Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2

# QEMU と SystemC を用いた NoC 向け仮想プラットフォームの開発

中島啓太[†]　稗田拓路[†]　谷口一徹[†]　冨山宏之[†]

多数のコアを用いて性能向上を図るメニーコア化が進むにつれ，スケーラビリティに優れた NoC（Network-on-Chip）が注目を集めている．本研究では，オープンソースの CPU シミュレータである QEMU と，業界標準のシミュレーション言語である SystemC を用いて，高速な NoC 向け仮想プラットフォームを開発した．開発した仮想プラットフォームを利用することで，NoC 上で動作するソフトウェアの開発を実機完成前の早期から行うことができ，開発期間の短縮を図ることができる．実験を通じて，本仮想プラットフォーム上で NoC の仮想環境を簡単に構築できること，実用的な時間で NoC をシミュレーションできること，様々な CPU に対応できることを確認した．

# An NoC Virtual Platform Based on QEMU and SystemC

KEITA NAKAJIMA,[†] TAKUJI HIEDA,[†]
ITTETSU TANIGUCHI[†] and HIROYUKI TOMIYAMA[†]

Network-on-Chip (NoC) is considered as a promising architecture for future many-core System-on-a-Chip (SoC) because it has better scalability for the number of cores than usual bus architecture. In this work. We have developed a fast virtual platform for NoC using QEMU and SystemC. The virtual platform enables software development for the NoC without hardware prototype board, thus the hardware/software design time can be shortened. Experiments demonstrate that our NoC virtual prototype is easy to use, very fast, and highly retargetable.

† 立命館大学　理工学部
College of Science and Engineering, Ritsumeikan University

## 1. はじめに

近年，半導体製品の製造技術が向上し，SoC（System-on-a-Chip）におけるコア数が増加傾向にある．コア数の増加は，主に処理速度や信頼性の向上，消費電力の低減を目的としている．すでにコア数が数十個を超えるメニーコア SoC も開発されており，目的とする性能が向上できる限り今後もコア数は増していき，更なるメニーコア化が進む．そのような背景の中，将来のメニーコア SoC に適したアーキテクチャとして NoC（Network-on-Chip）が注目を集めている．

NoC の典型例を次ページの図 1 に示す．NoC では，図 1 のようにルータを用いてチップ内ネットワークを構成し，ネットワークを介してコア間の情報をやり取りする．従来の SoC 設計で使われてきたバス・アーキテクチャよりも，コアの増加に対するスケーラビリティに優れている．アーキテクチャの見直しによりスケーラビリティの改善が図れる一方，NoC を採用したメニーコア SoC においても，従来のバスを採用した場合と同じ問題が生じる．その 1 つは，開発期間の一層の長期化である．

開発期間の長期化問題はメニーコア化以前から生じていたが，メニーコア化に伴いコア数が増えることでソフトウェアの規模・複雑さが増し，より深刻なものとなっている．開発期間が長期化することは，市場投入の時期を遅らせる原因となるため，開発したチップの売り上げ減少につながることが多い．また，長期化は開発コストの増加にもつながる．このように，開発期間の長期化問題はチップ製造企業にとって無視できない問題であり，この問題に対処するため従来のアーキテクチャでは仮想プラットフォームというツールが利用されてきた．

仮想プラットフォームは，あるプラットフォーム上で異なるプラットフォームの動作を再現するソフトウェアであり，チップ開発における有用なツールの 1 つである．仮想プラットフォームを使用し，チップのハードウェアを模倣する仮想環境を構築することで，ソフトウェア開発者は実機完成前の早期からプログラムの作成を始めることができる．これによって，ソフトウェア開発をハードウェア開発と同時並行に行えるので，全体としての開発期間を短縮できる．

従来のアーキテクチャ向けの仮想プラットフォームは複数提案されている．一方で，NoC 向けのものはあまり知られていない. NoC 向けに作られたツールは 1)や 2)のようなハードウェア設計を支援するシミュレータが多く，ソフトウェア開発の支援をする仮想プラットフォームのようなツールが不足している．

本論文では，ソフトウェアの早期開発を支援する NoC 向けの仮想プラットフォームを提案する．チップの開発期間を短くするためには，仮想プラットフォーム上に短期間で実機の仮想環境を構築できる必要がある．そのため，提案プラットフォーム上の CPU コアの実装には，高性能なオープンソースの ISS（Instruction Set Simulator）である QEMU を用いる．またルータ・周辺モジュール・モジュール間の配線を実装する

Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2

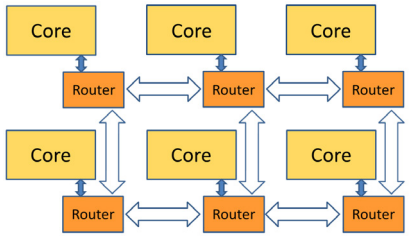



図 1　NoC の典型例（2D メッシュ型）

のに，システムシミュレーション言語の業界標準である SystemC を用いる．これらを採用することで，短期間での仮想環境の構築を可能にする．本プラットフォームを利用することで，NoC アーキテクチャを採用した SoC（メニーコア SoC を含む）を疑似的にシミュレーションし，各 CPU コア上で任意のプログラムを動作させることができる．

本論文の構成を示す．2 章では，関連研究について述べ，3 章では，開発した NoC 向け仮想プラットフォームについて述べる．4 章では，開発した仮想プラットフォームを用いた実験について述べ，5 章では，まとめを述べる．

## 2. 関連研究

本章では，初めに QEMU の実行モードについて簡単に述べたのち，QEMU を用いて実装された従来のアーキテクチャ向け仮想プラットフォームをいくつか紹介する．

QEMU には，フルシステムエミュレーションとユーザーモードエミュレーションの 2 つの実行モードがある．前者は ISS や多くの周辺装置のエミュレートによって，ホスト PC 上で別の OS を起動することができる．一方，後者は単に ISS のみを利用するモードであり，異なる CPU 用にコンパイルされたバイナリをホスト PC 上のプロセスとして実行できる．QEMU を用いた仮想プラットフォームとして有名であり，最も実用化されているのは，3)の Android Emulator である．

Android Emulator は，フルシステムエミュレーションモードの QEMU をベースにして実装されており，ARM プロセッサを搭載した携帯端末の仮想環境を構築することができる．ソフトウェア開発者は Android Emulator を用いることで，スマートフォンの実機がなくてもアプリケーションの開発・テストをすることができる．

同じように QEMU のフルシステムエミュレーションモードを利用した仮想プラットフォームとして，4)が提案されている．このプラットフォームでは SoC の仮想環境を構築することができ，ハードウェアとソフトウェアの協調シミュレーションを行うことができる．QEMU のソースコードに細工をすることでサイクル精度でのシミュレーションを可能にし，実機に近い精度での性能評価が可能となっている．

また，フルシステムエミュレーションに比べて高速動作するユーザーモードエミュレーションを利用した，5)のような仮想プラットフォームも提案されている．この仮想プラットフォームを用いることで，利用者は QEMU で実現された 2 つの ARM コアを持つ SoC の仮想環境を構築することができ，ハードウェアとソフトウェアの協調シミュレーションを高速に行うことができる．

提案仮想プラットフォームでは，主に QEMU のユーザーモードエミュレーションを利用する．本プラットフォームは，上記で紹介したプラットフォームに比べ，シミュレーションの抽象度が高く，NoC 上のソフトウェア早期開発に特化している．

## 3. NoC 向け仮想プラットフォーム

本章では，3.1 節で開発する仮想プラットフォームが持つべき性質を考え，その実現方法を述べる．そして，次の 3.2 節で実装の詳細を説明したのち，3.3 節で実装した仮想プラットフォームにおけるシミュレーションの流れを説明する．

### 3.1 仮想プラットフォームの検討

提案仮想プラットフォームが持つべき性質としては，以下の 2 つが考えらえる．

1. 迅速かつ容易に仮想環境を構築できる．
2. 実用的な時間内でシミュレーションできる．

1 番目の性質は，開発期間の短縮を目的とする本仮想プラットフォームには必須のものである．この性質を実現するために，QEMU と SystemC を組み合わせて用いた．CPU コアの実装に QEMU を用いることで，わざわざ自前で用意しなくても，6)で述べられた動的バイナリ変換により高速動作する CPU コアを利用できる．またコア以外のハードウェアの実装には，SoC シミュレーション言語として従来から利用されている SystemC を採用した．2 つを組み合わせることで，利用者は SystemC の記述に関する知識と，本仮想プラットフォームについての一部の理解があれば，簡単に仮想環境を構築できるようになる．さらに同時に，利用者は NoC の規則的な構造を仮想環境として迅速に構築できるようにもなる．それは，QEMU による CPU コア，および SystemC によるハードウェア（C++クラス）の双方が容易に複製できるためである．

2 つ目の性質も，支援ツールとして必須のものである．この性質は，シミュレーションを抽象度の高いトランザクションレベルで行うことで実現した．具体的には，NoC におけるコア間の送信・受信をトランザクションの基本単位とする．さらに，仮想環境におけるコアとルータ間のやり取りをソケット通信で行うことで，各コアを複数のホスト PC 上で分散シミュレートすることができる．これによってコア数が多い場合，複数のホスト PC を用いた分散処理によりシミュレーション時間の削減が図れる．

Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2

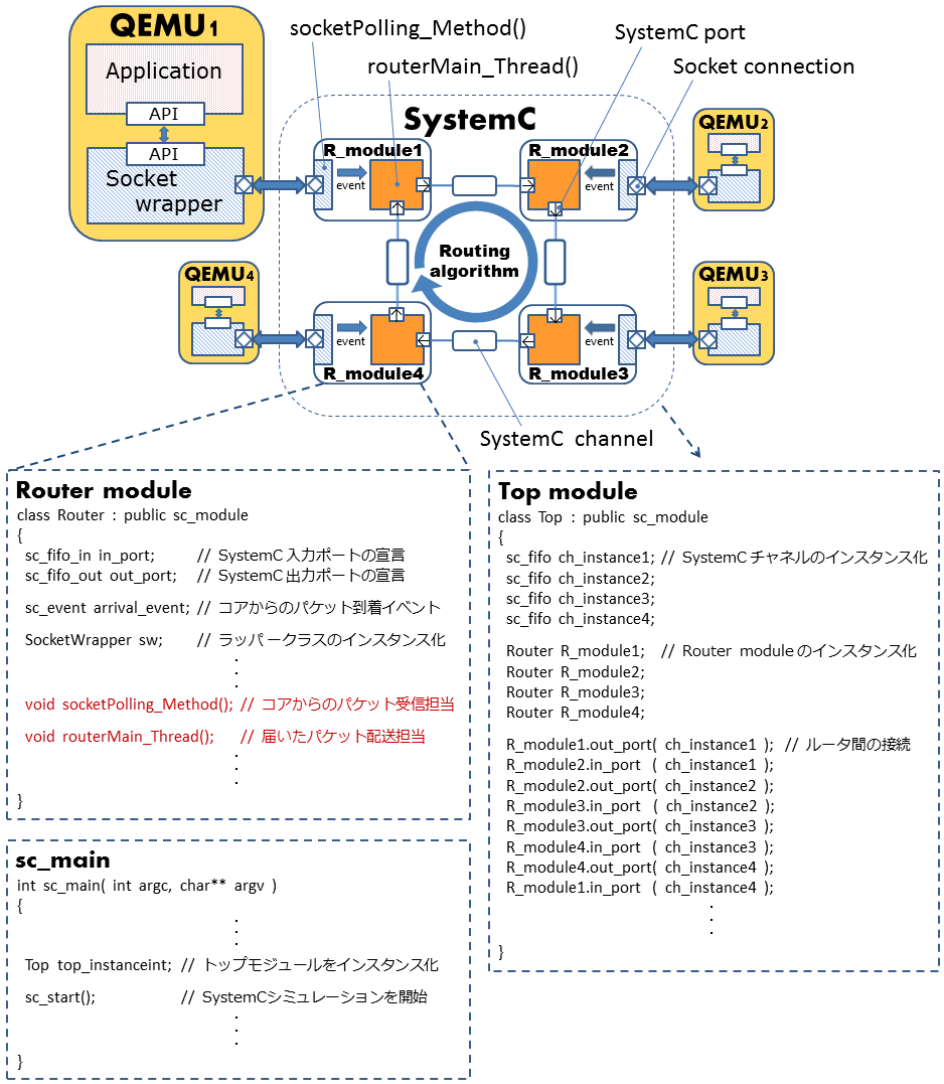



図 2　仮想環境の概要図（4 コア・リング型 NoC を構築した例）

### 3.2 仮想プラットフォームの実装

図 2 は，本仮想プラットフォーム上で 4 コア・リング型 NoC の仮想環境を構築した際の概要図である．この具体例を通じて仮想プラットフォームの実装について説明する．

開発した仮想プラットフォームのプログラムは, 主に以下の 4 つで構成されている．

1. コアとルータモジュール間のソケット通信を扱うソケットラッパーの記述
2. ルータモジュールの記述
3. モジュールのインスタンス化と接続関係の記述
4. SystemC によるシミュレーションを開始するための記述

1 つ目のソケットラッパーの記述とは, 図 2 の例だと Socket wrapper のことである．ソケットラッパーは，C++で記述された SocketWrapper クラスとして実装されており，QEMU 上のアプリケーション内および SystemC のルータモジュール内でそのインスタンスを生成し利用する形をとる．ラッパークラスのメンバメソッド（API）を利用することで，ソケット通信の詳細を気にすることなく，コネクションを確立したり，任意のデータを送受信したりすることができる．

2 つ目のルータモジュールの記述とは，図 2 の例だと Router module のことである．ルータモジュールは，NoC におけるルータの動作を模倣しており，対応するコアや接続関係にある他のルータモジュールから届くパケットをルーティングアルゴリズムに従って配送する．コアから届くパケットに対応するのが socketPolling_Method()であり，届いたパケットを次の宛先に送信するのが routerMain_Thread()である．この SystemC メソッドとスレッドの動作は重要なため，更に詳しく述べる．

socketPolling_Method()は，同じルータモジュール内のソケットラッパーを介してノンブロッキング受信を 1 回実行することで，対応するコアからパケットが届いているかを確かめるメソッドである．ノンブロッキング受信に成功した場合は，届いたことをイベントによって同ルータモジュール内の routerMain_Thread()に知らせる．SystemC のシミュレーションカーネルによって各ルータモジュールの socketPolling_Method()が繰り返し呼び出され，ポーリング動作を行うことで，コアから送信されたパケットに対応している．

routerMain_Thread()は，socketPolling_Method()と違って，SystemC のシミュレーションカーネルによって繰り返し呼び出されるのではなく，特定のイベントが発生した場合にのみ呼び出される．その特定のイベントは 2 つあり，1 つは先ほど述べた，コアからのパケット到着を知らせる socketPolling_Method()によるイベントである．残るもう 1 つは，他のルータからのパケット到着を知らせるイベントである．このイベントはルータモジュールの入力ポートから受け取ることができる．本プラットフォームにおいてルータ間のパケット送受信は SystemC のチャネルを介して行われており，送信

Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2

側のルータがチャネルにパケットを書き込んだら，受信側のルータは書き込みがあったことを入力ポートへのイベントという形で知ることができる．このスレッドは呼び出されると，パケットの宛先と利用者の定めたルーティングアルゴリズムに従って，対応するコアもしくは接続関係にある次のルータにパケットを送信する．

3 つ目のモジュールのインスタンス化と接続関係の記述とは，図 2 の例だと Top module のことである．ルータモジュールとチャネルをインスタンス化し，各モジュールのポートとチャネルを接続してやる．接続関係は任意に変えることができるので，利用者は任意のネットワークトポロジーを選択することができる．

4 つ目のシミュレーションを開始するための記述とは，図 2 の例だと sc_main()のことである．本仮想プラットフォームは SystemC のシミュレーションカーネルを利用するので，main()の代わりに sc_main()を利用しなければならない．この sc_main()内で，仮想環境全体を表すトップモジュールのインスタンスを作成し，sc_start()を実行することで SystemC のシミュレーションを開始する．

### 3.3 シミュレーションの流れ

本節では，前節の図 2 に示した 4 コア・リング型 NoC を引き続き例にして，提案仮想プラットフォーム上でのシミュレーションの流れを説明する．

シミュレーションの大まかな流れを図 3 に示す．初めに sc_main()を実行し，QEMU1 に対応するルータモジュール（R_module1）が TCP/IP ソケット通信のサーバとして接続要求の待ち状態に入る．次に QEMU1 が起動され，QEMU1 のソケットラッパーがソケット通信のクライアントとして SystemC 上の R_module1 に対して接続要求を行う．R_module1 は，その接続要求を受け取ると要求を許可しコネクションを確立する．同じようにして R_module2 と QEMU2，R_module3 と QEMU3，R_module4 と QEMU4 も順次コネクションを確立していく．

すべてのルータとコア間のコネクションが確立できたら，各ルータモジュールから対応するコアに対してアプリケーション実行開始の合図が送られる．合図をすべてのコアに送ると，NoC 仮想環境のシミュレーションが開始され，各ルータモジュールの socketPolling_Method()によるポーリングが開始される．

ポーリングは，セット単位で繰り返し行われる．1 セットは，各ルータモジュールの socketPolling_Method()を 1 回ずつ実行することから構成されている（図 3 の例では 1 セット 4 回）．1 セット内の各 socketPolling_Method()の実行順序はランダムであり，1 セットごとに SystemC のシミュレーション時間が 1 ずつ増えていく．

図 3 では，時間 10 において QEMU2 から QEMU4 宛のパケットを受信したと仮定している．時間 10 において R_module2 の socketPolling_Method()は，QEMU2 からのパケットを受け取り，自ルータ内の routerMain_Thread()にイベント投げる．イベントを受け取った R_module2 内の routerMain_Thread()は，同じ時間 10 に実行される予定であった残りの socketPolling_Method()が実行されたのちに起床される．この動作は，SystemC シミュレーションカーネル内のデルタサイクルを利用して実現している．R_module2 の routerMain_Thread()は，R_module3 につながるチャネルにパケットを書き込むことで，イベントを R_module3 の routerMain_Thread()に投げる．同じようにチャネルへの書き込みを通じて，R_module3 から R_module4 の routerMain_Thread()が呼び出され，パケットが伝播される．R_module4 の routerMain_Thread()は，パケットの宛先が自ルータと接続関係にある QEMU4 であるので，受信したパケットを QEMU4 に送る．その時間のすべてのパケットが宛先に届いたら，シミュレーション時間を 1 進めてポーリングが再開される．以上のような仕組みでコア間の通信がシミュレートされている．

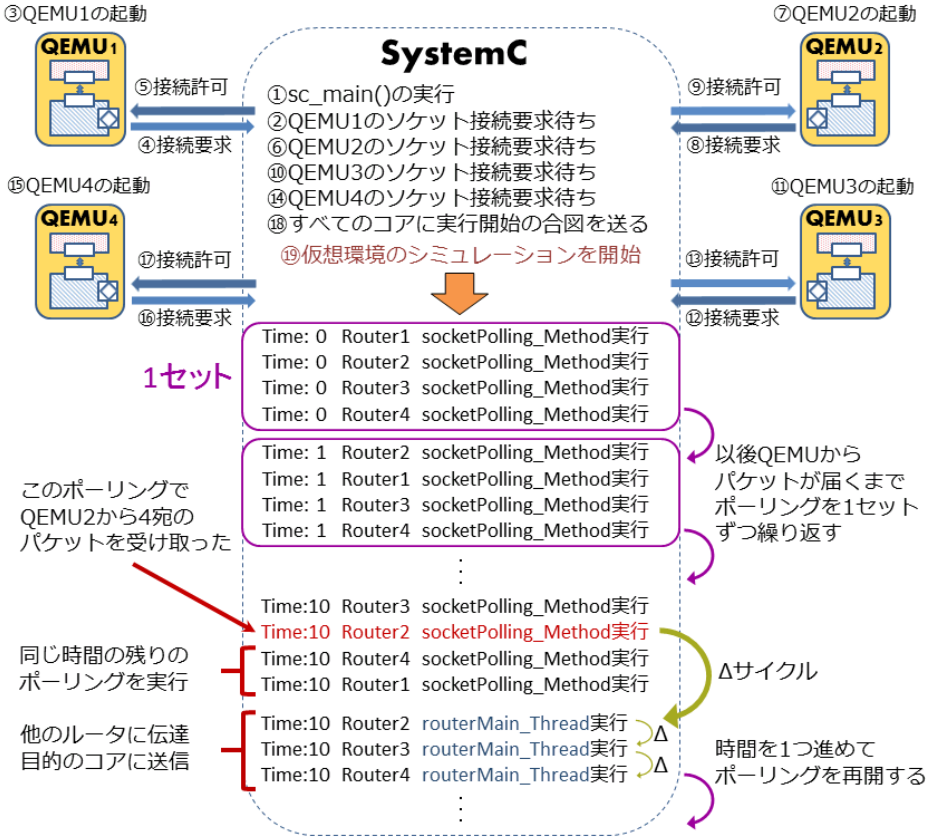



図 3　本仮想プラットフォーム上でのシミュレーションの流れ(4 コア・リング型 NoC)

Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2

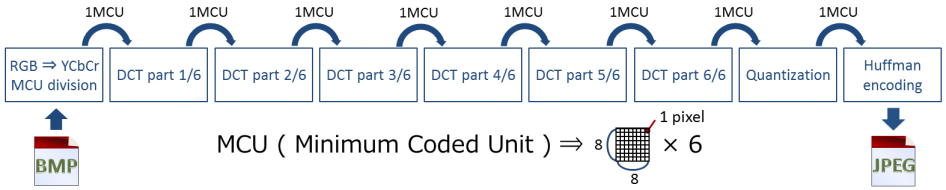



図 4　JPEG パイプライン圧縮プログラム

## 4.　実験

本章では，4.1 節で実験に用いる JPEG パイプライン圧縮プログラムについて説明したのち，続く 4.2，4.3 節で 2 つの異なる実験について述べる．

### 4.1　JPEG パイプライン圧縮プログラム

実験で用いる JPEG パイプライン圧縮プログラムの構成を図 4 に示す．パイプラインは全 9 ステージで構成されており，入力に BMP 形式の画像を用いる．

第 1 ステージでは，BMP 画像から RGB 値を抽出し，その RGB 値を，縦 16 画素，横 16 画素の正方形単位で取り出す．正方形 1 つ分の RGB 値は，輝度と色差を用いた YCbCr 表色値に変換されたのち，Y:Cb:Cr が 4:1:1 で間引きされ，図 4 に示す MCU（Minimum Coded Unit）という JPEG 圧縮の最小単位にまとめられる．まとめられた結果の MCU は逐一次の第 2 ステージに送られ，正方形単位で処理を繰り返す．

第 2 ステージから第 7 ステージは，離散コサイン変換（Discrete Cosine Transform）をするステージである．離散コサイン変換は割り算や掛け算を多用するため，他のステージに比べ実行に時間がかかる．全ステージの実行時間を平坦化するために，離散コサイン変換を 6 つのステージに分割した．分割した各ステージでは，第 1 ステージから送られてくる 1MCU 中の 6 分の 1 要素に対してそれぞれ離散コサイン変換を行う．

第 8 ステージでは，受け取った 1MCU に対して量子化を行う．量子化はすべて，あらかじめ用意した 1 つの汎用量子化テーブルを用いて行った．

第 9 ステージでは，受け取った 1MCU をハフマン符号化し，JPEG 画像を出力する．ハフマン符号化も，あらかじめ用意した 1 つの汎用ハフマンテーブルを用いて行った．

### 4.2　JPEG パイプライン圧縮実験

開発した仮想プラットフォーム上で，図 5 に示す 2D メッシュ型 NoC の仮想環境を構築し，各コア上で前節のプログラムを動かす実験を行った．本実験は，開発した仮想プラットフォームが 3.1 節で検討した性質を持っているかを確かめるためのものである．実験はコア数を 18，36，54，72，90，108 の 6 パターンに変えて行う．図 5 に示すように，X 軸方向のコア数は常に 9 個で固定し，Y 軸正方向にメッシュを拡大することでコア数を増やしていく．18 コアの場合は X 軸方向 9 列，Y 軸方向 2 行．54 コアの場合は X 軸方向 9 列，Y 軸方向 6 行といった具合である．図 5 の赤線で囲まれた行のように，各行 9 コアごとに JPEG パイプライン圧縮プログラムを 1 セット実行し，それぞれ同じ BMP 画像を入力して圧縮を行った．

実験で使用したホスト PC の情報を表 1 に示す．本実験では，CPU コアとして QEMU ユーザーモードの MIPSEL（MIPS のリトルエンディアン版）を選択し，ルーティングアルゴリズムとして XY アルゴリズムを用いた．また，実験は次の 2 通りの構成で行った．1 つ目は，Host PC1 のみで SystemC およびすべてのコアをシミュレートする場合．2 つ目は，Host PC1 と Host PC2 の 2 台を用いて分散処理を行った場合であり，Host PC1 上で SystemC および半分のコアを，Host PC2 上で残りの半分のコアをシミュレートした．

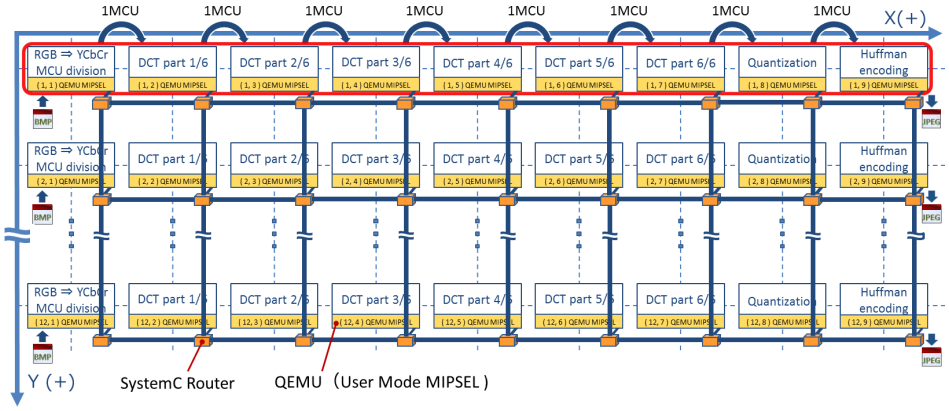
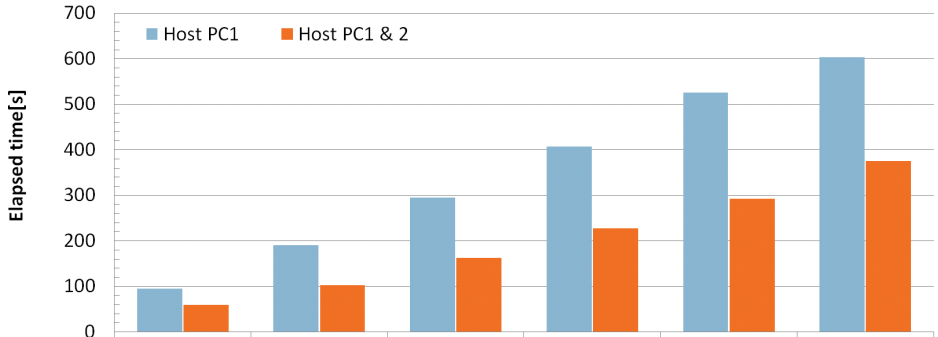



図 5　JPEG パイプライン圧縮実験

表 1　実験で使用したホスト PC 情報

| | Host PC1 | Host PC2 |
|---|---|---|
| CPU | Core i7 990x 3.46GHz (6cores 12threads) | Core i7 970 3.2GHz (6cores 12threads) |
| Memory | DDR3-1333 12GB | DDR3-1333 12GB |
| HDD | SATA3.0 2TB 5900rpm | SATA2.0 2TB 7200rpm |
| OS | Ubuntu 10.04 32bit | Yellow Dog for CUDA 64bit |
| Kernel | 2.6.32-37-generic | 2.6.18-164.ydl.3 |
| QEMU | version 1.0 | version 1.0 |
| SystemC | version 2.2.0 | version 2.2.0 |

Vol.2012-SLDM-155 No.1
Vol.2012-EMB-24 No.1
2012/3/2




| #Cores | 18 (N = 2) | 36 (N = 4) | 54 (N = 6) | 72 (N = 8) | 90 (N = 10) | 108 (N = 12) |
|---|---|---|---|---|---|---|
| Host PC1 | 95.09 | 190.86 | 295.22 | 407.3 | 525.4 | 603.56 |
| Host PC1 & 2 | 59.03 | 102.12 | 162.71 | 228.19 | 292.12 | 374.89 |

図 6　JPEG パイプライン圧縮実験の結果

実験結果を図 6 に示す．縦軸の経過時間は，Host PC1 で実行される SystemC 上で gettimeofday()関数を利用して測定した．すべてのコアとルータがコネクションを確立した時から，全コアのアプリケーション実行が終了するまでの経過時間である．

図 6 の結果から，実行時間はおおよそコア数に比例しており，妥当なオーバーヘッドで NoC 内のネットワークをシミュレートできている．コア数が 3 ケタに達する 108 コアの場合でも実用的な時間内でシミュレートできており，分散処理による実行時間の削減も図れていた．また本実験の準備をするため，開発した仮想プラットフォーム上で図 5 の NoC 仮想環境を構築したが，構築自体は 1 日以内で終えることができた．このことから，提案仮想プラットフォーム上で素早く仮想環境を構築できることが確認できた．

### 4.3　様々な QEMU の CPU アーキテクチャによる実験

次に QEMU の様々な CPU アーキテクチャで，JPEG パイプライン圧縮プログラムを 1 セット実行するという実験を行った．仮想環境として，先ほどの実験で用いた 2D メッシュ型 NoC を，X 軸方向に 9 列，Y 軸方向に 1 行の合計 9 コア構成に作り直して用いる．実験はすべて，表 1 の Host PC1 単体で行った．

実験結果を図 7 に示す．No QEMU は JPEG パイプライン圧縮プログラムを QEMU 上ではなく，ホスト PC 上のプロセスとしてネイティブ実行した場合の結果である．縦軸の経過時間は，すべて 4.2 の実験と同じ条件で測定した．

図 7 の結果から，QEMU を用いることで様々CPU アーキテクチャのコアを利用できることが確認できた．

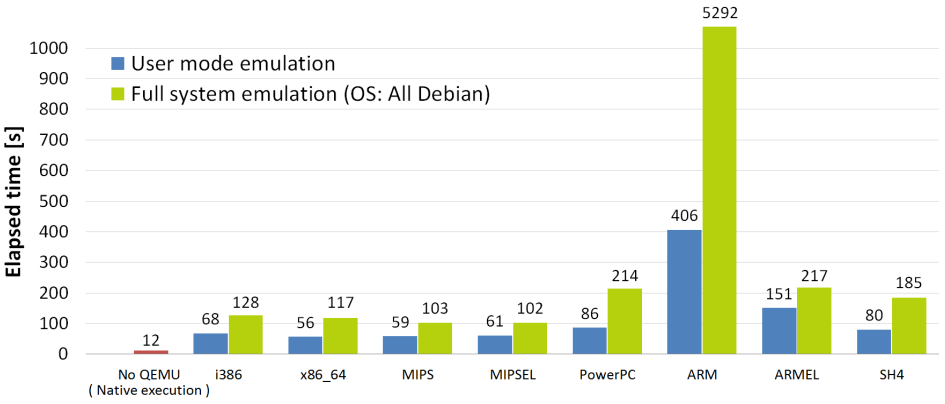



図 7　様々な QEMU の CPU アーキテクチャによる実験結果

## 5.　おわりに

本論文では，NoC 向け仮想プラットフォームの開発について述べた．開発した仮想プラットフォームを利用することで，NoC 上で動作するソフトウェアの開発を実機完成前の早期から行うことができ，開発期間の短縮を図ることができる．実験を通じて，本仮想プラットフォーム上で NoC の仮想環境を簡単に構築できること，実用的な時間で NoC をシミュレーションできること，様々な CPU に対応できることを確認した．

今後の課題としては，時間精度の向上，およびデータ粒度の詳細化が挙げられる．

## 参考文献

1) Gigli, S. and Conti, M.: A SystemC Platform for Network-on-Chip Performance/Power Evaluation and Comparison, *Proc. 7th Intelligent solutions in Embedded Systems*, pp.63-69 (2009).
2) Zhang, W., et al.: A Simulation Environment for Network-on-Chip Based on SystemC, *International Conference on Computer Application and System Modeling*, Vol.9, pp.V9-79-V9-83 (2010).
3) Android Emulator. http://developer.android.com/guide/developing/tools/emulator.html.
4) Chiang, M., Yeh, T. and Tseng, G.: A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.30, pp.593-606 (2011).
5) Peng, C., Chang, L., Kuo, C. and Liu, B.: Dual-Core Virtual Platform with QEMU and SystemC, *International Symposium on Next-Generation Electronics*, pp.69-72 (2010).
6) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proc. USENIX 2005 Annual Technical Conference*, pp. 41-46 (2005).