並列計算機のための相互結合網シミュレータ SPIDER

米田 卓司　　若林 正樹　　緑川 隆　　西村 克信　　天野 英晴

慶応義塾大学理工学部
神奈川県横浜市港北区日吉 3-14-1
Tel:045-560-1063 Fax:045-560-1064
email: komeda@aa.cs.keio.ac.jp

相互結合網の性能は, 確率モデルシミュレーションでその概略を予想する一方, トレース駆動シミュレーションにより詳細な解析を必要とする場合もある. しかし, 各々にシミュレータを実装するとユーザの負担が大きい. 相互結合網シミュレーション環境 SPIDER は, この様な負担を軽減させるために相互結合網シミュレータに共通する要素を抽出したライブラリである. ネットワーク資源, トポロジ, ルーティング・アルゴリズム, フロー制御等の結合網に関する諸特性は, 簡単なインタフェースを用意することで, 容易に所望する結合網を記述することができる. このことにより, 様々なシミュレータを簡単に構成して, 性能評価を行うことができる. 本稿では, 相互結合網評価環境 SPIDER の構成について述べた後, 本システムを使用し, 幾つかの結合網の転送能力に関する評価結果を示す.

相互結合網, シミュレーション, ライブラリ, 性能評価

An Evaluation System of Interconnection Networks : SPIDER

T. Komeda　　M. Wakabayashi　　T.Midorikawa　　K. Nishimura　　H. Amano

Keio University
3-14-1 Hiyoshi, Kouhoku, Yokohama, Japan 223
Tel:+81-045-560-1063 Fax:+81-045-560-1064
email: komeda@aa.cs.keio.ac.jp

Although various levels of performance analysis of interconnection networks is required for designing multiprocessors, it is difficult to build both probabilistic simulator and trace driven simulator for designer. In order to reduced the overhead to build such simulators, an interconnection network simulation environment called SPIDER is proposed. This system consists of library programs which includes various common functions in interconnection network simulators. Since each library program provides a simple interface, it is easy to describe various aspects including resources, topology, routing algorithm and flow control. After the introduction of this system, some evaluation examples are presented.

Interconnection Network, Simulation, Library, Performance evaluation

1 はじめに

近年, 高性能を目指した数百・数千台以上のプロセッサ要素から構成される並列計算機に関した研究, 実装が盛んに行われている. このような並列計算機では, 各プロセッサ要素間を結合させる手段として, 直接網や間接網などの相互結合網を用いて構成することが多い. 相互結合網は, トポロジ, ルーティング方式, フロー制御方式等で特徴づけられ, これらのパラメータが適切に選択されない場合, ネットワークがボトルネックとなり, 高い並列処理効率を得ることができない. そこで, 理論解析やシミュレーション, ハードウェアなどによる様々な性能評価が行われる. その中で計算機を使ったソフトウェアシミュレーションによる評価は, シミュレーション対象のハードウェアによる制約を受けないため, 設計や予備評価段階で有効な手段である.

このような目的のため, 様々な結合網やその結合網で構成された並列計算機システムを同じ条件下で評価できる汎用結合網シミュレータの開発が行われている. 1993年に九工大で開発された *INSIGHT*[7], 1995 年に筑波大で開発された *INSPIRE*[6] などがこれらの代表例である. これらのシステムは, 大規模な結合網の評価を目的としたため, 各プロセッサの動作は確率モデルによりモデル化される. 結合網は, 専用のネットワーク記述言語により記述され, それを変換する yacc と lex によるトランスレータをもつシミュレータ生成系によりシミュレータが生成されるため, 記述が容易で実行速度も速い. このため, 確率モデルにより大規模な結合網を高速にシミュレーションする用途に向いている.

しかし, 一方で時間を要してもプロセッサの動作を命令レベルシミュレーションやトレースドレブンシミュレーションなどで正確にシミュレートする要求もある. このようなシミュレータの例としては, トレース駆動型シミュレータの Stanford 大学の ATUM-2[1], 命令レベルシミュレータの慶應義塾大学の MILL[4], 複合式シミュレータの Stanford 大学の SimOS[3] 等が挙げられる. しかし, これらのシミュレータは, 結合網についてはバス結合等の簡単なものしか装備していない.

並列計算機の結合網を評価する場合, 確率モデルシミュレーションを用いて概略の評価を行ない, 詳細な評価はケースを絞ってトレースドリブンや命令レベルシミュレーションを行なうことが理想である。しかし, これらのシミュレータをそれぞれ目的に応じて実装したり, 使い分けたりすることは, 研究者にとって大きな負担となる.

そこで本研究では, ユーザに対する柔軟性をより高めた相互結合網シミュレーション環境 SPIDER を提案する. SPIDER は, 様々な相互結合網シミュレータに共通する要素を抽出し, ライブラリ化したものである. 本ライブラリを使用してシミュレータを実装する場合、ライブラリで用意された機能ブロックをそのまま使用することができるため、シミュレータ実装者は対象となるシステム特有の機能のみを実装するだけで良い。したがってシミュレータ実装者の負担を大幅に軽減することができる。

ライブラリの実装は, 我々の研究室で開発された並列計算機シミュレータライブラリ ISIS[5] を使用する形で行った. ISIS には並列計算機を命令レベルで評価するための機能ブロックが幾つか用意されているため, 乱数に基づく通信パターンだけでなく, 実際の並列アプリケーションに基づく通信パターンによる性能評価も行うことができる.

本稿では, 相互結合網評価環境 SPIDER の概要, シミュレータ作成方法, 幾つかの結合網に対して行ったシミュレーション結果について述べる. 以下, 第 2 章では設計仕様, 構成について述べる. 第 3 章では, 実際にユーザがシミュレータを実装する場合の記述方法について例を挙げて述べる. 更に第 4 章では本シミュレータを使用して, 直接網である二次元トーラス, ハイパキューブ, 間接網である MIN(Multistage Interconnection Network) に関する転送性能について調査, 検討を行う.

2 相互結合網評価環境 SPIDER

2.1 概要

SPIDER は, ハードウェアの機能ブロック (以降ユニットと呼ぶ) を構成単位とし, 相互結合網シミュレータを実装する上で再利用可能なユニットを個々に提供している. 図 1に SPIDER が想定している並列計算機システムのモデルを示す. また, ライブラリの実装には ISIS が提供する幾つかのユニットを使用している. 図 2にその様子を示す.

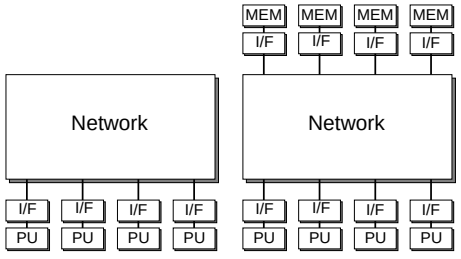


図 1: 評価対象モデル

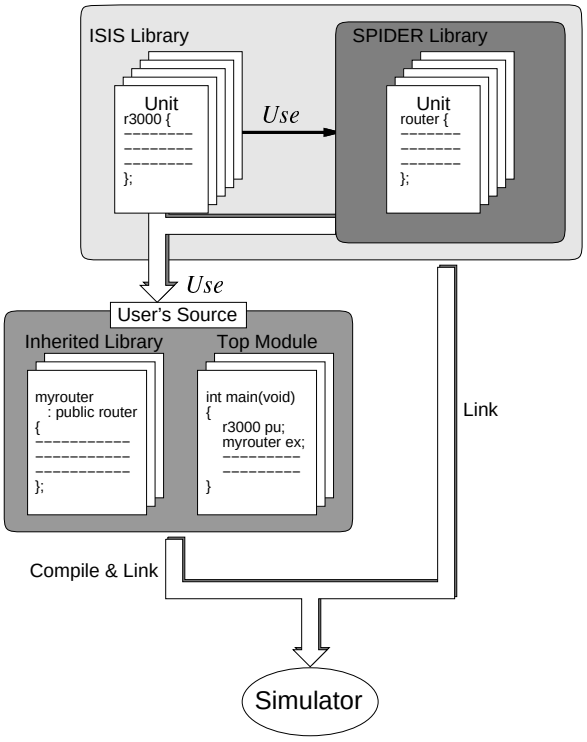


図 2: SPIDER の概要図

ISIS は, 我々の研究室で開発された並列計算機のためのシミュレーションライブラリであり, プロセッサやバス, メモリ, 同期ユニット等のユニットが提供されている. シミュレータの実装は, ISIS と SPIDER が提供するユニットを使用する形で行われる. その際, ライブラリが提供する機能の他に対象となるシステム特有の機能を追加したい場合, 継承を用いて新たなユニットを作り, それを用いてシミュレータの実装を行う.

2.2 設計仕様

SPIDER には, 以下に述べる事項が要求される.

1. 評価対象となるアーキテクチャに依存しない
2. シミュレーション方式に依存しない
3. シミュレータ実装者に負担をかけない
4. 生成したシミュレータが実用的な速度で動作する

ライブラリをアーキテクチャとシミュレーション方式に依存させないためには、個々の機能ブロックをユニットとして個別に実装する方法が有効である。シミュレーション方式によって機能が異なる部分—例えばネットワークインタフェースなど—については、その方式に応じたユニットを個別に用意することで対応が可能である。さらにユニット単位で独立にカプセル化することでユニット別にメンテナンスすることが可能になるため、実装や機能拡張のコストを軽減することができる。

シミュレータ実装者にかかる負担を最小限に留めるためには、様々な並列計算機に共通する機能ブロックをなるべく多く実装済みのユニットとして提供することはもちろんだが、ユーザ自身が新しいユニットを実装する際の負担も抑える必要がある。これを実現するために、ユニット間の共通機能を継承を用いて記述することにする。継承を利用すると、基本機能を持つユニットをライブラリ側で用意し、ユーザが基本機能を持つユニットを拡張することが可能となる。さらに、ライブラリ内のユニットを実装する際のコストを低くすることができるというメリットもある。

生成したシミュレータを実用的な速度で動作させるためには、ライブラリを実装する際に使用する言語が高速に動作する実行形式ファイル (a.out) を出力できる言語を選択する必要がある。一般に、ソフトウェアによる並列計算機シミュレーションは実行時間が長くかかるため、シミュレータの実装言語には C 言語など、高速に動作する実行ファイルを出力できる言語を選択するのが普通である。

以上の事から実装言語には, オブジェクト指向言語である C++を採用し, それぞれのユニットをクラスライブラリの形でユーザに提供することとする. また, クラス間の共通性を基底クラスとして抽出し, クラス階層を形成する.

さらに, 相互結合網の性能を評価する場合, 相互結合網を特徴づける要素の選択やハードウェア仕様, 通信パターンを容易に設定できることが必要になる. そこで, ライブラリを開発するに当たり以下に述べる事項も満足させなければならない.

(1) 詳細な評価を行うために, 各ユニットはクロック単位で状態遷移すること.

(2) 結合形態 (トポロジ) に応じて, 意図するノード間を柔軟かつ容易に接続および変更できること.

(3) 様々なルーティング・アルゴリズムに基づいたシミュレーションが行え, かつアルゴリズムの変更が容易にできること.

(4) 代表的なパケット転送方式である store and foward, wormhole, virtual-cut-through を選択できること.

(5) 通信方式として 1 対 1 およびマルチキャストに対応できること.

(6) バッファ長, パケット長, フリット長, ルーティングや調停 (アービトレション) にかかる処理遅延時間などをそれぞれ設定できること.

(7) ノード間の通信単位であるパケット内部の情報を容易に変更できること

(8) 様々なレベルでの性能評価を行うために, 相互結合網に投入する通信パターンをランダムから実際の並列アプリケーションまで対応していること

2.3 ライブラリ構成

上記で述べた (1) ～ (8) の相互結合網シミュレータに要求される条件を満足するように本ライブラリの開発を行った. SPIDER の構成は, 大きく分けて 3 つの機能ブロック, パケット, ルータ, ネットワークインタフェースからなる. 要件 (1) に関しては, 各機能ブロック (パケットを除く) をクロック入力によって状態遷移を行う基底クラス synchronous_unit の派生クラスとして実装することにより, その要件を満たすことができる. synchronous_unit クラスは, その派生クラスに対し状態遷移 (クロック入力), 状態の初期化 (リセット) 機能のインタフェースを規定する働きを持っている.

2.3.1 パケット

パケットはネットワーク内における通信の基本単位であり, 図 3 のようなクラス階層を形成している.

packet_base クラスは, パケットを表す全てのクラスの基底クラスであり, ユーザが新たにパケットを定義する場合は, この packet_base クラスの派生クラスでなければならない. header_packet_base クラスは, ネットワークを構成するユニットであるルータ内で, ルーティングや優先順位による調停の制御等を行うために使用される. network_interface_packet クラスは, header_packet_base クラスの上位階層に位置するパケットクラスとして定義されており, ネットワークインタフェースで使用される. header_packet_base, network_interface_packet クラスの機能は全て純粋仮想関数として提供され, 機能の働きは全てその派生クラスによって実装がなされるように設計されている. spider_system_packet クラスは, network_interface_packet クラスの派生クラスで実装済み初期パケットクラスとしてユーザに提供されている. 要件 (7) にあるユーザが機能の追加や内部情報の変更を行いたい場合は, 図に示すように spider_system_packet クラスの派生クラスとして実装することができる. しかし, 上記の設計には次のような欠点が存在する.

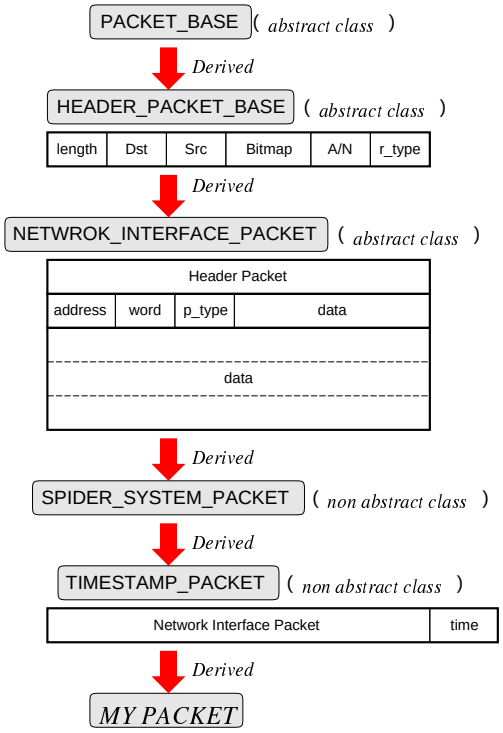


図 3: パケットのクラス階層

- フリットの概念が存在しない
- 多重継承を想定していない

一般的にパケットは, フリットの集合体として定義できる. 更にネットワークを通過するパケットは, フリット毎に転送される. しかし, フリットの概念を導入した場合, フリットに潜在する曖昧性から、ユーザのシミュレータ実装にかかる負担が増大するという問題が現れる. それは, フリットに内包するデータ集合がユーザの想定する対象毎に何通りも存在することに起因する. そのため, 例えば packet_base_class クラスからフリット番号を持った flit_base クラスを派生させた場合, この時点でこれ以上共通性を抽出できなくなり, ユーザが作らなければならないユニット数が増える. 図 4にそのクラス階層を示す.

次にクラス階層に多重継承を導入した場合, パケットという概念を扱う他のユニットをテンプレート化しなければならず, コードが複雑化し, コンパイラが正しいコードを生成できなくなる可能性がある.

以上のことから, 現状では, 多少柔軟性はなくなるが, パケットに対し上記の制約を設けることにより, シミュレータ実装に対する負担の軽減を図っている. しかし, 将来的にはフリットの概念を導入したパケット階層の実装を行う予定である.

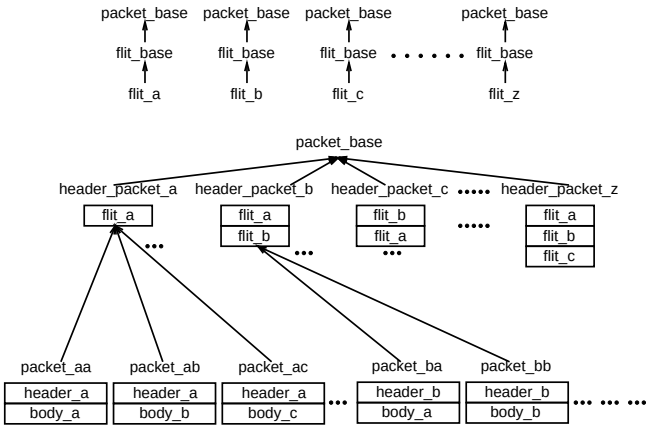


図 4: フリット概念を導入したパケットクラス階層

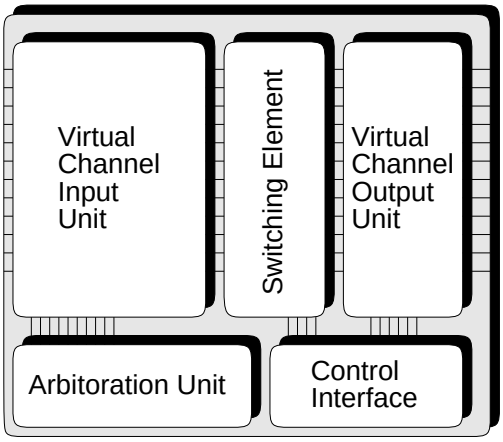


図 5: ルータのブロック図

2.3.2 ルータ

ルータは, 相互結合網を構成するための基本ユニットであり, 要件 (2) 〜 (6) を満たすように設計されている. 図 5に, 全てのルータの基底クラスとなる router_base クラス内部のブロック図を示す. 5 つの機能ブロックから構成され, 他のルータへの接続機能, 入出力幅, バッファ長, 遅延, 仮想チャネル数等の各種パラメータ値の設定・変更機能, フロー制御に必要な handshake 線の制御機能を持つ. router_base クラスは, 同期ユニットクラス synchronous_unit の派生クラスである.

router_base クラスには, 要件 (2) 結合形態の変更を可能にするために, 各ネットワーク資源における入力ポートと出力ポートを結合させるインタフェースを持っている. ネットワーク資源 (a) の (i) 番目の出力ポートをネットワーク資源 (b) の (j) 番目の入力ポートと結合する場合, 次にように記述することができる.

- a.out(i).connect(b.in(j))

また, 要件 (3) ルーティング・アルゴリズムにおけるフロー制御を行うために, 出力したいポートの状態と相手先の仮想チャネルの状態を知る必要がある. そこで, 次に示す 2 つのインタフェースが用意されている.

- is_phys_free(p) : そのネットワーク資源における出力ポートの混雑度を調べるための関数. p は出力ポート番号を表す.
- is_virt_free(p,v) : 出力ポート先の仮想チャネルバッファの混雑度を調べるための関数. p は出力ポート番号, v はそのネットワーク資源における出力ポート p の相手先 (入力ポート) の仮想チャネル番号を表す.

これらの関数により, パケットの出力先のバッファ状況を考慮した様々なルーティング・アルゴリズムを実装することができる. ルーティング・アルゴリズムを記述するインタフェースは, router_base クラス内で純粋仮想関数として定義されており, 結合網に合わせてその派生クラスで実装するように設計されている.

2.3.3 ネットワークインタフェース

ネットワークインタフェースは, 対象となるシステムによって構成方法が種々存在する. 例えば, 超並列計算機 JUMP-1 におけるネットワークインタフェース MBP[8] は, 直接メモリに接続されており, バスからのメモリアクセスも全て MBP を経由して行われる. 一方, Stanford 大の FLASH[9] では共有バスにメモリとネットワークインタフェース MAGIC が接続されており, メモリ制御とネットワークインタフェースの機能は分離されている.

このように様々な構成方法, 機能を有するユニットを実装する場合, 前述のパケットやルータのように共通機能を基底クラスとして提供し, 継承を利用してユニットを実装するよりもネットワークインタフェースが持っている機能ブロックを多くの実装済みユニットとして提供するほうが有効な手段である.

実装済みユニットとしては, ネットワークの入出力を制御する network_io_interface, パケットのエンコード/デコードを行う packet_encoder, packet_decoder, 一様乱数に基づいてメモリアクセスを発行する network_interface_processor 等が提供されている. network_interface_processor は, 確率モデルシミュレーションを行う際に使用されるユニットである. このユニットは, 同一パターンのアクセスを発行するため, 実装したシミュレータのデバッグ用としても使用することができる.

3　シミュレータ記述例

本節では, ハイパキューブ, 2 次元トーラス, MIN といった幾つかの結合網に対して, 資源, 結合, ルーティング・アルゴリズムを記述する場合の例を示す. トップモジュールにおける記述は, 図 6に示すように資源記述部, 結合記述部, 状態遷移記述部から構成される. 資源記述部は, ネットワー資源の定義及び入出力数や仮想チャネル数等の設定を行う. 結合記述部は, 資源記述部で用意した資源をトポロジに従って結合させる. 状態遷移記述部は, 各資源に対しクロックの入力を行う.

```
-----------------------------------------------------
int main(int argc, char* argv[]) {
    /*
     * 資源記述部
     */
          …
    /*
     * 結合記述部
     */
          …
    /*
     * 状態遷移記述部
     */
          …
}
-----------------------------------------------------
```

図 6: トップモジュール

3.1　ハイパキューブ

ハイパキューブシミュレータの記述例を図 7に示す. 但し, この記述例は結合網部分のみを示しており, 実際にはメッセージを発行するユニットをルータに結合しなければならない. また, 資源記述部において N 個のルータ ex が定義されているものとする. DIM は次元数を表す. 関数 switch_size() は, 2 つの引き数をとり, ex に第 1 引き数個の入力ポートと第 2 引き数個の出力ポートを与えている. 関数 channel_size() は 1 つの引き数をとり, ex の各入力に対し引き数で与えた数分の仮想チャネル数を与える. 関数 buffer_size() は, ex の入力の各仮想チャネルに引き数で指定された数のバッファを与える.

ハイパキューブのトポロジは, 一意な数字で識別されたノード番号 (以降アドレスと呼ぶ) を 2 進表現し, 0 〜 (DIM-1)bit のうち 1 つの bit を反転させたアドレスをもつノードに結合される. 結合記述部では, 前述した結合インタフェース connect() を用い, 手続指向的に記述している.

ルーティング・アルゴリズムは, *e*-cube ルーティングを用いる. 現在そのメッセージが存在するノードのアドレスと宛先ノードのアドレスを下位ビットから比較して, ある bit が異なれば, その bit の表す方向へ出力先を決定する. これを繰り返し, アドレスが一致すれば転送は終了する. 現在メッセージがいるノードのアドレスは, 関数 current_node() で調べることができる. この関数は引き数をとらず, 現在のアドレスを一意な 1 次元の値で返す.

```
int main(int argc, char* argv[])
{
    …
    for (int i = 0; i < N; i++) {
        ex[i].switch_size(DIM,DIM);
        ex[i].channel_size(1);
        ex[i].buffer_size(1);
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < DIM; j++) {
            ex[i].out(j).connect(ex[i^(1 << j)].in(j));
        }
    }
    …
}
void
myrouter::routing(const header_t& pkt,routing_table& buf)
{
    const i = current_node();
    for (int j = 0; j < DIM; j++) {
        if ((i & (1 << j)) != (pkt.Dst() & (1 << j))) {
            if (is_phys_free(j)) {
                if (is_virt_free(j,0)) {
                    buf.route_set(j,0); return;
                }
            }
            return;
        }
    }
}
```

図 7: ハイパキューブシミュレータの記述例

3.2　メッシュ/トーラス

単方向 2 次元トーラスシミュレータの記述例を図 8に示す. この例においても前例と同じく結合網部分だけを抽出している. 結合網構成は, X×Y で, 入出力数 2, デッドロックフリーを保証するために仮想チャネル数は 2 に設定されている.

出力ポート 0 を X 方向, 1 を Y 方向の転送にそれぞれ使用する. ルータ間の結合はアドレスの大きい方から小さい方へ向けて結合し, modulo 演算子%を用いてリング構造の記述を行っている.

ルーティング・アルゴリズムは XY ルーティングで, X 次元, Y 次元の順にメッセージの転送を行う. まず, メッセージの存在するノードのアドレスと宛先ノードのアドレスを X 次元方向で比較し, 異なっていれば X 次元方向へ転送を行う. X 次元方向のアドレスが一致したら続いて Y 次元方向の転送に移る. トーラスは潜在的にデッドロックの可能性を秘めているのでデッドロックフリーを保証するために仮想チャネルを使用し, *e*-cube ルーティングを用いている.

```
#define low 0
#define hi  1
int main(int argc, char* argv[])
{
    …
    for (int i = 0; i < Y; i++) {
        for (int j = 0; j < X; j++) {
            ex[i][(j+1)%X].out(0).connect(ex[i][j].in(0));
            ex[(i+1)%Y][j].out(1).connect(ex[i][j].in(1));
        }
    }
    …
}
void
myrouter::routing(const header_t& pkt,routing_table& buf)
{
    const x = current_node()%X;
    const sx = pkt.Src()%X;
    const dx = pkt.Dst()%X;
    …
    if (x != dx) { // X 方向
        if (is_phys_free(0)) {
            if (x == 0) {
                if (is_virt_free(0,hi))
                buf.route_set(0,hi);
            }else {
                if (sx < x) {
                    if (is_virt_free(0,hi))
                    buf.route_set(0,hi);
                }else {
                    if (is_virt_free(0,low))
                      buf.route_set(0,low);
                }
            }
        }
        return;
    }
    … // Y 方向
}
```

図 8: 2 次元トーラスシミュレータの記述例

3.3 MIN(多段結合網)

MIN(多段結合網) シミュレータの記述例を図 9に示す. 記述例にある MIN は 2×2 スイッチを用いた Omega 網であり, 各ノード間はシャッフル結合されている. 記述例の N は PU 数, ST は MIN の段数を表している.

結合記述は, 3 つの部分で記述されている. それぞれ PU-スイッチ間, スイッチ- スイッチ間, スイッチ-PU 間の結合を表している. 関数 shuffle() は, 第 1 引き数で与えられた数字を 2 進表現し, 第 2 引き数桁分だけローテートして, その値を返す.

ルーティング・アルゴリズムは, ディスティネーションルーティングを使用し, 宛先の識別子だけで経路を決定する. このアルゴリズムは, 宛先の番号を 2 進表現し, 上の桁から順に各ステージで 0 か 1 かを判別する. ここで, 0 の場合メッセージは上方の出力に 1 の場合は下方の出力に転送する. 例えば, 宛先の番号 101(5) であった場合, 下, 上, 下の順に転送すると宛先 101 に到着する. 記述例にある関数 check_binary() において, 現在メッセージが存在するステージにおける 0,1 を調べている. この関数は 2 つの引き数をとり, 第 1 引き数に宛先番号, 第 2 引き数にステージ番号を指定すると, そのステージにおける 0,1 が返される.

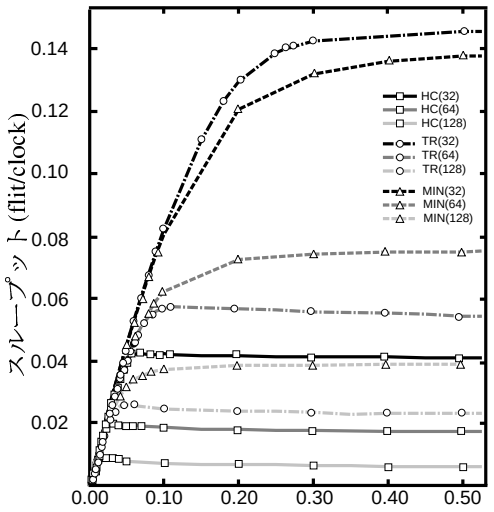
```
int main(int argc, char* argv[])
{
    …
    for (int i = 0; i < N; i++) {
        int j = shuffle(i, 1, N);
        pu[i].out().connect(ex[j/2][0].in(j%2));
    }
    for (int j = 0; j < ST-1; j++) {
        for (int i = 0; i < N/2; i++) {
            int k = shuffle(i*2, 1, N);
            int l = shuffle(i*2+1, 1, N);
            ex[i][j].out(0).connect(ex[k/2][j+1].in(k%2));
            ex[i][j].out(1).connect(ex[l/2][j+1].in(l%2));
        }
    }
    for (int i = 0; i < N/2; i++) {
        ex[i][ST-1].out(0).connect(pu[i*2].in());
        ex[i][ST-1].out(1).connect(pu[i*2+1].in());
    }
    …
}
void
myrouter::routing(const header_t& pkt,routing_table& buf)
{
    const i = current_node() % (N/2);
    const j = (current_node() - i)/(N/2);
    if (j > ST) return;
    const k = check_binary(pkt.Dst(), ST-j-1);
    if (is_phys_free(k)) {
        if (is_virt_free(k,0)) {
            buf.route_set(k,0);
        }
    }
    return;
}
```

図 9: MIN シミュレータの記述例

4 性能評価

本ライブラリを用い, ハイパキューブ, 2 次元トーラス, MIN の 3 つの結合網に関する転送性能の比較を行った. PU 数 64, 仮想チャネル数 1, 但し 2 次元トーラスはデッドロックフリーを保証するために 2, バッファ数 1, パケット転送方式は virtual-cut-through 方式, リンクのバンド幅 1(flit/clock), メッセージ長は固定で 256(bit), 総シミュレーション時間 100,000(clock) とする. 転送パターンはランダム転送で行う. 前述したユニット network_interface_processor を用いてメッセージ生成確率をパラメータとして変化させ, ネットワークの負荷を変えて評価を行う.

図 10にシミュレーション結果を示す. 縦軸はスループットで横軸はアクセス発行確率を表している. スループットは, 各 PU のメッセージ総受信量を, 各 PU が 1flit/clock でメッセージを受信した場合を 1 として正規化している. 図中の飽和する点がネットワークの最大スループットである.



HC:ハイパキューブ TR:トーラス MIN:多段結合網
() 内の数字は, フリット長 (単位 bit) を表す

図 10: シミュレーション結果

図より, 2 次元トーラス, ハイパキューブ, MIN の順に最大スループットが大きいことがわかる. 但し, フリット長が 128bit の場合, ハイパキューブの方が MIN よりも最大スループットが大きくなる. また, フリット長が 64bit の 2 次元トーラスの最大スループットは, フリット長が 32bit のハイパキューブ, MIN よりも小さいことがわかった.

各結合網でフリット長が変化した場合のスループットを比較すると, フリット長に比例して最大スループットが大きくなることがわかった.

5 おわりに

様々な相互結合網シミュレータで共用可能な機能を抽出したライブラリ, SPIDER を提案し, 結合網シミュレータに要求される条件の検討を行い, それに基づいて設計・実装を行った. また, 幾つかの結合網に関してシミュレータを実装する場合の記述方法を実際のプログラム例を挙げて説明し, その結合網に関する転送性能の評価を行った.

本ライブラリは, C++で記述されたクラスライブラリの形で実装した. パケット, ルータ, ネットワークインタフェースの各ユニットは, シミュレータ実装者の負担を軽減させるため, クラス間の共通性を継承を用いて記述できるよう設計した. シミュレータの記述は, ネットワーク記述言語をもつシミュレータ生成系と比べると記述しにくい面も存在するが, ユーザに対する柔軟性を考えると実用範囲内であると考えられる. また, シミュレーションの実行時間は, 1 ステップ当たり数 ms であり, 実行時間の面でも実用範囲内であると考えられる.

今後の課題としては, ネットワーク中の各資源の利用率等の統計情報を収集できるモニタ機能をもつユニットの設計が挙げられる. また, 複雑な結合網シミュレータを実装する場合, やはりユーザの記述ミスが多くなってしまう. そのため, デバッガビリティの改善なども今後はかる必要がある.

参考文献

[1] Richard L. Sites, Anant Agarwal, “Multiprocessor Cache Analysis Using ATUM”, Proceeding of 15th International Symposium on Computer Architecture, 1988.

[2] Rothberg. E, Smith. J. P, Gupta. A, “Working sets, Cache Sizes, and Node Granularity for Large Scale multiprocessors”, Proc. of the 20th ISCA, 1993.

[3] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, Anoop Gupta, “Complete Computer System Simulation: The SimOS Approach”, IEEE Parallel and Distributed Technology, Fall 1995.

[4] 寺澤 卓也, “マルチプロセッサチップのためのキャッシュメモリに関する研究”, 平成 7 年度 慶應義塾大学理工学部計算機科学専攻 博士論文.

[5] 若林 正樹, 米田 卓司, 天野 英晴, “並列計算機シミュレーションライブラリの提案”, 信学技報 VLD97 100-112, pp79-86

[6] 原田 智紀, 曽根 猛, 朴 泰佑, 中村 宏, 中澤 喜三郎 (筑波大), “並列処理ネットワークのための性能評価用シミュレータ生成系 INSPIRE” 情報処理学会研究報告,95-ARC-113, 113-9, pp.65–72, August 1995

[7] 柴村 英智, 久我 守弘, 末吉 敏則, “超並列計算機のための相互シミュレータ” 並列処理シンポジウム JSPP’93, pp.159–166, May 1993

[8] 松本尚, 平木 敬, “超並列計算機上の共有メモリアーキテクチャ”, 信学技報, CPSY 92-36, 1992.

[9] J.Kuskin and al. et. “The Stanford FLASH Multiprocessor” In Proc. The 21st ISCA, pp302-313, 1994