情報処理学会論文誌 Vol. 49 No. 10 3482–3492 (Oct. 2008)

MICS：システム設計のための フレキシビリティの高いシミュレーション環境

三好健文[†1] 杉野暢彦[†2]

近年，組み込みシステムでは面積や消費電力の制約のもとで高い計算能力を必要とするアプリケーションが求められ，ソフトウェアとハードウェアの両方を考慮した効率の良いシステム設計の重要性が高まっている．本論文ではシステム開発の初期段階において最適なソフトウェアおよびハードウェアを決定するために，様々な要素を容易に試行し実際の動作で性能を評価することができる柔軟性の高いシミュレーション環境を提案する．提案手法では，シミュレーション対象となるハードウェア要素およびデータフォーマットを，抽象化したインタフェースに基づいて取り扱う．まず，インタフェースの定義について述べる．次に，提案するシミュレーション環境のもとで，様々な種類および精度のシミュレーション要素を統一的に取り扱うことができ，容易に所望のアーキテクチャのシミュレーションを実行できることを示す．また，抽象的なインタフェースに基づくことで既存のシミュレーションツールの利用が容易であり，提案するシミュレーション環境の導入コストが小さいことを示す．本論文では，提案したシミュレーション環境を実際に実装し，シミュレーション速度と抽象化によるオーバヘッドについて評価する．これにより，要素数の増加にシミュレーション時間が正比例することおよび，抽象化によるオーバヘッドのコストが十分小さいことを示す．

MICS: Cycle Based Architecture Simulation Environment for System Design

Takefumi Miyoshi[†1] and Nobuhiko Sugino[†2]

Cycle based architecture simulation environment framework ‘MICS’ is proposed. Recent embedded systems require both sophisticated software and well-designed hardware, which has high computational performance to execute complicated applications within constraints over power consumption and chip sizes. In order to design an appropriate system, designers usually try various combinations of hardware and software, especially at the earlier design stages. The framework proposed in this paper provides the abstracted interfaces for simulating components and data, so that a new simulation environment for a system can be built by combining components required in its architecture. And assumed target architecture is driven by cycle-based virtual machine. It can simulate various components of different granularity or accuracy accordance with development stages. Therefore, system designers can easily try various architectures even at the early phase of system development by use of this Mics environment. Moreover, existing simulation tools can also be utilized by help of common wrapper interface, so that users can expand the simulation environment in very low cost. The proposed interfaces to define hardware units and data are implemented in Java, and some hardware architecture examples for cycle-based simulation are shown. And finally, simulating speed and overhead are evaluated for several examples.

1. はじめに

近年の組み込みシステムにおいては，高い処理能力が要求されるアプリケーションが増加している．しかしその一方で，設計時に組み込みシステムならではの厳しい実装スペースや消費電力の制約を満たす必要もある．そのため，低消費電力で高い計算性能が得られるマルチプロセッサアーキテクチャや，アプリケーションに特化した専用のハードウェアや信号処理用のプロセッサなどを組み合わせたヘテロジニアスなアーキテクチャなどによってシステムを実現する方法がとられることが多い．また最近では，システムの動作時に，そのときどきによって計算効率の高い演算器として振る舞うことができるリコンフィギュラブルアーキテクチャを用いてシステムを実現することも増えている．このようなターゲットアーキテクチャを用いて適切にアプリケーションを実現するためには，ハードウェアとソフトウェアの両方の処理における効果とコストをふまえたうえで，それぞれのあり方を検討するシステムアーキテクチャとしての設計がきわめて重要である．特にシステムアーキテクチャの設計においては，アーキテクチャ中のプロセッサの処理能力の評価だけではなく，システムとして構成するための通信路やキャッシュなど様々な要素を適切に検討する必要がある．バッファメモリ最適化[1)] やキャッシュのチューニング[2)] などによってシステムの全体性能が向上することは，よく知られた事例である．

システムを短時間で開発するために，アプリケーションの性能評価にターゲットアーキテクチャのシミュレータを用いる手法は広く用いられている．とりわけ，システムアーキテ

†1 東京大学大学院情報理工学系研究科創造情報学専攻
Department of Creative Informatics, The University of Tokyo
†2 東京工業大学大学院総合理工学研究科物理情報システム専攻
Department of Information Processing, Tokyo Institute of Technology

クチャを効率良く設計するためには，設計の初期段階から，対象としているアプリケーションの動作をターゲットシステムアーキテクチャ上で検証，評価できることがシミュレータには望まれる．現在，開発の現場では，対象とするハードウェアからソフトウェアまでの開発レイヤにおいて，それぞれ様々なシミュレーション手法が用いられている．たとえば，ハードウェアアーキテクチャの設計レイヤでは，HDL の RTL 記述を用いて設計したアーキテクチャをビヘイビア記述によるテストベンチでシミュレーションするほか，近年では，SystemC[3] や SpecC などの抽象度の高い言語を用いた実装と評価なども行われるようになっている．これらの言語を用いた設計の過程では，ハードウェアアーキテクチャの詳細な設計をクロックレベルによる高い精度でシミュレーションすることができる．一方で，ソフトウェアの設計と検証には，プロセッサ単位でのソフトウェア動作をシミュレーションする SimpleScalar[4]，BurstScalar[5]，SimCore[6]，GDB[7] など，精度や速度，目的に応じた様々なシミュレータが用いられる．

しかし，開発の初期段階において，用いるプロセッサやアーキテクチャのハードウェア構成の詳細を決定できることは大変稀なことであり，特に組み込みシステム開発においては，開発の初期段階で様々な仕様のハードウェアユニットを選別あるいは設計し，それらを組み合わせたうえでのソフトウェアの動作を評価，検証する必要がある．そのため，従来の詳細なアーキテクチャ設計に基づくシミュレータや，プロセッサごとに独立したシミュレータでは，仕様や構成を変更するたびにシミュレーション環境やソフトウェアの開発環境を逐一構築することが必要であり，システムアーキテクチャの設計が大変困難かつ時間を要する作業となっている．

これを容易にするために，開発の初期段階で決定可能な情報であるプロセッサの実行サイクル数や，ハードウェアユニット間の通信サイクル数などによってソフトウェアの動作をシミュレーションし，アプリケーションのボトルネックの検出や，変更による実行速度やコストを評価，検証できるシステムアーキテクチャ設計のためのシミュレータが強く求められている．

この解決のために，本論文では，(1) ハードウェアユニットを，抽象化したインタフェースに基づいたシミュレータ要素として実装し，(2) 各要素の処理に対する必要サイクル数によってシステム全体の動作を仮想的に実行する，サイクルベースのシミュレーション環境 MICS を提案する．これにより，柔軟にシステム構成を試行することができ，かつ，開発の初期段階から実アプリケーションに近いソフトウェアを動作させることができる．すなわち，(1) により，同じインタフェースに基づいて実装された要素は可換であることから，シミュレーション要素の組合せを容易に変更することが可能となり，(2) により，開発の段階に応じて，システムを構成する各要素のシミュレーション精度をユーザが任意に決定し，ソフトウェアを動作させることが可能となる．さらに，部分的に他のシミュレータツールや仮想的なモジュールを用いてサイクル数を算出することで，シミュレーション要素としてそれらを利用することができる．このため，シミュレータを再実装する手間を削減することが可能であり，提案するシミュレーション環境 MICS の導入コストは小さい．

2. 関 連 研 究

近年では，VHDL や Verilog HDL などの，高い抽象度でハードウェアを記述可能な言語およびそのシミュレーション環境がある．VHDL や Verilog HDL では，ビヘイビアレベルや RTL レベルによって高い抽象度から低い抽象度までの記述が可能である．論理合成および配置配線ツールにより，実際に FPGA や CPLD 上で動作させることができるため，開発の中後期において非常に強力である．しかし，開発のためのコストが大きく，またシミュレーションに必要とする時間も大きい．したがって，初期段階におけるシステム設計に用いることは困難である．

SystemC[3] は，ハードウェアアーキテクチャを抽象的に表現するための言語である．これは，C++のクラスライブラリであり，SystemC で書かれたプログラムをコンパイルすることで，所望の回路のシミュレーションを行うことができる．SystemC は C++であり，ハードウェアユニットはプログラムとしてテンプレートやクラスを使って記述される．ゆえに，様々なソフトウェアモジュールを利用することで，HDL と比較してより抽象度の高い記述ができる．モジュールは，開発の段階に応じて実際のハードウェアユニットと置換することができる．また，UML と SystemC に基づいた組み込みシステム向けのモデル駆動の設計環境[8] も提案されている．これは，C，C++，および SystemC のプログラムコードをモデル駆動の SoC 設計手法によって生成することができる．しかし，システム全体をシミュレーションし評価する際には，やはり SystemC をベースとした設計では，そのコストが大きくなってしまう．特に開発の初期段階において，システム開発者が対象とするハードウェアアーキテクチャの評価を行うためには，より抽象的な定義，たとえばパラメータの決定程度だけでシステム全体を評価することが望まれる．

システムレベルのシミュレーション手法としては，SimOS[9]（Simulation including OS behavior），ISIS[10),11)] や Balboa[12),13)] などがある．

SimOS は，MIPS R3000/4000 や Alpha などいくつかの CPU と，その上で動作する

IRIX や Linux などのオペレーティングシステムの振舞いをシミュレーションすることができる．また，ハードウェアアーキテクチャや OS などの階層が分離しており，選択可能な Mipsy や MXS，Embra [14] などのいくつかの CPU シミュレータが存在する．SimOS は，そのようなアーキテクチャを対象としたプログラムをデバッグし，その性能の評価を行うことができる非常に強力なシミュレーション環境である．しかし，開発の初期段階において様々なアーキテクチャを試行する場合においては，アーキテクチャ構成の変更が容易である必要があり，SimOS では難しい．

ISIS は，抽象的なインタフェースに基づくマルチプロセッサのためのシミュレーショフレームワークであり，多くのライブラリ群を持つ．設計者は，ライブラリあるいは自らで定義したユニットを組み合わせてアーキテクチャ全体の構成をプログラミングをすることで，所望のシミュレータを得ることができる．提供されているクラスライブラリや独自に設計したクラスライブラリを用いることで，抽象的なシミュレーション環境を定義し，開発の初期段階でシステムを評価することが可能である．しかし，対象とするアーキテクチャの構成を変更するたびにコンパイルする必要があるなど，アーキテクチャの試行が簡単であるとはいえない．

コンポーネントに基づくシミュレーション環境である Balboa は，スクリプト言語と，C++ コンポーネントおよび，それらを接続するインタフェースで構成されている．対象とするアーキテクチャを構成するための様々なハードウェアユニットを試行するためには，スクリプト言語を用いてシミュレーションユニットを記述するほか，システムアーキテクチャのもとで管理されるための統一されたモデルに基づいて，それらの接続などの関係および転送されるデータを定義する必要がある．

提案するシミュレーション環境 MICS は，これらと比べ，他ツールとの協調動作および外部設定による簡便なシステム構成の試行ができることに特に強みがある．また，個々のシミュレーション要素，特にプロセッサのシミュレータには，既存の様々なシミュレータを容易に用いることができるため，導入コストを小さくすることができるという特徴を持つ．

3. シミュレーション環境 MICS

提案するシミュレーション環境 MICS（以下，MICS と呼ぶ）を実装した MICS シミュレータのスクリーンショットを**図 1** および**図 2** に示す．なお，このシミュレータは，ここに示す GUI フロントエンドだけではなく，CUI フロントエンドも備えている．図 1 は，シミュレータのメインウィンドウであり，シミュレーションを制御するボタンとシミュレーション結果などを表示するためのメインウィンドウからなる．この図では，プログラムの実行サイクル，シミュレーション時間およびその性能が表示されている．また，図 2 は，シミュレーション対象のアーキテクチャの構成および，仮想ビットマップディスプレイを示している．仮想ビットマップディスプレイでは，対象プログラムを実機で動作させているときのように動作結果を確認することができる．MICS では，サイクルベースでプロセッサシミュレータやメモリ，キャッシュのほか，様々な精度で実現される仮想ビットマップディプレイや音声デバイス，ネットワークデバイスなどの周辺デバイスを混在させることができる．したがって，実際の対象アーキテクチャ上でプログラムを実行しているように結果を確認することが容易である．それゆえ，その結果をもとにシステムレベルでの最適化を実現することも可能となる．

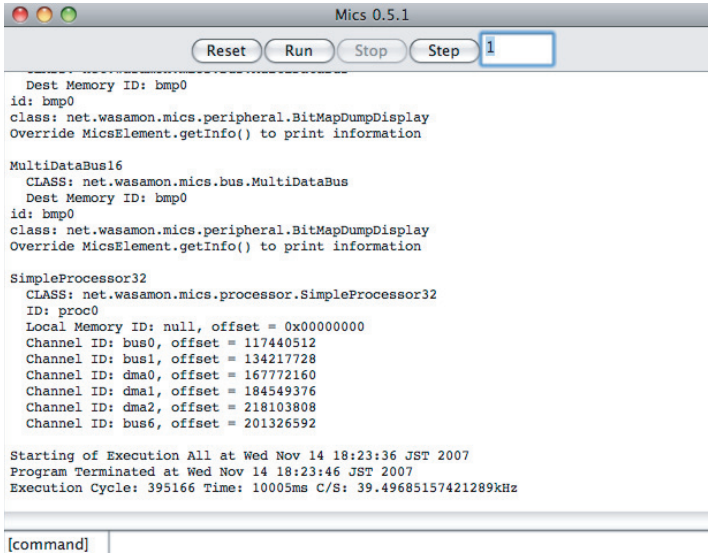


図 1　MICS の主ウィンドウ
Fig. 1　MICS main window.

MICS では，**図 3** に示すように，シミュレーション要素を MicsCompositeElement のインスタンスであるエンジンによって管理する．要素間の関係は XML によって記述されエンジンによって動的に管理されることで，それらの関係が疎になり，システム設計者は様々な

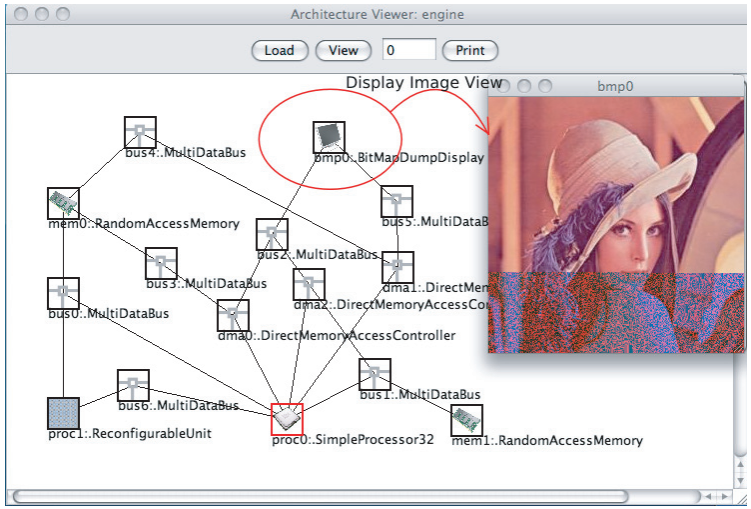


図 2　MICS アーキテクチャビューワと，仮想ビットマップディスプレイの表示
Fig. 2　MICS architecture viewer and instance of bitmap display.

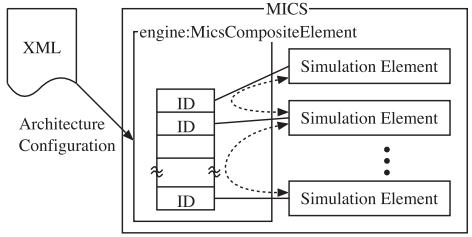


図 3　提案するシミュレーション環境 MICS
Fig. 3　Proposed simulation framework.

アーキテクチャ構成を容易に試行することができる．

3.1　シミュレータの構成要素

MICS においてシミュレーション対象となるすべての要素は，そのシミュレーションするハードウェア要素の性質に応じ，*ExecutableElement*，*DataBuffer*，*Channel* および *Channel-Connectable* の 1 つ以上のインタフェースを実装しなければならない．各インタフェースの示す性質については次の各項で述べる．これらのインタフェースは，要素をエンジンで管理するために必要なメソッドが定義された基本的なインタフェースである *IElement* を継承する．**図 4** は提案するインタフェースおよびエンジンのクラス図である．ここで，エンジンである MicsCompositeElement 自身もまた，*IElement* を実装するクラスであることに注意されたい．これは，MicsCompositeElement のインスタンスもまたシミュレーション要素として利用することができることを意味し，すなわち，シミュレーション要素の集合は階層的に定義できることを意味する．

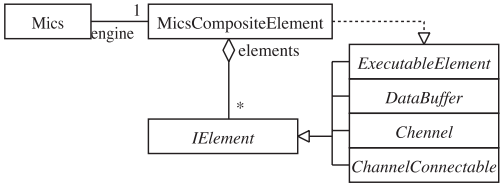


図 4　クラス図
Fig. 4　Class diagram.

提案するインタフェースを用いたシミュレーション要素の定義の例を **図 5** に示す．

3.1.1　*ExecutableElement*

ExecutableElement は，与えられるクロックに対して同期実行される要素が実装すべきインタフェースである．プロセッサやサイクルによる状態を持つシミュレーション要素が，このインタフェースを実装すべきクラスである．このインタフェースは，**図 6** に示すようにエンジンから定期的に呼び出される exec 関数群，*exec_first* と *exec_second* を実装することを規定する．MICS 上のシミュレーションでは，あるサイクルにおいて，まずすべての *ExecutableElement* を実装する要素の *exec_first* が呼ばれ，次に，それらの *exec_second* が呼び出される．これにより，同じサイクルにおける他のインスタンスの処理の影響を受けない処理と受ける処理を独立に記述することができる．*exec_first* および *exec_second* は，その実行結果を示す値として，それぞれ独立に次の 2 つの値からなる ExecInfo のインスタンスをエンジンに返す．

実行サイクル数　実行している処理が終了するまでに必要な残りサイクル数を示す．このサイクル数を消化するまで，エンジンは対応する exec 関数を呼び出さない．

終了可能フラグ　要素内に実行すべき処理が残っているかどうかを示すフラグ．たとえば，プロセッサなどで処理すべき命令が残っている場合には，このフラグを偽にする．エンジン内のループは，*ExecutableElement* 要素の exec 群から返された終了可能フラグが

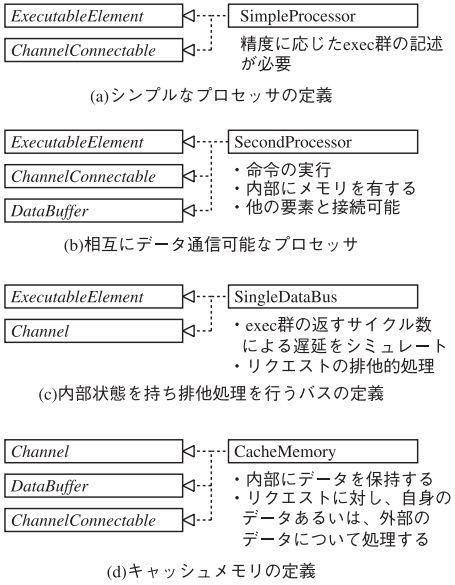


図 5　インタフェースに基づくハードウェアユニット定義の例
Fig. 5　Examples of MICS simulation units.

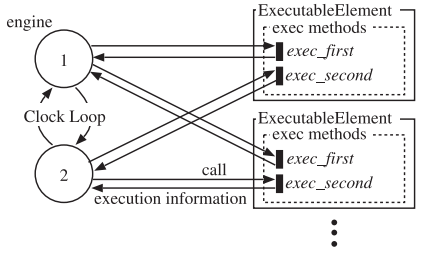


図 6　同期回路のシミュレーションモデル
Fig. 6　Simulation model of synchronous sequential Units.

すべて真である場合に停止する．

このようにモデルを単純化することで，精度の異なるシミュレーション要素が混在するアーキテクチャをサイクルベースでシミュレーションすることができる．このことは，3.4 節で述べる．

3.1.2　*DataBuffer*

DataBuffer は，データの集まりを保持し，読み書きすることができる要素が実装すべきインタフェースである．定義すべきメソッドの引数となるシミュレーション中に取り扱われるデータは，3.2 節で述べる *MicsData* によって抽象的に表現される．

3.1.3　*Channel*

Channel は，シミュレーションされる要素どうしにおいてデータを通信するための要素が実装すべきインタフェースである．このインタフェースを実装する要素においてデータアクセスのためのレイテンシや，通信の排他制御，ネットワーク構造を持つ通信路などのシミュレーションができる．

3.1.4　*ChannelConnectable*

ChannelConnectable は，非同期なデータ転送を受け付けることができるシミュレーション要素が実装すべきインタフェースである．このインタフェースでは，*Channel* インタフェースを実装したクラスに対するリクエストなど，他のシミュレーション要素からの非同期なデータ入力を受け付けるための *write_back* 関数を規定する．接続相手の要素は，*MicsData* を実装するクラスのインスタンスを引数として *write_back* 関数を呼び出すことで，非同期イベントを送出することができる．

3.2　MICS 内部のデータ形式

インタフェースに基づいてシミュレーション要素のクラスを記述するとき，MICS 内部で取り扱われるデータ形式は，インタフェース *MicsData* のもとで抽象化される．これは，アドレスおよび長さからなる低レベルのデータ形式や，複雑なデータアクセス形式のプロトコル，あるいはテキストによるコマンド形式のデータ通信などを統一的に扱う．これにより，様々なシミュレーション要素どうしの組合せをエンジンで気にすることなく取り扱うことができる．

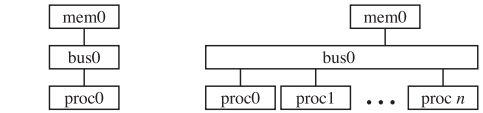
もちろんシミュレーション実行時に，予期しないクラスのインスタンスのデータがシミュレーション要素に受け渡された場合には，正しいシミュレーション動作を実現することはできない．しかし，これは実際のアーキテクチャにおいても，データフォーマットはデバイス間で共通でなければならず，その変換には何かしらの処理が必要となることであり，シミュレーション環境固有の問題ではない．

3.3　シミュレータ構成の定義

MICS では，ユーザによって定義された各要素の実装をシミュレーションの実行時に組み合わせることで，様々なアーキテクチャ構成を対象としたシミュレーションを容易に行え

```
<?xml version="1.0" encoding="euc-jp"?>
<mics>
 <element id="識別子" class="構成要素"/>
 <element id="識別子" class="構成要素" 属性="値">
   対象とする構成要素毎の設定
 </element>
</mics>
```

図 7　XML によるアーキテクチャ定義
Fig. 7　Architecture definition in XML.



(a)Single Processor　　(b) MultiProcessor

図 8　簡単なアーキテクチャ定義の例
Fig. 8　Example of simple architecture definition.

```
<?xml version="1.0" encoding="euc-jp"?>
<mics>
 <element id="mem0" class="RandomAccessMemory" length="0x7000"/>
 <element id="bus0" class="SingleDataBus" dest="mem0"/>
 <element id="proc0" class="SimpleProcessor" memory="0x7000">
   <channel id="bus0" offset="0x0a000000"/>
 </element>
</mics>
```

図 9　図 8 に示すアーキテクチャの定義
Fig. 9　Configuration description for Fig. 8.

る．**図 7** にアーキテクチャ構成を定義する XML 記述を示す．XML 記述では，ルートである mics 要素の下に，アーキテクチャを構成する要素に対応する`<element>`が列挙される．各`<element>`には，エンジンで要素を管理するための識別子とシミュレーション要素のクラス名を，ぞれぞれ属性値 id と class で指定する．ここで，要素の識別子はアーキテクチャ定義中で唯一の文字列でなければならない．`<element>`要素内の他の属性値および子要素は対象となるクラスに直接引き渡される．これにより，各クラスの独自の設定のために属性値および子エレメントの定義を利用することができる．

図 9 は，単純なメモリとプロセッサ，バスからなるアーキテクチャを構成する定義を示す．これは，**図 8** (a) に示されるアーキテクチャを定義する．ここで，クラス RandomAccessMemory は *DataBuffer* を実装したクラスである．さらに “proc0” の`<element>`の識別子だけを変えて複製することで，図 8 (b) に示すマルチプロセッサアーキテクチャを構成することができる．

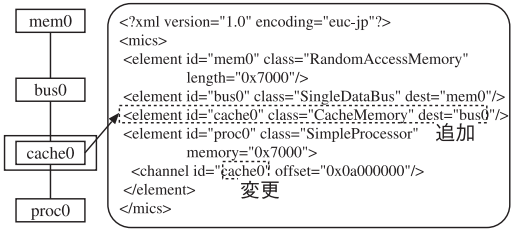


図 10　キャッシュを追加したアーキテクチャの例とその定義
Fig. 10　The architecture added cache and its description.

また，図 8 (a) のプロセッサとバスの間にキャッシュを挿入した **図 10** (左) に示すようなアーキテクチャを構成するためには，図 10 (右) のようにアーキテクチャ構成情報を変更すればよい．ここで，CacheMemory は，*Channel* と *ChannelConnectable* を実装して定義されたシミュレーション要素のクラスであるとする．

3.4　シミュレーション精度の混在

MICS のもとでは様々な精度のシミュレーション要素を同時に取り扱うことができる．すべての MICS における同期回路のシミュレーション要素は *ExecutableElement* を実装するため，*exec_first* および *exec_second* を持つ．これらが返す ExecInfo における実行サイクル数，すなわち次サイクルまでの必要待ち時間を用いてシミュレーション精度を混在させることができる．**図 11** に，MICS エンジンが *ExecutableElement* を実装するクラスのインスタンスを呼び出すシーケンス図を示す．anExecutableElement0 と anExecutableElement1 は *ExecutableElement* を実装する同期回路であり，MicsCompositeElement による exec 関数群の呼び出しに対し処理を行い，実行後 ExecInfo を返す．ExecInfo に () で付した値は，シミュレーション要素が ExecInfo に与える実行サイクル数，すなわち次に呼び出されるまでの必要サイクル数を示す．ここで，anExecutableElement0 は各サイクルに対して動作が定義されているサイクルレベルのシミュレーション要素であり，anExecutableElement1 は anExecutableElement0 よりも抽象度の高い，機能レベルのシミュレーション要素であるとする．図 11 に示すように，ExecInfo の実行サイクル数を消化する間は MicsCompositeElememt から対応する exec 関数が呼び出されることがない．この呼び出し制御によっ

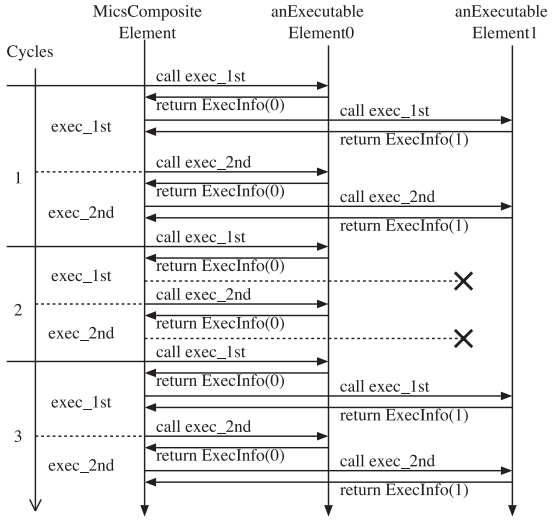


図 11　精度が混在するシミュレーションにおけるシーケンス図
Fig. 11　Sequence diagram for a mixed granularity architecture.

て精度の混在した要素を 1 つのターゲットアーキテクチャと見なしてシミュレーションすることが可能となる．

3.5　他のシミュレータとの協調

MICS では，定期的な動作を行うシミュレーション要素は *ExecutableElement* を実装しさえすればよい．したがって，**図 12** (a) に示すような，MICS シミュレーション要素として振る舞うラッパ要素を実装することで，容易に他のシミュレータを利用することができる．これによって，実績あるシミュレータを利用することができる．次に，具体例として GDB[7] との連携について説明する．

3.5.1　GDB との連携

GDB は GNU の提供するデバッグツールである．実アーキテクチャとの通信によってソースコードの命令レベルでのデバッグができるほか，ソフトウェアシミュレータとしてプログラムを実行することが可能である．GDB には数多くのプロセッサ向けのシミュレータの実装が含まれているため，GDB と連携することで MICS のために新規にプロセッサのシミュレータを実装することなく，既存のプロセッサのシミュレーションを行うことが可能となる．ここでは，MICS のシミュレーション要素として GDB のシミュレーション機能を利用する具体的な手法を述べる．

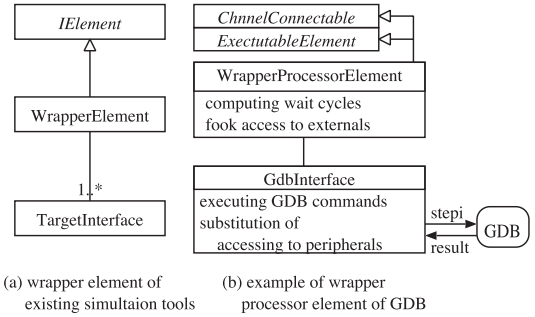


図 12　GDB とのインタフェース
Fig. 12　Wrapper simulation unit and its example for GDB.

図 12 (b) に，GDB を MICS の要素として定義する一手法を示す．ここで，GdbInterface は GDB プロセスを起動し操作するためのクラスである．また，*ExecutableElement* と *ChannelConnectable* を実装する WrapperProcessorElement が MICS のシミュレーション要素として振る舞う．MICS シミュレーション要素としてサイクルレベルのシミュレーションを行うために，WrapperProcessorElement は，GdbInterface が命令を発行するごとに次に発行する命令を解析し，その必要なサイクル数を算出する．また，メモリアクセスなどの外部の周辺デバイスが必要な場合には，WrapperProcessorElement で，アクセスする番地や専用命令をフックし，代替する．結果は GdbInterface を介して，適宜メモリやレジスタへ書き戻される．必要であればパイプラインのストールやメモリアクセスの遅延を考慮したサイクル数の算出などを行わなければならないが，それは求める精度に応じて実装すればよい．

各命令の実行サイクル数の算出および外部デバイスとのアクセスのフックのために，各プロセッサに対する独自のコードを記述する必要がある．機能レベルのシミュレータとして実装した MIPS シミュレータの場合は，Java ですべて実装すると 5,058 行であり，提案する手法による GDB を用いた実装では 3,327 行（GDB の入出力およびデータを処理する汎用のコード 1,388 行を含む）であった．

3.6　提案手法における制約

MICS では，各シミュレーション要素の定義は各設計者に委ねられている．様々な精度でのシミュレーション要素を混在させた場合，各要素単体のシミュレーション精度は自身で定

義した精度となるが，システム全体のシミュレーション精度は最も低い精度のシミュレーション要素の精度となる．

また，MICS ではシミュレーション要素の構成は，定義ファイルに従って実行時に決定される．そのため，あるサイクルにおけるシミュレーション要素の呼び出し順序を仮定することができない．したがって，複数の *ExecutableElement* において，それらが特定の順序で呼び出されることを前提として実装されている場合には，その前提と異なる順序で呼び出された場合のシミュレーション結果の正しさは保証されない．

これらの制約は，各シミュレーション要素を実装する設計者が責任を負うべきであるが，将来的には，MICS のもとで安全にシミュレーションできる要素を簡単に定義できるツールなどの整備によって，回避することができると考えられる．

4. 実装と評価

提案手法を実装し，そのシミュレーション速度および抽象化によるオーバヘッドを評価した．MICS の目的は高速なシミュレーションの実現ではないが，システム設計のための試行には現実的な時間でシミュレーションができなければならない．そこでまず，単一のバスで接続された対称型マルチプロセッサアーキテクチャを例題として，シミュレーション速度を評価する．次に，リコンフィギュラブルプロセッサを例題とし，その実装方法の違いによるシミュレーション手法のオーバヘッドを評価する．

評価を行ったホストコンピュータの環境は **表 1** に示すとおりである．

4.1 シミュレーション速度の評価

シミュレーション要素数の増加にともなうシミュレーション時間の増加を評価する．対象として仮定したアーキテクチャを **図 13** に示す．このアーキテクチャは，単一のバスで接続された対称型マルチプロセッサアーキテクチャであり，複数個の RISC 型のプロセッサを有する．各プロセッサは，Java による命令機能レベルのシミュレータである．プロセッサ数を 1 から 128 まで変化させた場合のシミュレーション時間を測定する．対象としたプログラムは，画像の部分検出を行うためのプログラムで，実行には 24,302,336 サイクルを必要とする．

シミュレーション時間を測定した結果を **図 14** に示す．各プロセッサ数に対応したシミュレーション時間の実測値を実線で，この結果に対して参考のために最小自乗法によって係数を求めた一次関数を破線で示している．実測値は，シミュレータを毎回起動し，同じプログラムを走らせる実験を 10 回行った平均値をプロットしたものである．プロセッサ数 1

表 1　ホストコンピュータ環境
Table 1　Host computer environment.

CPU	Intel Core2 6600 2.40 GHz
Memory	2.0 GB
OS	FreeBSD 6.2-RELEASE
Java	J2SE, 1.5.0-p4

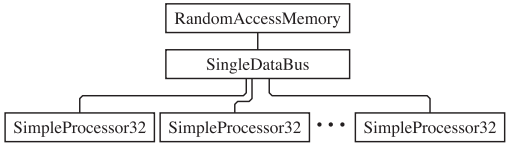


図 13　対称型マルチプロセッサアーキテクチャ
Fig. 13　Symmetric multiprocessing architecture.

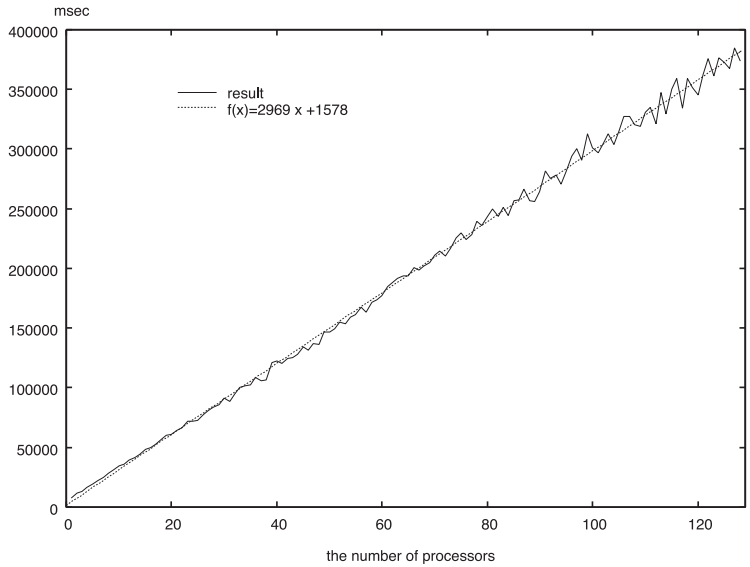


図 14　SMP アーキテクチャのプロセッサ数に対するシミュレーション時間の評価
Fig. 14　Simulating speed estimation of SMP, varying the number of processor elements.

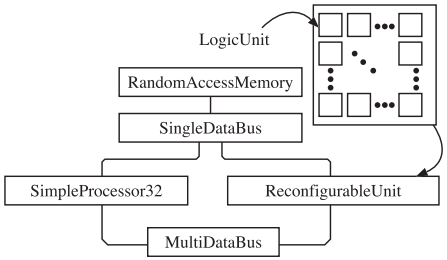


図 15　リコンフィギュラブルプロセッサを持つアーキテクチャ
Fig. 15　Target architecture which has reconfigurable processing unit.

の場合のシミュレーション速度は，3.06 M サイクル/秒と得られた．図 14 に示す結果より，MICS を用いたシミュレーションに必要な時間は，シミュレーション要素数の増加に対して正比例することが分かる．

4.2　抽象化によるオーバヘッドの評価

抽象化によるオーバヘッドを**図 15** に示すアーキテクチャを対象として評価する．ここで，リコンフィギュラブルプロセッサは，その演算機能が内部に持つロジックユニットの機能およびそれらの接続関係によって決定されるプロセッサである．各ロジックユニットでは加減算など 5 種類の演算および入出力データのシフト演算ができる．本論文では，内部に持つ個々のロジックユニットを機能レベルでシミュレーションできる実装を用いて評価する．

MICS では，リコンフィギュラブルプロセッサに相当するシミュレーション要素の実装手法として，たとえば，**図 16** に示す 3 つのパターンが考えられる．図 16 (a) は，内部のロジックユニットを個々の MICS シミュレーション要素として記述し，MicsCompositeElement のインスタンスとしてリコンフィギュラブルプロセッサを定義することで，内部のロジックユニットを動的に結合する．一方，図 16 (b) は，同様に内部のロジックユニットは，個々の MICS シミュレーション要素として記述されるが，それを静的に保持する固有のリコンフィギュラブルプロセッサクラスを定義する場合を示す．また図 16 (c) は，内部のロジックユニットを個々の MICS シミュレーション要素として記述せず，リコンフィギュラブルプロセッサを 1 つのクラスとして実装した場合を示す．図 16 (a) に示す実装の場合には，内部のロジックユニットの接続関係をアーキテクチャを定義するための XML によって記述することができる．そのため，個数や接続関係などの様々なパターンを容易に試行することができるという高い柔軟性を持つ．

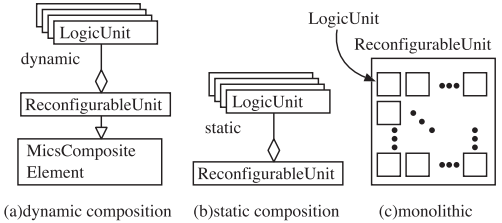


図 16　異る実装方法によるリコンフィギュラブルユニットの定義
Fig. 16　Definition of reconfigurable units in different types. In (a), the reconfigurable unit is defined as sub-class of MicsCompositeElement and its internal logic units are composed dynamically under MICS framework. On the other hand, the reconfigurable unit is defined by specific class which consists of logic units statically in (b), and the reconfigurable unit is written as a monolithic programming module in (c).

表 2　シミュレーション速度の比較
Table 2　A result of simulation time comparison.

	time (ms)	cycles/time (kHz)	# of lines
(a)	6,929	57.031	395
(b)	6,498	60.813	1,016
(c)	2,692	146.792	940

例として，これらのシミュレータを用いてグレイスケール変換プログラムを実行し，そのシミュレーション時間を評価する．例題としたプログラムでは，与えられた 256 × 256 ピクセルのカラー画像を 395,166 サイクルかけて白黒画像へと変換する．

シミュレーション時間および実装に要した行数を比較した結果を**表 2** に示す．(a) と (b) の結果は，MICS のもとで動的に構成を定義されたアーキテクチャによるシミュレーションのオーバヘッドが大きくないことを示している．また，1 つのモジュールとしてリコンフィギュラブルユニットを実装した場合が，この 3 つの例においては最もシミュレーション時間が短い．しかしながら，(a) は (c) に対し実行時間が約 3 倍程度であり，記述した行数は約 1/3 程度である．したがって，設計の初期段階では動的な結合を利用して容易に設計および検討をし，構成の確定に従って単一のクラスとして記述することでシミュレーション速度の向上をはかることができる．

5. ま と め

本論文では，まずシステムアーキテクチャの設計を対象としたシミュレーション環境 MICS を提案した．MICS は，抽象化したインタフェースの定義に基づいたシミュレーション構成要素の定義により，様々なハードウェアユニットの組合せを容易に試行できるフレキシビリティの高さを持つ．また，構成要素に対してサイクル数の算出だけを求めることにより，内部の設計に対する高い自由度を兼ね備えている．したがって，開発の段階に応じて，単純なスペックによる数値程度のシミュレーションから内部の動作に基づいた精度の高いシミュレーションまでの混在を可能にする．次に，MICS を実装し，様々なシミュレーションを実行することができることを示すとともに，マルチプロセッサアーキテクチャのシミュレーションを例として，シミュレーション時間がプロセッサ数に比例することを示した．またリコンフィギュラブルプロセッサの例によって，抽象化によるオーバヘッドについて議論し，たかだか数倍であることを示した．

今後の課題として，より大きなプログラムや複雑なアーキテクチャシステムの設計への応用と，実アーキテクチャでは評価困難な，シミュレータならではのパラメータを用いた評価手法への活用があげられる．また，精度の異なるシミュレーション要素との連携についての評価を行うことが必要である．さらに，MICS の抽象的なアーキテクチャ定義を用いた上流・下流設計の支援について検討し，より強力なシステム設計支援の実現を考えている．

MICS を http://mics.sourceforge.jp/で公開しオープンソースとして開発することで，ライブラリや適用対象の拡充を行うことを考えている．

参 考 文 献

1) Han, S.-I., Guerin, X., Chae, S.-I. and Jerraya, A.A.: Buffer memory optimization for video codec application modeled in Simulink, *DAC '06: Proc. 43rd Annual Conference on Design Automation*, New York, NY, USA, ACM, pp.689–694 (2006).
2) Viana, P., Gordon-Ross, A., Keogh, E., Barros, E. and Vahid, F.: Configurable cache subsetting for fast cache tuning, *DAC '06: Proc. 43rd Annual Conference on Design Automation*, New York, NY, USA, ACM, pp.695–700 (2006).
3) Liao, S., Tjiang, S. and Gupta, R.: An efficient implementation of reactivity for modeling hardware in the scenic design environment, *DAC '97: Proc. 34th Annual Conference on Design Automation*, New York, NY, USA, ACM, pp.70–75 (1997).
4) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An infrastructure for computer system modeling, *Computer*, Vol.35, No.2, pp.59–67 (Feb. 2002).
5) 中田　尚，中島　浩：高速マイクロプロセッサシミュレータ BurstScalar の設計と実装（シミュレータ），情報処理学会論文誌：コンピューティングシステム，Vol.45, No.6, pp.54–65 (20040515).
6) 吉瀬謙二，片桐孝洋，本多弘樹，弓場敏嗣：SimCore/Alpha Functional Simulator の設計と実装，電子情報通信学会論文誌 D-I，情報・システム，I-情報処理，Vol.88, No.2, pp.143–154 (20050201).
7) The GDB developers: GDB The GNU Project Debugger. http://www.gnu.org/software/gdb/
8) Riccobene, E., Scandurra, P., Rosti, A. and Bocchio, S.: A model-driven design environment for embedded systems, *DAC '06: Proc. 43rd Annual Conference on Design Automation*, New York, NY, USA, ACM, pp.915–918 (2006).
9) Rosenblum, M., Herrod, S.A., Witchel, E. and Gupta, A.: Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology: Systems and Applications*, Vol.3, No.4, pp.34–43 (1995).
10) Wakabayashi, M., Inoue, K. and Amano, H.: ISIS: Multiprocessor Simulator Library, *Applied Informatics*, Hamza, M.H. (Ed.), pp.198–200, IASTED/ACTA Press (1999).
11) Masaki, W. and Hideharu, A.: *Environment for multiprocessor simulator development*, pp.64–71 (2000).
12) Doucet, F., Shukla, S., Gupta, R. and Otsuka, M.: An Environment for Dynamic Component Composition for Efficient Co-Design, *DATE '02: Proc. Conference on Design, Automation and Test in Europe*, Washington, DC, USA, IEEE Computer Society, p.736 (2002).
13) Doucet, F., Shukla, S.K., Otsuka, M. and Gupta, R.K.: BALBOA: A component-based design environment for system models, *IEEE Trans. CAD of Integrated Circuits and Systems*, Vol.22, No.12, pp.1597–1612 (2003).
14) Witchel, E. and Rosenblum, M.: Embra: Fast and flexible machine simulation, *SIGMETRICS '96: Proc. 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, ACM, pp.68–79 (1996).

(平成 20 年 1 月 8 日受付)
(平成 20 年 7 月 1 日採録)

三好　健文（正会員）

昭和 56 年生．平成 15 年東京工業大学工学部電気電子工学科卒業，平成 17 年同大学院電子機能システム専攻修士課程修了．平成 19 年同大学院物理情報システム専攻博士課程修了．博士（工学）．同年東京大学情報理工学系研究科特任助教．現在に至る．自動並列化コンパイラ，HW/SW 協調設計に関する研究に従事．IEEE，電子情報通信学会各会員．

杉野　暢彦

昭和 39 年生．昭和 61 年東京工業大学工学部電気電子工学科卒業，平成元年同大学院物理情報工学修士課程修了．平成 4 年同大学院電子物理工学専攻博士課程修了．博士（工学）．現在，東京工業大学大学院総合理工学研究科物理情報システム専攻准教授．VLIW や DSP 向けのコード最適化，ディジタル信号処理アルゴリズムの実現技術の研究に従事．IEEE，電子情報通信学会各会員．