Vol.2011-SLDM-149 No.27
Vol.2011-EMB-20 No.27
2011/3/18

# 組込みシステム向け リアルタイム性能シミュレーション

池　　敦　　田宮　豊
タシ　デビッド　　桑村　慎哉

㈱富士通研究所

組込みシステムが複雑化して，マルチコア構成も一般的となっている現況では、各 CPU コアの機能・性能・電力等のシミュレーション処理について，より高い処理速度と処理精度を実現することが要求されている。機能シミュレーションについては、ホストコード実行時に動的変換を行う JIT コンパイル方式が主流となっている。しかし性能・電力のシミュレーションについては対応が遅れている。本論文では、JIT コンパイル方式の性能・電力のシミュレーションを精度を犠牲にせずに高速実行する手法を提案する。

# A Method of Fast Cycle Simulation for Embedded Systems

ATSUSHI IKE　YUTAKA TAMIYA
TACHE DAVID　SHINYA KUWAMURA

Fujitsu Laboratories Ltd.

On recent system-on-chips (SoCs), size of system has grown rapidly along with multi-core CPUs. Meantime how to simulate SoC's cycle-based performance and power as well as functionality, in both higher speed and higher accuracy, has become more and more critical issues. For a functional simulation, JIT-compile method which dynamically generates and executes host instructions on the fly during the execution is very effective in fast execution. But utilizing JIT-compile method for a cycle and power simulation has several challenges. In this paper, we investigate the reasons and propose a new method of fast cycle simulation using JIT-compile without sacrificing the accuracy.

## 1. はじめに

近年、組込みシステムの複雑化が進み、マルチコア構成も一般的となりつつある。また処理速度もモバイル端末では 1GHz を超えることも珍しくなくなり、これらに伴い、CPU コアのシミュレーションも、これまで以上に高い処理速度と、性能と電力に関して高い精度が要求されてきている。

このうち、機能レベルシミュレーションに関しては従来、CPU コアで実行される命令コードを一つずつ処理していくインタープリタ方式(図 1 左)が用いられた。しかしこの方式では命令毎に発生する命令デコードやレジスタ操作などのオーバヘッドが大きく、処理速度に問題があった。

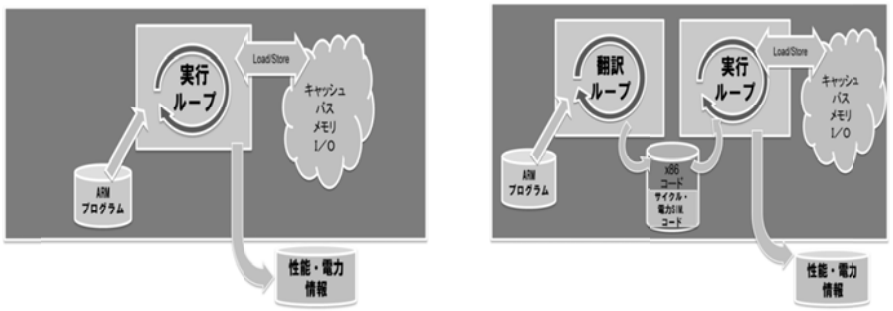


図 1　インタープリタ方式と JIT コンパイル方式

そこで近年では、命令コードをホスト CPU の命令コードに変換しながら処理を進める JIT（Just-in-Time）コンパイル方式(図 1 右)が主流となり、十分な能力を持つホスト環境を用意すれば、システムレベルでも比較的高速な機能レベルシミュレーションが可能となってきた。

例えば OSS（Open Source Software）である QEMU は JIT コンパイル方式を採用するシステムレベルの機能レベルシミュレータで、2GHz の Intel Core2 クラスホスト環境であれば、ARM Cortex-A8 を含む組込みシステムの機能レベルシミュレーションが 100MIPS を超える実用的なスピードで実行可能である。

JIT コンパイル方式による機能レベルシミュレーションでは、シミュレーション対象であるターゲット CPU について、実行中のプログラムに出現するターゲット CPU の命令をシミュレーションを実行するホスト CPU の命令に置き換え（翻訳ループ）、以降ではその置き換えた命令をできるだけ再利用して実行する（実行ループ）。この再利用率が一般に 99%程度まで高くなるため、翻訳ループのオーバヘッドが相対的に小

さくなり、結果インタープリタ方式に比較して極めて高速な実行が可能となる。

一方で、JIT コンパイル方式を性能・電力込みのシミュレーションに適用する手法については、余り進んでいない。

一般的な RISC 型 CPU 内部にはメインパイプラインの他に、命令 CACHE、データ CACHE、TLB、分岐ヒストリーバッファ、コプロセッサなど個別のステートマシンを有するサブコンポーネントが存在する。それらの内部状態の組み合わせによって命令単位で処理時刻がダイナミックに変動する。処理時刻を考慮しない機能レベルのシミュレーションと異なり、性能・電力込みのシミュレーションの場合には、こうした各サブコンポーネントの内部状態変化を無視すると精度が著しく低下してしまう。

例えばデータ CACHE がミスすると、メインメモリにアクセスするためのウェートが発生する。この時メインパイプライン内の後続命令に依存関係が無ければパイプラインはストールせずに処理を進めることができる(図 2 の mul 命令)。この場合、単純に CACHE ミスのペナルティ時刻を加算するという単純な処理では十分な精度は出せない。

内部状態の変化に対応するコードを命令毎に埋め込むことも考えられるが、それではインタープリタ方式と変わらないものとなり、処理時刻が非常に大きくなる。

近年ではチップ面積にゆとりが生じ高速化のためにより多くのリソースを追加できるようになってきた。メインパイプラインも単純なシングルイシュー型からスーパースカラ型、さらにアウトオブオーダ型などの複雑なマイクロアーキテクチャの採用が組込みでも急速に進んでおり、それに伴い TLB や CACHE などの状態変化による影響はさらに受けやすくなってきている。

こうした背景などから、高速かつ精度の高い性能・電力込みのシミュレーションに JIT コンパイル方式は一般に適さず、現実的な手法としてはほとんど活用されて来なかった。

本論文では、TLB や CACHE などのサブコンポーネントの内部状態に一定の前提を置いたホスト CPU の命令コードの生成と、その前提が崩れた際の動的な補正ロジックとを相補的に活用する JIT コンパイル方式を提案する。その結果、JIT コンパイル方式の高速性を維持したまま、性能・電力の十分な精度も同時に達成することが可能となった。

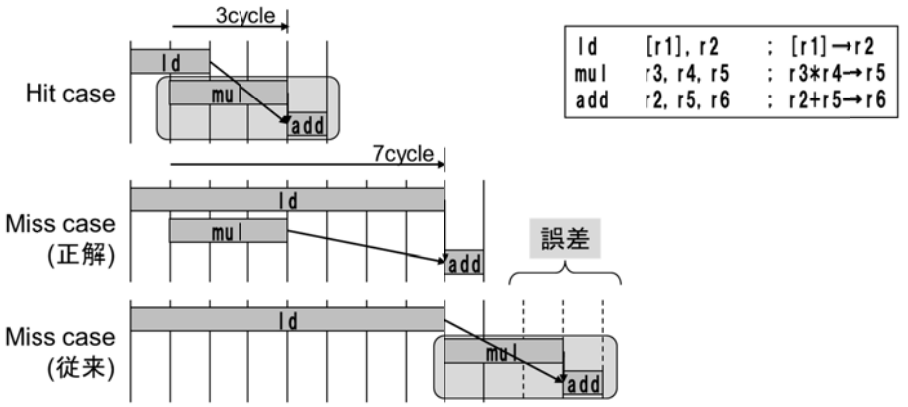


図 2　CACHE ミス時の処理例 (ld ミスが 1 ヶのケース)

## 2. 高速性能シミュレーション

本章では、JIT コンパイル方式でサイクルシミュレーションを行うにあたり、どういった手法と課題があるのかを考察し、その上で我々の手法を提案する。

なお説明を簡単にするため、以降では、ターゲット CPU を ARM、ホスト CPU を x86 とする。

### 2.1 従来方式

機能レベル処理の JIT コンパイル方式では、その翻訳ループにおいて、ARM 命令の一つずつを対応する x86 コードに変換していく。

サイクルシミュレーションの場合でも同様で、次図 3 の様に、性能レベル処理の x86 コードをその後に追加していく。

Vol.2011-SLDM-149 No.27
Vol.2011-EMB-20 No.27
2011/3/18

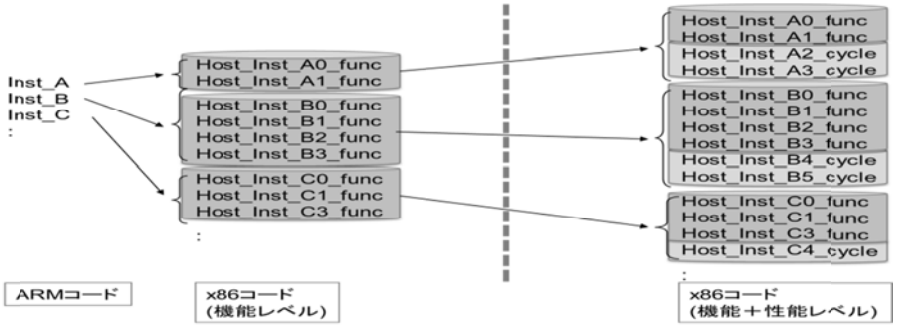


図 3　JIT コンパイル方式での性能シミュレーション

ARM の一命令に対して、機能レベル処理の x86 コードが平均 Nf 命令、性能レベル処理の x86 コードが平均 Nc 命令だとすると、シミュレーション時刻 Tsim は、

$$Tsim_{\text{性能レベル}} = Tsim_{\text{機能レベル}} \times \frac{N_f + N_c}{N_f}$$

という関係となる。

ここで例えば Nf が 8 命令だとすると、Nc も 8 命令であれば、性能レベルのシミュレーション時刻は機能レベルのシミュレーション時刻の 2 倍となる。同様に Nc が 80 命令であれば 11 倍、逆に Nc が 4 命令であれば 1.5 倍で済む。つまり Nc を如何に小さく抑えられるかが、性能レベル処理の速度上での大きな課題となる。

一方、先に述べた様に、TLB や CACHE といった内部ステートマシンを有するサブコンポーネントを持つ CPU は、それらの内部状態によって命令単位での処理時刻が変動し得るため、この変動分を考慮しないと精度が向上しない。これを先の図 2 で示した例で再び考えてみる。

まず ld/mul/add の各命令の実行レイテンシ情報として次表 1 を用いる。

| | 後続に依存関係有り | 後続に依存関係無し |
|---|---|---|
| ld 命令 | 2 サイクル | 1 サイクル |
| mul 命令 | 3 サイクル | 1 サイクル |
| add 命令 | 1 サイクル | |

表 1　命令のレイテンシ

またデータ CACHE のミスペナルティを 7 サイクルとする。

これらの情報を用いると、図 2 の各命令で生成される機能＋性能レベルのホストコードは例えば次の様になる。

ld 命令　<ld の機能レベル処理コード>
　　<CACHE がミスした場合に時刻を 7 サイクル進めるコード> ①
　　<時刻を 1 サイクル進めるコード(依存関係無し)> ②
mul 命令　<mul の機能レベル処理コード>
　　<時計を 3 サイクル進めるコード(依存関係有り)> ③
add 命令　<add の機能レベル処理コード>
　　<時計を 1 サイクル進めるコード> ④

ここで時刻を進めるコード①～④が性能シミュレーションに対応するホストコード部分である。また①は ld 命令が CACHE にミスした場合に発生するミスペナルティ分の 7 サイクルを加算するコードで、②～④が各命令の実行レイテンシを加算するコードである。

この一連のホストコードを実行すると CACHE がヒットする場合で 1＋3＋1＝5 サイクル、CACHE がミスする場合(図 2 の従来ケース)では 7＋1＋3＋1＝12 サイクル、それぞれ時刻が進むことになる。しかし CACHE がミスするケースの正解は、依存関係の無い mul 命令が先に処理されるため、11 サイクルではなく 9 サイクル(図 2 の正解ケース)である。つまりこの実装方法では 3 サイクル分の誤差が発生してしまうことになる。

またこの例では ld 命令が 1 つで、かつそれが CACHE ミスするだけであった。実際には ld 命令が複数同時に処理され同時に CACHE ミスしたりもする。さらにスーパースカラ型やアウトオブオーダ型の場合では、組み合わせはさらに複雑となりかつ誤差も拡大することになる。

### 2.2 誤差補正

次に発生した誤差を動的に補正する仕組みについて考えてみる。

再び図 2 の例を用いると、例えば次の様な情報を用意することで、発生する誤差を補正することを考えてみる。

A. ld 命令の結果に依存する最初の後続命令はいつか

Vol.2011-SLDM-149 No.27
Vol.2011-EMB-20 No.27
2011/3/18

上記 A.の情報は図 2 では 3 サイクルであるので、次の様なホストコードを生成することで CACHE ミスケースは 4＋1＋3＋1＝9 サイクルに補正され、正解ケースとも一致する。

ld 命令　<ld の機能レベル処理コード>
　　<CACHE がミスした場合に時計を 7 － 3 ＝ 4 サイクル進めるコード> ①'
　　<時計を 1 サイクル進めるコード(依存関係無し)> ②
mul 命令　<mul の機能レベル処理コード>
　　<時計を 3 サイクル進めるコード(依存関係有り)> ③
add 命令　<add の機能レベル処理コード>
　　<時計を 1 サイクル進めるコード> ④

さらに複数の ld 命令がミスするケースでも、次の B.の様な情報を用いることで補正することを考えてみる。

B.　ld 命令の前の ld 命令はいつ実行されたか

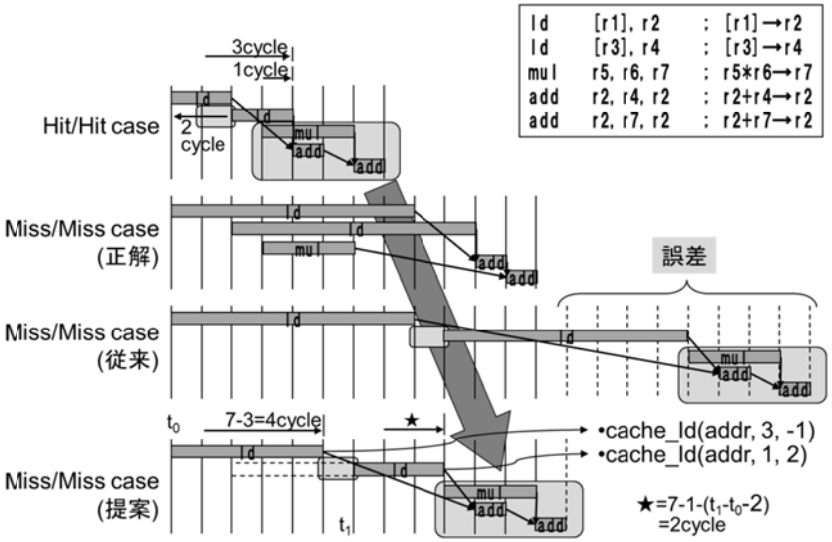


図 4　CACHE ミス時の補正例 (ld ミスが 2 ヶのケース)

図 4 は ld 命令が 2 つあり、かつそれぞれが CACHE ミスするケースである。補正を行わないと ld 命令それぞれで CACHE ミスのペナルティ 7 サイクルが加算され、誤差は 8 サイクルにまで大きくなってしまう。しかし B.の情報を用いることで、直近の ld 命令が直前で実行されており、2 つの ld 命令が並行して処理できることがわかる。これと A.の情報とを用いることで、図の様な補正を行うと、結果正解値に対して＋1 サイクル程度の誤差に修正することができる。

ここで示した各補正は、CACHE ミス時の誤差分を完全に補うことを保証するものではない。しかし多くのケースではこれでも十分な補正が可能であり、また補正のためのホストコード追加も小さくて済む。一方、より多くの情報を追加することでさらに精度を向上させることも可能である。

### 2.3 提案方式

補正情報 A.と B.は、あらかじめ前後の命令関係を把握することで抽出できるものである。また、各命令の実行レイテンシも、シングルスカラー型の CPU であれば単純であるが、スーパースカラー型やアウトオブオーダ型の CPU の場合は、単純なレイテンシ情報の積み重ねだけでは精度は上がらない。そこで、こうした補正情報やレイテンシ情報を正確に抽出するために、あるまとまった命令単位（例えばベーシックブロック）で一度サイクルシミュレーションを実行する。このシミュレーションを

**「予測に基づく第一のシミュレーション」**

と呼ぶ。予測に基づくとは、CACHE、TLB やプリフェッチ、分岐ヒストリバッファなどについて、より確率の高い挙動の方に固定してシミュレーションを行うことを指す。例えば CACHE は一般的にヒット率が非常に高いのでヒットに固定する。TLB もヒット率が高いので同様にヒットに固定する。この様な前提の下にシミュレーションを行い、各命令のレイテンシ情報と、補正情報 A.や B.の抽出を行う。その後、これらの情報を用いて、性能レベル処理用のホストコードを生成する。この際、各予測が外れた場合に、次の補正を行う処理コードも追加される。

ホストコードを実行する実行ループでは、上記予測が外れた場合に補正の処理が行われる。これは補正情報 A.や B.を用いた簡単な補正でも良いが、場合によってはその前後の命令列を再シミュレーションするということもあるために、この処理を

**「予測ミス時の第二のシミュレーション」**

と呼ぶ。

これら一連の提案フロー全体を次図 5 に示す。

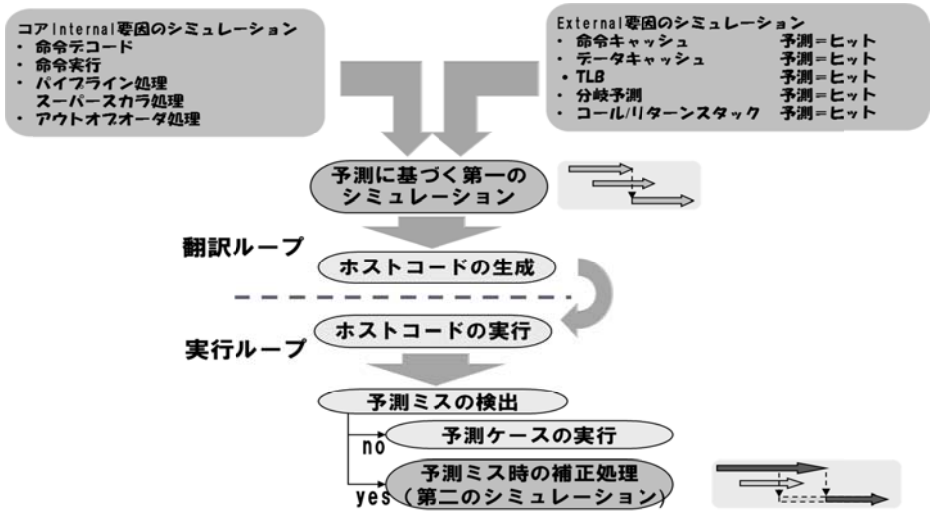


図 5　提案方式での JIT コンパイル型サイクルシミュレーション

## 3. 高速サイクルシミュレーションの処理量

本章では、提案する高速サイクルシミュレーション手法の各処理において必要となる処理量について考察する。

なお以下では、次の前提を置く。

| | |
|---|---|
| インタプリタで ARM 1 命令の機能処理に必要な x86 命令数 | 50 |
| インタプリタで ARM 1 命令のサイクル処理に必要な x86 命令数 | 100 |
| JIT コンパイル翻訳ループの実行時刻 | 全体の 1% |
| JIT コンパイル実行ループで TLB/CACHE 等の実行 | 全体の 20% |
| JIT コンパイル実行ループで TLB/CACHE 等の処理に必要な x86 命令 | 10 |
| JIT コンパイル実行ループで予測のミス率 | 10% |

表 2　処理量計算の前提

まず機能レベルシミュレーションに関して、インタプリタ方式と JIT コンパイル方式の処理量を比較する(図 6)。

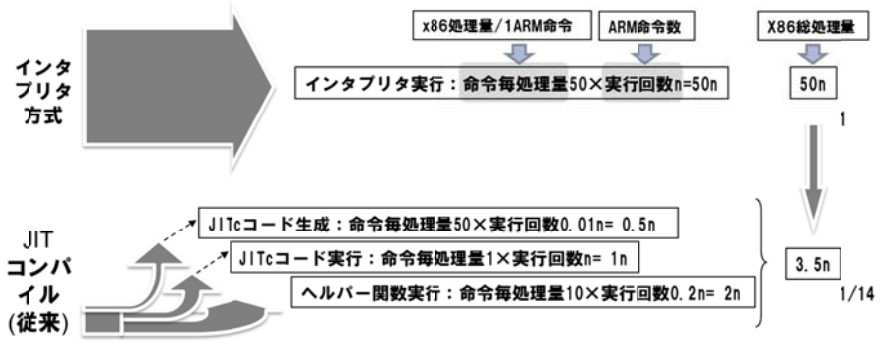


図 6　JIT コンパイル方式の処理量（機能レベル）

続いて、性能レベルシミュレーションを行う場合に、インタプリタ方式と JIT コンパイル方式の処理量とを比較する(図 7)。

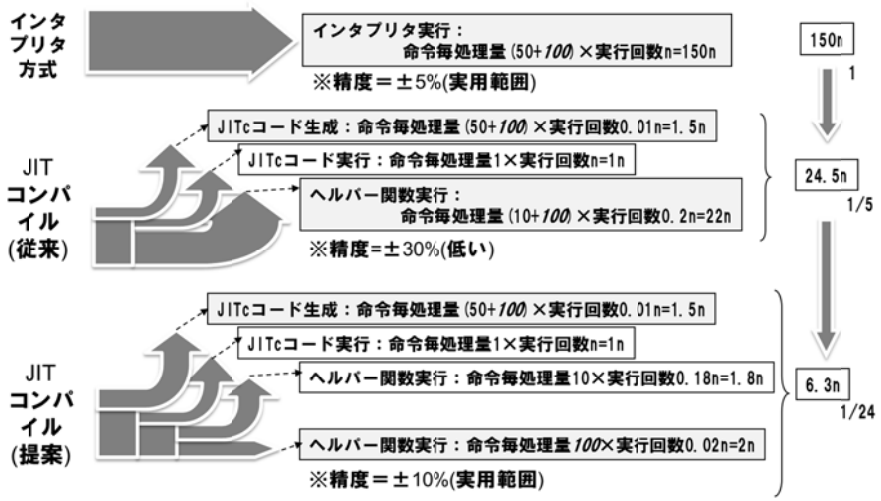


図 7　JIT コンパイル方式の処理量 (機能＋性能レベル)

これら比較によると、従来の JIT コンパイル手法では、機能レベル処理については非常に有効でありインタプリタ方式に比較すると 1/14 の高速化が期待できた。しかし性能レベル処理を含むと、インタプリタ方式の 1/5 程度の高速化しか期待できない。これを処理量で考えると、機能処理時の JIT コンパイルの 3.5n に対して 24.5n と、実に 8 倍ほど処理量が増える（遅くなる）。

これに対して、今回提案した手法によると、インタプリタ方式の 1/24 もの高速化が期待でき、これを処理用で考えると 6.3n となるので、機能レベル処理の 3.5n と比較しても 1.8 倍で済む。

さらに、本提案手法を Android Emulator (QEMU)に適用した場合、ARM Cortex-A8 で Android 2.2 (Froyo)の OS ブートおよび API Demos のサンプル実行が、機能レベル処理に比較して、それぞれ 1.5 倍、1.4 倍のスピードで実行可能であることを確認できた。

こうした高速化が可能となるのは、次図に示す様に、従来重たい処理であった第一のシミュレーションと、今回追加される補正のための第二のシミュレーションが、いずれも全体の実行に対して 1%、2%と非常に小さく、これらの処理が薄められた効果が大きいと考えられる。

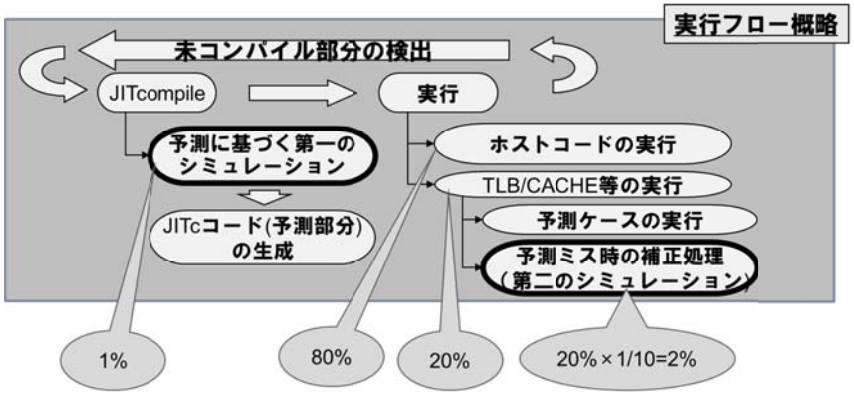


図 8　JIT コンパイル方式での処理量(概要)

## 4. おわりに

本論文のシミュレーション手法によれば，CPU の性能シミュレーションを高速かつ高精度に行うことが可能となる。

従来 JIT コンパイル方式は機能レベルのシミュレーション実行の高速化には非常に有効であったが、命令毎にあらかじめホストコードを生成しなければならないため、内部ステートマシンを個別に有する CACHE や TLB、分岐ヒストリーバッファといったサブコンポーネントを有する CPU の場合、これらの内部状態によって変化する処理を翻訳時に特定することが難しく、故に現実的な手法としては活用されにくかった。

今回提案するシミュレーション手法では、あらかじめ確率の高いであろう方向に条件を固定、まとまった命令単位で各命令のレイテンシ情報や補正情報を抽出する第一のシミュレーションを実施する。そしてその際に得られた補正用の付加情報をヒントとして、シミュレーション実行時にこの前提条件が外れると動的な誤差補正(第二のシミュレーション)を行い精度を補償するものである。

考察の結果、全体のサイクルシミュレーションとして重い部分は確率的に薄めることができ、その結果、十分な精度のサイクルシミュレーションを実現しても機能レベルシミュレーションに対して 50%程度のオーバヘッドだけで性能レベルシミュレーションが可能となることを確認、本手法の基本的な有効性を確認できた。

この様に本提案手法により、CPU を含むシステムの性能，電力のシミュレーションを高速に実施することが可能となり，従来では現実レベルでの実行が困難であった，大規模なシステム全体の性能，電力の評価や解析，予測などを容易に行えるようになることが期待できる。

## 参考文献

[1] “QEMU, a fast and portable dynamic translator,” in Proceedings of USENIX Annual Technical Conference, June 2005, pp. 41–46

[2] F. Bellard. QEMU. [Online]. Available: http://bellard.org/qemu/index.html

[3] Synopsys Inc., “Platform Architect,” http://www.synopsys.com/Tools/SLD/VirtualPrototyping/Pages/PlatformArchitect.aspx.

[4] CoWare Inc. CoWare Processor Designer. http://www.coware.com/PDF/products/ProcessorDesigner.pdf.

[5] J. Schnerr, O. Bringmann, and W. Rosenstiel. Cycle Accurate Binary Translation for Simulation Acceleration in Rapid Prototyping of SoCs. In Proceedings of the Design, Automation and Test in Europe (DATE) Conference, pages 792–797, 2005.

[6] VaST Systems Technology. CoMET. http://www.vastsystems.com/docs/CoMET_mar2007.pdf.

[7] J Schnerr et al, “High-Performance Timing Simulation of Embedded Software”, DAC 2008, June 8-13