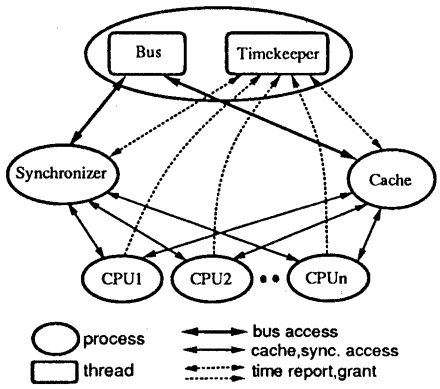

図 1: MILL の全体構成

2.1 対象マシン・モデルと使用形態

MILL は本来、対象とする並列計算機の形態に依存しないが、現在は ATTEMPT0 とセットで用いることを前提としているため、単一バスのバス結合型並列計算機をシミュレートするように作られている。仮想マシンのアドレス空間等は ATTEMPT0 と完全に一致しているので ATTEMPT0 上のプログラムはほとんどそのまま稼働する。

まず、MILL のユーザは ATTEMPT0 のモニタ機能を利用してプログラムの挙動を解析し、対象とするシミュレーションのパラメータを決定する。現在の MILL は、バス結合型マルチプロセッサを対象としているため、プロセッサ数、バスの転送能力、アービトレーション時間、キャッシュのサイズ、ブロック（ライン）サイズ、セット数、プロトコルと各ステップの実行時間等は、パラメータの形で簡単に選択できる。全く異なった交信形態や、異なる種類の CPU をシミュレートする場合は MILL 自体に手を入れる必要がある。MILL はビルディングブロック構成を取っており、各部の入れ換えは比較的容易である。

また、ATTEMPT0 上のプログラムのデバッグを MILL 上で行なうことが可能である。この場合問題なのがシミュレーションの遅さであり、MILL ではこの目的の高速版 MILL/D を用意している。MILL/D は時刻管理の部分を持たず、その実行速度は評価用の MILL の数十倍である。

3 MILL の詳細

MILL は以下の 4 つのブロックから構成されている。

1. プロセッサ部
2. キャッシュ部
3. 同期機構部
4. バス/共有メモリ/時刻管理部

これらのブロックの入れ換えにより、異なるアーキテクチャのシミュレーションが可能である。具体的には、CPU が異なる場合プロセッサ部を、バスをスイッチ結合に変更する場合バス/共有メモリ/時刻管理部をそれぞれ入れ換える。

MILL は SPARC Station 上で稼働しており、上記の 4 つのブロックはそれぞれ独立した UNIX プロセスとして実現され、それらの間の通信にはソケット（BSD UNIX の socket）を用いている。

3.1 プロセッサ部

プロセッサ部は対象の CPU を命令レベルでシミュレーションする。現在の MILL では ATTEMPT0 に合わせて MOTOROLA 社の MC68000 シリーズのインストラクションを実行するが、コンパイラ、アセンブラ、ローダが存在すれば、いかなる CPU でも同様の手法でシミュレーション可能である。

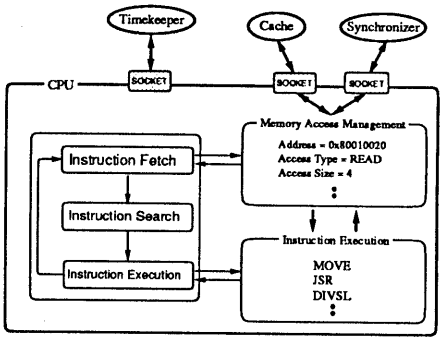

図 2: プロセッサ部

CPU プログラムはアドレッシングモード解析部、メモリアクセス管理部、実行制御部、各インストラクション実行部、その他の部分からなる。以下、シミュレーションの実行を追いながら、各部の機能について解説する。

1. シミュレーション対象の CPU のコンパイラ、アセンブラ、ローダを用いて、対象プログラムのオブジェクトを生成する。現在の MILL では MC68000 シリーズの CPU を用いているワークステーション上でこの作業を行なう。このオブジェクトファイルからテキスト、データの大きさ等の情報をとりだし、メモリを確保してオブジェクトをロードする。

   CPU のレジスタなどの資源は UNIX のプロセスコントロールブロック等で使われている形式に類似し

た構造体により表現する。これらはイニシャライズルーチンで初期化する。また、このルーチン内でスタック領域の確保も行なう。

2. プログラムカウンタの指す番地から実行が開始される。具体的にはメモリからインストラクションをフェッチし、そのインストラクションを実行する関数を呼び出す。呼ばれた関数内ではまず、インストラクション自身の一部や、拡張ワードを読み出してアドレッシングモード解析ルーチンに渡し、オペランドの実効アドレスを得る。

3. 実効アドレスが算出されるとインストラクション自身の動作が始まる。メモリアクセスが起きる場合はメモリアクセス関数を介して行なわれる。この関数内ではメモリの管理はアドレス順に連結された固定長ブロックのリストの形で行なわれる。共有メモリへのアクセス、同期機構へのアクセスもこのルーチン内で判定され、適当な処理が行なわれる。メモリアクセスはこの関数内で集中的に管理される。

4. 一つのインストラクションの実行が終了すると実行制御部に戻り、次のインストラクションフェッチをして`__exit`が呼ばれるまでこれを繰り返す。入出力等のシステムコールはUNIXライブラリ関数syscallを用いて実際に（シミュレータが）システムコールを行なう。

3.2 キャッシュ部

キャッシュ部はCPUやバスから独立した一つのUNIXプロセスである。

キャッシュ間ブロック転送、無効化などを行なうためにはバスをスヌープする必要があるが、バスプロセスを複数のキャッシュプロセスが監視するのは困難である。そこで、発想を逆転させてキャッシュがスヌープという受身の状態をとるのではなく積極的にバスに働きかけるようにする。

一例として、共有ブロックに対して書き込みが起き、他のCPUのキャッシュを無効化する場合を考えよう。この場合、アドレスをバスに出して他のキャッシュが反応するのではなく、書き込みを起こしたCPUのキャッシュがバスを獲得してそのブロックを共有している他のキャッシュの対応するブロックを無効にするところまでを行なう。この時問題になるのはタグメモリの共有とタグメモリへの書き込みの排他制御であるが、これはSun Microsystems社のSunOSが提供しているライトウェイトプロセス（Lightweight Process: LWP）ライブラリを利用して実現している。

具体的なキャッシュプロセスの構造は次のようになっている。シミュレーションに参加するCPUプロセスの数をnとする。キャッシュプロセスはn個のCPUプロセスに対して一つだけ存在する。キャッシュプロセス内にはn個のコンテキストが存在する。このコンテキストとは1CPU分のキャッシュメモリと、そのディレクトリメモリ、パケットを入れておくバッファなどから構成される構造体である。

次に、キャッシュプロセスの動作について解説する。

- CPUからのキャッシュに対するアクセスはCPUのメモリアクセス管理関数を介してキャッシュプロセスに届けられる。このプロセス間交信はソケットを介して行なわれる。この時送られる情報はアクセスアドレスやそのCPUプロセスのpid、アクセスの種類などである。キャッシュプロセス内にはCPUからのアクセスに対する窓口となるスレッドが常に存在してアクセスを受けつける。窓口スレッドはアクセスを受けつけるとパケット中のpidからそのCPUのコンテキストを選択し、パケットのバッファへのコピーなどの処理をした後キャッシュスレッドを生成する。

- キャッシュスレッドは生成される時に引数としてコンテキストの番号を渡される。これより後の処理はこのキャッシュスレッドとアクセスしたCPUとの間で進められ、窓口スレッドは窓口の仕事に戻る。このようにして、一時に最大n個のキャッシュスレッドが並列（並行）にキャッシュプロセス中に存在する事が可能で、これらのキャッシュスレッドはn個のCPUのコンテキストを共有している。

- 一つのキャッシュスレッドはコンテキスト中のパケットのコピーからアクセスアドレス、サイズ、Read/Write、リクエストidなどの情報を読み出し、

ヒット/ミスの判定を行なった後、指定されたプロトコルに従って無効化や、ブロック転送などの処理を行なう。

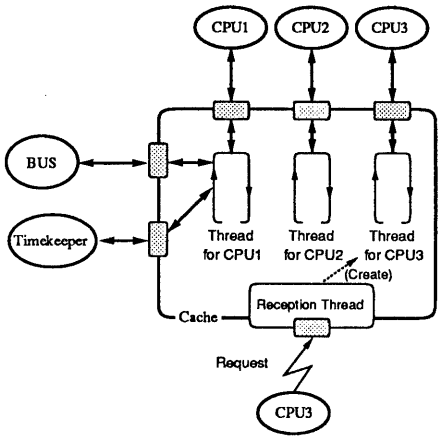

図 3: キャッシュ部

3.3 同期機構部

同期機構としてはシンクロナイザ [6] の他に、キャッシュ中に 取り込まれる/取り込まれない 共有メモリ領域を使った Test&Set などをサポートする予定であるが、現在実装されているのはシンクロナイザのみである。

シンクロナイザは一種のブロードキャストメモリに基づく同期機構で、プロセッサ間割り込みを伴う書き込み、バスを用いない読みだし、同期命令 Fetch&Dec 等が備えられている。

各 CPU に分散されているためキャッシュと構造が良く似ており、やはり LWP ライブラリを用いて実装されている。キャッシュと異なる点は共有メモリに対するアクセスがないことである。シンクロナイザに対するアクセスは Read、Write、Fetch&Dec の 3 つに分けられるが、Read 以外はバスを使うオペレーションであるため、キャッシュにおける共有メモリアクセスと同様にソケットを介してバスプロセスにリクエストを送り、バスを獲得する必要がある。

3.4 バス/共有メモリ/時刻管理部

バス（及び共有メモリ）アクセス処理部とシミュレーション全体の時刻管理を行なう部分は一つの UNIX プロセス中の 2 つのスレッドとして実現されている。両者が 1 つのプロセスにまとめられている理由は、バスアクセス処理部がアービトレーションの開始等、バスアクセスを起こした CPU の状態を遷移させる際に、各 CPU の時刻情報を利用しているためである。

3.4.1 バス/共有メモリ部

バススレッドはキャッシュあるいはシンクロナイザからの要求に応えてバスオペレーション、共有メモリ・ブロックの出し入れを行なう。

現在の MILL では Futurebus[7] に基づいたバスモデルを採用している。アービトレーションは現在のバスマスタがバスを使用している期間中に、オーバラップして行なことができる。これによって次のバスマスタが決定され、現在のバスマスタがマスタ権を放棄した時点で次のバスマスタに切り替わる。バスリクエストを行なったプロセッサの状態は以下の 4 状態にわかれる。

1. アービトレーション中
   同時刻にバスリクエストを出した 1 つ以上の CPU、あるいは、アービトレーション待ち状態から解放された CPU がアービトレーションを行なっている状態。1 回のアービトレーションに要する時間はパラメータとして与えられる。

2. アービトレーション待ち
   バスリクエストを出した時刻には、既にアービトレーションが開始されているか、次のバスマスタが決定しているため、次回のアービトレーションが可能となるまで待っている状態。次のバスマスタが決定している状態ではアービトレーションを行なうことは出来ないので、現在のバスマスタが、決定された次のバスマスタにマスタ権を譲渡した時点で、待ち状態から解放され、アービトレーションを行なうことが出来る。

3. バス使用中
   バスを占有して作業を行なっている状態。バス要求の

種類に応じて、バスを占有している時間はパラメータとして与えられる。

4. バス使用待ち
   アービトレーションに勝利して次のバスマスタとなり、現在のバスマスタがバスを解放するのを待っている状態。アービトレーションに勝利した時点でバスマスタが存在しなければ即座にバスマスタになることが出来る。

これらの状態の遷移を管理し、プロセッサ毎にアービトレーション参加回数、バス占有時間、バス待ち時間等の情報を得られるようにログを取る。

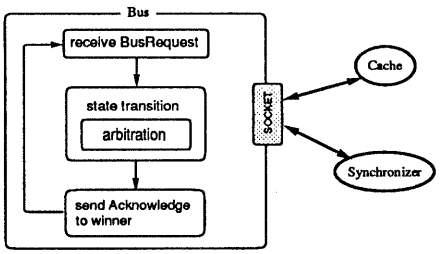

図 4: バス/共有メモリ部

3.4.2 時刻管理部

シミュレーションの時刻管理はCPU、キャッシュ、シンクロナイザの各プロセスから時刻の報告を受け、その時刻の小さい順に実行を許可することによって行なわれる。

- システムにグローバルな時計が一つあり、各CPUはシステムに加わるとその時点のシステム時刻を受けとって自分の時計をその時刻に初期化する。
- 各CPUはキャッシュアクセス、シンクロナイザアクセス等、他のCPUとのインタラクションを生じるイベントを実行しようとする場合、専用ソケットを介して時刻管理スレッドにそのイベントの生起する時刻を報告し、実行許可を待つ。この報告は実際にはCPUが行なう訳ではなく、アクセスが生じた時点でキャッシュプロセス、あるいはシンクロナイザプロセスが行なう。
- 時刻管理スレッドはその時刻に存在する全CPUの時刻報告を待ち、全て揃うとその中でもっとも時刻の小さいイベントに実行許可を与える。
- 共有資源へのアクセスを行なわないCPUが存在しても、時刻管理スレッドはこのCPU（キャッシュ、シンクロナイザ）からの報告を待ち続けるため、システム全体が停止してしまう。これを防ぐため、一定期間以上外部アクセスを生じないCPUは直接、時刻管理スレッドに対して定期的な時刻報告を行なう。

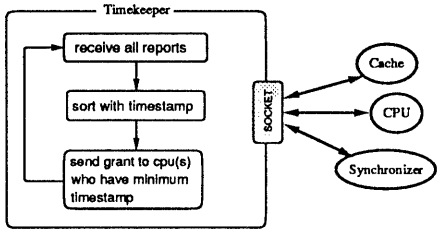

図 5: 時刻管理部

4 ATTEMPT0/1プロジェクト

次に、MILLと共に並列計算機開発環境を構成するテストベットATTEMPTに関して簡単に紹介する。このプロジェクトは慶應義塾大学とアルュメ株式会社により1988年より開始され、現在ATTEMPT0とMILLによる評価結果に基づきATTEMPT1が設計されている。ATTEMPT1は図6に示すように、Futurebus+[8]のサブセットを採用し、キャッシュはプロトコル可変のライトバックキャッシュを用いる。ATTEMPT0同様ローカルメモリは4Mbyte持つが、ジャンパ切替えにより共有メモリに変更できる機能は削除した。

ATTEMPT1に関してMILLは以下の問題点について判断するために使用されている。

- ATTEMPT0の評価結果[9]によると同期機構シンクロナイザはその基板面積の割に有効利用されていない。これを他のより簡単な同期機構でシミュレートできないか？

- ATTEMPT0 では非同期設計に基づく Futurebus のプロトコル制御を同期設計したため転送速度が遅くなった。ATTEMPT1 では、同じく非同期バスの Futurebus+のサブセットを採用するが、どのレベルで同期、どのレベルで非同期設計を用いるべきか？

- ATTEMPT0 は主記憶のアクセス速度が不足した。また、ディスクを持たなかった。ATTEMPT1 ではどのように主記憶を構成し、ディスクと接続すべきか？

これらの設計はまだ決定していないものも多く、後の機会に詳細を報告する予定である。

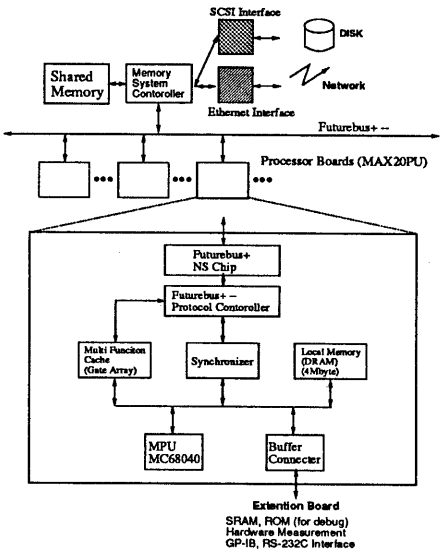

図 6: ATTEMPT1

5 おわりに

現時点では MILL は稼働を開始したばかりで、前節で述べた ATTEMPT1 の設計に関するデータは採集されていない。

参考までにテスト用に作成した共有メモリを用いた 10×10 の整数行列のかけ算を 9CPU（master1,worker8）で計算した結果を次に示す。MILL 上のシミュレーションは SPARC Station2 で約 6 分を要する。（CPU タイムでは master で 4.8 user + 6.9 sys）

pid	hit	miss	%	arb	m/w
23807	374	26	93.50	242	master
23812	247	26	90.47	41	worker
23814	247	26	90.47	41	worker
23816	227	25	90.07	39	worker
23818	227	25	90.07	39	worker
23819	227	25	90.07	39	worker
23820	228	24	90.47	38	worker
23824	247	26	90.47	41	worker
23831	247	26	90.47	41	worker

MILL の現在の実装ではシミュレーションに非常に時間がかかる。これは、インストラクションレベルでシミュレートしていること以外に主に次の理由による。

- ビルディングブロック化をプロセス単位で実現したため、ソケットを用いたプロセス間通信が頻繁に起こり、このオーバヘッドが大きい。

- ATTEMPT0 ではプロセッサ間割り込みを用いたプログラミングか可能であるが、現在の MILL はこの割り込みには対応していない。このため spin-lock せざるを得ない。この場合、spin-lock はシンクロナイザ等で行なわれるので上記のオーバヘッドが特に大きくなる。

- 現在の時刻管理の方法は共有資源へのアクセスを基準に考えられている。このため、例えば、master-worker タイプのプログラムでは master が worker に与える仕事の準備をしている間は、内部的な計算をしている時間が比較的長いために、この CPU の時刻だけが突出してしまう場合がある。この間、worker 達は spin-lock 等で待っているため、各自の時刻の進み方はゆっくりしており、master が準備を終えてもなかなか計算に入れないばかりでなく、master も待たされてしまう。この傾向は master の計算量が多いほど、worker の数が多いほど、強く現れるので CPU 数の多い場合ほどシミュレーション時間も長くなってしまう。

このようにMILLには多くの問題点が残されており、これらの改善が今後の課題である。

6 謝辞

本報告にあたり、数々の御指導を頂いた東京工科大学の工藤知宏講師に深く感謝致します。また、MILLの実装について貴重な御助言を頂いた木村哲朗氏ほか慶應義塾大学計算機科学専攻天野研究室の皆様に感謝致します。最後に、ATTEMPTプロジェクトを支えて下さるアルュメ株式会社の皆様に感謝致します。

参考文献

[1] A.Agarwal, R.L.Sites, M.Horowitz, *"ATUM : A New Technique for Capturing Address Traces Using Microcode"*, Proc. of 13th ISCA pp.119-127, 1986

[2] R.L.Sites,A.Agarwal, *"Multiprocessor Cache Analysis Using ATUM"*, Proc. of 15th ISCA pp.186-195, 1988

[3] C.B.Stunkel, W.K.Fuchs, *"Analysis of Hypercube Cache Performance Using Address Trace Generated by TRAPEDS"*, ICPP pp. I33-I40, 1989.

[4] H.Davis, S.R.Goldschmidt, J.Hennessy *"MULTIPROCESSOR SIMULATION AND TRACING USING TANGO"*, 1990

[5] B.W.O'Krafka, A.R.Newton, *"An Empirical Evaluation of Two Memory-Efficient Directory Methods"*, Proc. of 17th ISCA pp.138-147, 1990

[6] H.Amano, T.Terasawa, and T.Kudoh *"Cache with Synchronization Mechanism"*, Proc. of 11th IFIP, pp. 1001-1006, 1989

[7] *"IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus"*, IEEE Jun., 1988

[8] IEEE P896 Working Group of the Microprocessor Standards Committee, *"DRAFT STANDARD: Futurebus+"*

[9] 鳥居 淳, 天野英晴, "並列計算機テストベッド ATTEMPT の交信機構の評価", 並列処理シンポジウム JSPP '91 論文集, pp.205-212, May 1991