命令レベルシミュレーションの
高速実行に関する研究

平成 20 年度

近江谷　康人

論文要旨

市販マイクロプロセッサは年率 50～20%の速度性能の向上を果たしてきた．その結果，従来は専用ハードウェアを併用して行ってきたシミュレーションを，市販 CPU に搭載したソフトウェアのみで実現できるようになってきた．CPU 性能を利用したコンピュータのシミュレーションの実用範囲が拡大されてきているが，その実行性能には更なる高速化が求められている．本研究は，CPU 性能を利用したシミュレーションにおいて，バイナリ互換のプログラムを機械命令シミュレーションで実行する高速化方式に焦点を当てたものである．まず，命令レベルシミュレーションの基本となるインタプリタ方式の研究を行い，次に更に高速性を要求される用途を対象とした動的バイナリ変換方式の研究を行った．

インタプリタ方式は，特別なハードウェアなしに命令セットの異なる CPU を用いてバイナリ互換のコンピュータ製品を実現するのに適したシミュレーション技術である．半導体プロセスの微細化に伴い CPU に搭載できる機能が拡大し性能が向上してきたが，開発費の高騰によりコンピュータメーカ独自の命令セットを持つCPUの定期的な開発が困難になってきた．そのため，高性能の市販マイクロプロセッサを用いてバイナリ互換を実現するアーキテクチャシミュレーションは，有効な手段である．なかでも C 言語実装によるインタプリタ方式はアセンブリ言語記述されたものに比べて性能が低いが，原理が単純なため例外仕様などの厳密な実装が容易で，ホストアーキテクチャ依存度が低いなどの特徴を持ち，開発費，設計品質，保守性の点で有利である．

本研究では，C 言語実装によるインタプリタの実行速度性能を高める手法を研究し，インタプリタ試作による評価を行った．試作は，5 種類のシミュレーション対象の命令セット，2 種類の RISC 型のシミュレーション実行マシン，3～5 種類の実装方式による計 45 種類に及んだ．この結果，高性能なマイクロプロセッサを用い C 言語でインタプリタを実装し，次の 2 点を明らかにした．

(1) インタプリタの共通処理部（コアループ）はその処理時間の比率が 70～80%と高いため，コアループの試作のみで速度性能の目安がつくこと．

(2) C 言語に適合したコアループの実装（改良 function 方式）を採用するとシミュレーションの速度性能を 1.3～2.2 倍までに向上させ，それによりアセンブリ言語記述の理想的なインタプリタの 80%程度の速度性能を達成できること．

バイナリ変換方式には，専用ハードウェアがないと例外処理が課題となり異なる命令セットへのエミュレーション適用が困難になるという制約があるが，その高速性を活かした応用分野として性能解析ツールへの適用がある．高性能な CPU を搭載した組込み機器開発では，パイプラインストール，メモリウォール問題，ソフトウェアの肥大化などにより，製品のシステム性能の見積りが困難になってきている．ハードウェア構成だけでなくソフトウェアの改善にも利用できる性能解析ツールが求められており，キャッシュメモリやバス及びプログラム挙動を捕捉するために命令レベルのシミュレーションが必要となる．特に，設計段階で様々なプログラムの評価データをインタラクティブに採る場合には，シミュレーション速度が問題になる．

本研究では，キャッシュメモリやプログラム動作の解析ツールである ESPRIT/sim の速

度性能の向上をねらい，簡易的な実装手法を用いた動的バイナリ変換アクセラレータの高速実行の研究をした．この動的バイナリ変換アクセラレータは，次に示す手法などを採ることにより実行速度と開発投資の点で有利である．

- 組込み機器で今後主流となる異種マルチプロセッサに対応している．
- 3 階層方式のトランスレータによりトランスレータとシミュレーション対象の命令セットの依存性及びトランスレータとホスト命令セットへの依存性を極力排除し，更にトランスレータのコードの共通部を増やす．
- 開発費に見合った変換の最適化レベル及び変換対象とする命令種の実装を選択できる．

ここでは，SPEC CINT95 を用いた評価と SPLASH-2 を用いた評価を行った．その結果，インタプリタ方式の 3〜14 倍の速度性能の向上を実現しており，本方式はアクセラレーションとして有効である．

Abstract

Commercial microprocessors have increased their performance at 20%-50% per year. As the result, software based simulations have substituted for simulations using hardware accelerators. Although computer simulation becomes more practical due to the improvement of CPU performance, there is still a requirement to make them faster. This research focuses on high-speed execution of computer simulation that executes binary-compatible programs by utilizing CPU peak performance. This paper describes two technologies of "interpreter" and "dynamic binary translator".

The interpreter is a useful technique for attaining a binary compatible computer by using a CPU that has a different instruction-set-architecture without any special hardware. Highly advanced semiconductor process technologies boost up not only CPU functionality and performance but also developing cost of a new computer product. An architectural simulation using commercial high-performance microprocessors is a cost-effective way for developing a new computer with keeping binary compatibility to solve this increasing development cost issue. In particular, an interpreter that has a simple structure and uses architecture-free C-language description is practical for considering development cost, quality, and maintainability.

This research evaluated the execution speed of 45 sets of interpreters written in C-language; five types of legacy instruction set architectures, two RISC hosts, and three through five types of different implementations. As a result of this evaluation, it has appeared that: (1) The core-loop, which is common to all instructions, wastes 70 to 80 percent of the whole execution time and it proves that a prototyping of core-loop is usable for rough performance estimation. (2) Proposed implementation techniques to fit C-language called as "improved function method", raise simulation performance up to 1.3 through 2.2 times of original performance, and they achieved approximately 80 percent performance of ideal simulator performance written in an assembly language.

The binary translator has constraints such that it requires hardware assist for exception handling. However, it is an effective technology for high-speed simulators such as performance analyzing tools. Nowadays, embedded system is becoming more difficult to develop. Because, forecasting system performance and achieving the planned performance are complex tasks, due to issues such as pipeline-stall and memory-wall lying on enlarged-software with high-performance microprocessors. To solve these issues for software improvement as well as hardware design, analyzing tools are expected to simulate binary instruction codes, for analyzing cache memories, buses, and program behavior. In particular, the simulation speed-up of a design stage is to reduce developing time with an interactive environment.

This research has sought for dynamic binary translation accelerators employed by "ESPRIT/sim", which is an effective analyzing tool. This binary translator is useful in both of execution speed and developing cost by adopting as following methods. (1) Heterogeneous multiprocessor simulation capability that will be required for embedded systems. (2) Three-layered-translator that minimizes dependency between translators and simulated instruction-set-architectures or dependency between translators and host instruction-set-architectures, and that increases common parts for translators. (3) Selections of optimum translation level and translation instruction types to be translated, in order to meet development cost as planned. This research evaluated the binary translated simulation which ran on SPEC CINT95 and SPLASH-2 benchmarks, and showed the speed up of three times to fourteen times against to target interpreters.

目次

図目次

表目次

第1章 序言

本章では研究の背景と目的，従来の研究，本研究の特徴を述べる．また，最後に本論文の構成を示す.

1.1. 研究の背景と目的

コンピュータ製品の新機種を開発して市場投入するときには，ユーザ資産の継承のために前機種との互換性の維持が極めて重要となる．コンピュータメーカは独自アーキテクチャ路線を採る機種向けに命令互換の CPU の開発を継続してきた．市販のマイクロプロセッサを使用した機種向けには命令互換のCPUを採用して互換性を維持してきた. しかし，半導体技術の進歩により集積度の向上はしたがその一方 LSI 開発費が高騰したため，独自アーキテクチャ仕様の CPU 開発は採算をとれなくなってきている．また，市販のマイクロプロセッサでは命令セットアーキテクチャの淘汰が進み，命令セット互換で，かつ高性能なCPUチップの供給を受けることが困難となりつつある. そこで, コンピュータメーカは，メモリや周辺回路の高速化，既存 LSI と FPGA の組合せなどの工夫により互換 CPU を実現し後継機種の開発を行っている.

そのようなアプローチとは別に命令互換性を維持する方法に，異なる命令セットアーキテクチャの CPU を用いたバイナリ変換またはエミュレーション[†1]と呼ばれる技術がある[1][2]．エミュレーションの方式には，プログラム実行前に命令の変換が必要な静的バイナリ変換方式，プログラム実行時に変換を行う動的バイナリ変換方式，古典的なインタプリタ方式がある．新命令セットアーキテクチャへのソフトウェアの移植は，プログラムコードのリコンパイルやソース変換で対応できるが，リコンパイルによる既存の潜在障害の顕在化やコンパイラの障害などが発生する．それらへの対策としてシステム試験が必須となり，その期間と費用の確保が課題となる．そこで，ソフトウェアの移植が要らないエミュレーション技術とそれをソフトウェアにより実現したエミュレータが実用化されてきた[2][3].

静的バイナリ変換は，コードの最適化による高い性能が期待できるが，制約が多くその適用には運用上の課題がある．動的バイナリ変換は実行頻度の高い部分のみ変換しつつ実行する方式で，実用的かつ性能面でも優れている[2]．しかし，最適化処理と例外処理が複雑化し安定した動作を保証するには開発費が増え試験評価期間が延びる問題がある.

インタプリタ方式は，命令を 1 語ずつ解釈と実行をするため実装が単純であり，命令による命令語の書換え（自己修飾）や商用機で厳密性が要求される例外動作も正確に処理できる特長を持つ．命令間に渡る相互作用を限定することができ既存の診断プログラムによ

[†1] シミュレーション対象のマシンに対し，その機能を損なわず性能も匹敵することからシミュレーションと呼ばずにエミュレーションという用語が使われている.

る網羅的な検証が可能である．そのため，開発費は動的バイナリ変換方式に比べ一桁から二桁少なくて済むといわれている．欠点は，動的バイナリ変換方式に比べ数倍から十倍ほど遅いことである．しかし，インタプリタ方式のエミュレータが実用的な速度で動作する製品分野がある．マイクロプロセッサは年率 55%[†2]程度の速度性能の向上しその恩恵を受けて，エミュレーション機の速度性能も同様に伸びてきた．その伸びに対して，企業活動や生産活動など基幹業務で扱うデータ量の増加は年率 10%程度以下であり，それに要求されるコンピューティングパワーはピーク時や用途拡大を勘案してもその伸びは年率 20%以下[†3]である．そのため，これらの分野では，インタプリタ方式で賄える処理が年々増えることになる．

インタプリタ方式の実装には，多くの場合，より高い性能を得るためにアセンブリ言語が使用されてきた．一方，開発工数が少なく，ホストへの依存性が低く長期的にソースコードが活用でき，エミュレータの改良開発と保守を行う技術者の確保が容易な C 言語実装が望まれている．

しかし，現状では，C 言語実装は単純に性能が低いこと以外にも次の問題点があるといわれている．

- エミュレーション性能は，エミュレーション対象の命令セットアーキテクチャ（レガシーISA）に対する，ホストの命令セットとの類似性，ホストのマイクロアーキテクチャとの相性，C 言語コンパイラの最適化性能の影響を大きく受ける．
- C 言語実装は，ホストが決まっているアセンブリ言語による実装に比べて，選択肢が広がる分，開発着手前に行う性能見積りが困難となる．
- 性能が記述スタイルやコンパイラに依存するため，チューニングは試行錯誤的となり性能を改善できる保証をすることが難しく，結局は開発のリスクを負ってしまう．

このような問題に対し，我々は，C 言語を使用しても，開発前の性能見積り及びチューニングによる性能向上は可能であり，しかもレガシーISA とホストの組合せによらず同様な手法で同様な効果が得られると考える．本研究では，それを可能にする実装手法とその効果を明らかにすることを目的とした．

組込み機器の分野では，従来は専用ハードウェアで実現していた領域にもマイクロプロセッサが多く使われるようになってきた．まず，ソフトウェアを利用した機器の機能向上がある．また，処理をソフトウェア化することで柔軟性を持たせるとともに，最新の高性能なプロセッサの演算能力を利用した処理性能の向上も行われている．マイクロプロセッサは，動作周波数の向上，多段パイプライン，スーパスカラ，オンチップキャッシュメモリなどの技術により性能向上をしてきた．しかし，それらの挙動が複雑なため，設計段階でのシステム性能の見積りが困難となり，更に見積り値に達しないときの原因究明と解決にコストが掛かるという問題が出てきている．特に組込み機器では製品コストや発熱など

[†2] 1986～2002 年までは SPECINT 性能で年率 50～55%，それ以降は 20%で推移.

[†3] 平成 7~11 年の全産業の労働生産性の成長率は 0.74%，最も高い情報通信産業は 8.29%でその中でも昭和 60 年～平成 11 年では電気通信の 13.1%が最大である.

の制約が強いため，主記憶周りの速度性能が抑えられメモリウォール問題による性能低下が起きやすく，ハードウェアの特性に合わせたソフトウェア改善が求められることが多い．

ソフトウェアの改善にはプロファイラを用いたチューニング方法があるが，次のような問題がある．

- プロファイル専用に生成された命令列と最適化コンパイル済みの命令列の差異がありレポートされた部分を改善しても期待したほどの改善効果が出にくい．
- 処理が遅い原因の解明に十分な情報を得られない．
- 短い周期のタスク切替えにより発生するキャッシュミス問題の解析に使用できない．

更に最近はマルチコアやマルチスレッドを搭載した CPU 製品が普及し，ソフトウェアによる細粒度の並列処理が行われる傾向がある．そのような並列動作の高速化のため，解析とチューニングのツールとしてシミュレーションのニーズが高まってきている．ハードウェアを SystemC などでモデル記述したシステムシミュレータはそのような目的にも使用できるが，その処理速度は遅く大規模ソフトウェアのシミュレーションには膨大な時間が掛かり実用的でない．そこで，性能解析を目的とした高速なシミュレータが必要となる．

性能解析に用いるシミュレータは，次のように分類できる．

1） トレースドリブン型：プログラムの実行と解析を分離して実行する．

2） 実行ドリブン型：プログラムの実行と解析を並行して行う．これには命令シミュレータコアが必須となる

(a) インタプリタ方式：1 命令ごとにシミュレーション対象の命令をホストの機械命令に通訳しながら実行する．

(b) バイナリ変換方式：バイナリ変換方式は，シミュレーション対象の命令をホストの機械命令に翻訳して実行する．

インタプリタ方式は 1 命令ごとに，命令の取り出し，命令デコードの結果による分岐，オペランドフィールドの切り出し，オペランドの読出し，演算，結果の格納，プログラムカウンタの更新を繰り返す．インタプリタの構造とソースコードが公開されているシミュレータの代表例として SimpleScalar[7][34]が挙げられる．インタプリタ方式の処理は逐次性が高くパイプラインを乱しやすいため，最新の高速マイクロプロセッサを使用してもその性能を引き出すのが難しいアプリケーションである．シミュレーションのために処理時間が何倍遅くなったかを示す指標として“スローダウン”がよく用いられるが，インタプリタによるスローダウン値は 20～200 倍程度[†4]といえる．

一方，バイナリ変換方式は，事前またはシミュレーション実行時に，シミュレーション対象の命令をホストの機械命令に変換する．そのため，命令の取り出しからオペランドフィールドの切り出しまでの実行処理時間を減らすことができる．更に，最適化によりシミュレーション処理を減らすことができ，インタプリタ方式の数倍から十倍の速度性能の向上を期待できる．品質と開発コストの面では，実装が容易かつ移植性が高いことからイ

[†4]古い事例では 20～80[5]，最近の例では 119～246[5]や 20～45[38]が報告されている．

ンタプリタ方式が有利となる．逆に，これらはバイナリ変換方式の短所となる．

組込み分野では異種マルチプロセッサシステムが増えており，これに対応するため，シミュレーション対象のプロセッサに制約がなく柔軟性の高いシミュレータのニーズが高まると予想される．市販のシミュレータや LSI ベンダが提供するシミュレータでは現在，シミュレーション対象のプロセッサの種類やそのシミュレーション機能が限定されている．一方，高機能かつ高速なシミュレータを自製するには開発コストが掛かり，性能解析のためだけに高速シミュレータを何種類も用意するのは難しい．

組込み機器開発ではその製品の性能向上を目指し，市販プロセッサの選択，オフチップメモリの適正化，プログラムからのメモリアクセス方法やソフトウェア構造の改善，コンパイラオプションの精査が求められている．そして，それらを勘に頼らずに着実に効率よくできるツールとして，シミュレータの研究開発が必要となる．

本研究では，次のような要求に適合するシミュレータとそのシミュレータコアの効果的な実現手法を探求している．

- 市販プロセッサを対象とした性能解析を高速に行える．
- プロセッサメーカやサードベンダに依存せず，安価な開発コストで実現したい．
- 異種マルチプロセッサシステムに柔軟に対応でき，拡張性が高い．

本研究は，その実現手段として簡易実装ながら設計品質の確保が容易で拡張性があり高性能なバイナリ変換アクセラレータを提案している．なお，シミュレータコア自身が直接計測するのは命令数であり，コア単体での主な用途には，性能解析の初期段階での粗い性能の把握や，詳細評価を行うプログラム箇所までのセットアップがある．このコアに，キャッシュシミュレーションなどの機能を付加して高速化へ応用することを本研究の目的としている．

以上のように，本研究は命令レベルシミュレーションの高速実行として，インタプリタ方式とバイナリ変換方式の２つを対象としている．インタプリタ方式では，コンピュータエミュレーション向けに実用的な C 言語記述のインタプリタに焦点を当てて，その性能の課題を解決する．バイナリ変換方式では，性能を解析評価するシミュレータの高速化に焦点を当てて，簡易実装による高速性能の実現を探求する．

1.2. 従来の研究

本節では，従来の研究の概観を述べ，また命令レベルシミュレーションの基本となる方式を説明する．1.2.1ではコンピュータエミュレーションについて，背景説明としてその歴史と対象について述べる．1.2.2では，性能評価シミュレーションの基本方式と従来研究を簡単に述べる．1.2.3ではインタプリタの原理，1.2.4では静的バイナリ変換の原理，1.2.5では動的バイナリ変換の原理を説明する．

1.2.1. コンピュータエミュレーション

1.2.1.1. エミュレーションの歴史

コンピュータエミュレーションは，もともとシミュレーションの解釈実行の低速性をハードウェアの助けを借りて高速化しようとして考案された. IBMsystem/360 による 1401 のエミュレーションや汎用エミュレータ MLP-900 など，マイクロプログラミングに技術により発展してきた[25].

ソフトウェアによる別アーキテクチャのエミュレーションは，文献[1][2]に紹介されているように, 静的バイナリ変換による HP3000（1987 年），IBM system/370 から IBM RT PC に変換する Mimic（1987 年），VAX や MIPS から Alpha へ変換する MX/Vest（1993 年）がある．アップルコンピュータ社では 68K から PowerPC にアーキテクチャ変換をするときに OS にエミュレータを搭載して 68K バイナリしかない既存アプリケーションも動作するようにした（1994 年）．同社は今回，PowerPC からインテル製の x86 に移行するため Transitive 社のエミュレーション技術を利用した Rosetta [58]（2007 年）を Mac に搭載している．

比較的最近では，インテル社と HP 社が協力して VLIW（Very Long Instruction Word）の IA-64 アーキテクチャの Itanium を設計した(2000 年)．HP 社では既存の PA-RISC のアプリケーションとの互換性維持によりユーザの移行を促進するため，動的バイナリ変換エミュレータ Aries[3] を OS に搭載した. HP 社はソフトウェアの解析と最適化を行うツール Dynamo[59]などで蓄積した技術をそれに応用している．その一方，HP 社は既存アーキテクチャの PA-RISC 機も Itanium2 が出るまで市場投入しており，余り公表されていないがメモリを多量に使うアプリケーションでは性能が充分ではなかったと推測されている．インテル社では IA-32（x86）互換の動作モードをハードウェアとして Itanium に搭載していたが，同一クロック周波数の x86 機（Xeon）に比べるとかなり性能が低かった．そのため，当初計画されていた IA-32 から IA-64 への移行ができず，IA-32 との併売と x86 機への 64 ビット機能搭載の道をたどった. このように, コンピュータメーカが新アーキテクチャへの移行の一時しのぎとして使われるのがエミュレータの 1 つの形態である．そのため，メーカはエミュレータの開発にある程度のリソースを割くことができるため，エミュレータの品質確保は容易といえる．

エミュレータの別の使い方に，DEC 社の FX!32（1996 年）の事例がある．マイクロソフト社の OS である WindowsNT は当初 x86 の他に DEC 社の Alpha にも移植されたが Alpha 向けのアプリケーションが少ないため，DEC 社は x86 用のバイナリをエミュレーションした．FX!32 は，静的に変換したバイナリをディスク上にファイルとして残すことにより高速化を図っており，命令エミュレーションそのものよりアプリケーションインターフェイス（API）に工夫がある．

これらに対して，他のメーカが決めた命令セットをエミュレーションする互換ビジネスでの成功例は少ない．トランスメタ社の Crusoe は x86 の命令を VLIW アーキテクチャに動的に変換する CMS（Code Morphing Software）[41]を搭載し，当時インテル社が重視していなかったモバイル PC の低消費電力化に成功した．この動作原理は，インテル社が

Pentium-Pro で実現した CISC 命令セットから RISC 型のマイクロアーキテクチャ（μOPS）へのハードウェアによる変換を，ソフトウェアで実現したようなものである．このエミュレーションが高速な理由は x86 アーキテクチャに合うハードウェア構成（例：オペランドアクセスなど）を採り，例外処理によるオーバヘッドをなくしているからである．これも短期的には成功したがインテルが Pentium-M で省電力化に取り組んだ結果，淘汰された．

一方，エミュレーションによるコンピュータビジネスに完全に成功した分野がある．それはオフィスコンピュータやメインフレームの一部などのビジネスコンピュータであり，基幹業務を対象としたものである．これらの製品分野ではコンピュータに求められる性能は基幹業務の扱うデータ量の伸びに依存し，年率高々20%程度である．1980 年以前は，マイクロプロセッサより一桁程度以上高速であったが 1990 年代の前半に追いつかれ 2000 年頃には完全に逆転した．そのため，マイクロプロセッサを使って既存の命令セットを模擬しコンピュータ全体をエミュレーションする新製品の開発が可能となった．その中でも，エミュレーションが非常にうまく働いたのは IBM の System38 の後継機 AS/400[47]である．これはホストの命令セットを一般に開放せず，命令とデータを分離していたため，独自の CISC 命令セットをから PowerPC†5に移行しコストパフォーマンスを改善したバイナリ互換の改良機種を市場投入できた. 富士通では CISC アーキテクチャのオフィスコンピュータ K シリーズを，上位機は Ultra-Sparc をベースに下位機は x86 をベースに GRANPOWER6000 シリーズにアーキテクチャ転換した（1998 年）．ここでは独自 OS をマイクロカーネル化してネイティブ移植し，ライブラリやアプリケーションはエミュレーションした. エミュレーションはインタプリタと OCT[19]という バイナリ変換で実現しているが，バイナリ変換による速度性能の向上はインタプリタの 2〜4 倍に留まっていた．NEC では，オフィスコンピュータ S シリーズを Express5800/600 シリーズに替えた．そのシリーズは，互換機能も提供しておりインタプリタによるバイナリ互換機能があるといわれていた．また同社のメインフレームもエミュレーションにより実現されていたようである．IBM 社のメインフレームユーザ向けに System370〜390 互換機能を搭載した FLEX-ES[20]は，I/O を含めた互換性を実現し IBM と提携して販売を行っていた．

エミュレーション対象は CISC が主であり RISC はその対象とならないという意見も一部にある．しかし，PowerPC をエミュレーションする Rosetta[58]や Sparc の命令を IA-64 に変換する QuickTransit[24]の例もあり，CISC 同様に RISC もエミュレーション対象となると考えられる．

エミュレーションの更にまた別の使い方の研究として，VLIW（Very Long Instruction Word）アーキテクチャをホストとしてバイナリ変換を適用し命令実行の高速化をねらうものがある．その代表として IBM の DAISY [28]が著名であるがプロジェクトが停止し，BOA[40]に引き継がれている．

エミュレーションを前提としたマシンアーキテクチャには JavaVM[50]がある. これはバ

†5 市販の PowerPC と異なり十進演算，エラー検出強化などされた専用アーキテクチャである．

イトコードと呼ばれるインタプリタ実行が容易な仮想命令を定義したもので，java インタプリタと JIT[51]などのバイナリ変換でホストの命令に変換して実行する†6.

仮想マシンは 1960 年代の IBM/360 から始まり 1980 年代に多用されていた．それとは，趣の異なるタイプの仮想マシンが，PC を中心に 2000 年代には普及し始めた．それは OS のみならず I/O を模擬するもので，ドライバレベルの実装に価値がある．IBM の System370[22]では仮想マシンを考慮して特権命令のみエミュレーションすればよいように命令体系が整理されていたが，x86 などマイクロプロセッサではそのような考慮がない†7. そのため，VMWARE[53]はバイナリ変換によるエミュレーションを併用して仮想化を実現していた．オープンソースソフトウェアのエミュレータとしては QEMU[54]が有名である．その変換対象に ARM や PowerPC も挙げられているが，x86 同士の OS 間のエミュレーションが主流となっており性能や構造に関する論文などの情報は乏しい．この他に，ホビーとしてゲーム機のエミュレーションも多く使われているようであり，また，ソフトウェアの開発環境という目的でのエミュレーションも多いが，どれだけ普及しているかを示すデータはない．

この他に，宇宙や防衛機器のように長期間†8に渡って同一製品の供給や保守を必要とする分野もあり，製品を構成する部品枯渇の問題が出てきているものもある．部品枯渇への対策として FPGA による等価回路の実現や命令エミュレーション[23]も検討されている．

1.2.1.2. エミュレーションの対象と方式

エミュレーションは，まず，その対象により表 1に示すように分類できる．また，図 1にコンピュータのエミュレーション技術をマップに示す．この図で縦軸に目安としての性能を，左半分は異種命令セットを，右半分は同一命令セットを示す．灰色でハッチングした部分はアプリケーションレベルなどシミュレーションの対象を示しているが，背景の白い部分に置かれているものはシステム全体のエミュレーションを示す．なお，参考として性能評価や解析を目的としたシミュレータも斜体文字を使ってその一部を示した．

このようにコンピュータのエミュレーションには，命令のエミュレーションが必要である．最近は，エミュレーションに動的バイナリ変換が使われることが多く，インタプリタの高速化に関する論文は見当たらない．しかし，インタプリタは動的バイナリ変換に併用されておりこの高速化は必要である．また，バイナリ変換はアプリケーションに大きく依存して性能が高くなる場合と低くなる場合があるが，インタプリタは安定した特性を持つ．実装面ではインタプリタは開発コストが安く，安定した品質を得るのが容易であるという特長がある．

---

†6 バイトコードをハードウェアで実行するものやアクセラレータもあるが，一部のコードはソフトウェアとして実装が必要である．

†7 最近は x86 にも仮想マシン専用のハードウェア(VT)が搭載されるようになり緩和されている．

†8 PC などの機器の 3〜4 年に対して，産業用機器で 7〜10 年，防衛機器では 15〜30 年ともいわれている．

表1 エミュレーションの形態と方式

分類	対象	説明	例
エミュレーション範囲	CPU	異なる命令セットの CPU の命令をエミュレーションする	（省略）
	アプリケーションプログラム	プログラムの API（アプリケーションインタフェイス）を模擬する	QEMU，FX!32
	OS	他 OS のコマンドやユーザインターフェイスを模擬する	Cygwin，SoftPC
	I/O	OS が対応している旧式の I/O をエミュレーションする	VMWARE，Xen
	システム	別のコンピュータを模擬する，1 台のコンピュータで複数のシステムを実現する	JavaVM，DAISY, BOA, VMWARE，Xen
アドレス変換	論理アドレス	アドレス変換はハードウェアと OS に任せてエミュレーションしない．（アプリケーションレベル）	QEMU（ユーザモード）
	物理アドレス	論理アドレスから物理アドレスへの変換動作も模擬する．（フルシステム）	BOA
命令セット	異種命令セット	異なる命令セットアーキテクチャをエミュレーションする．	（省略）
	同種命令セット	OS などの模擬，プログラムデバッグなどの手段として同一命令セットをエミュレーションする．	QEMU，VMWARE

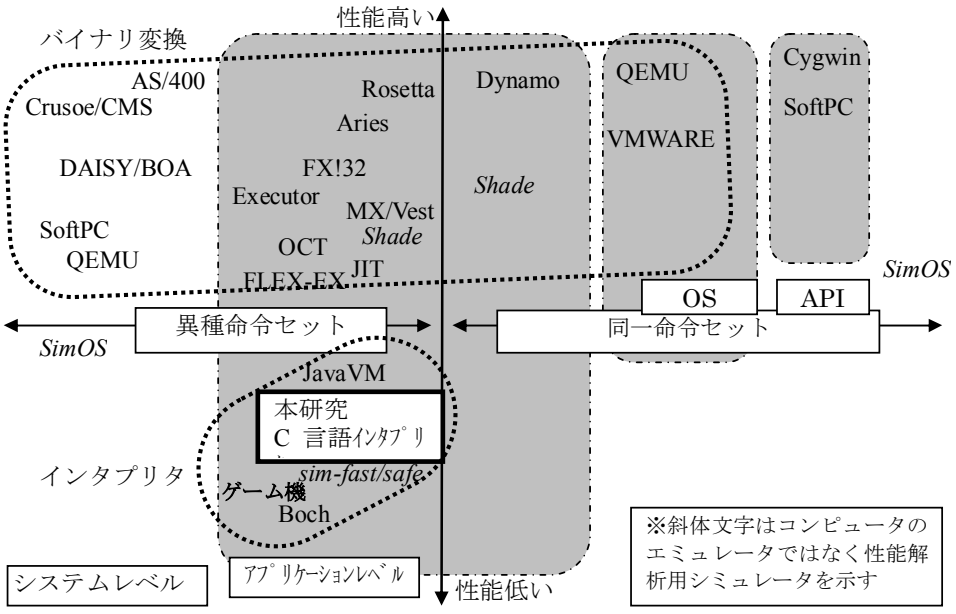


図1　エミュレーション技術のマップ

1.2.2. 性能評価シミュレーション

コンピュータシステムの性能評価には，解析モデルによる手法，実機計測による手法，シミュレーションによる手法がある．

解析モデルによる手法では，評価対象システムの構成要素の挙動として各要素のアクセス頻度とそれぞれの処理時間，データのやり取りの関係が分かっている場合は，比較的容易に計算できる．また，バスやネットワークなど待ちが発生するものについては，待ち時

間を簡易的に算出する手段として確率による待ち行列シミュレーション結果を使うこともある．

実機計測による方法は，測定条件となるベンチマークプログラムを実行して，計測器により事象と時間を記録し集計・解析する手法である．メモリの記録容量が小さかった 1980 年代前半位までは結果を磁気テープに，その後はディスク，最近は RAM に記録している．この方法では，実装の高集積化とともに採取できる情報の分解能が低下するため，大規模なシステムでは使用できなくなってきた．計測器を使わずに行う方法としては，IBM の System370 の CPU に装備したプログラム事象記録機構（PER）[22]を用いて割り込み処理で自己計測をする方法，1990 年代後半からは PPC750 や Pentium-Pro などマイクロプロセッサに搭載した性能モニタカウンタなどによる計測方法も使われてきた．

これらに対して，評価対象システム全体をモデル化してシミュレーションする方法や，各要素をモデル化して組み合わせた全体のモデルを構築してシミュレーションする方法がある．コンピュータの能力向上とともにこのようなシミュレーションが主流となってきた．シミュレーションによる手法の長所は，事象の計測分解能を細かくできまた集計が容易である，また事象の発生原因が解明できることである．特に性能評価目的のシミュレータでは，性能の低下原因の究明が比較的に容易である．シミュレーションの一般的な課題としてまず，シミュレーション対象が複雑になることにともないモデル化のレベルと精度のトレードオフの見極めが難しくなることが挙げられる．精度を上げるためモデル化の粒度を細かくすると，シミュレーションに時間が掛かりシミュレータの実行時間性能の向上が課題となる．LSI 設計時の論理の正当性を検証する論理シミュレータには，ソフトウェアで実装するシミュレータの他に，シミュレーション専用のコンピュータや FPGA などを使用したハードウェアエミュレータも使用されている．一方，性能評価を目的とするシミュレーションには，一般にソフトウェアによるシミュレータが用いられる．

シミュレーションによる性能評価は，確率モデルシミュレーション，トレースドリブンシミュレーション，実行ドリブンシミュレーションに大別できる．

1.2.2.1. 確率モデルシミュレーション

確率モデルによるシミュレーションは，古典的かつ簡易的な手法である．システムを構成する多数の要素が互いに独立に動作し，バス，ディスク，LAN などのようなリソース競合を発生させるようなシステムにおいて，動作対象のプログラムなどが特定できない場合には有効な手法である．この手法では，指定した確率分布によりトランザクションのタイプ，データ長，トランザクション間隔などのパラメータを変えシステムの性能の評価を行う．モデルが粗いため精度は良くないが，処理速度の遅い LAN や WAN などが処理速度の速い CPU に対して与える影響を見るシミュレーションなどに効果がある．なお，キャッシュメモリや分岐予測機構などは，アクセス履歴によりミス率が決まるため確率モデルシミュレーションにはそぐわない．

1.2.2.2. トレースドリブン型シミュレーション

トレースドリブン型シミュレーションは，実機による計測または別のシミュレーション結果から採取したトレース情報を入力とし，評価対象をモデル化したシミュレータを用い

て解析する手法である．この方式はトレース採取とシミュレーションを分離できるため，シミュレータの構築とデバッグが容易な点が長所となる．一方，この方式の本質的な課題は，トレースのオン状態とオフ状態で評価対象の挙動が大きく変わってしまうことにある．

トレースドリブン型シミュレーションでは，トレース対象情報の採取，トレース情報の削減と圧縮，トレース処理（シミュレーション）のフェーズに別けられる[4]．トレースの採取方法は，バイナリロードモジュールを実行するものにはダイレクト実行とエミュレーションがある．ダイレクト実行にはマイクロコードを改造した System360（1960 年）や VAX8200 の例がある[4]. また System370 のプログラム事象記録機構や EWS にハードウェア機能と割り込みルーチンを使った方法も採られた．エミュレーションは，インタプリタを実装した Spa[4]や Mable[4]が著名である．また，インタプリタは実行時間が掛かるためプリデコードを行った命令をインタプリタが実行する SPIM[4]，更にそれを最適化したスレデッドコードを実装した Talism[4]と gsim[4]もある．バイナリロードモジュールには適用できない手法として，アセンブラレベルなど付加コードを挿入する方法には，TRAPEDS[4], MPtrace[4], Pixie[4]などがある．これらの実装は容易であるが，ソースコードが必要，コード量の増大によりワークロードが異なること，対象が限られることなどの課題があり，バイナリロードモジュールに適用できる方式が一般的である．

トレースドリブン型のシミュレータは CPU アーキテクチャを設計・評価する目的で使用され，特にキャッシュメモリ，TLB，パイプライン設計の効果を評価するのに使用された．

1.2.2.3. 実行ドリブン型シミュレーション

実行ドリブン型は，シミュレーションにより抽出した情報をもとにその場で解析用のシミュレータを動かす．その処理は更に，コンピュータのプログラムをシミュレーションするフロントエンド（例：命令シミュレーション）と，詳細解析を行うバックエンド（例：キャッシュシミュレーション）に別けられる．これらの両者をまとめて実行するか，別プロセスまたは別スレッドで行うかは実装により異なる．

実行ドリブン型では，バックエンドのシミュレーションを通じて得た時間情報をフロントエンドにフィードバックすると，より正確な挙動をシミュレーションできる．CPU の高速化に伴い，実行ドリブン型で課題であった事象の抽出 （例：キャッシュのヒットミスの判定) のオーバヘッドが実用上問題にならなくなり, その後, この方式が主流になった [4].

実行ドリブン型の代表的なものについては，5.1でその特徴を述べる．

1.2.2.4. シミュレーションの粒度と複合方式

コンピュータのシミュレーション対象は，命令，メモリアクセス，キャッシュメモリという順に細かくなる．プロセッサ内部では，パイプライン，アウトオブオーダのスケジューリング，それらの制御回路の順に詳細になる．プロセッサ外部では，バストランザクション，バス信号，バッファリング回路，メモリ制御回路（リフレッシュやバンクビジー制御）と順に細かくなる．シミュレーションのモデル化の粒度は，命令レベル，簡易パイプラインレベル／バストランザクション，アウトオブオーダパイプライン／バス信号のレベルなどに別けられ，概して処理速度は一桁ずつ遅くなる．逆に粒度を粗くする方向では，高級言語レベル（C 言語)，関数レベル，タスクレベル，プロセス（プログラム）レベルと広げ

ていく場合もある．シミュレーションの長所の 1 つはその客観性にある．そのため粗い方では，高級言語記述のソースコードを命令セットの異なる CPU でコンパイルしたバイナリで解析するか関数レベルまでであり，それより上位の粒度まで粗くしたシミュレーションは余り使われていない†9.

粒度を細かくすれば，精度は良くなるがその分大幅にシミュレーション時間が掛かるようになる．そのため，粒度の粗いシミュレーションで全体の傾向を掴み，性能上支配的な部分のみ詳細に評価するというアプローチは一般的である．そこで，全体の傾向を見るのと評価対象箇所に至るセットアップに高速なシミュレータを使い，評価対象箇所は詳細なシミュレータを使い，その中間地帯ではキャッシュシミュレーションのように履歴を残すシミュレータを使うという複合型のシミュレーション方式がある．SimOS [14]，SimpleScalar[34]の Version3，本研究対象の ESPRIT/sim はそのような使い方に対応している．

図 2に性能評価シミュレータの位置づけとして，横軸に粒度，縦軸に速度としたマップを示す．図 3には，プロセッサ構成，バイナリ変換の有無，フルシステムシミュレーション機能の有無による分類を示す.

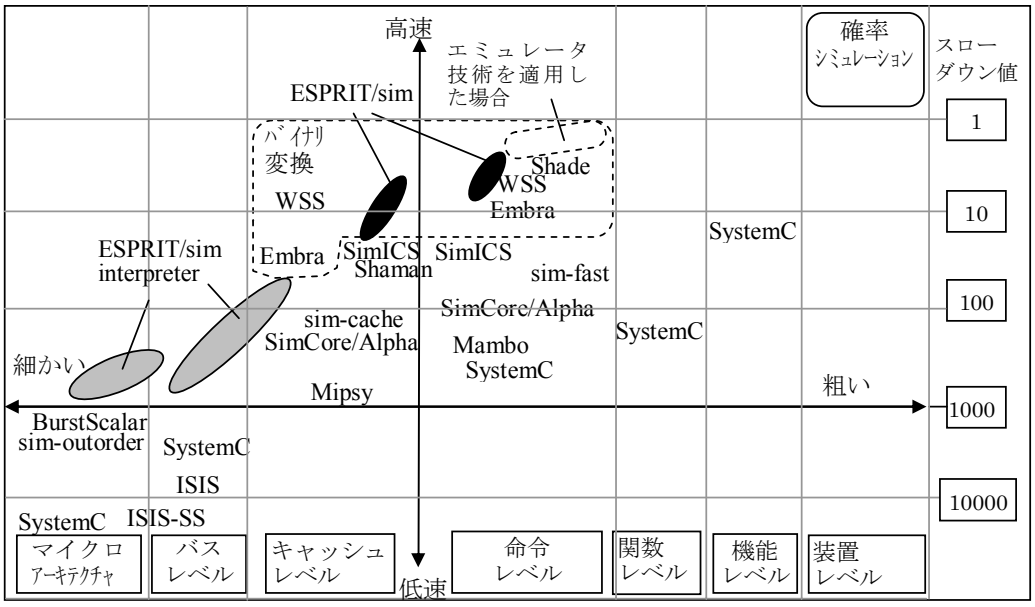


図2 性能評価シミュレータの位置づけ （ユニプロセッサ）

†9 SystemC では実行速度性能が低いことから抽象度を上げてモデル化してタイミング情報を付加するという研究もある[43].

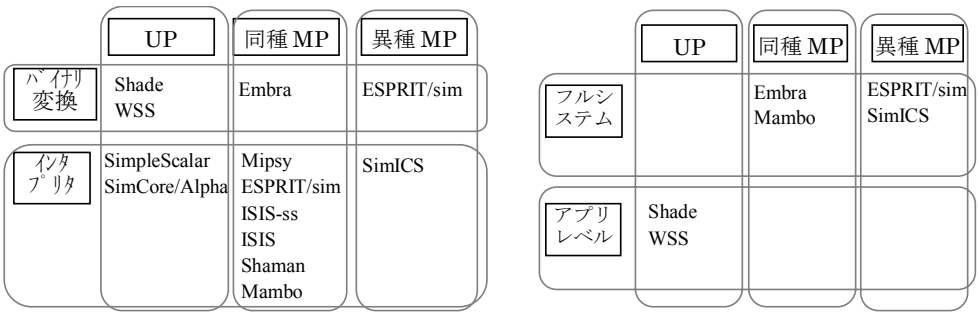


図3 性能評価シミュレータの分類

1.2.3. インタプリタ方式の原理

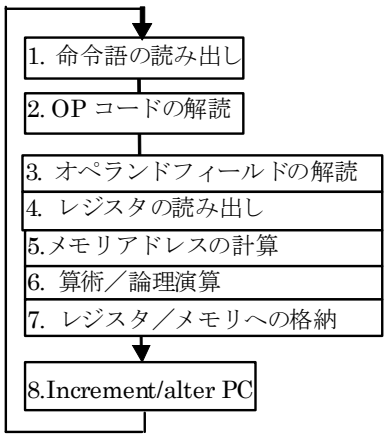


図4 インタプリタ方式の処理フロー

図 4にインタプリタ方式の基本的な処理フローを示す．インタプリタ方式では，(1) 命令語の読出し，(2) 命令語に含まれる命令フィールドの解読 (デコード)，(3) オペランドフィールドの解読，(4) レジスタオペランドの読出し，(5) メモリオペランドのアドレス計算と読出し，(6) 演算，(7) レジスタまたはメモリへの結果格納，(8) プログラムカウンタの更新，などを 1 命令ずつ逐次的に繰り返す．一般に，(3)～(7)の処理は命令の種類により異なり行われない処理もある．(2) の解読処理は分岐命令を使って実装される．この繰返し処理をコアループと呼ぶ．シミュレーション対象の命令セットアーキテクチャをレガシーISA，エミュレーションを行うコンピュータをホストと呼ぶ．レガシーISA のプログラムカウンタ，汎用レジスタ，浮動小数レジスタ，制御レジスタ，状態語などは，ホストのレジスタまたはメモリに割り付ける．

1.2.4. 静的バイナリ変換の動作原理

静的バイナリ変換は，プログラムのロード前またはロード時に，命令コード領域を解析してホストの命令列に変換を行う手法で，Moxi（1986 年）や MX/Vest（1993 年）が静的バイナリ変換に分類されている[15].

一般的に，命令実行の流れが途切れる分岐命令までを “分岐ブロック”と呼ぶ．分岐ブロックは，そのブロックにジャンプしてから最初の出口までを基本ブロック，その基本ブロックが集まり全体で複数の出口を持つ塊を拡張ブロックと区別する[15] (条件付分岐が途切れて無条件分岐となるところまでが拡張ブロックとなる)．図 5に静的バイナリ変換の処

理フローの例を示す．まず，命令コードを解析して分岐ブロックを抽出する操作が行われ，分岐先も順次見つけていく．次に各分岐ブロックに対して個々の命令を順にホストの命令列に変換していく．変換されたコードの効率は，変換前と後の命令セットアーキテクチャ同士の親和性やホストで使用できるレジスタの数により異なる．一般にレジスタをメモリに割り付けた場合でも約 4 命令に置換できる†10．この変換では最適化が行われることが多く，定数やオペランドの組合せを評価して単純な命令やより高速な命令に置換したり，分岐ブロック同士を並べ替えたり連結して分岐命令を減らすことも行われる．

静的バイナリ変換では，命令と定数データを正確に区別できないことが多くデータもプログラムとみなして余分に変換する傾向がある．命令のコード量は一般に 3 倍になるといわれている．また，ダイナミックリンクライブラリのコード相当分は事前に生成できないため，確実なもののみ生成するか分岐ブロックの検出に留めて他の変換手法と組み合わせた実装を採るものもある．現在でも静的バイナリ変換をよく使用しているものとして，JavaVM[50]の JIT[51]が挙げられる．

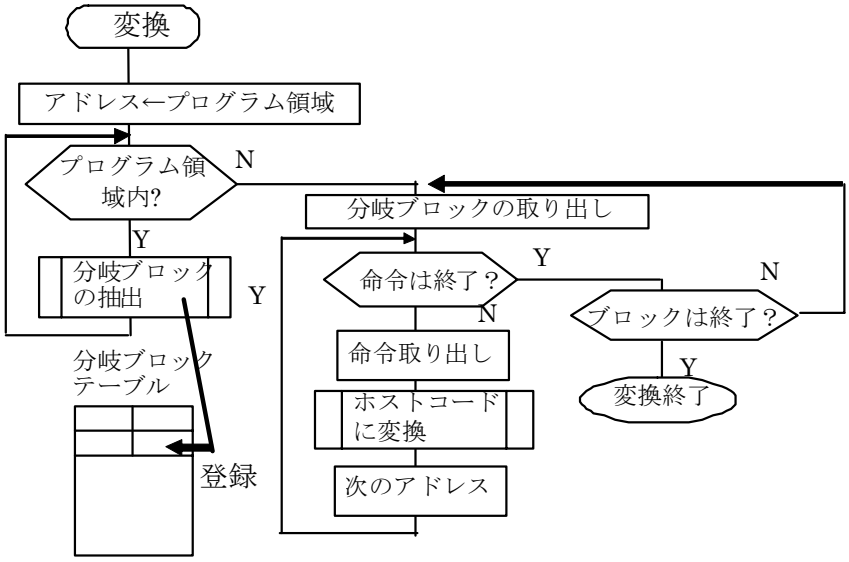


図5 静的バイナリ変換の処理フロー例

1.2.5. 動的バイナリ変換の動作原理

動的バイナリ変換は，命令シミュレーションを実行しながら変換する手法である．System370 の命令を PC/AT に変換する Mimic （1987 年）が著名である．図 6に一般的な動的バイナリ変換の原理的なフロー図を示す．分岐ブロックに対する考え方は静的バイナリ変換と同じであるが，動的バイナリ変換では，分岐発生をきっかけとして分岐ブロックを必ず検出できる．静的変換との大きな違いは，バイナリコードへの変換結果をコード変換キャッシュという限られたバッファに格納し，しかも実行されないコードは変換しないことである．図に示した閾値による制御やインタプリタは原理的には必須ではないが，閾

†10 RISC ホストでも，オペランドのロード 2 回，演算 1 回，ストア 1 回を基準とし，それより多い命令，短い命令もあり 3～5 命令になることが多い．アドレス変換をシミュレーショする場合は更に掛かる．

値を制御することにより実行頻度の高いコードのみ変換対象として頻度が低いコードはインタプリタで動作させることが可能となる．したがって，利用可能なメモリ容量が小さいときには静的バイナリ変換より有利であるといわれている．また，条件付分岐命令の分岐発生の頻度が高い場合には，分岐先（taken）と非分岐先（inline）の命令列生成を逆にする高速化処理[3]も行いやすい．

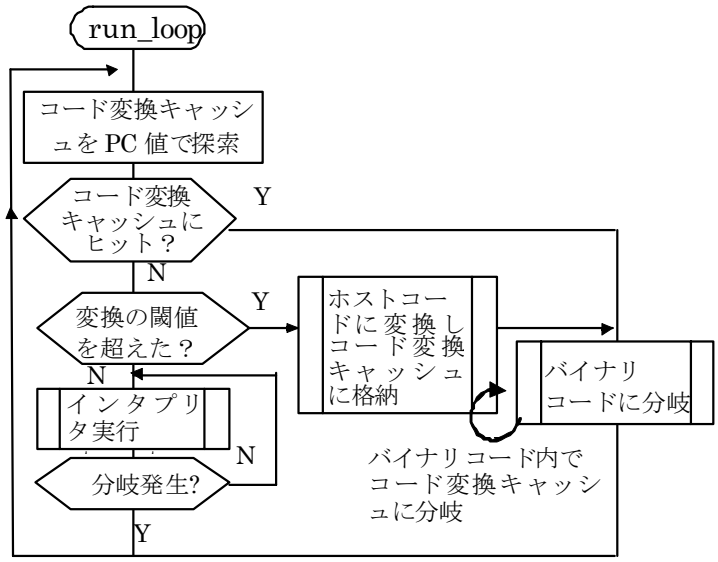


図6 動的バイナリ変換の処理フロー例

性能解析を目的としたシミュレータでは，Sparc と MIPS にバイナリ変換を適用した Shade[15]，マルチプロセッサ動作に適用した Embra[29]が著名である．エミュレーション用途では，PA-RISC から IA-64 への変換を行う Aries[3]などあり最近のバイナリ変換はこの方式を使用している．JavaVM の JIT の中でも SUN の HotSpot[52]はこの範疇に入る．

1.3. 本研究の特徴

コンピュータシステムを別の命令セットアーキテクチャのホストを用いて高速にシミュレーションするエミュレータでは，インタプリタは古くからある原理方式であり，近年ではバイナリ変換方式が主流となっている．どちらの方式も実用性があり実際に使用されているが，それらの実装方式には技巧的な要素が多いため工学的な立場から扱いにくく，実装方法を詳細に述べて性能との関連を評価した研究は少ない．インタプリタは原理的に性能が低いが，マイクロプロセッサの高性能化の恩恵により，その性能は充分実用に耐えうるレベルになってきた．その結果，一般に公表はされていないが使用されている分野が増えてきている．しかしながら，高性能なマイクロプロセッサはパイプライン，スーパスカラ，分岐予測などを多用しているため動作が複雑化し，インタプリタとして実現できる性能値の見積りは容易ではなくなってきた．更に C 言語記述のインタプリタはコンパイラを介在するため，インタプリタの性能の予測とチューニングは一層困難となってきている．

本研究はこの課題に取り組んだ点に独自性があり，C 言語記述のインタプリタ性能の予測と性能チューニングを容易化する高性能なインタプリタを実現する手法を提案し，評価し

たものである．C 言語で記述したインタプリタの実装は，switch～case 文による記述をして命令のデコードと実行処理を混在させるものが一般的である．本研究では，命令種ごとに関数を用意して命令デコード後にそれらの関数に間接分岐する手法“function 方式”を提案している. switch～case 文による記述方式では膨大な case 文の集合の最適化を C コンパイラに任せるため，その最適化の仕組みや効果が予測できず，わずかなソースコード修正が性能を大きく変えてしまうという課題がある．function 方式は，そのようなコンパイラへの依存性がない. また, 個々の命令処理を関数化したことにより発生しうる新たなオーバヘッドも，隠ぺいできる提案手法である．本研究の特長は，エミュレーション対象の命令セットやエミュレーションを実行するホストのアーキテクチャによらず速度性能の向上が可能なインタプリタの実装方法を提案していることであり，その効果を評価により実証している. なお, function 方式は, x86 をホストとすると switch 方式に対する優位性が RISC ほど顕著でなくなるが，自由に使用できるレジスタ本数が増える 64 ビットのアーキテクチャモードの使用が一般的になれば RISC 同様に有効性が見えてくると考える．

バイナリ変換は本来ハードウェアと機能を分担することによりその真価を発揮するものであるが，ソフトウェアのみで実現できるバイナリ変換技術を利用したものとして，性能解析を目的としたシミュレータがある．このシミュレーションの速度向上への要望は強いため，高速なシミュレータではバイナリ変換が使用されている．

バイナリ変換そのものは現在では新たな技術ではないが，一般に，任意のプロセッサの組合せができないなど拡張性の課題がある. 一方, バイナリ変換機能を搭載したシミュレータを自製するには開発コストが掛かるという課題がある．本研究では異種プロセッサの混載，モデルの拡張性，高い設計品質，高速性，低開発コストなど相反する条件を満足するバイナリ変換アクセラレータの方式を提案している．最近の高性能マイクロプロセッサでは 2 次キャッシュメモリの容量が増大しパイプラインが深化しており，本研究は，バイナリ変換方式の実装手法の効果が従来と変化しつつあることに着目した．そして，バイナリ変換の構造を単純化して，高性能化と低開発コストの両立の目途を付けたことが本研究のバイナリ変換方式の特長である．

バイナリ変換の方式に，エミュレータ並の最適化を行えば更に速度性能の向上が可能であり，そのようなアプローチを採るバイナリ変換には速度性能は及ばない．しかし，市販 CPU や IP を組み合わせた組込みシステムの性能評価を安価な開発費で行う目的には，本研究のような簡易的な手法が有効である．

1.4. 本論文の構成

本研究では，命令レベルシミュレーションの高速実行に関して，その主要な原理であるインタプリタ方式とバイナリ変換方式の両方を研究している．それらの方式の研究にあたり，インタプリタ方式では命令エミュレータを対象としている．また，バイナリ変換方式では，コンピュータシステムの性能評価向けのシミュレータに搭載する動的バイナリ変換を対象としている．

それぞれについて図 7に示すように，背景と従来の研究，本研究での提案，同提案の評価

の順に述べる．

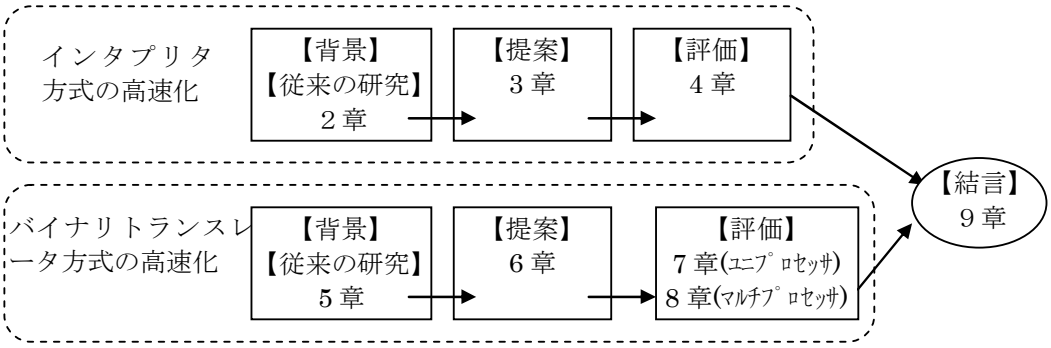


図7 本論文の構成

第2章から第4章まではインタプリタ方式のシミュレーションの高速化について述べる．まず第 2 章では本研究の技術面から見た背景，インタプリタ方式の性能上の課題を定性的に述べ，次に事例研究を用いて実装に伴う性能上の課題を示す．第 3 章では C 言語実装に適したインタプリタの実装方式として”改良 function 方式”を提案する．第 4 章では提案した改良 function 方式の性能を評価する．

第 2 章では，2.1で命令エミュレーション技術として，本研究のエミュレーションの扱う範囲，エミュレータの有効な分野，C 言語記述による実装の要求，インタプリタ方式の実装と課題を述べる．そして，本研究の着眼点と新規性についても述べる．従来研究の事例としてインタプリタ方式の速度性能を分析した文献が見当たらないため，2.3でインタプリタの実装事例に基づいて性能解析を行っている．2.4では既存のインタプリタの記述で使用されている switch 方式の問題点を示す．なお，その評価に先立ち，2.2で評価方法について述べている．

第 3 章では，3.1でまず，改良 function 方式の提案に至った“function 方式の原型”とその改良すべきポイントを述べる．次に3.2で原型の速度性能の解析から得た改良指針を述べ，3.3で改良 function 方式の説明との提案を行う．なお，3.2では function 方式の原型の速度性能値については触れずに改良 function 方式と併せて4.1～4.4にまとめて示す．

第 4 章では提案方式の有効性を示すため，改良 function 方式の速度性能を評価する．まず，4.1～4.4では2.3.4～2.3.6と同様の評価を行う．次に，提案した方式によるエミュレーション性能を様々な観点から論じる．具体的には，4.5では x86 をホストにしたときの課題を述べる．4.6ではレガシーISA を 3 種類追加して 5 種類に増やしてその有効性があることを示す．4.7では前節までは影響が小さいとして取り扱わなかったが一般に性能への影響があると考えられる項目を評価する．4.8で第 2 章から第 4 章までのまとめを行う．

第5章から第8章まではバイナリ変換方式のシミュレーションの高速化に関して述べる．まず，第 5 章では性能評価を目的とした代表的なシミュレータと本研究の適用対象とした ESPRIT/sim についてその特徴を述べる．次に，第 6 章で性能評価シミュレータの高速化方式の提案を行う．ここでは ESPRIT/sim に搭載するバイナリ変換アクセラレータとして提案を具現化する．第 7 章ではユニプロセッサシステムを対象として SPEC CINT95 にて

性能評価を行いその有効性を示す．第 8 章では，同種マルチプロセッサシステムと異種マルチプロセッサシステムの速度性能の評価を行い，有効性を示す．

第 5 章では，まず，従来研究として代表的なシミュレータの特徴を5.1で述べる．次に，性能評価シミュレータの高速化手段を整理し，方式としてバイナリ変換の有効性と課題を5.2で示す．5.3では，本研究のベースである性能評価シミュレータ ESPRIT/sim の特徴を述べる．5.4では異種マルチプロセッサ向けのシミュレータの課題を示して，本研究の目標とする．5.5と5.6では，それらの従来研究と対比する形で ESPRIT/sim の位置づけ，性能，有効性と新規性の要素を個別に説明する．

第 6 章ではまず, 6.1で本研究の着眼点を整理して新規性との関係を述べる. 次に, それを踏まえて ESPRIT/sim の高速化のため実装する動的バイナリトランスレータの設計方針を6.2で述べる．6.3～6.8ではその方針を具体化した実装方法を提案する．

第 7 章では, 7.1で評価方法を定義し, 7.2では検証方法を述べ, 7.3で SPEC CINT95 で測定した性能の総合的な評価結果を示す．その後，7.4と7.5では提案方式の各手法それぞれの部分効果について述べる．また，本研究では簡易実装による高速性能をねらっているため，その効率について7.6で論じる．提案している動的バイナリ変換アクセラレータはシミュレータのコアであり，その波及効果について7.7で述べる．7.8ではユニプロセッサシミュレーションの速度性能についてまとめる．

第 8 章では，マルチプロセッサシミュレーションの速度性能の向上効果を評価する．8.1では評価方法を述べ，8.2では検証方法を述べる．8.3で同種マルチプロセッサシステムのシミュレーション速度性能を，8.4では異種マルチプロセッサシステムのシミュレーション速度性能に関して評価する．8.5ではマルチプロセッサシミュレーションの速度性能についてまとめる．

最後に，第 9 章ではインタプリタとバイナリ変換のそれぞれの研究についてまとめ，今後の展開について述べる．

第2章 C言語実装を用いたインタプリタ方式の命令エミュレータ性能

本章では，専用ハードウェアを用いずにソフトウェアのみで実装されたインタプリタ方式のエミュレータの動作原理を概説する．そして，事例研究を通して C 言語記述を用いたインタプリタの速度性能上の課題を探求する．

本章の構成を述べる．2.1ではインタプリタを中心としたエミュレーション技術を概説する．本研究ではレガシー命令セットとホストの組合せによらずに，性能見積りが容易で性能向上が可能なインタプリタを C 言語で実装できることを示すため，事例研究と提案方式のそれぞれに共通な評価を実施した．2.2では，この評価方法について述べる．2.3ではインタプリタの事例研究として命令セットアーキテクチャ SimpleScalar/PISA [7]のシミュレータ sim-safe[7][34]を取上げ，性能評価を行う．まず2.3.2で評価対象に PISA を選んだ理由とインタプリタの実装を解説する．PISA は仮想アーキテクチャのため，その実装方法を実際の MIPS アーキテクチャ [8]に似た仕様に改造した命令セット MIPSlike を定義しそのインタプリタ sim-MIPSlike を製作したので それについて2.3.3で述べる．2.3.4では sim-safe の速度性能の解析を，2.3.5では sim-MIPSlike の速度性能解析を行っている．2.3.6でコアループ性能を述べ，2.4で switch 方式の問題点をまとめる．

2.1. 命令エミュレーション技術

本節では命令エミュレーション技術全般に関して述べ，本研究が C 言語記述のインタプリタ方式のエミュレータをターゲットとした背景を説明する．

まず，2.1.1で命令エミュレーションが扱う動作範囲，2.1.2でエミュレータが産業上どのようなときに使われるかを述べる．2.1.3で高速なバイナリ変換方式とインタプリタ方式の関係を述べ，2.1.4で C 言語実装のインタプリタに対する期待と課題を示す．2.1.6では，インタプリタ実装と性能に関する影響を定性的に述べる．

2.1.1. 命令エミュレーションの範囲

エミュレーションには，フルシステムエミュレーションとアプリケーションレベルエミュレーションがある．フルシステムエミュレーション[14] [20]は，オペレーティングシステム（OS）も含むシステム全体をエミュレーション対象としてアドレス変換動作も模擬する．アプリケーションレベルエミュレーション[2][19]は，OS のカーネル部とライブラリの一部をホストのネイティブ命令で動作させ OS の残り部分とアプリケーションを模擬実行する．後者ではアドレス変換とメモリ保護は TLB やマップレジスタにより行われるため，エミュレーションの速度も前者に比べて速い．アプリケーションレベルエミュレーションに静的または動的バイナリ変換方式を用いた場合には，ＯＳはセグメント例外やページ例外を検出すると，変換後のプログラムカウンタ値からレガシーのプログラムカウンタ値の推定を行う．それに対し，インタプリタ方式ではエミュレータのプログラムカウンタ値をそのまま用いて簡単に例外処理を実現できる．

アプリケーションレベルエミュレーションはOSの移植を必要とするが，エミュレータの単体性能はフルシステムエミュレーションより高速である，入出力装置を含むシステムの機能拡張への対応が容易である，更にカーネルの実行時間の比率が高くなるとシステム全体の速度性能が向上するなどの長所により，実用範囲が広い．カーネルの時間比率とインタプリタの速度性能の関係を例で示す．エミュレーション対象のシステム（レガシーシステム）の処理時間のうち，OSカーネルとネイティブ化したライブラリが占める元の時間の占有率をαとする．レガシーシステムに対してホストのネイティブ速度性能の向上比をβとする．エミュレーション対象部をエミュレーションしたときの実行時間をホストがその部分をネイティブとして動作したときの実行時間で割った値，すなわち何倍の時間が掛かったかを示すスローダウン[†11]をγとする．ホストによるOSカーネル部の実行時間はα/β，エミュレーション対象部であるアプリケーションの実行時間は$(1-\alpha)\gamma/\beta$となる．仮に$\beta=50$, $\gamma=30$とすると，$\alpha=0.2$で全体の処理時間は0.48となり，速度性能の向上率は0.48の逆数の2.1倍となる． αが0.5，0.8と増えるに従い速度性能は3.2，7.4倍と増加する．図8に処理時間の変化の様子を示す．本研究では，このように性能面での実用性が高いアプリケーションレベルエミュレーションを例に，インタプリタ性能を論じ[†12]評価を行う．

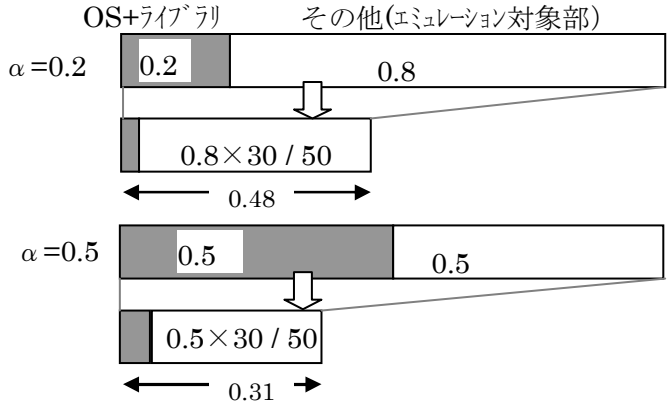


図8アプリケーションレベルのエミュレーション処理時間

2.1.2. エミュレータの有効な分野

マイクロプロセッサの速度性能は年率1.5倍から1.55倍の割合で高速化されてきた[21]．一方，メインフレームやオフィスコンピュータメーカでは，基幹系の業務データを処理するユーザが多いためそのデータ量の増加すなわち企業の成長率に見合った性能トレンドで

[†11] スローダウンは，シミュレーション後の処理時間をシミュレーション前の処理時間で割った値であり，シミュレーションをしたことにより何倍遅くなったかを示す指標である．

[†12] レガシーの命令語を1語ずつ解読してはホスト命令列を生成しメモリに格納してから実行する方式も存在するが，一般のRISCプロセッサに適用すると性能劣化が著しく実用的でないため，ここでは検討の対象とはしない．また，命令のデコード時間短縮のため，プログラムロード時にプリデコードや変換を行う方法(例：文献[4]で紹介されているSPIM)，命令は書き変えられないものとみなして命令処理完了前に次の命令読出しを行う方法，デコード結果を速見表に格納しプログラムカウンタを索引として引く方法[5]などの高速化手法も提案されている．しかし，これらの手法を用いると，命令語を書き換えて実行する”自己修飾”や命令語を定数として参照することを許すなどのインタプリタ方式の利点が失われてしまう．そこで，本論文ではレガシーを正確にエミュレートするというインタプリタ方式の利点を失わない基本的な方法のみを検討の対象とする．

コンピュータ製品を供給してきた．そのため，CPU 単体の速度性能ではマイクロプロセッサに追い越されその格差が拡大している．その関係をここで少し説明する．マイクロプロセッサの速度性能の向上を年率 1.5 倍とし，これらの基幹業務向け製品の速度向上が年率 1.2 倍とすると，10 年で 9 倍，15 年で 28 倍に性能格差が拡大する．年率 1.15 倍では 10 年で 19 倍，15 年で 54 倍に性能格差が拡大する．エミュレーションによるシステム性能スローダウンを 20 と仮定すると，1992 年当時にマイクロプロセッサの 1.4 倍の速度性能を持っていた基幹業務向けコンピュータは，年率 1.2 倍の速度性能の向上を続けても 2007 年にはマイクロプロセッサを用いたエミュレーションでもほぼ同等の速度性能を実現できる．1.15 倍の速度向上でよい場合には必要な性能の 1.9 倍の速度性能を持つ製品を実現できる†13．動的バイナリ変換方式を採用した場合にはスローダウンが小さいため，適用可能時期が早まる．そのような原理により，1998 年頃よりエミュレーション方式によるメインフレーム[20]やオフィスコンピュータ[19]が販売されてきた．

インタプリタ方式の製品分野での近年の適用事例を公開文献より見つけるのは困難であるが，性能レンジが低い機種を中心に適用されてきた．従来ハードウェアによりバイナリ互換を実現していた場合には，FPGA などの比較的高価な部品が削減される原価低減効果，シリーズ全体の LSI 開発品種数を上位機種に限定し開発費を削減する効果がある．上位機向けに動的バイナリ変換を適用する場合でも，下位機向けには原価を最適化した別のホストアーキテクチャを選択することがある．下位機にはインタプリタ方式を適用すると，開発費が安く開発期間が短くて済む．更に上位機の動的バイナリ変換に先駆けた製品投入が可能という効果も出る．

コンピュータ製品以外の分野にも，CPU 開発や市販 CPU チップ入手の課題がある．防衛機器では，ソフトウェアの蓄積も多く製品のライフサイクルが長く，その間に基板レベルでの交換や新基板への置換が行われるため，米国では市販 RISC チップによるエミュレーション研究[23]もある．またエネルギー分野や産業分野での適用事例は報告されていないが，計測処理するデータ量の増大率が高くない領域が広いと考えられ，更にインタプリタ方式は動的バイナリ変換方式に比べて性能変動がほとんどないため，リアルタイム動作に優れており，適用できる分野も今後増えてくると思われる．

2.1.3. インタプリタ方式とバイナリ変換方式

バイナリ変換方式はインタプリタ方式の数倍から十倍程度の速度性能があるといわれている．しかしエミュレータの分野ではそれらの速度性能について公表されたデータはほとんど無い．その理由としては，性能は条件により大きく変わり，またバイナリトランスレータ†14を単体で販売することがなく積極的に数値を出す必要がないためと考えられる．エミュレーションによる性能低下を示す指標としてネイティブ動作の何倍遅いかを示すスローダウン値が使われている．文献[15]の例によると，インタプリタのスローダウン値 25

†13 マイクロプロセッサの性能向上は2002年頃からは年率1.2倍程度の伸びに鈍化したため正確には2007年にこの性能は実現できず，2.1.1で述べたOSの割合（α）の効果を併せて実用化されている．

†14 “バイナリトランスレーション”は用語“バイナリ変換”として記載している．その方式を実装したものは“バイナリ変換器”ではなく“バイナリトランスレータ”に統一して記載している．

～80 に対し，Shade のバイナリトランスレータのスローダウン値は 8～15 である．速度性能はスローダウンに反比例するため，バイナリトランスレータはインタプリタの 3～5 倍の速度向上をするといえる．エミュレーション専用のハードウェアを持たない富士通の Granpower600 の OCT[19]ではインタプリタに対して 2.3 倍と 4.8 倍の速度性能の向上例が記載されている．したがってエミュレーション専用のハードウェアがない場合はこの程度の速度性能向上であると考える．

バイナリ変換方式には静的バイナリ変換と動的バイナリ変換がある．一般的なオブジェクトコードでは命令コードとデータの区別がつかないため，実際にコードが命令として実行されるときに変換を行う動的バイナリ変換が最近の主流である．動的バイナリ変換では図 6（12ページ）に示すようにインタプリタを併用する．バイナリ変換でインタプリタを併用したときの性能向上率について述べる．バイナリ変換の 1 命令あたりの速度性能向上率を A とする．A はインタプリタ実行時間をバイナリ変換で実行した時間で割ったものとなる．併用時にインタプリタを使って実行されるレガシー命令の割合を Pi とする．動的バイナリ変換の全体の速度性能向上率を U とする．U のおよその値は式（1）で表現できる．ここで，A を 50，Pi を 0.01 と仮定すると式（2）に示すように U は 34 倍となる．すべてがバイナリ変換で実行された理想条件では Pi は 0 となりそのときの U_{ideal} は A と同じになり，バイナリ変換の利用効率 E は式（3）となり 67%に留まる．インタプリタの速度性能を上げることができて相対的に A が小さくなると E は改善される．A=10 では U は 9.2 倍となり，E も 92%までその効率を高めることができる．

$$U = \mathrm{f}(Pi, A) = \frac{1}{Pi + \dfrac{1 - Pi}{A}} \qquad \text{式（1）}$$

$$U = \frac{1}{0.01 + \dfrac{1 - 0.01}{50}} = 33.5 \qquad \text{式（2）}$$

$$E = \frac{U}{U_{idea}} = \frac{33.5}{50} = 0.67 \qquad \text{式（3）}$$

したがって，バイナリ変換を適用する場合でもインタプリタの速度性能の向上が重要となる．

2.1.4. C言語記述によるインタプリタ実装の要求と課題

従来，インタプリタ方式の高速エミュレータはアセンブリ言語記述の機械語を用いて実装されてきたが，ホストの RISC 化に伴った C コンパイラの発展により機械語による局所的な最適化効果が小さくなったといえる．現在でも有効なアセンブリ言語記述の長所には次の 3 つが挙げられる．

（1）関数を使わずに間接分岐できること

（2）アドレス制約を課すことにより分岐先アドレスの計算量を削減できること

（3）使用頻度の高いグローバル変数をレジスタ割付可能なこと

更に，機械語による高速化技法としては，パイプラインストールを減らすため変数をレ

ジスタに割り付け，それでも発生するパイプラインストール要因をなくすため命令処理関数内で次命令の解釈を並行処理する手法[16][17]がある．この手法は，本論文で述べる改良function 方式の 1.5～2 倍の速度性能にまで改善でき，バイナリ変換の速度性能に近付けることができる．しかしこの手法は，ホストアーキテクチャへの精通が必須であり，出現頻度が低い命令処理まで合わせてアセンブリ言語記述が必要である．また，エミュレータの改良開発と保守のためにそのようなスキルが必要でありその人材を確保し続けることは非常に難しい．

一方，C 言語でエミュレータを実装すれば，開発工数が少ない，ホストへの依存性が低く長期的にソースコードが活用できる，エミュレータの改良開発と保守を行う技術者のスキルの制約もなくなり人材の確保が容易になる．したがって C 言語実装が望ましいと考えられている．しかし，現状では，C 言語実装は単純に性能が低いこと以外にも次の問題点があるといわれている．

- エミュレーション性能は，エミュレーション対象の命令セットアーキテクチャ（レガシーISA）に対する，ホストの命令セットとの類似性，ホストのマイクロアーキテクチャとの相性，C 言語コンパイラの最適化性能の影響を大きく受ける．C 言語実装は，ホストが決まっているアセンブリ言語による実装に比べて，選択肢が広がる分，開発前に性能見積りを行うことが困難となる．
- 性能が記述スタイルやコンパイラに依存するため，チューニングは試行錯誤的となり性能を実現できる保証が難しく，結局は開発リスクを負ってしまう．

しかし，我々は，C 言語を使用しても開発前の速度性能の見積り及びチューニングによる性能向上は可能であり，しかもレガシーISA とホストの組合せによらず同様な手法で同様な効果が得られると考える．これは，最近の高性能なマイクロプロセッサでは，多段パイプラインやスーパスカラを採用しているものが多いため，エミュレーション性能のボトルネックは演算実行そのものにはなく，命令語の読出し，命令デコード，各命令処理への分岐，次の命令の準備などの共通処理部分（コアループ）にあると考えられるからである．しかし，C 言語を利用したインタプリタの速度性能の解析はほとんど行われておらず，公開された定量的な評価事例がない．そこで本研究では，(1) コアループの処理時間が全体に占める比率が高いことを定量的に示し，(2) レガシーISA やホストへの依存度が低く，C 言語標準に適合し，確実に性能改善が可能なコアループの実装手法を提案する．

2.1.5. インタプリタの高速化における本研究の着眼点と新規性

本節では，本研究の着眼点を述べて新規性を示す．まず，概念レベルの着眼点について述べる．

最近のエミュレータの研究ではバイナリ変換に関するものしか公表されておらず，インタプリタ性能について定量評価する必要がある．しかしながら，インタプリタは古典的な手法であり，その性能を評価した文献が見つからない．また，最近は高性能な RISC マイクロプロセッサが普及しているため，処理性能を支配する条件が従来とは変わり新たな評価が必要となると考える．

C 言語記述のインタプリタ性能の予測は困難といわれているが，我々は予測の容易化は可能であると考えている．それは，インタプリタには間接分岐が必要であり，また遂次性が

高いプログラム処理がコアループに多く存在すると考えるからである．最新のプロセッサは高速化のために色々とハードウェアに工夫をしてきたが，遂次性が高いプログラム処理の存在により，そのような工夫されたハードウェアが活かしにくくボトルネックになりやすい．そこで，コアループを定量評価すれば性能の目途が着くと考える．

また, C 言語を用いてもコアループや実行頻度の高い命令をチューニングする手法は存在すると我々は考える．バイナリ変換はインタプリタと比較して高速といわれその性能は断片的に公表されており，そのバイナリ変換のスローダウン値からインタプリタのスローダウン値を逆算して求められる．そのインタプリタのスローダウン値が大きいことが，改善の余地があると考えた 1 つの理由である．また，性能解析を目的としたインタプリタ方式のシミュレータに関しては，性能が公表されたものが多い．我々のインタプリタの設計経験と比べるとこれらのスローダウンが大きいことが 2 つ目の理由である．C コンパイラが生成した機械命令を調査でき，ホストのプロセッサのパイプライン構造が概略でも理解できれば，ある程度のチューニングは可能となる．RISC プロセッサの構造は一般に類似性があるため，特定のホストへのチューニング手法も他ホストにも共通な効果があると考える．

ここからは，具体的な着眼点について述べる．

一般にインタプリタとして公開されている C 言語のソースコードには, switch～ case ～ break 文を用いた記述が多いようである．この記述は間接分岐をするコードに変換される．分岐先で行う処理をコンパイラが最適化することになるが，その最適化の対象範囲をコンパイラが定めることになるが，コードの塊が大きくなるとその最適化による性能向上が困難になると考える．そこで，命令の種類ごとに関数にすれば，コンパイラが対象とする最適化の範囲が定まり確実に最適化が行われる．個々の関数間のコード共有がされなくなるが，インタプリタのコード量は元々小さいため命令キャッシュメモリに収まり問題にはならない．

C コンパイラはパイプラインの最適化として，命令の順番を入れ替えるリソースのスケジューリングを行っている．しかし，プログラムの処理単位と次の処理単位との関係を的確に把握できない場合や，グローバル変数のアクセスが多いと，リソースのスケジューリングが期待したほど行われないようである．その結果，コンパイル後のホストのコードでは命令間の並列動作や連続動作ができなくなり処理性能が低下すると考える．そこで，次のように考えた．

- コンパイラが生成しやすい単位にプログラムの処理を別ける．具体例としては，関数に別けてコンパイラがその範囲のみを見れば済むようにする．
- コンパイラが変数をレジスタに割り付けられるように，変数の制約を緩和する．グローバル変数が関数内のある部分で頻繁にアクセスされ更にシングルスレッド用にコンパイルされる条件では, グローバル変数アクセスにメモリだけではなくレジスタも使ってアクセスをするように最適化が行われることが多い．しかし，その関数から他の関数を呼び出して戻ってきたときには他の関数内でそのグローバル変数が変更される可能性があるため，実際に変更する関数がない場合でもコンパイラは判断できずレジスタを使った最適化がされない．そのため余分なメモリアクセスが発生し高速化を阻む．このような場合には，グローバル変数のほかにコピーとして関数内にローカル変数を持ち，書込みは両方に行い読出しはローカル変数から行うようにプログラム記述をする．ロー

カル変数はレジスタに割り付けられる可能性が高いため, パイプラインストール要因となるメモリ読出しは行わず高速なレジスタから読出しが行われる. グローバル変数への書込みは, 通常ほかの処理と並行して行われるのでその処理時間は隠ぺいされやすい.

- レジスタが不足するためにローカル変数をメモリ割付けしてアクセスするようにコンパイルされる場合には, プログラムの記述をレジスタが足りるように軽い処理に分割する. インタプリタの処理には, 実際の出現頻度の低いが複雑な処理が結構ある. それらのために, 多くの変数や中間変数が必要となりレジスタ不足が生じやすくなる. このような場合には典型動作を除いた処理を別の関数にするなど単純な処理と複雑な処理を分離し, レジスタ不足を回避することにより典型処理を高速化できる.
- 間接分岐命令があると一般に命令キューからの命令供給が途切れるため, その分岐後の命令列で発生するパイプライン乱れを吸収できなくなりパイプラインストールが発生しやすくなる. そこで, 命令処理関数への分岐実行直後の数命令がよどみなく流れるようにする. 具体例として, 命令処理関数の例を挙げる. 加算を行う ADD 命令などの命令処理では, インタプリタはレガシーISA のレジスタが格納されているメモリ配列からデータ値を読み出すが, その前に命令コードが格納された変数からレジスタフィールドの切り出す必要がある. レガシーレジスタ 2 個分のメモリ読出しはパイプラインとして連続動作できる可能性があるが, 命令コードの読出しが終わらないとレジスタフィールドの切り出しをする命令がストールし, それに続くレガシーレジスタのメモリ読出し命令のディスパッチもできなくなる. RISC の場合には命令語長などの関係から, 命令コードの格納された変数やレガシーレジスタの格納された配列変数のアドレス計算も必要となる場合もあり, それらもパイプラインストール要因となる場合が多い. これらのストールを回避する手段は, メモリ参照が不要となるように予めホストのレジスタにデータを埋め込むことであり, 関数呼出しの引数として渡すなどの方法で対応できる.
- インタプリタでは, 各レガシー命令に対応した処理動作が終わりコアループ処理に戻ると, そこでもパイプラインストールを発生しやすい. 具体的には, 前者でセットしたフラグ類を後者で参照するために, リードアフターライトのパイプライン干渉が発生しやすい. これもレジスタ渡しで解決できる.
- コアループでは, 命令コードのデコードは分岐命令で実行する. このときに命令のタイプによりフィールドの切り出し位置や回数が異なるため, 条件付分岐命令を使って判定を行うことが多い. しかし, マイクロプロセッサの分岐予測はプログラムカウンタやスタックポインタなどの履歴を用いた予測であり, 変数のデータ値を用いた分岐予測まで行っていない. そのため, インタプリタでは一般のプログラムに比べて分岐予測ミスが発生しやすい. また, 一般のプログラムではレジスタ間の演算命令に比べてアドレスオフセット付きのメモリアクセス命令や即値を使った命令の頻度が比較的に高いという知見から, インタプリタ処理では 1 つの命令フィールドのみの命令デコードで済む確率が高いという予見ができる. この両者から, 命令デコード方式を評価して最近のマイクロプロセッサに適した方式を選定する必要があると考える.

ここからは, 本研究の新規性と有効性について述べる.

まず, C 言語記述のインタプリタ性能に関して, 性能向上を評価した研究で公開されたものは珍しく文献から探すことができないため, 新規性または有効性があると考える. なお,

エミュレータを目的としたインタプリタ以外では，分析はされてはないが工夫による性能向上効果を報告した研究として，SimCore/Alpha[5]が挙げられる．

次に，インタプリタの処理はコアループが支配的であるという知見を持っている人は多いと考えられるが，それを定量的に評価した研究も少ないと思われる．これも文献から探すことができていない．したがって，コアループの割合をインタプリタの改善前と改善後の両方について評価しているところにも新規性がある．

C 言語でインタプリタを記述する方法として，後述する switch 方式と function 方式があるのは一般的に知られている．しかし，文献や公開されたソースコードでは前者が多く後者は少ない．これらの得失を評価した研究も見当たらないため，両者の性能の評価を行ったところに，本研究の新規性がある．同様に，function 方式の有効性を検証したことにも本研究の新規性がある．

インタプリタの性能向上効果の評価として，本研究では 5 種類のレガシーISA，3 種類のホストを対象とし計 63 種類[†15]のエミュレータを試作した．本論文は，その評価結果を示しており，客観性がある多量なデータを提示しているところに有効性がある．手法そのものには新規性はないが，色々な視点からの評価の対象項目を挙げており，そのデータを併せて提示しているところに新規性があると考える．

2.1.6. インタプリタ方式の実装と課題

本節では，インタプリタ方式のエミュレータ実装の方法と，それにより発生する性能への影響を述べる．専用ハードウェアを用いずにソフトウェアのみで実装されたインタプリタ方式のエミュレータの動作原理を簡単に解説して分類する．

2.1.6.1. 対象とするホストアーキテクチャ

インタプリタを実行するホスト CPU としては，一般的な RISC 方式を対象とする．すなわち，パイプライン制御，スーパスカラ実行，キャッシュメモリ，RISC 型[†16]の命令セットを前提とする．

2.1.6.2. レジスタのマッピング

エミュレーションの高速化のためにレガシーISA のレジスタをホストのレジスタにマッピングするのは常套手段である．しかし，次のような場合にはメモリに割り付ける．

(1) ホストの利用可能なレジスタ本数がレガシーに比べ充分に多くない．

(2) レガシーISA のレジスタ本数が多くレジスタ番号ごとに命令コードをデコードするとデコーダが膨大になる．

[†15] RISC 用に switch 方式を含めて 45 種類ある．そのうち，PISA の 2 種は sim-safe のわずかな修正のみである．これらのほかに x86 用に 18 種あり合計 63 種類である．

[†16] 高性能なマイクロプロセッサは RISC の命令セットで動作している．インテル社や AMD 社の x86 は命令体系は CISC であるが，Pentium-Pro 以降は μops という RISC 動作に変換している．（なお 2008 年に登場したローエンド組込み機器向けの ATOM は CISC である．）

(3)　プログラム言語の制約によりレジスタへの割付けができない.

C 言語実装では，レガシーISA のレジスタは変数として記述され，特に配列として記述したものはメモリにマッピングされる.

2. 1. 6. 3. 命令デコードと分岐処理

命令語には，命令動作を示すフィールドとオペランドを指定するフィールドがある．インタプリタは命令動作を示すフィールドを参照して命令動作固有の処理に分岐し，命令実行処理を行う. 命令動作を示すフィールドは, 単一のフィールド(opcode) または複数フィールド（opcode と subop）から構成され，フィールドの個数，ビット長，ビット位置はレガシーISA により異なる．それぞれの命令動作固有の処理に分岐する方式には，直接分岐方式と間接分岐方式の二つがある．直接分岐方式は，opcode のビット列に固定値を連結または加算したアドレスに分岐する．間接分岐方式は，同様にして生成したアドレスをポインタとして用いて分岐する．直接分岐方式は性能面に有利であるがアドレスの制約が厳しく，保守性に問題があり，何よりも C 言語では仕様上記述ができないため，用いられるのは間接分岐方式に限られる. この方式を opcode と subop のデコード及び分岐方法により更に表2に示すように分類する.

表2 命令デコード方式の分類

項目番号	方式	説明
(1)	逐次多段分岐	opcode，subop の順に 1 フィールドずつデコードと分岐を逐次的に繰り返す. subop を伴う opcode は専用の分岐先エントリを持ちそこで更に subop のデコードと分岐をする.
(2)	一括分岐	opcode と subop フィールドをデコードしそれらの結果より一括して分岐をする.
(3)	一括冗長分岐	命令フィールドが f1，f2 からなる命令形式と f1，f3 からなる命令形式があるときに f1, f2, f3 を結合した冗長ビットを含む全ビットをデコード対象とする．分岐先エントリへのポインタの数は（1）(2) よりも多くなる.

（1）は最悪時にはフィールド個数分の間接分岐を繰り返すため（2）に比べて遅いように見える．しかし，(2) にはフィールド切り出しの演算，フィールド位置ごとの処理への分岐またはメモリ読出しによる表引きが必要となる．そのため，ホストの実行ではデータ依存性によるパイプラインストールや分岐予測ミスなどが生じやくなり，実際は（1）より遅くなることが多い．(3) の方式は，フィールド位置が連続していてポインタ表がキャッシュメモリに入れば最も高速となる．これら（1）～（3）のいずれでもパイプラインストールを発生することが多い.

2.1.6.4. 分岐命令によるプログラムカウンタの更新

レガシー命令が分岐を伴わない“非分岐命令”のときは，プログラムカウンタの更新は固定値または命令語長の加算のみでよい．しかし，分岐命令では，命令によって異なる条件に基づいて分岐発生の有無とプログラムカウンタ値が決まる．インタプリタ処理の繰返しであるコアループでは，分岐命令と非分岐命令の動作が混在するためエミュレータの実装方法によっては分岐有無の判定が必要となり，コアループ形成の分岐と併せてホストの

パイプラインストール要因の一つとなる.

2.1.6.5. 例外判定

プログラム例外の検出はエミュレータに必須な機能である．命令には例外が発生し得る命令と発生し得ない命令があるため，個々の命令処理で例外を検出し，異状を示す例外フラグをセットするのが一般的である．このため，コアループでは例外フラグの判定が必要となる．この判定に関するホストの分岐予測は多くの場合ヒットするため後続命令列の投機実行による高速化が期待される．しかし，分岐条件が確定する前に次の予測を行うことが多くホストによってはパイプラインストールの要因の一つになる.

2.1.6.6. オペランドのデコードと演算処理

演算の入力と出力を示す汎用レジスタ番号を示すフィールドの切り出しは，シフトとANDにより行う．その結果をインデックスとしてメモリ配列から読出しを行うことで，演算のソースデータとなる汎用レジスタからのデータを得る．そのデータに対して，命令種ごとに算術・論理演算またはメモリ読出しを行う．これらの一連の動作自体は，データ依存性が存在するため並列には動作できないが，フィールドが複数存在する場合は，スーパスカラによる整数実行ユニットが並列動作する．また，複数のメモリアクセスもパイプライン動作によりオーバラップ可能であるため，パイプラインストールは小さい.

2.1.6.7. インタプリタ方式のエミュレータのまとめ

以上，インタプリタ方式の原理と実装に伴う性能への影響を述べた．特にコアループで行う命令デコードと分岐処理，プログラムカウンタ更新，例外フラグ判定が性能上の課題であることを示した.

2.2. インタプリタ方式エミュレータの速度性能の評価方法

本節では，レガシー命令セットとホストの組合せによらずに，性能見積りが容易で性能向上が可能なインタプリタを C 言語で実装できることを示すため，事例研究と提案方式のそれぞれに共通な評価を実施した．ここでは，この評価方法を述べる.

性能評価に使用した，レガシーISA，ホスト CPU アーキテクチャ，評価プログラムの要約を表 3に示す.

表3 評価対象の要約

	評価対象	記載箇所
レガシーISA	PISA, MIPSlike, PowerPC, SH4, M32R	2.2.1
ホスト CPU	MPC7450, SparcIIIi　（参考:Core Duo）	2.2.2
評価プログラム	I-LOOP, SPEC CPU95	2.2.3

2.2.1. レガシーISA の選択

従来からのエミュレーションの適用対象は主に CISC であった．1990 年代には低価格マイコンを除き,プロセッサの主流はRISCになり,現在ではそのRISCも淘汰され種類が減っ

ている．PowerPC や Sparc をエミュレーションするソフトウェア製品の出現 [24]など，今後は，CISC と併せて RISC もエミュレーション対象になると考える．エミュレーションの評価では，CISC と RISC それぞれ数種について行うのが望ましいが，評価環境の入手と構築，更に同一の基準で測定できるベンチマークプログラムの適用可否が課題となる．本論文では，レガシーISA に RISC を使用してまず評価を行い，CISC にインタプリタを適用したときの速度性能については4.7.5で述べる．

2.2.1.1. レガシーISA の選択理由

まず評価対象として，SimpleScalar [7]の PISA（Portable ISA）を用い，インタプリタの特性を解析した．SimpleScalar は教育の目的及びプロセッサアーキテクチャの評価を目的としたツールセットで，C コンパイラ，ライブラリ，シミュレータが WEB 公開されている．PISA を使用した理由は，次のとおりである．

(1) 命令セット仕様が公開されている．

(2) sim-fast，sim-safe などの形でインタプリタ実装例がソースコードとして公開されている．

(3) SPEC CPU95 [13]ベンチマークの実行にはレガシーOS の模擬が必要であるが，同ツールセットの静的ライブラリの利用によりシステムコールのシミュレーションが容易に実現でき，コンパイル済みの SPEC CPU95 のバイナリが利用できる．

PISA は仮想マシンであり，一般的なマイクロプロセッサに比べデコードが単純である．そこで，PISA の命令フィールドを MIPS-IV [8]相当に改造した MIPSlike を定義し，これの評価も行った．更に，本議論が特定のレガシーISA に依存しないことを示すため，PowerPC 命令セット[9]，SH-4 命令セット[10]，M32R 命令セット[11]の評価も行った．これらの命令セットを扱うことにより，命令語長（32 ビット，16 ビット，混在），命令フィールドの数と長さ，遅延分岐の有無，条件コードの有無，命令機能への修飾の有無，可変長データを扱う命令の有無など，多様的な評価ができる．

2.2.1.2. SimpleScalar/PISA（PISA）

SimpleScalar/PISA は 115 種の命令と 32 種のシステムコール実装からなる．図 9に示す 64 ビット長の命令語形式を用いており，命令フィールドは図中に灰色で示した 1 バイトに単純化されている，MIPS-IV のサブセットの機能を持つが遅延分岐命令は採用していない，ビッグとリトルの両エンディアンがある，アドレス境界が整列されてないデータ向けの LWL/LWR 命令や平方根を求める SQRT 命令などが特徴的である．MIPS-IV ではメモリアクセスにはアドレス境界の制約があり違反すると例外になるが，PISA では例外検出をしない仕様になっている．

	0-23	24-31	32-39	40-47	48-55	56-63
(1)	-	opcode	RS/FS	RT/FT	RD/FD	RU/SHA
(2)	-	opcode	RS/FS	RT/FT	UIMM/IMM/TARGET	
(3)	-	opcode	TARGET			

図9 SimpleScalar / PISA の命令形式

2.2.1.3. MIPSlike

PISA の命令コードを MIPS ライクの 32 ビットの命令語形式に変更した MIPSlike を定義する．図 10に示すように MIPS-IV の命令語形式と同じく，J 命令と JAL 命令は（3），16 ビット即値を持つものは（2），浮動小数演算は（4），そのほかでは（1）の形式を採る．命令フィールドは図に灰色で示すように，命令語形式により 1～3 個まで変化する．NOP 命令は MIPS-IV では形式 (1) であるが，最速動作の評価のため形式 (3) として定義した．また，メモリアクセス時のアドレス境界の例外検出を追加し，MIPS-IV など一般的な RISC マイクロプロセッサの仕様に合わせた．

	0-5	6-10	11-15	16-20	21-25	26-31
(1)	opcode	RS	RT	RD	-	subop
(2)	opcode	RS	RT	UIMM/IMM/TARGET		
(3)	opcode	TARGET				
(4)	opcode	subop1	FT	FS	FD	subop2

図10 MIPSlike の命令形式

2.2.1.4. PowerPC

PowerPC の命令形式の代表的なものを図 11に示す．約 200 種類の命令から構成され，subop フィールドが最大 10 ビットある．分岐命令は BO/BI/AA/LK フィールドにより修飾される，演算命令の Rc フィールドによる 4 ビットの条件コード生成指定，lmw / stmw などの可変長命令，倍長演算命令，積和演算など強力な命令セットが特長である．

0-5	6-10	11-15	16-20	21-25	26-30	31
opcode	S	A	UIMM/SIMM/BD			
opcode	BO	BI	BD		AA	LK
opcode	LI				AA	LK
opcode	D	A	B	subop		Rc
opcode	D	A	B	C	subop	Rc

図11 PowerPC の命令形式の例

2.2.1.5. SH-4

SH-4 は約 230 種類の命令を持ち，図 12に示すように命令語長は 16 ビットに固定されている．遅延分岐命令と非遅延分岐命令の両方を持つ．整数系はシフトや除算などが簡素化され，浮動小数は積和演算（FMAC），平方根（FSQRT）が強化されているなどの特長がある．

0-3	4-7	8-11	12-15
opcode	Rn	UIMM/	IMM
opcode	subop	UIMM/	IMM
opcode	Rn	Rm	subop
opcode	Rn	subop2	subop1
opcode	subop3	subop2	subop1

図12 SH4 の命令形式の例

2.2.1.6. M32R

M32R は図 13に示すように 32 ビットと 16 ビットの命令語長を持つ約 150 種の命令から

構成されている．2 オペランドの命令を中心とした整数命令を基本とし，命令機能の修飾をするフィールドもなく単純化されている．

0-3	4-7	8-11	12-15	16-19 20-23 24-27 28-31
opcode	LI			
opcode	RD	subop	RS	UIMM/SIMM/BD
opcode	RD	UIMM/ IMM		Another Instruction
opcode	subop	UIMM/ IMM		Another Instruction
opcode	subop2	Subop1	RS	Another Instruction

図13 M32R の命令形式の例

2.2.2. ホストアーキテクチャの選択

エミュレータを組み込んだシステムを製品化するときには，CPU チップの製造販売期間が長く, 独自 OS を移植するのに必要な仕様やエラッタ情報が少量ユーザにも入手可能であることが，ホストアーキテクチャ選択の前提となる．また，性能の測定が容易であり，結果の解析が行えることも条件とした．C 言語コンパイラには，メーカが製品としてサポートしているものを使用する．

この条件に合うホストアーキテクチャとして PowerPC を選択し, CPU に MPC7450 [12] 867MHz を搭載したマシンを使用した．この条件が近いものとして更に Ultra-SparcIIIi（1.28GHz）も使用した．両者ともにワークステーションに用いられており，容易に利用することができる．加えて PowerPC は，組込み用途にも使用されており，それ自体もレガシーアーキテクチャとして評価できる．更に，パイプライン構造が公開されており比較的容易に解析できる利点がある．

評価に利用したホストマシンの仕様, OS, コンパイラとそのバージョン, 最適化オプションを表 4に示す.

表4 評価対象のホストマシンの仕様

プロセッサ，コア周波数	パイプライン段数	命令発行数	1 次キャッシュ	2 次／3 次キャッシュ	マシン名	OS，C コンパイラ，最適化オプション
MPC7450, 867MHz	5	3	命令:32Kbyte データ:32Kbyte	2 次:256Kbyte 3 次:2Mbyte	Power Mac G4	OSX(BSD), gcc3.1, -O3
Ultra SparcIIIi, 1.28GHz	14	4	命令:32Kbyte データ:64Kbyte	2 次:1Mbyte	SunFire V240	Solaris, SUN C5.6, -xO4

本評価では，エミュレータの製品化の条件に合わないため x86 はホストの対象とはしていなかった．しかし，x86 を除くと正当な評価にならないという意見もあったため，参考として評価に加え測定結果を表やグラフに併記した．x86 に関する評価結果の説明は，4.5で x86 の課題と併せてまとめている．使用した x86 マシンの仕様を表 5に示す.

表5 参考評価対象のホストマシンの仕様

プロセッサ，コア周波数	パイプライン段数	命令発行数	1次キャッシュ	2次／3次キャッシュ	マシン名	OS，Cコンパイラ，最適化オプション
Core Duo, 1.06GHz	12	3	命令:32Kbyte データ:32Kbyte	2次: 2Mbyte	ノートPC CF-R6	Windows-XP(SP2), VC++ Ver6, /O2

2.2.3. 評価プログラム

インタプリタの速度性能を計測するため，簡易プログラムI-LOOPと，SPEC CPU95を使用した．

2.2.3.1. I-LOOP

命令個々の実行クロックサイクルの計測に簡易プログラム（I-LOOP）を使用する．I-LOOPは，同一命令を繰り返して実行する単純なループから構成される．ここに，即値とレジスタのORを取るORI命令のコード例を示す．r15にはあらかじめループ回数を初期設定しておき，測定対象命令の処理時間は，N の個数をゼロにした測定結果との時間差から計算した．

I-LOOPのコード例

```
1. LOOP:  ORI     $1,$1,1              // 1個目
2.        ORI     $2,$2,1              // 2個目
                  :
8.        ORI     $8,$8,1              // N個目（N=8）
9.        ADDIU   $15,-1               // ループカウンタを-1
10.       BGT     $15,$0,LOOP          // ループの繰返し
11.       ORI     $2,$0,1
12.       SYSCALL                      // システムコール
```

2.2.3.2. SPEC CPU95

エミュレーション対象のプログラムの違いによる性能への影響を見るため，SPEC CPU95[13]のCINTとCFPを使用した．SPECのホームページ記載の評価結果（result）によるとCPU95では100MHz～1GHzのCPUが，CPU92では40～350MHzのCPUが報告されている．インタプリタ方式によるネイティブからの速度性能の低下（スローダウン）を30程度とすると30～100MHz程度のCPUと同等の負荷を想定できる．マイクロプロセッサの今後の性能向上を考慮して,今回の評価では負荷の重いCPU95を用いた.SPECはTLBやキャッシュメモリの影響を考慮して，新版の度にメモリのワーキングセットを拡大してきている．そのため，エミュレーション対象としてCPU95では重過ぎる可能性が考えられるが，4.7.3と4.7.6にて後述するようにインタプリタ方式のエミュレータではそれらの影響はほとんど考慮する必要はない.

評価対象はCINTのすべて（099.go，124.m88ksim，126.gcc，129.compress，130.li，132.ijpeg，134.perl，147.vortex）と，CFPからは命令数が 4×10^9 個以下の102.swim，110.applu，141.apsi，145.fpppp，146.wave5を選んだ．入力データとしては標準としてtrainセットを使用した．trainでは命令数が少ない128.m88ksimはtestを用い，129.compressには“40000 e 2231”を設定し，命令数が 100×10^6～3000×10^6 個になるよ

うに調整した.

エミュレータの基本的な分析や，対象プログラムの差による影響を調べる際には，これら 13 本のプログラムすべてを用い，他の評価にはシステムコールの種類が少なく評価が容易な 099.go に“30 11”をパラメータ設定して用いた.

2.2.4. 評価用インタプリタと実用版との違い

評価用インタプリタは，実用的なインタプリタと次の点が異なるが，それらの影響は無視できる.

- エミュレータから見えるメモリの論理アドレスの範囲
- システムコールの呼び方と戻り方
- エミュレータを実行する OS
- コアループへの出入り（例外検出後の処理）
- 評価用はシグナル処理実装を省略
- I/O やタイマー割り込みの発生頻度

命令数の計数は実用インタプリタでは実装しないが，評価用では評価の都合上実装したままとする.

2.3. インタプリタ方式のエミュレータ実装事例に基づく性能解析

本節では，まずインタプリタ方式のエミュレータの C 言語実装を2.3.1で述べる．具体的な実装事例として SimpleScalar（Ver.3.0）のシミュレータ sim-safe の構造を2.3.2で解説し，2.3.3で一般のプロセッサと類似の命令語形式を採った sim-MIPSlike の試作について述べる．2.3.4では sim-safe を2.3.5では sim-MIPSlike の速度性能をそれぞれ評価し，インタプリタの処理時間を支配しているコアループは2.3.6で述べる．最後に，これらの実装方式の課題を2.4に示す.

2.3.1. インタプリタ方式のＣ言語実装

まず，インタプリタ方式の C 言語実装に関して一般的な事項を述べる.

2.3.1.1. C 言語でのリソース割付

汎用レジスタや浮動小数レジスタは一般に，レジスタ番号をインデックスとしてアクセスできるように配列に割り付ける．プログラムカウンタ，状態語レジスタ，制御レジスタは個別の変数とする．これらを総称してエミュレータ変数と呼ぶことにする.

C 言語で用いられるローカル変数は, コンパイラの最適化によりレジスタに割り当てられる可能性があるが，スタック領域であるためスコープの関係から他関数から直接アクセスができない．一方，グローバル変数は，一般的なコンパイラではレジスタに割り付けられないが，どの関数からもアクセス可能である．エミュレータ変数はインタプリタの各所で利用するため，グローバル変数として実装される傾向が強く，まとめて構造体として実装することが多い.

アセンブリ言語記述ではレジスタの用途を決めておきプログラムカウンタなど一部のエミュレータ変数をレジスタに割り付けることにより高速化するのに対し, C 言語実装では一般的には明示的なレジスタ割付けができず，コンパイラ任せになる．

2. 3. 1. 2. ホストのレジスタの割付

インタプリタをコンパイルする際に，エミュレータ変数，デコード表，ポインタや中間変数がホストの汎用レジスタに割り付けられるが，この割付けが性能に及ぼす影響は大きいため注意が必要である．すなわち，ホストのユーザモードのレジスタ本数，そのうちコンパイラの規約で割付け可能な本数，更にセーブとリストアのオーバヘッドなしに関数が使用できる本数により，エミュレーション性能が影響を受けることになる．

2. 3. 1. 3. 命令固有処理への分岐

アセンブリ言語での実装ではジャンプテーブルを使用した間接分岐が多用されるが, C 言語では，(a) switch～case～break 文による記述，または，(b) 配列を用いて関数への間接分岐する記述を用いる．便宜上，(a) を switch 方式，(b) を function 方式と呼ぶことにする．switch 方式では，case 値の上限値チェックを行うコードが生成される．

2. 3. 2. SimpleScalar のシミュレータ sim-safe

SimpleScalar では，高速にシミュレーションを行う sim-fast，例外条件チェックが追加されている sim-safe などのシミュレータが公開されている．これらはエミュレーションを直接目的にしたものではないが，エミュレータ並みの高速性を持つ．これらは2.3.1.3で定義した switch 方式に該当する．本節では，実装事例として評価に用いる sim-safe の構造を解説する．

2.3.2.1. sim-safe のエミュレータ向けの事前改良

sim-fast は sim-safe に比べて 20%ほど高速であるが，エミュレータとして必須な例外検出処理などが欠けている，そこで，SimpleScalar Ver.3.0 の sim-safe をベースとし，更にエミュレータ向けに事前改良を行ったものを sim-safe として評価に用いた．改良箇所は次の点である．

- sim-fast にも共通であるが 2Gbyte のアドレス空間をフルにアクセスする目的と TLB を模擬する目的などからメモリ空間をページ管理しており，そのために性能が Ver.2.0 より劣化していることがわかった．そこで，フラットなメモリ空間 0～0x17FFFFFF 用の領域を用意しそのアドレスをポインタ（memoryp）から取得しアクセスするように適正化した．これにより約 4 倍の速度性能の向上を実現した．
- デバッガ Dlite の呼出しや表示の判定，メモリアクセス数のカウント，命令数カウント値による終了判定を削除した．なお命令数のカウントは評価上必要なため 64 ビット変数を 32 ビット整数に変更しオーバヘッドを削減した．

なお, SimpleScalar の各シミュレータでは図 9 (2.2.1.2) に示した命令語をプログラムロード時にプリデコードして enum 値に変換し，インタプリタ実行時は enum 値を使用している．本来，プリデコードを用いないコードで評価すべきであるが，1 バイトのコード間の変

換であり性能的には差が無いことを確認したため，そのままの形で使用した．これ以降では，改良を施した sim-safe を評価対象として説明する．

2.3.2.2. sim-safe のエミュレータ変数

レガシーレジスタの構成を次に示す．なお，実際の記述はマクロを用い複雑なため，展開後の内容を示している．

レガシーレジスタ構成の記述

```
typedef unsigned int UINT;
struct {
  UINT regs_R[32];                                              // 汎用レジスタ
  union { double d[16]; float f[32]; UINT l[32];} regs_F;// 浮動小数レジスタ
  struct {
      UINT hi, low;                                             // 乗除算の結果
      int fcc;                                                  // 浮動小数演算フラグ
  } regs_C;
  UINT regs_PC;                                                 // プログラムカウンタ
  UINT regs_NPC;                                                //  次の命令アドレス
} regs;
```

上記の他に，次に示すような変数が用意されている．fault を除きグローバル変数である．memoryp はメモリ領域の先頭を示すポインタであり，エミュレータ実装に依存し，ゼロ番地からのオフセットまたは値ゼロがセットされる．本評価では主記憶イメージの先頭を指す．

そのほかの変数

```
struct { UINT a, b} inst;                                   // 命令語(上下) global
UINT sim_num_insn;                                          // 命令数計数 global
int  fault;                                                 // 例外発生フラグ local
unsigned char *memoryp;                                     // メモリ領域 global
```

2.3.2.3. sim-safe における命令固有の処理の実装

SimpleScalar の各シミュレータは#define 文によるマクロを用いて処理コードを組み立てている．ここでは，“LW RD, offset(RS)”の例を示す．この命令は，データ 4 バイトをロードする命令で，16 ビットのオフセットとインデックスレジスタを用いている．

#define 文マクロにより命令コードを記述している例

```
#define LW_IMPL                                             ¥
{   _result と _fault の変数宣言                            ¥
    _result = READWORD(GPR(BS) + OFS, _fault);              ¥
    if(_fault!=md_fault_none)DECLARE_FAULT(_fault)          ¥
    SET_GPR(RT, _result);                                   ¥
}
```

この定義はプリプロセッサにより変換され下記のような冗長かつ解読が困難なコードに展開される.

マクロ展開された後のコードの例

```
case LW: { word_t _result; enum md_fault_type _fault; _result = ((_fault) =
md_fault_none, addr = ((regs.regs_R[(inst.b >> 24)]) + ((int)(( short)(inst.b
& 0xffff)))), (*((word_t *)(memoryp+addr)))); if (_fault != md_fault_none)
{ fault = (_fault); break; }; (regs.regs_R[((inst.b >>16) & 0xff)] = (_result)); };
break;
```

これをコンパイルすると，値が変化しない_fault の判定による break などの冗長部が削除され，下記の記述と等価な機械語コードが生成される.

コンパイラで最適化された機械命令を C 言語で記述した例

```
case LW: regs.regs_R[(inst.b >> 16) & 0xFF] =
*((UNIT *)(memoryp + regs.regs_R[inst.b >> 24] + (inst.b << 16 >> 16)));
break;
```

2.3.2.4. sim-safe のコアループ処理の解説

sim-safe のコアループの構造を下に示す．この場合，コアループは，各命令に固有の処理を含む形でマクロ展開される.

sim-safe のコアループの構造

```
while(1) {
    regs.regs_R[0] = 0;                          // 値ゼロを保証
    inst.a = *((UINT *)(memoryp + regs.regs_PC));
    inst.b = *((UINT *)(memoryp + regs.regs_PC+4));
    sim_num_insn++;
    op = inst.a & 0xFF;
    switch (op) {
     case A: ……; break;                         // 命令 A
     case B: ……; break;                         // 命令 B
     default: _panic(…);
    }
    if(fault != 0)                               //例外発生し得る OP のみ実行対象
       fatal("…");
    regs.regs_PC = regs.regs_NPC;
    regs.regs_NPC += 8;
}
```

このコードでは, op に例外発生条件がない場合には fault の判定は行わないように最適化されている．また，2 つの変数 regs_NPC と regs_PC の使用により2.1.6.4で述べた“分岐有無の判定”も不要とし，regs_PC の更新と inst のロードとの間が空いているためパイプラインがストールしにくいように工夫されている.

2. 3. 3. sim-MIPSlike の試作

PISA は仮想的なプロセッサであり，デコードが単純であり，例外検出も不足している．より現実的なプロセッサのエミュレータを解析を行うためには，市販されているプロセッサの命令語形式と同様の評価が必要である．そこで，sim-safe を次のように改造し，2.2.1.3 で定義した MIPSlike の命令形式を実行する sim-MIPSlike を試作した.

- 命令語は上位 32 ビット（inst.a）に集約
- プリデコードを止め inst.a の値にシフトと定数マスクを行う逐次多段方式による分岐

を採用

- switch-case 文の入れ子を 3 レベルまで用いた switch 方式の採用
- 各命令処理では MIPS-IV と同様のフィールドよりレジスタ番号などのフィールドを切り出し
- LB,LBU,LWL,LWR,SB,SBU,SWL,SWR 命令を除く全メモリアクセス命令に，アドレス境界条件を比較して違反時には例外フラグ（`fault`）のセットを追加

2.3.4. sim-safe の速度性能の解析

本節では，sim-safe と sim-MIPSlike の速度性能の解析を行う．

性能の指標としてレガシー命令 1 個を実行するホストの平均クロック数を CPI（Clock cycles Per Instruction）として用いる．CPI の数値が小さいほど速度性能は高い．まず，2.3.4.1では sim-safe の I-LOOP 性能を測定し，理想的なアセンブリ言語実装コードと比較する．2.3.4.2では SPEC CINT/CFP95 の速度性能の結果より評価プログラムによらず CPI 値が同様になることを示す. 次に2.3.4.3では, C 言語実装によって発生した sim-safe のオーバヘッドを解析する．

2.3.4.1. sim-safe の I-LOOP 性能

sim-safe を MPC7450 と SparcIIIi 用にコンパイルし，I-LOOP 実行時間とプロセッサ周波数より，それぞれの命令の CPI を測定した．この結果を表 6と表 7の“Measured CPI”として示した．I-LOOP で繰返し実行する命令には，次の 4 つを選んだ．

・NOP： コアループの速度性能を示す

・ADDU：オペランドが 3 個ありオペランドデコードが多い

・LW： メモリアドレスの計算からメモリロードが遅延要因となる

・BEQ： 分岐判定とプログラムカウンタ値を使った計算が特徴的である

ホスト MPC7450 用には，レジスタ割付とパイプライン最適化を施した理想的なインタプリタをアセンブリ言語で作成し，I-LOOP により実測した CPI（Ideal CPI）と，sim-safe の実測結果を比較した．SparcIIIi ではパイプライン詳細動作が不明なため，コンパイラ生成のアセンブリ言語ソースから求めた命令数（Actual）を机上作成した理想命令数（Ideal）と比較した．結果として，sim-safe の速度性能は，理想コードに対して SparcIIIi では 39～57%，MPC7450 では 40～47%の速度性能であり，理想コードとの差が大きいことが分かる．

表6 MPC7450 の sim-safe エミュレーション性能

Instruction	Measured CPI	Ideal CPI	Ideal / Measured
NOP	33.3	15	45%
ADDU	40.0	17	42%
LW	40.8	19	47%
BEQ(taken)	45.2	18	40%

表7 SparcIIIi の sim-safe エミュレーション性能

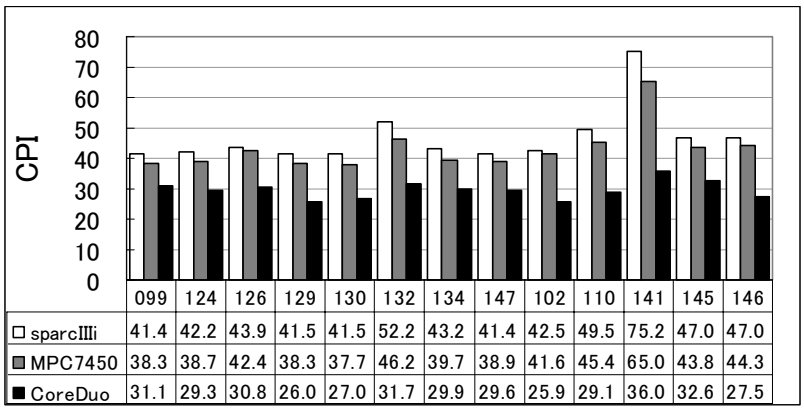
Instruction	Measured CPI	Instruction steps		
		Actual	Ideal	Ideal/Actual
NOP	36.9	31 (28+3)	12	39%
ADDU	40.9	43 (28+15)	23 (12+11)	53%
LW	37.2	45 (28+17)	23 (12+11)	51%
BEQ(taken)	40.1	47 (28+19)	26 (12+14)	57%

2.3.4.2. sim-safe の SPEC CINT/CFP95 の速度性能

次にSPEC CPU95を用いて平均CPIを測定した. 結果を図14に示す. 132.ijpeg, 110.appl, 141.apsi を除いた 10 本の結果では，これらの平均に対し CPI の増減は SparcIIIi で 9%, MPC7450 で 10%以内の範囲に収まっている．132.ijpeg，110.appl と 141.apsi の CPI が他より大きいのは，乗算命令が（mult）がそれぞれ 1.3%，0.7%，3.8%の出現頻度ながら 1 命令の実行に SparcIIIi で 1404 サイクル，MPC7450 で 776 サイクル掛かっているためである．これらは，乗算命令の実装変更により改善できる．

図 15に命令種別毎の頻度を示す. 頻度はロード (load), ストア (store) , 分岐 (branch), 整数演算と論理演算 (int / log) を行う基本命令, 乗除算 (mpy / div), 浮動小数演算 (float), その他（others）に分類してある．棒グラフでは，この順に下から積上げて表示している．その他は，LWL/LWR/SWL/SWR/SQRT/SYSCALL などレガシーISA 固有のものを示す．折れ線により基本命令の合計（basic）を示しているが，CINT では 132.ijpeg の 98.6%を除き 99.4%以上となっている．この結果より load～int / log までの単純な命令の比率が高く，またそれらが ISA 固有の複雑な命令に変わる頻度も小さいことから，評価プログラムの違いによらずインタプリタは同様な CPI 値を示すといえる．

図 16に，異なるレガシーISA 間での命令種の比率を 099.go で測定した結果を示す．この図からも，基本命令の比率は同様に高く，ISA 固有の複雑な命令の影響は無視できるといえる．これよりレガシーISA 及び評価プログラムに依存せずにインタプリタは同様な CPI 値を示すと考えられる．なお，他の ISA の評価結果はここでは示さず，提案方式の評価結果と併せて4.6の表 12（57ページ）と図 23（57ページ）に示す.



	099	124	126	129	130	132	134	147	102	110	141	145	146
□ sparcIIIi	41.4	42.2	43.9	41.5	41.5	52.2	43.2	41.4	42.5	49.5	75.2	47.0	47.0
▩ MPC7450	38.3	38.7	42.4	38.3	37.7	46.2	39.7	38.9	41.6	45.4	65.0	43.8	44.3
■ CoreDuo	31.1	29.3	30.8	26.0	27.0	31.7	29.9	29.6	25.9	29.1	36.0	32.6	27.5

図14 sim-safe の SPEC 95 エミュレーション性能

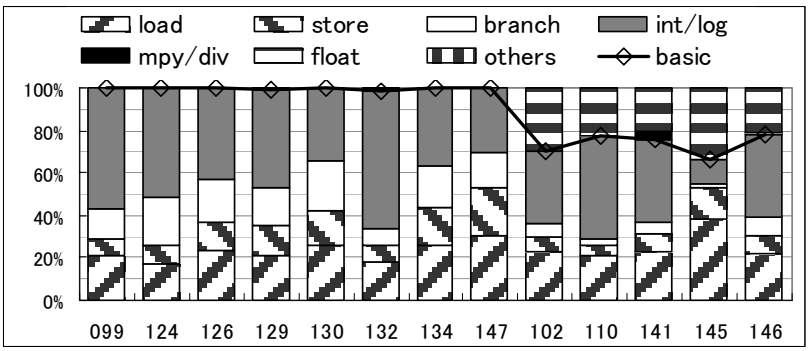


図15 SPEC 95 の命令種別毎の頻度（PISA）

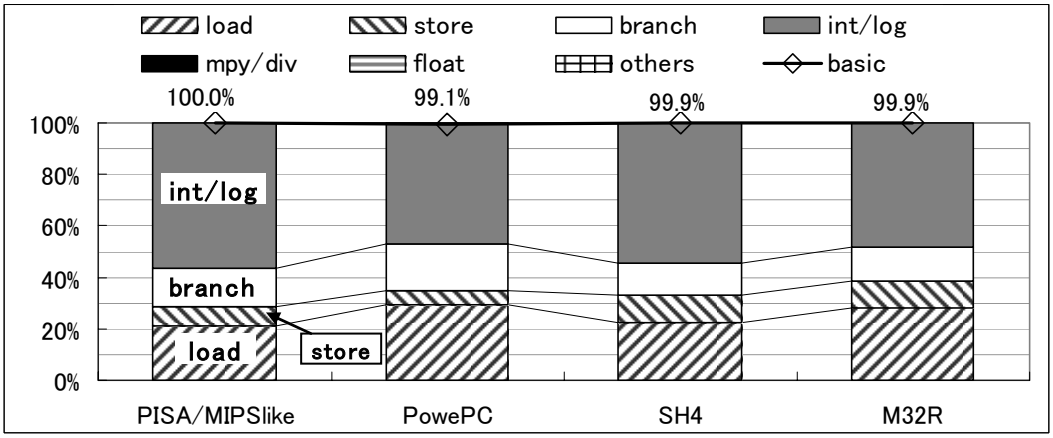


図16 複数 ISA の命令種別毎の頻度（go 30, 11）

2.3.4.3. sim-safe のオーバヘッド

sim-safe の速度性能の解析にあたり，評価プログラムへの依存性が低いことを2.3.4.2で確認できたため，オーバヘッドの分析は命令個々に行うことにした．コンパイラの最適化コードが PowerPC より優れていた Sparc の機械語コードを分析し，各命令ごとにコアループ部と命令固有の処理部分のオーバヘッドを別々に分析した．オーバヘッドは，

オーバヘッド率 ＝（*実行したホスト命令数*―*理想のホスト命令数*）／ *理想のホスト命令数*

の式で表現する．コアループの“*理想のホスト命令数*”は表 7（38ページ）の 4 列目の括弧内に示した値 12 個であり，コアループの“*実行したホスト命令数*”は表 7の 3 列目の括弧内左に示した値 28 個である．これらの値を式に代入して求めた（28 ― 12）／12 ＝ 1.33 がコアループのオーバヘッドとなる．ADDU 命令は理想のホスト命令数の 23 個に対して実際は 43 個掛っている．これの命令固有部では，理想のホスト命令数は 4 列目の括弧内右側の数値 11 個で，実行したホスト命令数は 3 列目の括弧内右側の数値 15 個から，（15―11）／11＝0.36 がオーバヘッド率となる．各レガシー命令を構成しているホスト Sparc の命令をロード（LD），ストア（ST），分岐（B），その他（Others）の“ホスト命令種別”に分類して，オーバヘッド率をグラフにした結果を図 17に示す．このグラフでは，共通処理となるコアループ部と命令固有の処理部を分けて表記している．コアループは 2.33 倍の時間が掛かりオーバヘッド率は 1.33（133%）のオーバヘッドとなり，その 50%

相当分がロード命令（LD）である．ADDU の固有部の合計（36%）の半分もロード命令である．ホスト命令種別でもロード命令が約 20%ほどと大きく，パイプラインを考慮するとクロックサイクルに占める割合はそれ以上に大きいと考えられる．コアループと命令固有部ではコアループの改善が効果的であることが分かる．ロード処理（LD）が多い主な原因は下記の 3 点である．

- 各種アドレス値（間接分岐のアドレス表の開始ポインタ，memoryp，regs のアドレスなど）を必要分ホストのレジスタに持つことができない．
- 値が変化するプログラムカウンタ（reg_PC, reg_NPC），命令数計数（sim_num_insn）の最新値のコピーをレジスタ上に残す最適化をしていない．
- 次の命令処理向けにエミュレータ変数（hi，lo）を先読みするように最適化されているが，実際にはそれらの使用頻度が低いため逆効果となっている．

また other には，ポインタ値を新たにレジスタに設定するときの定数生成の処理もレガシー命令 1 個当たり 2～3 回含まれている．例外判定などを含む命令間でのコードの共通動作をコンパイラが併合し，そのための分岐も余分に発生している．PowerPC では，このようなコードをコンパイラが生成する傾向は更に拡大してレガシー命令処理間の併合処理の分岐が 2 個に増える命令もあり，更に低速化している．なお，ソースコードのわずかな違いにより，このようなオーバヘッドの現れ方が大きく変わる．これらのオーバヘッドを削減する改良が必要となる．

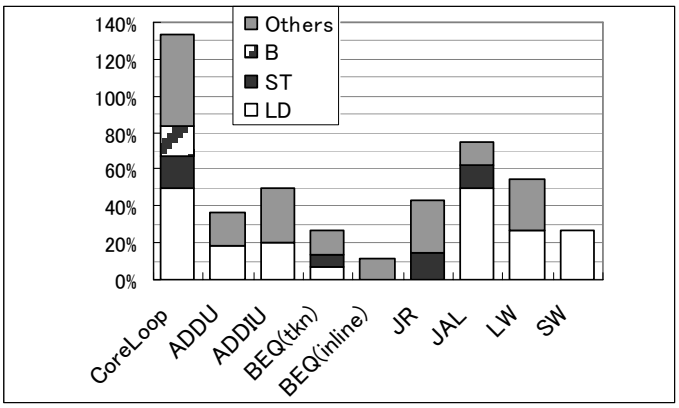


図17 オーバヘッドとなっている命令数の割合

2. 3. 5. sim-MIPSlike の速度性能

レガシーISA の対象を今度は MIPSlike にし，その速度性能を2.3.4と同様に解析し，I-LOOP と SPEC CPU95 の結果を示す．

2.3.5.1. sim-MIPSlike の I-LOOP 性能

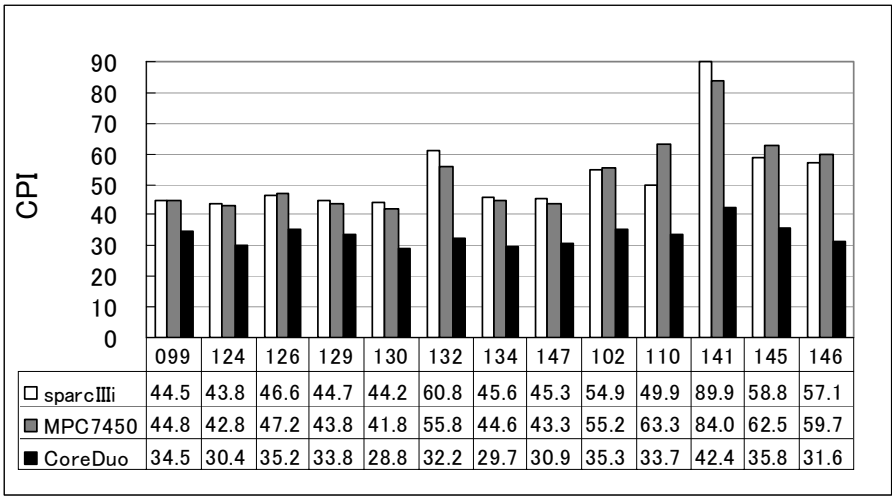
ホスト MPC7450 では I-LOOP 測定の結果は sim-safe に比べて，2 回の命令デコードを行う命令では 13.0 サイクルの増加, 3 回行う命令では 44.4 サイクル増加した. それに対し，sim-safe と同様に 1 回で行う命令では 2 サイクルの削減となった．この削減のうち 1 サイクル分は inst.b の読出しが不要となったためである. SparcIIIi ではこれらは各々, 10.8 増，26.8 増，1.3 減となった．MPC7450 で 3 回デコード時のサイクル数増加が大きいのは，ア

ドレス配置の関係で分岐後の命令キューのフィルが追いつかず命令間の並列実行度が低下していることも要因となっている．

2.3.5.2. sim-MIPSlike の SPEC CINT/CFP95 の速度性能

SPEC CPU95 による CPI 測定結果を図 18に示す．I-LOOP の結果に比べて，SparcIIIi では sim-safe に対する増分が 0.4～12.3（平均 6.0），割合では 1～29%（平均 12%），MPC7450 では 4.1～18.9（平均 9.9）と 11～43%（平均 22%）と低く抑えられている．命令デコードが，1 回で済むもの (opcode)，2 回要るもの (+subop1)，3 回要るもの (+subop2) の割合を図 19に示す．これより 1 回のデコードで済む確率が高いために CPI 増加が小さくなっていることが分かる．

評価プログラム間の CPI 値の差異は，2.3.4.2の図 14（38ページ）で示した sim-safe より拡大され，SparcIIIi で 21%，MPC7450 で 29%となった．これは CFP でのデコード回数の差異が影響しているためである．命令デコードが複雑化したことにより，CPI 値のバラツキが増大してはいるが，似た値を示すことに変わりは無い．評価プログラムごとに異なるデコード回数の比率を考慮することにより，CPI の目安は付けることができる．



	099	124	126	129	130	132	134	147	102	110	141	145	146
□ sparcⅢi	44.5	43.8	46.6	44.7	44.2	60.8	45.6	45.3	54.9	49.9	89.9	58.8	57.1
▩ MPC7450	44.8	42.8	47.2	43.8	41.8	55.8	44.6	43.3	55.2	63.3	84.0	62.5	59.7
■ CoreDuo	34.5	30.4	35.2	33.8	28.8	32.2	29.7	30.9	35.3	33.7	42.4	35.8	31.6

図18 sim-MIPSlike の SPEC CPU95 エミュレーション性能

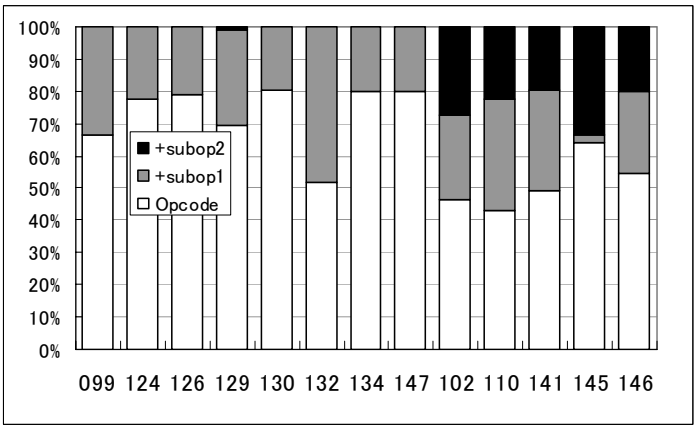


図19 SPEC CPU95 の命令デコードの割合

2.3.6. コアループの時間比率

sim-safe と sim-MIPSlike の両者のコアループの処理時間を評価する．表 8にそれらのコアループが占める処理時間を比率で示す．CPI は SPEC CPU95 のベンチマーク 13 本の算術平均，“Core”はデコード回数ごとの頻度にコアループ時間を乗じたもので 13 本の平均である．“Core%”は Core の処理時間が CPI に占める割合である．コアループの割合(Core%)は，sim-safe で SparcIIIi が 79%，MPC7450 が 77%，sim-MIPSlike では 78%と 73%と高い．sim-MIPSlike ではデコードが複雑になったことによりコアループの割合がより高くなるが，非コア部での例外チェックの CPI も増えているため結果として同様な割合になった．

コアループの時間比率が高いことは，コアループに着目した性能改善が効率的であり，コアループの計測でエミュレータ性能の目安がつくといえる．

表8 コアループの処理時間に占める比率の平均

Host	sim-safe			sim-MIPSlike		
	CPI	Core	Core%	CPI	Core	Core%
SparcIIIi	46.8	36.9	79%	52.8	40.9	78%
MPC7450	43.1	33.3	77%	53.0	38.7	73%

2.4. switch 方式のインタプリタの問題点

sim-safe と sim-MIPSlike は switch 方式によるデコードを使用しており，結果としてオーバヘッドが大きく高い性能が得られないことを示してきた．本節では，インタプリタの速度性能の改善の実施にあたり switch 方式の問題点を挙げる．

switch 方式は，小さなプログラムを対象としたときには記述性と性能面で優れていると考えられるが，大きなプログラムの速度性能をチューニングする観点からは，次の問題がある．

- コンパイラがどのような基準で各 case を最適化しているかが明確でない場合が多い．特に，ソース中の出現頻度の高い変数の最適化や類似コードの併合が行われる．また分岐の記述が増えると最適化の対象単位が小さく限定されるようにみえる．
- 小さなコードで実験し良い最適化結果が得られても，コード量が増大したときに同様な結果が得られる保証がない．また，コンパイラを変えた場合やバージョンアップした後にも同様な結果が得られるかどうか不安がある．
- チューニング時に，ソースコードと機械語コードの対応付けを行う際に，switch 文全体の中から該当箇所を見つけるのが困難である．
- ソースコードのわずかな変更で全体の速度性能が影響を受ける．また，各命令固有の処理が併合されるために，修正用パッチコードの適用を行う際などの保守面に支障がある．
- 逐次多段分岐の記述では switch 文が入れ子になり可読性が大幅に悪化し，記述誤り箇所の特定に時間が掛かる．

2.5. まとめ

本章では，その研究範囲としてアプリケーションレベルのエミュレータをターゲットに，移植性が高く保守が容易な C 言語記述のインタプリタ方式について，性能の予測とチューニングが容易なインタプリタを研究することにした.

2.1.1から2.1.3では命令エミュレーション技術全般について説明を行い, 2.1.4では C 言語記述のインタプリタの要求と課題について述べた．2.1.5では，本研究の着眼点と新規性について述べ，2.1.6ではインタプリタの実装と速度性能を中心とした一般的な課題について紹介した. C 言語記述のインタプリタの速度性能を明らかにしてその課題を客観的に抽出するため，2.2では評価方式を定めた．評価対象のレガシーアーキテクチャは，SimpleScalar の PISA と仕様を MIPS-IV に近付けた MIPSlike, PowerPC, SH4, M32R の 5 種にした．シミュレーション実行するホストには MPC7450 と SparcIIIi を，評価プログラムには I-LOOP と SPEC CPU95 を選んだ.

2.3では，C 言語記述による実装として SimpleScalar の sim-safe の例を示し，MIPSlike 用に記述をした sim-MIPSlike の試作を示した．次に sim-safe と sim-MIPSlike のそれぞれの I-LOOP と SPEC CPU95 の 13 本のプログラムの速度性能の測定結果を示した．それらの結果，次のことが分った.

- 実際のインタプリタの速度性能は，理想的なインタプリタの速度性能の 40～50%程に留まる.
- コアループの処理時間の占める割合は 70～80%程と大きい.
- オーバヘッドの原因を分析した結果コアループに着目した改善が可能である.

また，2.4では，これらの実装で用いている switch 方式の課題として，最適化の結果はコンパイラ任せとなり得られる最適化の効果も不透明なため，そのままではチューニングが困難であることを述べた.

第3章 C言語に適したインタプリタ実装方式の提案

前章の2.3.4では switch 方式である sim-safe の速度性能の調査と分析を，2.3.5では命令コードのエンコードを一般化した sim-MIPSlike の速度性能の調査と分析を行い，2.4でそれらの実装に使用された switch 方式の課題を挙げた．本章では，CPI 短縮を確実にする実装方式として“改良 function 方式”を提案する．3.1では，まず“function 方式の原型”について述べ，3.2で function 方式の原型の速度性能の課題を示し，3.3でその課題を解決した改良 function 方式を説明する．なお，これらの function 方式の評価は第 4 章で行う．

3. 1. function 方式の原型とその試作

function 方式は，レガシー命令それぞれの命令固有の処理に対して専用の“命令処理関数”を設ける一般的な方式である．switch 方式で余分な処理となる opcode の上限チェックのオーバヘッドを除くと，理論的には function 方式は switch 方式を超える性能向上はできず，WEB などで公開されたソースコードでもほとんど用いられた例がない．

しかし，この方式は，グローバルスコープとなる命令処理関数を目印としてアセンブリ言語ソースやアドレスマップを用いた解析が容易であり，switch 方式の課題を解決し，特に，レガシー命令個々に着目した性能改善が容易になる利点がある．

function 方式では，コアループと各命令処理関数のスコープが異なるため，その間の情報授受にグローバル変数を用いる．コアループから命令処理関数に引き渡す情報には，命令語（inst），プログラムカウンタ（regs_PC）が，コアループが受け取る情報には次のプログラムカウンタ値（regs_NPC）がある．まず，この function 方式の原型として PISA 用の emu-PISA-base と MIPSlike 用の emu-MIPSlike-base を試作した．emu-MIPSlike のコード例を次に示す．

emu-MIPSlike-base の宣言文のコード例

```
typedef unsigned int UINT
/*--- レジスタ変数，分岐テーブルの例 ----*/
struct regs_t {
        UINT regs_R[32];
        union {double d[16];float f2};UINT l[32];} regsF;
        UINT hi, lo, fcc, regs_PC;
} regs;
unsigned char *mmp;
UINT expcode;
UINT (*ftbl[64])( ), (*ftbl_cop1[32])( ), (*ftbl_spec[64])( );
```

この例に示した配列 ftbl は opcode による関数分岐，ftbl_spec は subop による関数分岐を，ftbl_cop1 は subop1 による関数分岐用の関数へのジャンプテーブルである．ftbl の各要素にはデコードが 1 回で済む ins_addiu や 2 回目のデコードを行う ins_SPEC や ins_COP1 などがセットされている．ins_SPEC は ftbl_spec を使って更にデコードを行い ins_addu 関数などを呼び出す. 不正命令のときは一旦 ins_opex が呼ばれそこで例外発生フラグを立てる．コアループは，sim-safe と似ているが，ftbl を使った関数呼出し，例外発生を示すフラグ expcode がグローバル変数になっていることが大きく異なっている．

emu-MIPSlike-base の命令処理関数のコード例

```
void ins_addu( ){
        regs.regs_R[(inst>>11)&31] = regs.regs_R[(inst>>21)&31]
                +regs.regs_R[(inst>>16)&31];
}
/*---- デコード専用関数の例 ---*/
void ins_SPEC( ){ ftbl_spec[inst&0x1F]( ); }            // 図 10(1)のコード
void ins_COP1( ){ ftbl_cop1[(inst>>21)&0x1F]( ); }   // 図 10 (4)のコード

/*--- 不正命令処理関数の例 ---*/
void ins_opex( ){ expcode = OPEX_CODE;}
```

emu-MIPSlike-base のコアループのコード例

```
/*--- コアループの構成例 ---*/
void coreloop( ) {
  sim_num_insn=0;expcode=0;
  for(;;){
        inst=*((UINT *)(mmp+regs.regs_PC));
        sim_num_inst++;
        regs.regs_R[0] = 0;
        (*(ftbl[inst>>26]))();
        if(expcode != 0)
                goto exp;
        reg.regs_PC = regs.regs_NPC;
        regs.regs_NPC += 8;
   }
exp: …
        return;
}
```

3. 2. function 方式の原型の改良指針

emu-PISA-base を実装しその解析を行った結果, function 方式の原型でのいくつかの問題点が明らかになった．これらの問題点とその解決指針を次に示す.

1） 各命令処理関数がレジスタを多く使用する場合には関数の出入口で行うセーブとリストア処理が発生してオーバヘッドとなる．この問題を解決するためには，中間変数が少なくなるように処理を簡潔に記述する必要がある.

2） 各命令処理関数が更に関数を呼び，呼び出された関数もセーブ／リストアを行い処理が遅くなる場合がある．したがって，実行頻度の高い命令処理関数ではコンパイラが，呼び出す関数を命令処理関数内に展開できるようにする.

3） 各命令処理関数がグローバル変数にアクセスする際に，間接アクセスするようにコード生成される．これはコンパイル時に決定されないデータセグメントのグローバル変数のアドレスを解決するためである．このため，“変数のアドレスをコード領域に置いたポインタ定数として，プログラムカウンタ相対番地から読む手順”を踏み，処理が遅くなる．これを防ぐためには．グローバル変数を構造体として固めてこの手順の実行回数を削減する工夫が有効である.

4） 命令処理関数からコアループに引き渡す情報にグローバル変数を使用すると，キャッシュメモリへの書込みとコアループでの読出しが接近しパイプラインス

トールを引き起こす．これを回避するためには，グローバル変数を極力利用せず，戻り値を使用する必要がある．

これらを考慮した，改良 function 方式を次節以降で述べる．なお，emu-PISA-base の速度性能値は参考として第 4 章に示し，ここでの解説は省略する．

3. 3. 改良 function 方式の提案

function 方式の原型の問題点を解決した改良 function 方式のエミュレータ emu-MIPSlike-opt のソースコード例を示す．次に，改良のポイントとしてデコード，関数への引数の渡し方，戻り値の扱いを説明する．なおコード中で“x_”が頭につく変数は，ローカル変数でレジスタ割付けが行われることを期待している．

emu-MIPSlike-opt の宣言文のコード例

```
typedef unsigned int UINT
/*--- レジスタ変数，分岐テーブルの例 ----*/
struct regs_t {
    UINT regs_R[32];                                    // (a)
    union {double d[16];float f2};UINT l[32];} regs_F;
    UINT hi, lo, fcc, regs_PC;                          // (b)
    unsigned char *memp;                                // (c)
    UINT expcode;
    UINT (*ftbl[522])(UINT, struct regs_t *);           // (d)
} regs;
typedef struct regs_t REGS;
```

emu-MIPSlike-opt の命令処理関数のコード例

```
UINT ins_addu(UINT ic, REGS *rp){
    UINT x_npc = rp->regs_PC+8;                         // (e)
    rp->regs_R[(ic>>11)&31] = rp->regs_R[(ic>>21)&31]
           +rp->regs_R[(ic>>16)&31];
    return x_npc;                                       // (e)
}
/*---- デコード専用関数の例 ---*/
UINT ins_SPEC(UINT ic, REGS *rp){    /*図 10(1)のコード*/  // (f)
    return rp->ftbl[(ic&0x1F)+64](ic,rp); }
UINT ins_COP1(UINT ic, REGS *rp){    /* 図 10 (4)のコード*/ // (g)
    return rp->ftbl[((ic>>21)&0x1F)+224](ic,rp);}
/*--- 不正命令処理関数の例 ---*/
UINT ins_opex(UINT ic, REGS *rp){
    return (rp->expcode = OPEX_CODE) | 1;}              // (h)
```

emu-MIPSlike-opt のコアループのコード例

```
/*--- コアループの構成例 ---*/
void coreloop() {
 UINT x_pc, x_ic, x_icnt; x_pc=regs.regs_PC; x_icnt=0;
 UINT(**x_fp)(UINT,REGS*); x_fp= x_rp->ftbl;              // (i)
 unsigned char *x_memp = mmp;
 for(;;){
    x_ic=*((UINT *)(x_memp+x_pc));                        // (j)
    regs.regs_R[0] = 0;
    x_pc=(*(x_fp+(x_ic>>26)))(x_ic,&regs);                // (k)
    regs.regs_PC = x_pc;                                  // (l)
    x_icnt++;
    if((x_pc & 3) == 0) continue;
    else goto exp;                                        // (m)
  }
exp: …
    instrucntion_count = x_icnt; return;
}
```

このコードとその中に示した注釈（a～m）を用いてこれより改良 function 方式の説明を行う．

3.3.1. 改良 function 方式のデコード

コアループでは，逐次多段分岐を用いて opcode による分岐を行う（k）．2 回以上のデコード用には命令処理関数と同様にデコード用関数を用意する（f, g）．

3.3.2. 改良 function 方式の引数

改良 function 方式では，関数に渡す引数を下記のようにする．

- function 方式における命令デコードに次ぐボトルネックは，命令語を取得し，フィールドデコードをしてレジスタ番号を計算し，ホストメモリ上の汎用レジスタ（regs_R）を読み出すパスである．このパスを短縮するため，命令語（x_ic）を引数として渡す（j，k）．
- 次に汎用レジスタのアドレス解決を不要にするため引数として&regs を渡す（k）．このときに構造体 regs のメンバの先頭を sim-safe と同様に汎用レジスタとしておけばアドレス計算に余分な命令サイクルが掛からない（a）．
- LW 命令などレガシーのメモリ空間にアクセスする命令は，変数 mmp を引数としてレジスタ渡しをした方がよいように見える．しかし，ホストによっては命令処理関数がセーブ／リストアなしに使用できるレジスタ数を越してしまい逆効果となる場合がある（Sparc，PowerPC ともに該当）．そこで，mmp のコピー（memp）を構造体 regs_t のメンバに加えておく（c）．この場合，その読出しサイクルはレガシーレジスタ番号のデコードやレガシーレジスタ値の読込みサイクルと並行動作が可能になる．
- プログラムカウンタを参照する命令の割合は，SPEC CPU95 のベンチマーク 13 本では最大 21%，平均 11%であり，引数とせずに構造体メンバのままで構わない（b）．
- 逐次多段分岐の場合には，分岐テーブルのポインタ値の読出しがボトルネックになる．この解決のため，メモリのポインタ同様に構造体 regs_t のメンバとすることを考える．しかし，ポインタ値の読出しとインデックス計算を並行して行うことが可能な場合，インデックス計算は 1 または 2 サイクルで完了してしまい，やはりポインタ値の読出

し待ちになってしまう．そこで，構造体 regs_t の最後の部分にメンバとして分岐テーブル全体を入れておけばポインタ値の読出しが不要となり高速になる（d）．

- コアループからの参照用には分岐テーブルのポインタとしてローカル変数（x_fp）を用意する（i）．

3.3.3. 改良 function 方式の戻り値

命令処理関数からコアループへ引き渡す情報のオーバヘッドを削減するため，次に示す工夫をする．

- プログラムカウンタは命令処理関数で更新して関数の戻り値とし（e），コアループでローカル変数（x_pc）として使用しつつグローバル変数に格納する（k, l）．
- 例外発生を示すフラグも戻り値（x_pc）の冗長ビット（下位ビット）に埋め込み（h, k），命令境界違反と併せてレジスタのまま判定ができるようにする（m）．
- 不正命令の検出をコアループで判断すると遅くなるため，命令処理関数として用意し（h），例外検出を一箇所にまとめる．

なお，命令処理関数内でプログラムカウンタの更新とリタンを別けて記述しているのは，エミュレータ変数の読出しオーバヘッドを隠蔽するスケジューリングがコンパイラの最適化のみでは不十分なためである．

3.4. まとめ

本章では，switch 方式の課題を解決しチューニングにより性能向上を容易にさせるため，命令種ごとに命令処理関数を設ける function 方式を原理とした方式を提案した．まず，3.1 で“function 方式の原型”を示し，次に3.2でその速度性能の課題を述べ更にそれらを解決する“改良 function 方式”への指針を示した．3.3では改良 function 方式の実装例を emu-MIPSlike-opt として示し，特に効果の大きいデコード，関数への引数の受け渡し，関数からの戻り値など実装のポイントを述べて，処理が高速でチューニングが容易な方式として提案を行った．

第4章 改良 function 方式によるエミュレータの速度性能の評価

本章では，3.3で示した改良 function 方式の速度性能の評価結果を述べる．なお，function 方式の原型についても参考として併せて提示する．

まず，4.1で emu-PISA，4.2で emu-MIPSlike の速度性能の評価結果を述べ，4.3で簡単にまとめる．次に4.4で改良後のコアループの時間を述べる．参考としてホストを x86 にしたときの速度性能に関して4.5で結果と課題を示す．次に，これら4.1と4.2の傾向が他のレガシーISA でも同様であることを4.6で示す．最後に，今まで敢えて触れなかったそのほかの要素について性能との関係を論じる．

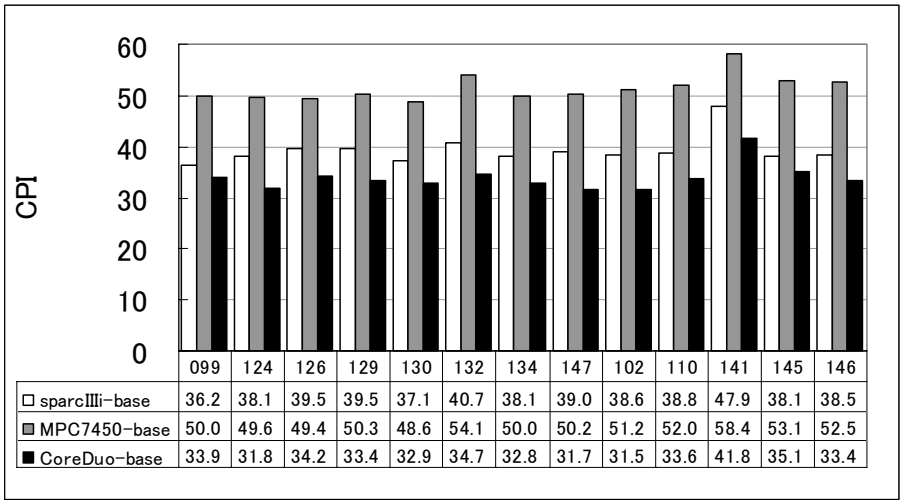
4. 1. エミュレータ emu-PISA の速度性能の評価

PISA をレガシーISA として，function 方式の原型 （emu-PISA-base）と改良 function 方式（emu-PISA-opt）のそれぞれを試作した．ホストは SparcIIIi，MPC7450，x86（Core Duo[†17]）である．それらの CPI 性能を図 20 (52ページ) に示す．図の上のグラフは function 方式の原型（*-base）を，下のグラフは改良 function 方式（*-opt）を示している．

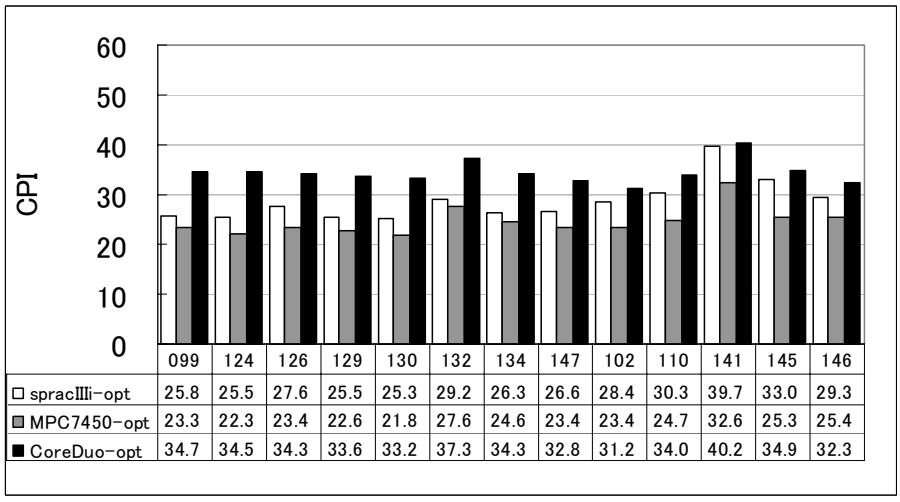
このグラフより，SparcIIIi と MPC7450 では原型に比べて改良 function 方式の CPI が大幅に短縮され速度性能が向上していることが分かる．SparcIIIi で 65～87%（13 本の平均は 73%），MPC7450 では 45～56%（平均 48%）に短縮できている．sim-safe の CPI に対しては SparcIIIi では 53～70%（平均 61%），MPC7450 では 50～61%（平均 57%）に短縮できている．

ここで，データの意味を解説をしておく．原型では，SparcIIIi よりもパイプライン段数が短い MPC7450 の方が，CPI は逆に大きく性能が悪くなっている．これは，グローバル変数のアドレス解決のためのオーバヘッド，更にそのためにレジスタのセーブとリストアが発生しているためであり，PowerPC 用 gcc コンパイラの特性である．一方，改良 function 方式では原型で生じた課題を3.3で述べたように改善しているため，CPI も大幅に削減されている．また，原型と改良 function 方式のいずれにおいても 132.ijpeg と 141.aspi の CPI 値が他の 11 本より大きい．これは乗算を繰返し加算で行っているため，乗数のビットが 0 か 1 かによる分岐予測ミスが大きいことが原因である．

[†17]Core 2 Duo ではなくそれより性能の低い Core Duo である．この評価時には Core 2 Duo は入手できていない．第 7 章と第 8 章の評価はその後，デスクトップ型の Core 2 Duo 搭載の PC を入手して行った．



	099	124	126	129	130	132	134	147	102	110	141	145	146
□ sparcIIIi-base	36.2	38.1	39.5	39.5	37.1	40.7	38.1	39.0	38.6	38.8	47.9	38.1	38.5
▒ MPC7450-base	50.0	49.6	49.4	50.3	48.6	54.1	50.0	50.2	51.2	52.0	58.4	53.1	52.5
■ CoreDuo-base	33.9	31.8	34.2	33.4	32.9	34.7	32.8	31.7	31.5	33.6	41.8	35.1	33.4



	099	124	126	129	130	132	134	147	102	110	141	145	146
□ spracIIIi-opt	25.8	25.5	27.6	25.5	25.3	29.2	26.3	26.6	28.4	30.3	39.7	33.0	29.3
▒ MPC7450-opt	23.3	22.3	23.4	22.6	21.8	27.6	24.6	23.4	23.4	24.7	32.6	25.3	25.4
■ CoreDuo-opt	34.7	34.5	34.3	33.6	33.2	37.3	34.3	32.8	31.2	34.0	40.2	34.9	32.3

図20 emu-PISA の CPI

次に，emu-pisa-base と emu-pisa-opt を使って I-LOOP 実行したときの CPI 値を，アセンブリ言語記述を使って最適化した理想コードの実行結果の CPI 値と比較した結果を表 9 に示す．これらのホストは MPC7450 である．ideal は理想コードを示す．括弧内の値は理想コードの CPI をそれぞれの CPI で割ったもので, 速度性能の達成度を示す. 改良 function 方式ではアセンブリ言語記述には及ばないが，性能が向上し理想的なアセンブリ言語記述の 80%以上の速度性能を実現できていることが分かる．

表9 ホストが MPC7450 の emu-PISA の CPI 値

Instruction	ideal	base(ideal/base)	opt(ideal/opt)
NOP	15	28.7 (52%)	18.2 (82%)
ADDU	17	47.4 (36%)	20.4 (84%)
LW	19	49.8 (38%)	21.2 (89%)
BEQ(taken)	18	51.4 (35%)	22.2 (81%)

以上のように，改良 function 方式では function 方式の原型の CPI 値を大幅に短縮し，switch 方式である sim-safe の CPI 値をこれも大幅に改善して, 理想コードに近い速度性能を実現できた．

4. 2. エミュレータ emu-MIPSlike の速度性能の評価

レガシーISAのMIPSlikeにfunction方式を適用したエミュレータemu-MIPSlikeのCPI性能を図21に示す. 上のグラフはfunction方式の原型で, 下のグラフは改良function方式を示す. 改良function方式を適用したemu-MIPSlikeのCPIはsim-MIPSlikeのCPI値に対し, SparcIIIiでは56～85%（13本の平均68%), MPC7450では48～61%（平均57%）に短縮できている. 表10にMPC7450の理想コードのCPIとの比較を示すが, MIPSlikeでも同様に理想コードに近づいていることが分かる. LW命令はアドレス境界チェックが必要ため, 表9（52ページ）に示したemu-PISAより2サイクル増えている. 理想コードでは投機実行によりその影響を隠蔽できているのに対し, コンパイラではポインタmempを読み出す命令のスケジュリーングが不十分なため遅延を生じているが, 現状のコンパイラの限界と考える.

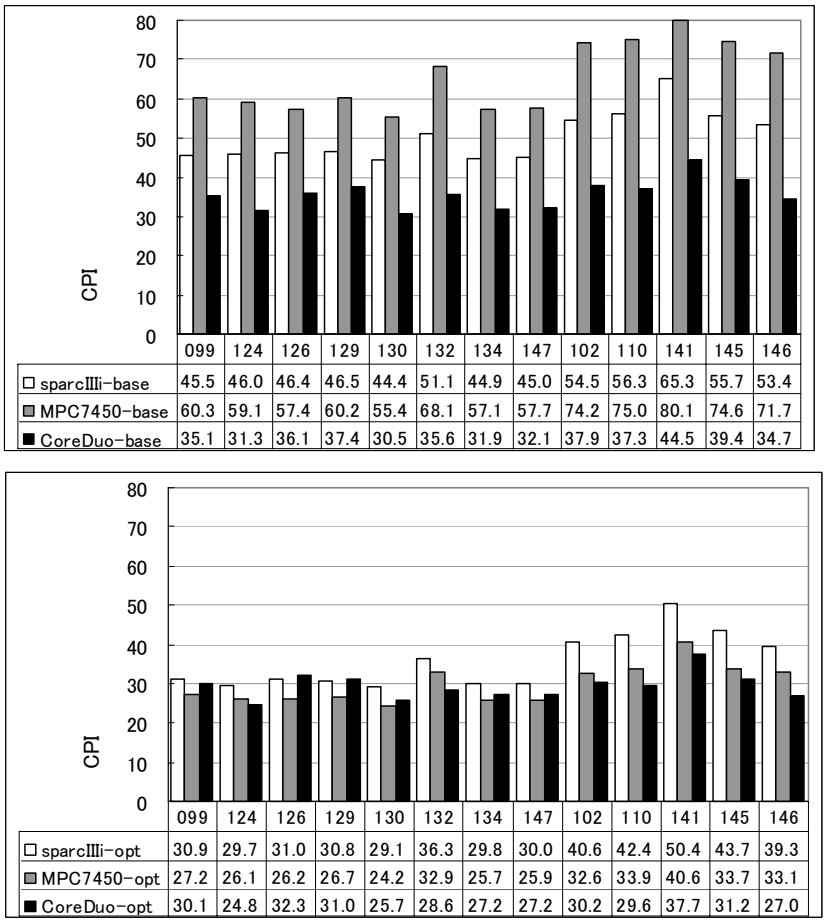


図21 emu-MIPSlikeのCPI

表10 ホストがMPC7450のemu-MIPSlikeのCPI値

Instruction	ideal	base (ideal/base)	opt (ideal/opt)
NOP	15	29.7 (51%)	18.2 (82%)
ADDU	27	74.6 (36%)	31.6 (85%)
LW	19	53.7 (35%)	23.5 (81%)
BEQ(taken)	18	51.8 (35%)	22.3 (81%)

これらより，一般のマイクロプロセッサと同様に複雑なデコードを必要とする MIPSlike では，switch 方式に対する改良 function 方式の優位性は更に拡大し，理想コードに一層近い手法であるといえる.

4. 3. function 方式による性能向上のまとめ

前節までは速度性能の向上を CPI 値とその短縮率を使って表現していたが，本節では function 方式の switch 方式に対する優位性を相対性能値として，まとめて示す.

SPEC CPU95 の 13 本の CPI を算術平均値より求めた switch 方式の sim-safe, sim-MIPSlike をそれぞれ 100%としたときの相対性能を図 22に示す.これより，function 方式の原型では，ホストのコンパイラに依存し switch 方式より性能が向上するもの(SparcIIIi）と劣化するもの（MPC7450）に別れる．しかし，改良 function 方式ではホストによらず switch 方式の 148～177%の速度性能に向上でき，RISC ホストでの function 方式の有効性が確認できた.

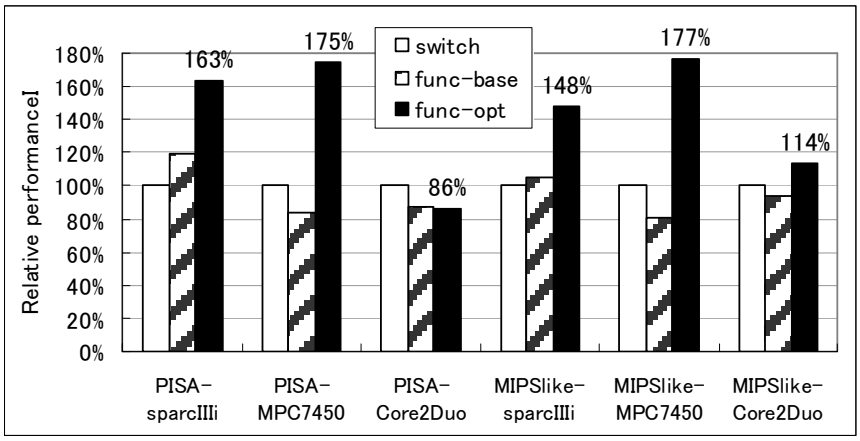


図22 function 方式による性能向上

4. 4. コアループの時間比率

function 方式によるコアループの占有時間について述べる．emu-PISA-opt と emu-MIPSlike-opt の SPEC CPU95 のコアループの時間比率を表 8（42ページ）と同じ計算方法と形式を用いて表 11に示す．すなわち，レガシー命令 1 個の実行に掛かるホストサイクル数 (CPI) のうちコアループのサイクル数 (Core) が占める割合を “Core%” とする．評価プログラム 13 本に対し，emu-MIPSlike の Core%は Sparc で 61～83%，MPC7450 で 65～85%の範囲を示した. MPC7450 では PISA と MIPSlike の Core の差は小さいのに，SparcIIIi ではその差が拡大している.これは 2 段階目以降のデコードでの分岐パイプラインストールが長いことと，シフトとマスクが合計 2 命令掛かることによる．switch 方式から function 方式に改良されたことによる Core%の変化は，MPC7450 ではほとんど無い．一方，SparcIIIi では Core の削減値が，PISA で 16.4，MIPSlike で 14.0 に対し Core 以外の削減値が PISA で 1.9 と MIPSlike で 1.9 であり, Core の削減効果が大きかったため Core%

が逆に減った.

表11 コアループの処理時間に占める比率平均

Host	emu-PISA-opt			emu-MIPSlike-opt		
	CPI	Core	Core%	CPI	Core	Core%
SparcIIIi	28.6	20.5	72%	35.7	26.9	75%
MPC7450	24.6	19.4	79%	29.9	23.5	79%

２つのレガシーISA と２つのホストの計 4 個のインタプリタのいずれも，function 方式を使って実装したことにより理想状態に近いインタプリタ性能の向上が実現できた．また，コアループの占める割合の平均も 72~79%と高いといえる.

4.5. x86 をホストにしたときの速度性能向上と課題

x86 をホストとして使用したときの速度性能を示す.各エミュレータの CPI 値は図 14(38ページ), 図 18 (41ページ), 図 20 (52ページ), 図 21 (53ページ) に既に示してある. switch 方式に対する function 方式の相対性能は，図 22（54ページ）に示すように，ホストが x86 では，function 方式の速度性能の向上効果が薄らいでいる.

インテル社の x86 ではあらゆるアプリケーションの速度性能を向上させるべく，NetBurst アーキテクチャ以降の実行トレースキャッシュ，ループ検出，間接分岐予測，プログラムのアクセスパタンによるプリフェッチ機能などの各種の高速化機能が搭載されてきた．そのため，特定のアドレスへのアクセスが集中するインタプリタのようなプログラムでは，アドレス配置の影響を受けてそのオーバヘッドによりプログラムの改善効果が隠れてしまうことがよくあり，試行錯誤を含むチューニングが困難であることが課題である.

また，チューニングにあたり命令セット上の課題もある．それは汎用レジスタが 8 本しかなく，しかも C 言語でセーブ／リストアなしに使用できるレジスタは 3 本しかない．そのため，一般的に関数内では変数が 3 個の時，関数呼出し側では 6 個以内のときは高速である．それに対し，その数を超えるとメモリ演算か push/pop 命令によるレジスタのセーブ／リストアが必要になる．インタプリタ実装では，単純な動作ですむ sim-safe のように扱う変数が少ない場合は switch 方式が効率よく動作し，多くなると function 方式が若干有利になる．x86 向けの func-opt では，命令処理関数へ渡す引数を命令語のみに限定しレジスタ渡しにしているが，加算命令を処理する命令処理関数では，レジスタが 3 本不足するため，7 命令余分に掛かっている．また，複数回の命令デコードを行う命令では，通常，コアループよりデコード専用関数経由で命令処理関数を呼び，命令処理関数からは直接コアループへ戻るようなコードが生成されることを期待している．しかし，x86 では一旦デコード処理関数に戻りレジスタをリストアするため, function 方式も効果的には動作できない.

これらの結果，図 22にて PISA では switch 方式，func-base，func-opt の順に性能が良く，MIPSlike では func-opt は switch 方式の 9～23% (平均 14%) の速度性能向上に留まっている.周波数あたりの速度性能が優れている Core Duo のような最新の x86 プロセッサでも，このようにレジスタ本数が制約となり，インタプリタの速度性能の向上が困難となっている.

4. 6. そのほかのレガシーISA での評価

次に，改良 function 方式の優位性を，他のレガシーISA 向けのエミュレータを試作して評価した．

レガシーISA には，PowerPC，SH4，M32R を，ホストには今までと同様に SparcIIIi と MPC7450 を用いた．評価プログラムにはシステムコールの実装が軽くて済む 099.go を用いている．コンパイラと最適化オプションは，PowerPC は gcc3.1（-03），SH4 は gcc3.3（-O2），M32R は gcc2.9（-O2）である．パラメータは“30 11”とし碁の対局が 114 手で終了するようにした．このときのレガシー命令数は PISA と MIPSlike が 1.74×10^9 個，PowerPC が 1.54×10^9 個，SH4 が 2.08×10^9 個，M32R が 2.07×10^9 個である．CPI の測定結果を表 12（57ページ）に示す．括弧内の数値は switch 方式を 100%とした相対速度性能である．PISA と MIPSlike の速度性能値が，図 14（38ページ），図 18（41ページ），図 20（52ページ），図 21（53ページ）とは微妙に異なっているが，それは評価プログラムの設定パラメータの違いによるものである．

PISA は命令形式と命令仕様が単純であり3.3で示した改良で充分であったが, 市販の ISA では更に個別の改良を加える必要がある．PowerPC では，LK/Rc フィールドをデコード条件に加えて異なる命令処理関数として実装して，命令処理関数内での分岐判定をなくし，中間変数を減らしてレジスタ割付けを改善した．また BO フィールドの値により変化する分岐命令の機能の判定順番を頻度順に変更した．M32R では，上位 12 ビットによる一括冗長分岐を使用した. SH4 では 16 ビットの命令語を 0xFF0F で AND した結果に対しての一括冗長分岐を行い，ビット 8-11 のデコードは逐次分岐を組合せて実現した．これらの改良を行った結果を表 12中に func-extra として示している．

PowerPC の CPI 値が大きい理由は，MIPSlike に比べて命令仕様が複雑であり，lmw/stmw などの強力な命令がある，CISC 同様にメモリアドレス制約が無くまた分岐命令が複雑であること，インデックスレジスタ番号がゼロのときの特殊処理などによる．M32R と SH4 では 16 ビットの命令語のため, レジスタフィールドの数は少なくオペランドデコードの回数も少ないが，opcode が 4 ビットしかないため 1 回の命令デコードで済む確率が低くなり func-opt の改善効果が薄まっている. また M32R では命令語長の判定が必要なこと，SH4 では遅延分岐命令と非遅延分岐命令が混在し更に遅延分岐制約による“スロット不当命令例外”検出が必要なこと，がオーバヘッド増加の要因になっている．

次に今まで CPI で表現してきた function 方式の速度性能の向上効果を，switch 方式を 100%とする相対性能として PISA と MIPSlike も含めて図 23(57ページ)にグラフで示す．各項目はレガシーISA とホストの組合せを示している．数値の詳細は表 12（57ページ）にも括弧書きで示している. ここで示すように SparcIIIi と MPC7450 では func-opt により，switch 方式の 1.17～1.82 倍の速度性能を実現した．更にその改良後に顕著になったオーバヘッドを func-extra として改善し，1.33～2.25 倍までの速度性能の向上ができた．

ホスト Core Duo では PISA を除き性能は向上し，1.08～1.26 倍となった．PISA で性能低下したのは，switch 方式は簡単なインタプリタでは性能が出やすい特性があり，PISA はそれに該当し，更にレジスタ本数の制約に対し PISA は x86 命令セットで使用できるレジ

スタ本数 6 本に収まりやすいことによる．図 24 (58ページ) に，各ホストのエミュレーションによるスローダウンを示す．このスローダウンは，go（30,11）のエミュレーション時間をホストのネイティブ実行時間で割った値であり，値が大きいほどインタプリタの速度性能が低いことを意味する．SparcIIIi が MPC7450 よりスローダウンが大きいのはクロック周波数とパイプライン段数の違いにより説明できる．ネイティブ動作の時間は Core Duo が 1.46 秒，SparcIIIi が 1.47 秒とほぼ同一時間であるが，クロック周波数が低い Core Duo は SparcIIIi よりスローダウンが 9～37%大きく，インタプリタ性能の向上が小さい．

以上のように，改良 function 方式を使用することにより実質上レガシー命令セットによらずに，switch 方式の速度性能を改善でき，レジスタ本数の多い RISC ホストではその効果が得られることを示した．

表12 他のレガシーISA の CPI と相対性能

Host	Legacy	switch	func-base	func-opt	func-extra
SparcIIIi (Sparc)	PISA	41.2	36.8 (112%)	26.1 (158%)	--
	MIPSlike	44.5	45.6 (98%)	31.1 (143%)	--
	PowerPC	54.8	54.3 (101%)	43.8 (130%)	38.8 (141%)
	SH4	49.0	55.1 (89%)	42.0 (117%)	369 (133%)
	M32R	48.9	57.6 (89%)	44.7 (115%)	30.4 (169%)
MPC7450 (mpc)	PISA	41.7	49.5 (84%)	22.9 (182%)	--
	MIPSlike	45.8	59.8 (77%)	26.8 (171%)	--
	PowerPC	61.2	69.9 (88%)	38.3 (160%)	33.9 (180%)
	SH4	52.4	73.0 (72%)	36.3 (144%)	31.7 (165%)
	M32R	56.4	80.0 (70%)	37.3 (151%)	25.1 (225%)
Core Duo (x86)	PISA	30.5	31.7 (96%)	36.0 (85%)	--
	MIPSlike	35.1	34.3 (102%)	32.6 (108%)	--
	PowerPC	40.8	43.7 (93%)	38.9 (105%)	36.8 (111%)
	SH4	35.7	39.1 (91%)	33.3 (107%)	33.0 (108%)
	M32R	37.2	38.3 (97%)	34.2 (109%)	29.4 (126%)

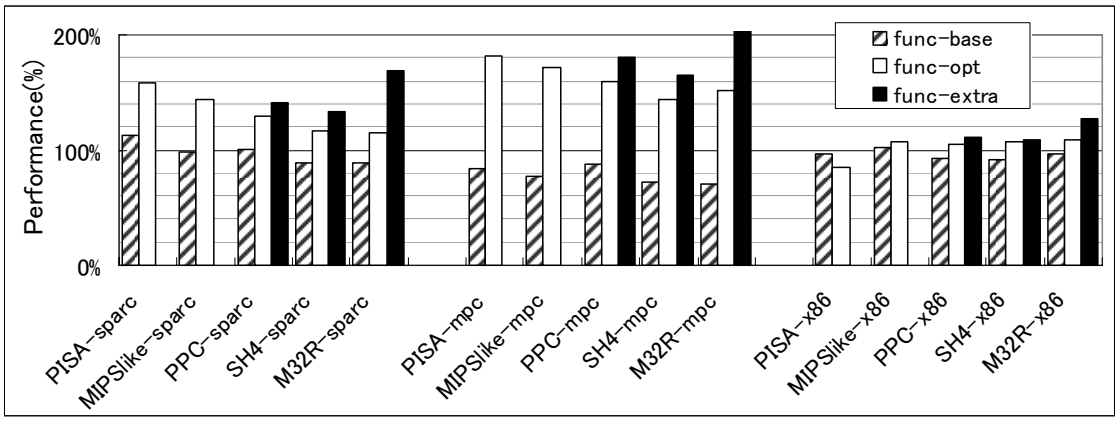
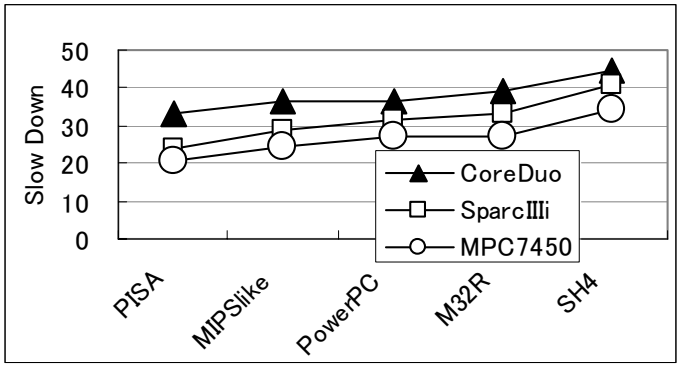


図23 全方式と全レガシーISA の速度性能比較



（値は小さいほど速度性能が高い）

図24 ホスト別のスローダウン

4.7. 性能に関するそのほかの要素に関する議論

これより前の章節では，コアループ処理，その中でも逐次多段分岐を中心に性能改善の評価を行ってきた．それは，そのほかの要素の影響が小さいからである．結論としては，CISC の条件コード生成のオーバヘッドは比較的小さく，またそれ以外に取り上げた項目の影響度は極めて小さく無視可能である．本節ではそれを確認する意味で敢えて取り上げる．

まず，4.7.1で一括分岐方式，4.7.2で分岐のオーバヘッド，4.7.3でキャッシュミス，4.7.4でエンディアン，4.7.5で CISC レガシーへの適用，4.7.6で OS 性能への影響について論じる．

4.7.1. 一括分岐方式の有効性

今までの議論では定量的に触れなかった一括分岐方式の速度性能に関して述べる．

一括分岐方式では，opcode の値により subop フィールドの有無判定とフィールド位置を選択する必要があるが，その選択に分岐命令を用いると，逐次多段分岐に性能面で劣ってしまう．これは，実際のプログラムでは2.3.5.2の図 19（41ページ）に命令のデコード回数を示したように，逐次多段分岐では 1 回の間接分岐で済んでしまう確率が高いこと，一括分岐方式ではその判定・選択処理で投機実行の分岐予測がはずれる確率が高いことによる．レガシーISA が PowerPC，ホストに 2 ビットの分岐予測機構を持つ MPC7450，評価プログラム 099.go を用いて，分岐命令と予測ミス回数をシミュレーションした．その結果，1 回デコードの命令（実行頻度 65.6%）と 2 回デコードの命令（実行頻度 34.4%）を判定する処理で発生する分岐予測ミス率は 42.6%と非常に大きく投機実行ミスのペナルティが効いてくることを確認した．

また，分岐を使わず，opcode 値より表引きにより関数ポインタ位置の計算を，

```
((opcode >> optbl[opcode].shift)&optbl[opcode].mask)
+ optbl[opcode].offset
```

のようにシフト値，マスク値とオフセット値を使う方法がある．この方法では表の構造体メンバをアクセスする際のキャッシュメモリアクセス時間が増加し，更にレジスタ本数が不足しやすくなるため，分岐を使った方法より実行が遅くなる．別の改善例としては，シ

フト値とマスク値とオフセット値を 32 ビットの 1 つのデータにまとめてキャッシュメモリ参照を 3 回から 1 回に減らし opcode と subop1 までの 2 フィールドまで一括分岐する方法がある．このように改良された一括分岐方式では，2 回以上デコードを行う確率が高い，分岐パイプラインが深い，ホストの汎用レジスタ本数が多い，という条件が重なると逐次多段分岐方式より一括分岐方式が有利となりうる．func-opt にこの改善を施した一括分岐方式を適用して 099.go （30，11）実行した．その各レガシーISA の CPI 値の結果を表 13に示す．括弧内は func-opt を基準とした CPI 増加値である．レガシーISA が MIPSlike や PowerPC のように opcode が 6 ビットのものでは CPI が増加し効果がない．一方 SH4 や M32R では opcode が 4 ビットと短く 2 回以上のデコードがそれぞれ 79.4%，86.5%と大きいため CPI が減り効果が出ている．MPC7450 よりパイプライン段数が深く使用可能レジスタ本数が多い SparcIIIi では，opcode が 6 ビットでは CPI 増はより少なく，4 ビットでの CPI 削減効果はより大きくなっている．SparcIIIi では，一括分岐方式は逐次多段分岐方式に比べて SH4 は 6.8%，M32R が 10.6%高速になっている．

表13 一括分岐方式の CPI

Host	MIPSlike	PowerPC	SH4	M32R
SparcIIIi	33.1 (+2.1)	46.2 (+3.9)	39.4 (-2.7)	40.4 (-4.3)
MPC7450	33.1 (+6.3)	43.5 (+5.2)	35.6 (-0.7)	36.2 (-1.1)

このように一括分岐方式は，定性的には有効な領域が存在するが，レガシーの命令語長が 32 ビットでは逐次多段分岐に劣り，16 ビットでは4.6で述べたように一括冗長分岐が可能となるため，性能面での効果は見られない．

4.7.2. 分岐のオーバヘッド

インタプリタ方式のエミュレータでは，分岐予測ミスが多くオーバヘッドになるといわれている．次に，分岐予測ミスについて検討する．

例外条件検出処理においては，通常，分岐予測ミスが発生することはないが，データ値の判定やレガシー命令の条件付分岐命令の処理では予測ミスが発生する．図 9（2.2.1.2）に示す IMM（符号付定数）や TARGET（分岐オフセット）の生成において，命令語のビット 48 により判定し負ならば 0xFFFF0000 を OR する実装では値が正のときの速度性能は向上するが，分岐予測ミスによるオーバヘッドが増大し平均性能は低下してしまう．このような処理は，算術シフトと cast の記述により符号拡張演算を行うことで回避でき，sim-safe でも実施されている．

一方，sim-safe のように 64 ビットの結果を得る 32 ビット乗算を，シフト，判定，加算を乗数ビット分繰り返す実装法を採ると，乗数のビット値により分岐予測ミスが発生する．この場合は long long 宣言を用い倍精度乗算に相当するコードを生成する方法や，16 ビット乗算 4 回と加算による方法などが有効となり，これによる分岐オーバヘッドは削減可能である．

インタプリタの実装において原理的に削減できない分岐予測ミスの要因には，レガシーISA の条件付分岐命令がある．PISA では SPEC CPU95（13 種）全命令中 1～18%が条件付分岐命令の可能性がある分岐命令であり，各々の分岐確率を 50%とし平均をとると，レ

ガシー命令あたり 1 個あたり 4.5%の分岐予測ミスが発生することになる．CPI 換算では，SparcIIIi では 0.27，MPC7450 では 0.15 のオーバヘッドとなる．func-opt の場合の CPI 値は SparcIIIi が 28.6，MPC7450 が 24.6 であり，それらへの影響は 0.9%と 0.6%と無視できる範囲である．

4.7.3. キャッシュミスの影響

インタプリタ方式のエミュレータでは，命令キャッシュは通常ヒットする．データキャッシュでは，エミュレータ変数はヒットするが，レガシーのデータとともにレガシーの命令もデータキャッシュを使用するためそのミス率は増加する．実在するレガシーISA の PowerPC（MPC7450）の実行例として 099.go（5, 6）のキャッシュシミュレーションした結果を表 14に示す. MPC7450 でのネイティブ動作ではデータ一次キャッシュのミスが 165×10^3 回であり，エミュレーション動作では 5.4 倍の 888×10^3 回のミスに増大した．しかし，エミュレーションにより命令数が 68 倍に増大したのにキャッシュミスは 5.4 倍に留まったためその影響は大幅に薄まっている．この影響を CPI 換算にすると 0.2 以下で CPI 値 33.9 の 0.5%以下となり，これも無視可能な範囲である．

表14 MPC7450 ネイティブとエミュレーション動作のキャッシュミス回数

	命令数	命令１次キャッシュミス回数	データ１次キャッシュミス回数
Native(N)	25,331,926	307,163	165,420
Emulation(E)	1,721,410,099	308	888,293
E/N	68.0(IPI)	0.0010	5.37

4.7.4. レガシーとホストのエンディアンの差異

レガシーISA とホストのエンディアンが異なる場合は，アドレスに対し 16 ビットアクセスならば 0x2 を，8 ビットアクセスならば 0x3 との排他論理和を取る命令 1 個の追加により補正が可能である．この命令によるオーバヘッドは，SPEC CPU95 13 本で CPI 増加が平均 0.06，最大 0.12 である．なお emu-PISA と emu-MIPSlike はリトルレンデイアンのレガシーをビッグエンディアンのホストで実装した結果である. また PowerPC のようにレガシー命令にアドレス境界制約がない場合には境界を跨ぐ場合に 1 バイトずつ処理することで実現しているが，境界を跨ぐ確率自体が低いため，性能低下に対する影響はほとんど無い．したがってエンディアンの違いも無視可能な範囲である．

4.7.5. CISC レガシー命令セットへの適用効果

レガシー命令セットが CISC のときの速度性能を考察する. 同一のプログラムで比較すると CISC は RISC に比べて命令数が少なくて済むため, インタプリタの処理時間の支配項であるコアループの実行回数が減りその分，処理は高速になる．一方，CISC では一般的に条件コード生成（フラグのセット）が必要なためそのオーバヘッドの増加が懸念される．

オーバフローやキャリーなどの条件コードの生成は，C 言語で実装すると 4～10 命令程度を要する煩雑な操作に変換されるため，エミュレーション性能上の課題と考えられがちである．RISC 系のレガシーISA には条件コードを使わない比較・分岐命令を備えたものが

多く, 条件コードを使用している PowerPC でも性能への影響は大きくない. その評価例 [6] のデータを表 15に示す.

表15フラグ生成を伴う命令の実行頻度

	099.go	129.compress	130.li
算術演算	0.058%	0.429%	0%
論理演算	0.013%	0%	1.684%
シフト演算	0%	0%	0%

CISC 系レガシーISA のインタプリタ性能を議論する. 実在する CISC 命令セットによる評価は, 開発環境と動作可能な汎用ベンチマークプログラムの入手が困難である. そこで, MIPSlike に IBM System370 の条件コード [22]に相当する条件コード生成を実装して SPEC CPU95 による評価を行った. 条件コードは 0～3 の値であり命令により生成方法を, 論理演算ではゼロと非ゼロの ZN, オーバフローの無い演算では正負符号を含めた ZMP, オーバフローやキャリーを含む CO, 浮動演算結果の FLT に分類した. 各々の実装ではホスト固有の条件コードレジスタを参照した実装は行わずに, 単純に比較分岐, シフト, 論理演算を行う C 言語記述をした. MIPSlike ではもともとオーバフロー検出する ADD と ADDU を区別しているが, CISC では一般にそのような区別が無いことが多いため ADDU, ADDIU などはすべて CO と定義した. 図 25に各条件コードの生成頻度（左軸, 縦棒）と, 生成による CPI 増加オーバヘッド（右軸, 折れ線）を示す. オーバヘッドはエミュレータ func-opt を基準とし, それに条件コード生成を追加したことによる CPI 増分を func-opt の CPI で割った値である. CPI 増オーバヘッドは, MPC7450 では CINT が 11%, CFP が 19%, SparcIIIi では CINT が 13%と CFP が 14%と比較的小さい. CINT では SparcIIIi はパイプライン段数が深く分岐オーバヘッドが多い分 MPC7450 よりオーバヘッドが大きいが, CFP ではプログラムにより傾向が異なっている. 生成頻度の平均は 43%で CO が 30%占めている. これはアドレス加算によく使用されている ADDU 命令と ADDIU 命令の頻度が併せて 26%あり, System370/XA の条件コード生成しない LA 命令相当のものも多く含まれていると考えられる. この命令種の統計では, 演算がないロード命令とレジスタ間のムーブ命令なども 25%を占め, 仮に CISC 化によりそれらが一切なくなり命令数がその分減ったとしてもオーバヘッドが 1.3 倍程度に増えるに過ぎない. 以上のように, レガシーが CISC で条件コードの生成確率は高いが, オーバヘッドはそれほど大きくないと予想され, コアループが支配項であることには変わりない.

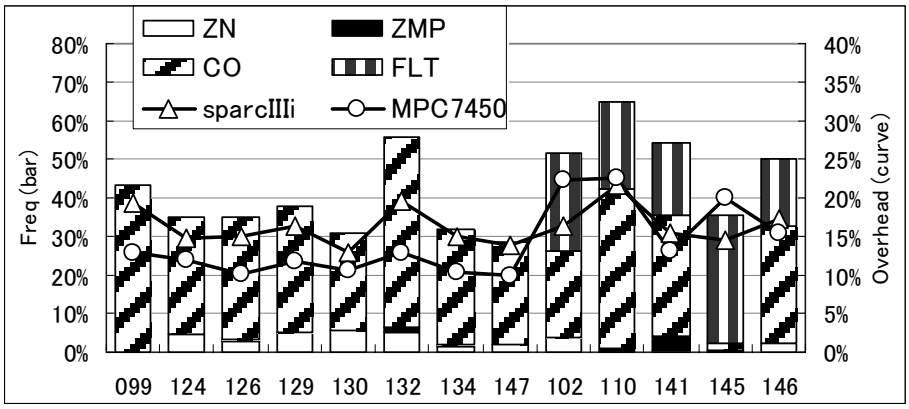


図25 SPEC CPU95 による条件コード生成頻度と CPI 増加

4.7.6. エミュレーションのオペレーティングシステム性能への影響

エミュレータの動作とオペレーティングシステム（OS）の性能の関係について簡単に触れる．OS のカーネル部分をホストのネイティブコードとして移植する場合は，OS 単体の実行性能は2.1.1で示した値 β を上限として著しく向上する．一方，性能の低下要因としては，エミュレータとインターフェイスをとるシステムコール，互換性を維持する入出力装置のアクセスなど考えられるが，その実行頻度は一般の命令に比べて極めて低く SPEC CPU95 ではシステムコールは 2.4ppm が最大であり，その影響は無視できる．アプリケーションプログラムはエミュレータを介して実行されるため，レガシーの環境に比べて，エミュレータの命令とエミュレータ変数の分だけメモリ使用量が増加する．インタプリタ方式ではそれは一般に数十 Kbyte，今回製作した中で最大の PowerPC 向けエミュレータ PPC-mpc-extra の例ではコアループ部と命令処理関数のコード部が約 24Kbyte であり，TLB ミスの増加も小さい．本論文の評価対象であるアプリケーションレベルのエミュレーションでは，複数のアプリケーションが並行動作するときにはエミュレータを複数プロセス実行することになる．エミュレータのコードと定数並びにアプリケーションのコードと定数は，それぞれ単一コピーを複数プロセス間で共有できるため，キャッシュミスの増加とアプリケーション起動時の負荷の増加は小さいといえる．一方，命令 TLB エントリの共有領域の実装は OS とホストアーキテクチャに依存し, エントリの共有ができないときには, エミュレータプロセスごとに数エントリが必要となる．データ TLB は，エミュレータプロセスごとにスタック分として 1 エントリと3.3に示した regs_t の分 1～3 エントリほど増加する．したがって，小さなアプリケーションを複数流すと TLB ミス回数は増える傾向にあると予想できる．しかしながら TLB ミスオーバヘッドはスローダウンにより薄まりシステム動作全体では無視できる．

4.8. まとめ

本研究では，インタプリタ方式のエミュレータのデコード方式を2.1.6で分類し，2.3.2で C 言語による実装方式の事例として SimpleScalar の PISA の sim-safe を取上げ，2.3.4で評価を行った．その結果，PISA の一部の命令を対象としたアセンブリ言語記述の理想コードに対し sim-safe は 0.39～0.57 倍の速度性能に留まっていること，SPEC CPU95 の 13

本のプログラムの CPI を測定し CPI 値が同様であることと，理想コードに対するオーバヘッドの原因を示した．また，PISA の命令はデコードが 1 回で済む特殊性があり一般化するため，MIPS-IV ライクな命令語対応に改造した sim-MIPSlike も2.3.5で評価した．sim-safe と sim-MIPSlike のコアループが 73～79%の時間を占め，その速度性能の改善が効率的であることを示した．sim-safe に見られるような switch 方式では，最適化がコンパイラ任せとなり性能向上を解決する着実な手法となっていないなどの問題がある．

このため，第 3 章では switch 方式の対極にある function 方式として，まず function 方式の試作の原型である emu-PISA-base の問題点を挙げ, それを解決する改良 function 方式の実装例 emu-MIPSlike-opt を説明した．

第 4 章では，emu-PISA と emu-MIPSlike の速度性能を評価した．改良 function 方式で I-LOOP では理想コードに対し，emu-PISA-opt で 0.81～0.89 倍，emu-MIPSlike-opt では 0.81～0.85 倍に相当する速度性能を実現した．SPEC CPU95 では RISC ホストでの速度性能向上は大きく switch 方式に対し PISA は 1.63～1.75 倍, MIPSlike は 1.48～1.77 倍の速度性能を得た．更に，別のレガシーISA として PowerPC，SH4，M32R のエミュレータを switch 方式と改良 function 方式の両方を試作し，改良 function 方式が 1.33～2.25 倍の速度性能の優位性があることを示した．これらによりレガシーアーキテクチャによらず改良 function 方式が C 言語実装のインタプリタ方式のエミュレータ実装に有効であるといえる．参考としてホストを x86 とした評価を実施し，レジスタ本数が制約となり性能向上は困難であるが，実用レベルのレガシーISA では switch 方式よりは優れていることを確認した．また，コアループの占める割合が 70～80%と高いことから，コアループの試作と評価により性能値の概略見積りが可能となるといえる．

第5章 性能評価シミュレータとバイナリ変換技術

本章では，性能評価を目的としたシミュレータの高速化手法とバイナリ変換について，従来研究を述べる．また，本研究のベースである ESPRIT/sim の概要も述べる．本研究の適用ターゲットは，異種マルチプロセッサを含む高性能な組込みシステム向けのシミュレータである．本論文では，論点をバイナリ変換アクセラレータに絞っている関係上，シミュレータの全体像を本章に記載する．また，従来研究との差異や性能や機能の優劣に関しても本章で述べる．

5.1では代表的なシミュレータの特徴を述べる．次に，5.2では，シミュレーションの高速化にあたりバイナリ変換と他の手法との関係を示し，また，エミュレーションと性能評価シミュレーションでバイナリ変換に対する取組みがどう違うかを述べる．これにより，性能評価シミュレータにバイナリ変換を搭載する効果と課題も示す．5.3では，本研究のベースである性能評価シミュレータ ESPRIT/sim の特徴を述べる．5.4では異種マルチプロセッサ向けのシミュレータの課題を示して，本研究の目標とする．5.5と5.6では，それらの従来研究と対比する形で ESPRIT/sim の位置づけ，性能，有効性と新規性を個別に説明する．

5. 1. 代表的な性能評価シミュレータ

本節では性能評価目的に使用されている，実行ドリブン型を中心とした代表的なシミュレータを概観する．5.1.1で Shade，5.1.2では SimOS と Embra，5.1.3で SimICS，5.1.4で SimpleScalar，5.1.5で SimCore/Alpha，5.1.6で ISIS，5.1.7で ISIS-SimpleScalar，5.1.8で Mambo，5.1.9で WSS，参考として5.1.10で SystemC 記述のシミュレーションに関して述べる．これらのうち，ごく最近の研究には，Mambo，WSS，SimCore/Alpha，ISIS-SimpleScalar が挙げられる．なお，11ページに示した図 2と図 3もこれらの関係を示している．

5. 1. 1. Shade

Shade [15]はサンマイクロシステムズが開発した高速なシミュレータであり，トレースドリブン型のシミュレータのフロントエンドとしてトレースを出力している†18（発表 1994 年）．これは，エミュレーション目的ではなく性能解析を目的としてバイナリ変換を使用したことで著名である．シミュレーション対象は Sparc 命令セットまたは MIPS-I 命令セットであり，それを Sparc（32 ビット版）のバイナリに変換してシミュレーションを行い，トレースをメモリ上に吐く．文献に明記はされていないが，生成されたバイナリを呼ぶ部分はアセンブリ言語で記述されているとコード例から推定できる．シミュレーション対象はアプリケーションレベルのソフトウェアである．シミュレーション対象の add 命令をホ

†18 トレース出力しているが Shade 自体は実行ドリブンとして分類されている．

ストの ld 命令，ld 命令，add 命令，st 命令，プログラムカウンタを更新する inc 命令に変換している．

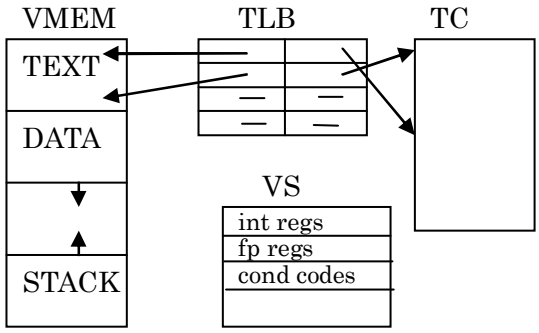


図26 Shade のデータ構造

バイナリ変換は実行されるコードのみを変換する動的バイナリ変換であり，その変換結果は図 26に示す TC（Translation Cache）に格納される．変換前のアドレスと変換後のアドレスはソフトウェアで作った TLB という表で関係付けられ, その表は N ウェイのセットアソシアティブ構造を採る．Shade は TC に可変長のエントリを順に詰めていく方式を採り, 満杯になるとフラッシュする. 分岐命令の変換には, TC 内の他エントリに分岐するチェイン構造を採る．また，Shade に関する別の文献[46]には，TC の管理方法に FIFO を用いて前方参照の分岐のみチェインしてサイクリックに TC を使用する方法も示されている. シミュレーション対象のレジスタ類は構造体 VS に保持され, プログラムカウンタはホストレジスタに置かれている．バイナリ変換では，プログラムが命令を書きかえる自己修飾が課題となる．Shade では，RISC でよく行われているキャッシュメモリをフラッシュする命令をきっかけとして無効化処理をしている．Shade にはいくつかのモードがあり，トレースを生成せず単に命令実行をする，命令数のみカウントする，プログラムカウンタのみトレースする，キャッシュアクセスしたアドレスをトレースするというモードの中から選択ができる．変換性能としては，最速のモードで Sparc から Sparc へ変換する例として命令数で平均 2.9 命令掛かるものと 5.9 命令掛かるものが示されている．処理時間は，ネイテイブ動作の 3.1 倍と 6.6 倍掛かっている．Sparc から Sparc への変換の中で遅いものは，ネイティブ動作の 84 倍の時間が掛かっている．

5. 1. 2. SimOS と Embra

SimOS はスタンフォード大学で開発された MIPS R4000 系のシミュレータである（発表 1995 年）. 複数のシミュレーション方式を動的に切り替えながら実行を行う複合型のシミュレータであり，シミュレーションの精度と速度のトレードオフを選択可能にしている．特に OS の振舞いに対してハードウェアの挙動をシミュレーションする目的のため，SimOS は，アドレス変換を含むフルシステムシミュレーションが可能である．

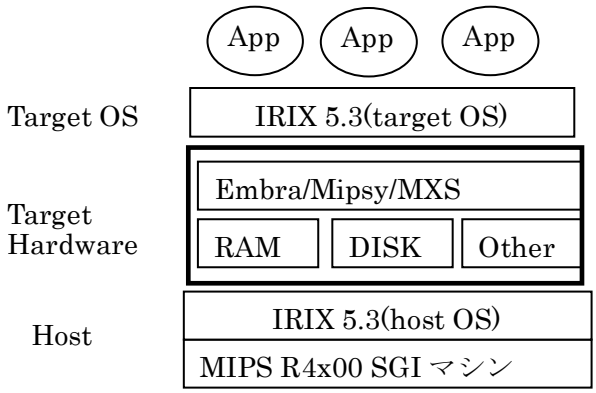


図27 SimOS の動作環境

図 27に SimOS の動作環境を示す. シミュレータのコアは 3 種類ある. 命令レベルシミュレータ Mipsy は，MIPS プロセッサのパイプラインを命令フェッチ／デコード／実行の 3 段階でラフにモデル化している．一方，MXS はプロセッサ内部のパイプラインを詳細にシミュレーションする.

EmbraはShadeと同様に動的バイナリ変換を使用しているが, OSカーネルのシミュレーションを行うためアドレス変換機能も備えている（発表 1996 年）．アプリケーション実行のシミュレーション時間は，ネイティブ動作の 3.5～9 倍，1 命令をシミュレーションするのに掛かるホストのクロック数を示すCPI値は7.5～27である. キャッシュシミュレーションを行った場合は更にその 1.2～3 倍程度の処理時間となる.

Embra はマルチプロセッサシミュレーションも可能で, 4-CPU シミュレーションの例では CPU 数に比例した処理時間が示されている. キャッシュシミュレーションを行った場合は更にその 6～10 倍の時間に延びる．文献 [37]によると，後に 1998 年に発表された論文のデータよりキャッシュメモリシミュレーション時にはネイティブの 130 倍の時間が掛かると評価されている．アドレス変換のため各 CPU が 4Mbyte の速見表を持つなど，メモリ負荷が大きいことが大幅に時間が掛かる原因と考えられている.

OS 込みのシミュレーションでは，シミュレーション時間はネイティブ動作時の 60～100 倍の時間が掛かり CPI 値では 67～330 である．しかし，このように性能低下は大きくても Mipsy の 16～18 倍は高速である.

UNIX では複数プロセス間で同一プログラムを実行したときにテキスト領域がシェアされることなどから，バイナリ変換対象のコードは物理アドレスで管理している．Embra の速度性能を Shade と直接比較はできないが, 物理アドレスで管理しているためか Shade に比べると性能が遅いように見える.

Embra には，ホストとシミュレーション対象の両方がマルチプロセッサのときに並列シミュレーションする Parallel-Embra という機能もある．しかし，評価アプリケーションが並列性の高い SPALSH-2 でも, ホストが 4-CPU 時に 1-CPU の 1.4～1.8 倍程度にしか性能向上していない．これはキャッシュメモリの動作チェックのオーバヘッドが大きいことが原因であると解析されている.

5. 1. 3. SimICS

SimICS [36][37]は，スウェーデンコンピュータサイエンス研究所で開発されたシミュ

レータで，システム全体のシミュレーションをねらったものである（発表 1997 年）．開発に 7 年の歳月と 20 人年掛けたと報告されている[37]．現在は Virtutech など商用に転用され，Sparc 以外の命令セットも対象にしている．

この特長は，シミュレーション対象の CPU の命令をメモリアクセスなどの部品に分解したスレッデッドコードと呼ばれる中間コードに一旦変換して，そのコードをインタプリタ実行する方式を採っている．そのため，一般のインタプリタより高速である．また，複合型シミュレータの構成を採っていない．アドレス変換とキャッシュメモリアクセスをシミュレーション対象としてその使い方が最適になるように，メモリ容量を圧迫しないデータ構造を用いている．図 28に示すように，アドレス変換には 4Kbyte ページごとに STC（Simulation Translation Cache）と呼ぶ 512 エントリの変換バッファを，

｛リード，ライト，実行 ｝×｛ ユーザモード，特権モード｝

の計 6 組用意しているが合計 24Kbyte と小さい．またキャッシュメモリアクセスにはラインサイズを 16byte に固定した 256 エントリの MRU（Most Recently Used）のアドレスを保持した STC を用い，キャッシュメモリのミス評価処理のオーバヘッドを削減している．マルチプロセッサ用には，キャッシュメモリの共有エントリ間のリンクを持っている．ただし，STC のアクセスに Sparc の 64 ビット命令を使った処理やスレッデッドコードへの分岐の goto 文に gcc の方言を使うなど移植性は良いとはいえない．

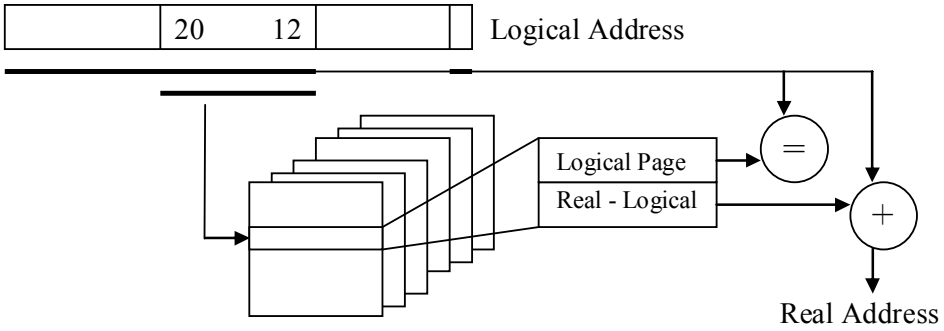


図28 STC の構造

Sparc から Sparc へのシミュレーションとして，CPI 値が 23～113 の速度性能が示されている．

5. 1. 4. SimpleScalar

SimpleScalar（発表 1997 年）[7]はプログラムコードが公開[34]されている高速なシミュレータとして著名である．ウィスコンシン大学で開発され，MIPS アーキテクチャをベースにした仮想アーキテクチャ PISA†19用に，C コンパイラ，リンカー，ライブラリと各種シミュレータが公開と提供されている．そのため，SPEC CPU95 をそのままコンパイルして，公開されたシミュレータを使った評価ができることから，多くの研究者が利用あるいはシ

ミュレータの拡張を行っている．

シミュレータはインタプリタ形式であり，用途ごとにカストマイズする構成を採っている．シミュレータの種類には，命令を単純に実行する sim-fast，不正実行をチェックする機能を持つ sim-safe[†20]，キャッシュシミュレーションを行う sim-cache，実行命令種などを調査できる sim-profile，それらにアウトオブオーダのパイプラインと分岐予測ミスなども含めた詳細パイプラインシミュレーションを行う sim-outorder[†21]がある．インタプリタの構造は，既に2.3.2で示したとおりである．

命令ごとに異なるインタプリタの実行コード，メモリアクセス，パイプライン制御などの情報は表をイメージした C 言語のマクロで定義され，C プリプロセッサにより用途別のコードに展開される構造を採る．このため，比較的高速な実装ができるが，可読性に課題がある[5].

PISA 用には C 言語で記述されたシステムコールシミュレータが実装されおり，POSIX 対応の OS には移植が容易である．Windows の VC++には移植できないが，Cygwin 環境下では gcc で実装できる．そのほかの命令セットとして PowerPC，Alpha，ARM のソースも公開されているが，AIX，OSF など現在では余り一般的でない環境が必要なため，PISA を使ったシミュレーション速度性能との比較がよく行われている．

SimpleScalar の Ver.3.0 以降では，複数のシミュレータと複数の命令セットが利用できるが，それはソースコードを共通化しただけである．もともと，コンパイル時に定数やアドレスが確定する構造を使って性能の最適化を図っている．そのため，複合型のシミュレータ，マルチプロセッサ機能，異種プロセッサの混載を行うように拡張すると，その高速性の特長が薄らいでしまう．

5.1.5. SimCore/Alpha

SimCore/Alpha は，CPU シミュレーションを目的としシミュレータのコアの高速化と可読性をねらって C++で Alpha を記述したものである（発表 2005 年）[5]．高速化のためにバイナリ変換などの手法は採らずに，インタプリタを使用している．プログラムカウンタでハッシュした速見表でデコードのオーバヘッドを削減しており，その効果は 2.3 倍あると報告されている．それは，一般の C 言語によるデコード記述の処理よりここでの C++記述のデコード処理が遅いため，効果が大きめに出ている．CPI は，130～180 程度であり，ホストにパイプラインが深い Xeon を使用していることを差し引いても CPI が大きく高速とはいえない[†22].

[†19] MIPS は 32 ビットの命令長を持ち 1～4 段のデコードが必要であるが，PISA では 64 ビットの命令長にして命令種を限定して 1 バイトにコードを納めてユーザ拡張を許している．

[†20] 実在のアーキテクチャに比べると例外チェック機能が非常に少ない仕様となっている．

[†21] 分岐予測ミスによる投機実行失敗のペナルティ，命令プリフェッチ機構による命令キャッシュのアクセス，キャッシュミス時の最初の語をフェッチした後にパイプストールが解除される機能など多くの機能が欠落しているが，これを使用した研究が多い．

[†22] ESPRIT/sim の Xeon で PowerPC シミュレーションで C 版で CPI が 90，C++版で 120 であり，SimCore/Alpha の 169 は大きい．

5.1.6. ISIS

ISIS は，慶應義塾大学が開発したマルチプロセッサシステムのシミュレータ構築用のライブラリである（発表2001年）. シミュレーションのターゲットのアーキテクチャやシミュレータの利用方法は限定せずに，システムの構成要素となる部品のみを提供しているため，多目的な用途に利用できる．この ISIS のライブラリを使用することによりシミュレータ開発の労力を軽減できる［44].

ISIS は C++言語により記述されたクラスライブラリであり，C++の特長であるクラスの派生，コンストラクタによる初期化，STL（Standard Template Library）と呼ばれるテンプレートを使用したビット幅のカスタマイズなどを使いモデル化を容易にしている.

ISIS は各部品をユニットとし，部品間の通信はポートを経由してパケットとして定義した手段でモデル化されている．図 29にそれらの関係を，図 30にそれらの基本クラスの定義を示す．パケットの実体は派生クラスで定義される．ポートではパケットの送受信を行う put と get, ポート間の接続と切断を行う connect と disconnect が定義されている. 各ユニットはクロック同期で動作する．それらの同期は clock_in 関数と clock_out 関数を呼び出して 2 フェーズで処理するため，個別に記述したユニット間の並列動作が機能する.

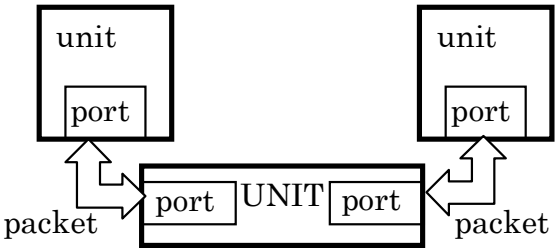


図29 ISIS の基本クラス

```
class packet {
public:
   virtual packet *new_packet( ) const=0;
}

class port {
public:
   void put(packet *);
   packet *get( );
   void connect(port&);
   void disconnect( )
   bool have_packet() const;
}
```

```
class unit {
public:
   virtual void clock_in( )=0;
   virtual void clock_out( )=0;
   virtual void reset()=0;
}
```

図30 ISIS の基本クラスのメンバ

ISIS には MIPS の R3000 のモデルが含まれている. 命令レベルシミュレーションを実行したときの速度性能は，ホストが PentiumIII（600MHz）では，単純ループやメモリコピー動作時に 0.3MIPS である．CPI 値に換算すると 2000 程度である．同ホストで SPLASH-2 ベンチマークの並列シミュレーションは，シミュレーション対象の CPU が 1 個あたりで

0.16MIPS，CPI 換算では 3600 程度である．シミュレーション対象の CPU が増えた場合は，その数に比例して処理時間が増える特性を持っている．

5.1.7. ISIS-SimpleScalar

ISIS に用意されたプロセッサが単純なパイプライン構造でありスーパスカラに対応していないため，ネットワーク結合の最新システムの評価には不十分となってきた．そこで，慶應義塾大学ではプロセッサのアーキテクチャ評価に使われている SimpleScalar のプロセッサモデル sim-outorder をクラスライブラリに改造して，ISIS と融合させた（発表 2007 年）．これにより，SimpleScalar もマルチプロセッサ機能を備えかつ，プロセッサ間の結合方式を容易にシミュレーションする環境も提供できるようになった．

ISIS-SimpleScalar のシミュレーション速度性能は SimpleScalar ツールセットの sim-outorder の約 1/4 であり，SPLASH-2 ベンチマークでは CPI は 14000 程度となる．そのため，シミュレーションデータの事前デバッグ，キャッシュミスなどの評価を行うには速度性能が課題となりうる．なお，シミュレーション対象の CPU 数（プロセッサ数）が増えた場合は，シミュレーション時間は CPU 数に比例した時間より短くなる傾向がある．これは，CPU 間のメモリアクセス競合などのために CPU がストール状態となり，CPU のシミュレーションの負荷が軽くなるためである．

5.1.8. Mambo

Mambo は，IBM 社が開発したシミュレーション環境である（2003 年発表）．SimOS をベースに PowerPC を含むシステムをシミュレーションするもので，PowerPC と x86 系の PC など複数のホスト環境で動作する実用的なシミュレータである．システム全体をシミュレーションできるが，そのためか性能はホストが AMD 社の Athron1.2GHz のときに 2.5MIPS（CPI 値は 500）と低い．Cell Broadband Engine の開発環境のシミュレータも Mambo をベースにしている．それに，PowerPC のブート動作用に動的バイナリ変換が搭載されているが，動的バイナリ変換の速度性能に関して記述した資料は見当たらない．

5.1.9. WSS

WSS（Workload Specific Simulator）（豊橋技科大学，発表 2005 年）[32]は，命令レベルシミュレータの高速化を行うために，アプリケーションのバイナリモジュールを読んでシミュレーションするコードを C 言語で書き出し，それを gcc コンパイラでコンパイルしてホストコードを生成する静的バイナリ変換方式のシミュレータである．この実装は，SimpleScalar の sim-fast のインタプリタの実行部のコードを記述しているマクロを printf 文に書き換えて実現している．通常のインタプリタではコアループは，命令コードの取り出し，命令フィールドの切り出し，デコードのための分岐，レジスタが割りつけられたメモリ配列からのデータの読出し，それらの演算，レガシーレジスタのメモリ配列への書込み，プログラムカウンタの加算，コアループへの分岐を繰り返す．それに対して，WSS は命令分岐から次の命令分岐までを 1 つの単位とするブロック内の複数の命令列を次のような C 言語として出力する．

```
L800:       r[1]=r[2]+r[3];
            r[5]=r[5]+(-1);
            if(r[5]> 0) goto L800;
```

そのため，バイナリ変換と同等のコード生成が期待できる．この方法にはホストの命令セットアーキテクチャを知らなくても済むという長所がある．また，コンパイラに備わった最適化を利用した高速化も期待できる．

この方法の問題点として，次の点が挙げられる．

- もとのプログラムより遙かに大きなソースコードを一括して生成するため，大きなプログラムではコンパイラの限界を超す可能性がある．
- コードとデータが別れていないプログラムに対応できない．
- ダイナミックリンクライブラリを使用したアプリケーションには適用が困難となる．

更に，コンパイル時間が掛かる，ディスク溢れやまたコンパイル時のエラーや警告などを気にする必要があるなどの課題もある．なお，コンパイル時間として，SPEC CPU95 で最大のものが 18 分，残り 17 本が平均 85 秒と報告されている．

5.1.10. SystemC

SystemC [42]は，C++言語のクラス構文を利用してハードウェアとソフトウェアの双方を記述を可能にし，それらの協調検証をねらったものである．1999 年に Open SystemC initiative [42]が創設され，2003 年に Language Reference Manual がリリース，2005 年に IEEE の規格として承認された．現在，主に画像や通信などのアルゴリズムから LSI 実装までの連続性が強い分野を中心に，普及してきている．

```
# include "systemc.h"
SC_MODULE(adder)                    // モジュール（クラス）宣言
{
  sc_in<int> a, b;                  // ポート
  sc_out<int> sum;
  void do_add( )                    // プロセス
  {
    sum = a + b;
  }
  SC_CTOR(adder)                    // コンストラクタ
  {
    SC_METHOD(do_add);              // カーネルへの do_add の登録
    sensitive << a << b;            // do_add のセンシビティリスト追加
  }
}
```

図31 SystemC の記述例

図 31に記述例を示す．“SC_”や“sc_”で始まるキーワードは C++のキーワードに置換される．SC_MODULE は各部品に対応するクラスを定義し，SC_CTOR はコンストラクタとなる．モジュール本体には，処理そのものであるプロセス，他部品とのインターフェイスを取るポートまたは上位概念のチャネルがある．ISIS がクロックドリブンの制御であるのに対し，SystemC はイベントドリブン記述が可能である．プロセスの記述が複数サイクルを使う場合は，wait()文により他部品からのイベントまたは次サイクルまで待つ記述ができる．そのような並列動作は，SystemC カーネルが行っているスレッド間のスケジューリングで実現されている．

記述の粒度の自由度は大きいが，詳細な記述を行うとシミュレーション速度は非常に遅くなる．表 16にシミュレーション粒度または抽象度を違いとした記述レベルの例とシミュレーション速度のオーダを示す．SystemC ではシミュレーション対象回路のクロック周波数を実時間と比較するため，命令処理数よりクロック周波数を用いて性能を表すことが多い．この表ではそのクロック周波数で速度を表記している．

表16 SystemC の記述レベルとシミュレーション速度性能

名称	説明	シミュレーション速度 （1GHz Sparc，2GHz Xeon レベル）
RTL レベル	クロックサイクル	1～100KHz
BCA（バスサイクル精度）	バスサイクル数が合致	10KHz～1000KHz
TF（タイムドファンクショナル）	機能ごとの概略サイクル数が合う	100KHz～10MHz
UTF（アンタイムドファンクショナル）	機能動作が合う	

SystemC は C++で記述されるため HDL 記述よりは高いシミュレーション速度性能が見込める．しかし，それでも性能評価用のシミュレータに比べると性能は低い．そこで，トランザクションレベル，Approximately-timed などの概念を導入して抽象度を上げた使い方が多い．また，シミュレーション対象のソフトウェアをホスト用に直接コンパイルさせて処理時間情報をバジェットとして追加するモデル化手法 [43]のように100MHzの動作をねらうなどの変形はあるが，それでも依然として性能は低い．

5.2. 性能評価シミュレータの高速化

本節では，性能評価シミュレータの高速化手法と課題を整理する．

5.2.1. 性能評価シミュレータに搭載されたバイナリ変換方式

性能評価目的のシミュレータに使用されているバイナリ変換方式には，静的バイナリ変換と動的バイナリ変換がある．静的バイナリ変換の有利な点は，ホストのアーキテクチャに最適化したバイナリを生成できる，分岐のオーバヘッドを小さくできる，データキャッシュメモリを圧迫しないことである．不利な点は，使用頻度の低いコードも変換するため変換処理に時間が掛かる，変換後のコードを格納する領域として広いメモリ領域が必要となることである．更に，命令コードとデータの分離が困難である，命令コードの自己修飾への対応ができない，ダイナミックリンクライブラリに対応できないという実用上の課題もあるため，使用上の制約が大きい．それに対して，動的バイナリ変換ではこれらが課題とならずかつ実装が容易なため，主流となってきた．

WSS が行っているバイナリを解析し C 言語に展開したシミュレーションコードをファイル出力してコンパイラを通す変換方法は，静的変換の派生形でありその短所と課題を受け継いでいる．つまり，ダイナミックリンクライブラリに対応できない，静的に解析できるバイナリモジュールに限定される，などの課題がある．また，コアループがないため，マルチプロセッサシミュレーションに必要なシミュレーション対象の切り替えが困難になる．

加えて，静的変換より更にコンパイル時間が長く掛かる．そのため，パイプラインの詳細シミュレータのフロントエンドとして使う場合はよいが，キャッシュシミュレーションでもシミュレーションに比べて 10 倍ほどのコンパイル時間が掛かり効率的でない．

5.2.2. インタプリタ方式の高速化方式

インタプリタ方式は，命令語の読出し，デコード，実行処理への分岐がバイナリ変換に比べて多いため，低速である．バイナリ変換以外の手法により，その速度性能の課題を緩和する方式として次の(a)～(d)のレベルがある†23.

a) **プログラムロード時の事前デコード**：
命令コードのデコードをプログラムロード時に行う方法があり，SimpleScalar/PISA はこの方法を採っている．これにより，複数フィールドに別れたコードのデコードの処理時間を短縮できる．しかし，命令とデータがセクション内に混在したバイナリモジュールではこの方法が使えない．

b) **ハッシュテーブルによるデコード結果の再利用**：
これは，プログラムカウンタ値を使ってハッシュテーブルを引き，デコード結果をテーブルに格納する方法である．SimCore/Alpha でも使用している手法であるが，デコードがもともと速いとその効果は小さい．バイナリ変換方式の機能を限定した場合（114 ページの表 22に記載した ESPRIT/sim の変換レベル 1 のサブセット）の動作と効果の一部と考えることができる．

c) **インタプリタ関数を呼び出すバイナリ命令列**：
インタプリタの命令実行部を function 方式で実装し，それらの関数呼出しをコアループで行わずにバイナリに展開されたコードから呼び出す．これは(b)より高速であり，デコード時間のほとんどの時間を削減できる．表 22（114ページ）に記載した ESPRIT/sim の変換レベル 1～3 がこれに該当するが，従来研究では言及されてない．

d) **スレデッドコード**：メモリリード，メモリライト，演算実行などの部品を関数として定義しておき，あらかじめまたは動的にプログラムを解析してそれらを呼び出すバイナリを生成する方式である．部品となる関数が互いに次の関数に直接分岐をすることによりデコードや分岐のオーバヘッドを削減できる．SimICS はこの方式を採用しており，ホストとレガシーISA への依存性がバイナリ変換より軽減できる長所がある．SimICS ではシミュレーション対象 ISA の命令仕様をデータ定義し SimGEN というジェネレータで変換しているが，そのデータ定義のデバッグが課題である．バイナリ変換と方式は異なるが，ホストへの依存性も強いようである．　更に，標準の C／C++には変数値により飛び先を変更する goto 文の仕様はないが，SimICS ではそれが可能な gcc の goto 文の方言も利用している．この方式は，十分な性能とはいえないがインタプリタとバイナリ変換の中間の処理性能を実現できる実用的な方式の 1 つであるといえる．

---

†23 インタプリタ自体の高速化手法として，本研究の第 3 章で提案している手法やホストのスーパスカラの並列性を引き出すために複数命令を同時にインタプリタ実行する方式[16][17]は，ここでは特に扱わない．

5.2.3. 並列処理によるシミュレータの高速化

科学技術シミュレーションでは，複数のコンピュータを並列使用したシミュレーションが一般的となっている．一方，シミュレーション対象をコンピュータにした場合は，シミュレータの構成要素となるモデル間の結合度が強くかつモデル内で閉じた計算量が小さく並列シミュレーションに不向きなせいか，コンピュータの並列シミュレーションに関する研究は少ない．

*N*個の CPU からなるシミュレーション対象を *N*個のスレッドに分割して *N/K*個のホストに CPU を割り付ける方式がある．この例としては SimOS の Parallel-Embra が挙げられるが，ホストが 4-CPU 時に 1.4～1.8 倍程度にしか性能向上しないなどスケーラビリティは悪い．キャッシュシミュレーションに PC クラスタを使用した例としては Shaman[48]がある．これも，ホスト 8 個で 6 割，16 個で 3～4 割の速度性能しか出せないなどスケーラビリティは悪い．

このようなデータ並列を指向した方式に対して，ホストごとにシミュレーションを担当する部分を時間方向に分割する方式がある．これは，まずトレースドリブンシミュレータのフロントエンド部を用いて粗いシミュレーションを行い，その結果から区間を分割する．次に，複数ホストでそれぞれの区間に対して詳細なシミュレーション（バックエンド）を同時に行う[60]．この方式では，フロントエンドとバックエンドの処理性能が大きく異ならないと効果が出ない．また，隣り合った区間との重複処理が必要となる．そのため，パイプラインシミュレーションのように時間が掛かるシミュレーションでは，効果があるといえる．

このように，従来は 4 台までのマルチプロセッサやネットワーク接続のクラスタを中心に，並列シミュレーションの研究が行われてきた．現在，マルチコアを搭載した PC が普及してきており，2 次キャッシュメモリなどの共有によるスレッドレベル並列シミュレーションに関する研究が，今後は増えると思われる．

5.2.4. エミュレータと性能評価シミュレータの差異

特別なハードウェアを用いずにバイナリ変換のみで，命令セットの異なるアーキテクチャを厳密かつ高速にエミュレーションするのは困難であり，性能を優先すると機能や例外動作に制約が出る[†24]．しかし，使用目的を性能評価用のシミュレータに限定すると，下記の理由によりバイナリ変換方式のアクセラレーションがより容易となる．

- 通常の動作では例外発生がしない（ページフォールト例外などは必要）．
- 浮動小数のアンダーフローなどによる精度の違いが，探求対象である性能の評価や解析に及ぼす影響は小さい．
- 既存のバイナリモジュールにある潜在的な障害まで互換動作を求められることは稀で

[†24] 専用ハードウェアなしにバイナリ変換でエミュレーションしたものは数少なくアップル社の Executor (M68K→PowerPC)くらいであり，その安定性は課題であったといえる．同社の最近の事例である Rosetta では PowerPC と x86 のそれぞれに対応したバイナリの両方を持っておりエミュレーションは補完的な役割に留めている．

ある.

- 互換エミュレータ並みのバイナリ変換の信頼性は必須ではない.
- エミュレータでは従来使用してきた機種より処理が遅いとその存在価値が薄れるが, 性能評価目的のシミュレータでは性能に対する厳密な指標がないためインタプリタより数倍高速なだけでもその価値はある.

したがって，性能評価シミュレータのバイナリ変換への取組みは，エミュレータ用途のバイナリ変換に比べて容易となる.

5.2.5. 性能評価シミュレータの高速化に関する課題

一方，性能評価を目的としたシミュレーションゆえに次の点が新たな課題となる.

- 性能評価のためだけにバイナリ変換機能の開発に多大なコストを掛けられない.
- ある程度の品質を確保しないとシミュレーションができないため, バイナリ変換の導入の敷居は高くなる.
- 複数の命令セットの CPU が混載可能なシミュレータの開発では，CPU を供給しているコンピュータメーカが異なるため必要な命令セットのバイナリ変換機構を取り揃えるのは困難となる.
- 性能評価に必要な命令数などの情報やマルチプロセッサシミュレーション時の CPU の切替え処理などが別途必要となる.

5.3. ESPRIT/sim

ESPRIT は，三菱電機がコンピュータ開発に使用していた性能解析手法を社内の組込み機器向けに一般化した技術の総称[†25]であり, ESPRIT/sim はそのシミュレータの呼称である. 同社では 1980 年代後半からシミュレーションによる性能の分析と解析が主流となり, 1990 年代前半までは CISC 系の独自アーキテクチャの CPU をその解析対象としていた．1990 年代後半から筆者を中心に，汎用 CPU を部品としたコンピュータシステムを対象とするシミュレーション技術とシミュレータ構築を行ってきた．1997 年には PPC750[†26]搭載製品の速度性能の解析を目的に PPC750 のスーパスカラパイプラインシミュレータを製作した．その後，MIPS，M32R などの各種シミュレータを自製し，2002 年には動的バイナリ変換を実装し，シミュレータソースコードの統合化を行った．2005 年にはシミュレータのソースコードを C 言語から C++へ移植を行い，異種マルチプロセッサシミュレーションへの対応を行った．更に同年，学術研究の一環として他のシミュレータ研究と比較評価のために SimpleScalar の命令セットも実装した（発表 2006 年）.

図 32に ESPRIT/sim のイメージを示す．シミュレータのコアとして各種の命令エミュレータ，メモリ，キャッシュ，バスのモデル，I/O シミュレータ，解析とレポート機能を持

[†25] Embedded System PerfoRmance Improvement Technology の略，2002 年以降よりこの呼称を使用

[†26] IBM 社製品は PPC750，モトローラ（現在のフリースケール社）製品は MPC750

つ．入力には，キャッシュミスやメモリなどのアクセスペナルティ及びキャッシュメモリ構成のようなパラメータ，並びにアプリケーションのバイナリがある．出力は回数や時間情報などボトルネックを示す情報である．

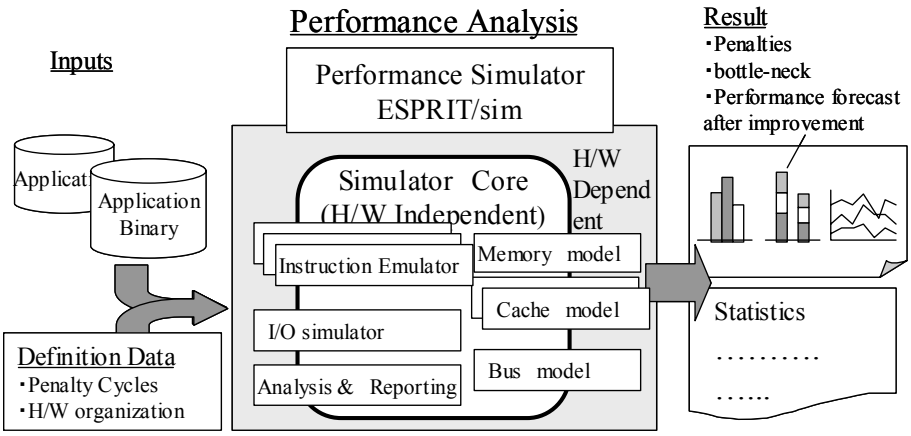


図32 ESPRIT/sim の処理イメージ

統計情報として次のようなものが採取できる．

- 命令実行回数
- クロックサイクル数
- 命令の種類と実行回数
- メモリアクセス（読書き，ページ，記憶素子）回数
- キャッシュメモリアクセス（ヒット，ミス，ライトバック，無効化）回数
- TLB アクセス（ヒット，ミス，無効化，パージ）回数
- 関数（呼出し回数，命令数，サイクル数，キャッシュヒット／ミス数，TLB ヒット／ミス数，関数の親子単位の呼出し回数）
- 分岐（実行，成立／非成立）回数，アドレスごとの同回数
- 分岐予測（ヒット／ミス）回数
- パイプライン（事象ごとの回数，事象ごとのオーバヘッドサイクル数）

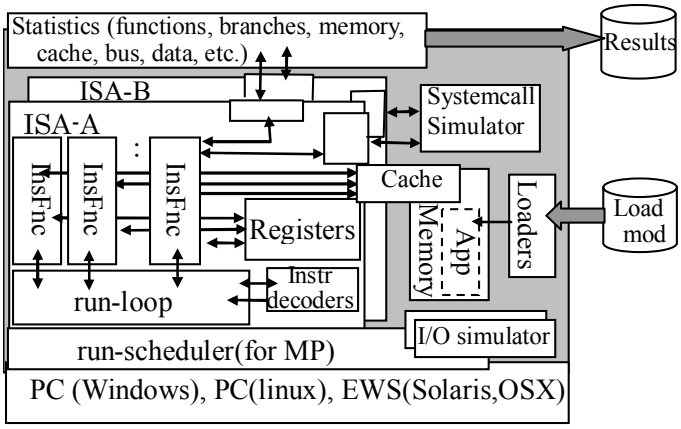


図33 ESPRIT/sim の構成

ESPRIT/sim の構成を図 33に示す. 本シミュレータは, 命令セットアーキテクチャ (ISA) ごとに各命令に対応した命令処理関数(InsFnc)とデコーダ及びインタプリタの処理を繰り返すコアループ(run_loop)を持つ．レガシーISA のレジスタやキャッシュメモリは CPU ごとに持つ．共通なものとして，メモリ，I/O シミュレータ，システムコールシミュレータ，統計機構，コマンドインタプリタ，ローダなどがある．複数 CPU 構成時や I/O 動作時にはスケジューラ(run-scheduler)が run-loop の切り替えを行う.

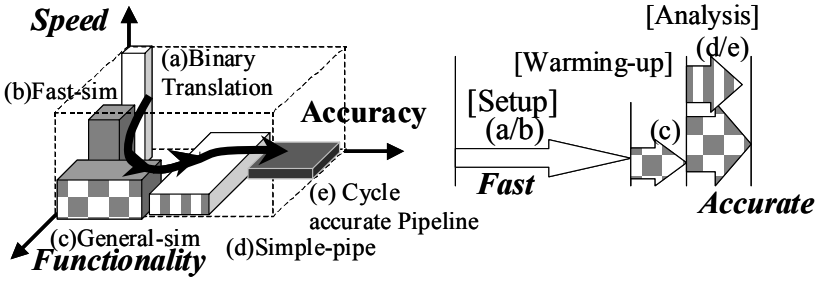


図34 ESPRIT/sim のシミュレーションコアの特性と使い方

シミュレータコアには，採取する統計機能を限定した高速インタプリタと，統計対象を個別に選択できる解析用シミュレータを標準装備している．更に，インオーダ命令発行をモデル化した簡易パイプラインシミュレータと，アウトオブオーダ命令発行／投機命令実行／命令プリフェッチが扱える詳細パイプラインシミュレータがあるが，それらの実装の有無は ISA 依存である．これらは，それぞれ別のコアループとして実装されている．ESPRIT/sim は複合型のシミュレータで，図 34に各シミュレータコアの特性と使い方のイメージを示す．実行速度（Speed），機能（Functionality），精度（Accuracy）は相反するものである．バイナリ変換（a）または高速インタプリタ（b）は速度重視のときに使用し，（d, e）パイプラインシミュレータは精度を重視するときに用いる. 各種の統計を採る, CPU 内部よりキャッシュメモリやメモリおよびバス周りの遅延が支配的な場合は（c）の解析用インタプリタを使用する．組み合わせて使う場合は図の右側に示したように，（a）または（b）でプログラム動作の立ち上げ，（c）でキャッシュメモリの状態を反映し，(d）または（e）で詳細な解析を行うという順に使う.

ESPRIT/sim がシミュレーション対象としている ISA（レガシーISA）には，次のものなどがある.

- PowerPC
- MIPS32/64
- SH4
- M32R/M32RII
- SimpleScalar/PISA

ホスト ISA には，x86 (Windows，Cygwin，Linux)，Sparc (Solaris)，PowerPC (OSX) がある.

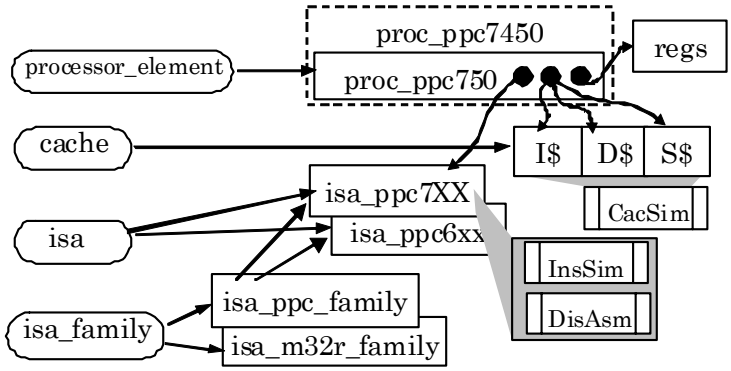


図35 ESPRIT/sim のクラスの例

図 35に ESPRIT/sim のモデル記述で使用しているクラスの関係を示す．各プロセッサは processor_element と isa_*の両方の派生クラスとして proc_*に定義される．isa_*は命令セット依存の処理を，processor_element はハードウェアの実体を受け持つ．同じ CPU のシリーズでもキャッシュメモリの構成が異なるものは proc_*ごとにカストマイズできる．各 CPU の処理は基本クラス processor_element のポインタから仮想関数としてアクセスされ，proc_*のメンバー関数，isa_*のメンバー関数の順にアクセスされる．次に processor_element クラス，isa の派生クラス及び proc_*クラスの定義例を示す．

processor_element クラスの定義例

```
class processor_element {
protected:
public:
   processor_element();
   char *processor_type_name;
   class isa_family *isa_fam;
   class isa *isa_ptr;
      :
   struct mpc_regs_dummy_s regs;                        // 各 ISA ごとのレジスタ定義
   unsigned long pc, pc_real;
   class cache_memory *i_cacp, *d_cacp, *s_cacp;      // キャッシュメモリへのポインタ
   struct code_cache_control_s cc_ctrl;        // コード変換キャッシュの制御とポインタ
   struct ins_def_s virtual *get_decode_ptr
   (unsigned long code, unsigned long addr) = 0;      //命令デコーダ
   void virtual translate(unsigned long addr) = 0;    // 命令列のバイナリ変換を起動
   bool translate_unit_instr_isa(unsigned long *lc, int cc_ix);
                                                        // 命令 1 個のバイナリ変換
   int virtual run_dbt_core(int cnt, int *cont) = 0; // バイナリ変換モードの run_loop
   void virtual run_fast_core(int cnt, int *cont) = 0;// 高速インタプリタの run_loop
   void virtual run_normal_core(int cnt, int *cont) = 0;// 解析用インタプリタの run_loop
      :
   int virtual print_instr(char *s, unsigned long code,
      unsigned long adr, unsigned long code2 = 0) = 0;
}
```

isa の派生クラスの定義例

```
class isa_ss_pisa: public isa, isa_ss_pisa_family {
public:
   isa_ss_pisa();
   struct ins_def_s *get_decode_ptr_isa(unsigned long code, unsigned long addr);
   void translate_isa(unsigned long addr) { cpe->translate_common(addr); }
   bool translate_unit_instr_isa(unsigned long *lc, int cc_ix);
   int run_dbt_isa_core(int cnt, int *cont);
   void run_fast_isa_core(int cnt, int *cont);
   void run_normal_isa_core(int cnt, int *cont);
      :
   int print_instr_isa(char *s, unsigned long code_upper,
      unsigned long code_lower, unsigned long adr);
}
```

proc_ss_pisa クラスの定義例

```
class proc_ss_pisa: public processor_element, isa_ss_pisa {
public:
   void translate(unsigned long addr) { translate_isa(addr); }
   bool translate_unit_instr(unsigned long *lc, int cc_ix)
        { return translate_unit_instr_isa(lc, cc_ix);}
   int run_dbt_core(int cnt, int *cont) { return run_dbt_isa_core(cnt, cont);}
   void run_fast_core(int cnt, int *cont) { run_fast_isa_core(cnt, cont); }
   void run_normal_core(int cnt, int *cont){ run_normal_isa_core(cnt, cont);
      :
   int print_instr(char *s, unsigned long code, unsigned long adr,
      unsigned long code2 = 0)
         { return print_instr_isa(s, code, code2, adr); }
}
```

図 36に，ESPRIT/sim に実装した動的バイナリ変換の処理フローを示す．コード変換キャッシュ（CC）は，レガシー命令アドレス，ホスト命令アドレスのペアからなるディレクトリ （CCD）とホスト命令列を格納した配列 （CCE）から構成される．実行頻度を計数するカウンタディレクトリ（CTD）の値が閾値を超すとトランスレータを呼び出す．変換済みのコードが見つかった場合は CCE 内のホストコードに分岐する．CCE から溢れるか分岐が成立すると，このループに戻るようにホストコードは生成される．インタプリタが実行されるとき，図中の“Interpreter”が命令デコード後に InsFnc を呼び出す．すなわち，インタプリタとトランスレータのそれぞれのコアループが密結合した一体構造が採用されている．付録に SimpleScalar/PISA 用の run_loop 部のコードを示す．

なお，ESPRIT/sim はコンパイラ依存のインラインアセンブルや C 言語の方言は原則的に使用していない．例外は，ホスト PowerPC 用のキャッシュの無効化処理と，ホスト x86 の RDTSC （Read Time-Stamp Counter）命令を使用した正確な時間測定を行うインラインアセンブル関数のみである．

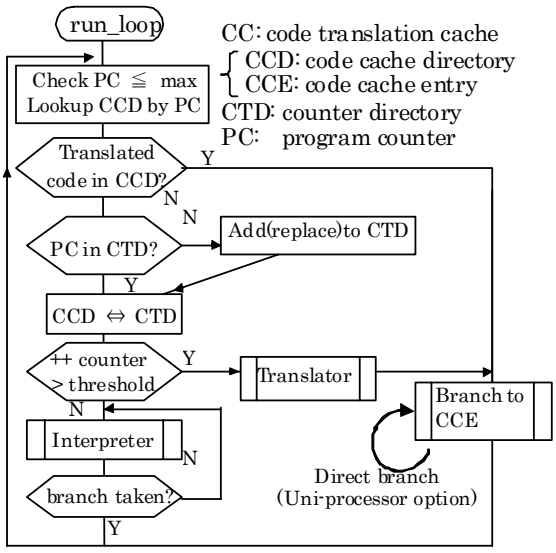


図36 動的バイナリ変換の処理フロー図

5.4. 性能評価シミュレータの課題と ESPRIT/sim の目標

性能評価を目的としたシミュレーションの対象は，従来は高性能なコンピュータシステムであり，そのためマルチプロセッサシミュレーションも対称型がターゲットであった．しかし，最近は組込み機器も高性能化しており，性能評価シミュレーションの対象となってきた．特に異種マルチプロセッサを含むマルチコアや複数チップから構成されたシステムの性能評価の需要が増えると考えられる．本節では，異種マルチプロセッサシステムの性能評価を目的としたシミュレータに共通な課題を述べ，ESPRIT/sim が解決する目標として示す．

5.4.1. シミュレーション速度性能

設計の最終段階では，シミュレーションは少ない回数で済むため，高速性よりも正確さが優先される．一方，その前段階の試行錯誤的な解析は，実行回数が多くターンアラウンドタイムが重要となるため，高速性が要求される．要求される速度性能値は，絶対的な処理時間よりもむしろ，シミュレーション対象の実処理時間から何倍遅いかを示すスローダウンで表すのが適切である．スローダウンの一般的な目標数値は定義されていないが，ここではシミュレータの目的から 10 倍程度のスローダウンを目標とした．

一般に，インタプリタ方式のシミュレーションのスローダウンは 20～200[†27]である．C++ のクラスを利用した場合には，仮想関数の間接呼出し，統計項目や拡張機能のオンオフ判定，命令ステップ実行の判定，マルチプロセッサの並行動作シミュレーションのオーバヘッドなどを勘案すると，評価使用時の速度性能は低下しスローダウンは更に大きくなる．

[†27] 古い事例では Shade の 20～80[15]，最近の例では SimCore/Alpha の 119～246[5]，ESPRIT/sim の 20～45[35]より 20～200 と表現した．

シミュレータの速度性能の向上のために，解析機能別に複数のシミュレータコアを搭載して目的に応じて切り替える複合型を採用するのは一般的である．ESPRIT/sim も，同様に解析用シミュレータのほかに，高速命令インタプリタと動的バイナリ変換を組み合わせて使用できる．それらの高速なシミュレータを使用して解析対象のポイントまでのセットアップを行い，そのポイント以降は解析用シミュレータなどを用いてキャッシュシミュレーション解析を行うなどの切替えを行う．

5.4.2. 拡張性と保守性

異なる命令セットアーキテクチャ（ISA）のプロセッサが混載可能なシミュレータでは，ISA 依存の機能とプロセッサ依存の仕様[†28]の組合せが膨大となる．そのため，シミュレータの拡張性の維持と保守が一層困難となるという課題がある．

この課題に対し，ESPRIT/sim は記述言語を C++に替えてクラスと名前空間を使って拡張性と移植性を改善した．図 35（79ページ）に示すように ISA ごとに基本クラス isa の派生クラスとして isa_*を用意する．すべてのプロセッサは，基本クラス processor_element と派生クラス isa_*の更なる派生クラスとして実装され，仮想関数を用いた統一手順にて呼出しが行われる．プロセッサのリソースであるレジスタは processor_element のメンバに持ち，キャッシュメモリはクラス cache に持つ．

このような構造と統一した呼出し手順の採用により，ESPRIT/sim では派生モデルの追加と保守が容易になり [35]，利点となる．また，InsFnc はクラス化すると宣言やデコード表の記述が煩雑になるためクラス化せずに，名前空間を用いて命令処理関数名の重複を許している．そのため，複数の ISA 間の命令の名前の区別も不要となり開発と保守が容易化できる．クラス化と名前空間の採用により拡張性と保守性が向上する反面，シミュレータのコンパイル時に定数やアドレスが定まらないことが原因となり，ISA を固定した C 言語による記述方式に比べて性能が 1～3 割低下することが移植経験から分かっておりそれが短所となる [35]．しかし，動的バイナリ変換を使用するとホストコードの生成時にアドレスなどを計算してホストコードに反映できるため，それらの短所も克服できる．

5.4.3. シミュレータの検証コスト

コンピュータメーカや半導体メーカは，製品のアーキテクチャに対応した診断プログラムやテストセットを豊富に持つ．それに対し，市販マイクロプロセッサのユーザである組込み機器の製造販売企業が，シミュレータを開発するときは，開発と検証のコストが問題となる．

命令セットシミュレータの設計は，マイクロプロセッサのマニュアルを見て行うのが一般的である．このため，まず命令仕様の理解に誤りが発生する可能性がある．次に実装時にもミスが発生する．C／C++言語で記述されたインタプリタはケアレスミスが少ないのに対し，バイナリ変換ではバグの発生が増えることが開発の経験上分かっている．更にホストの命令仕様の理解誤りのバグも加わる．またバイナリ変換では，メモリ破壊など潜在的

なバグがあるとほかの条件が変わったときにそれが顕在化することがある．このようにバイナリ変換方式を採用するとバグが増えるため，検証コストが課題となる．

検証には，まず，プログラムの挙動と期待値が分かっているテストセットを使用した機能劣化試験が必要である．次に，バイナリ変換処理の複雑性に対応して増加するようなバグをなくすため，より高度な試験が必要となりこのコストが上昇する．

ESPRIT/sim では，検証コストを下げつつ検証カバレッジを向上させるため，バイナリ変換対象のレガシー命令間の組合せへの依存性を排除し，コード変換キャッシュの履歴への依存性も軽減する．また，各種の ISA に共通の C 言語で記述したテストセット [35]を用意し，それを用いて試験を行っている．これは，同一の試験ソースコードを C コンパイラでコンパイルして共通に適用できるため，試験データの作成と試験データ自身の検証の共通化が図れる，試験工数と検証品質の確保に役立つ．それに加えて，アプリケーション実行動作をインタプリタの結果と比較する機能を併用し，そのテストセットを補完[†29]している．このような方法で品質を確保できるのも，複雑性の軽減によるところが大きい．

ESPRIT/sim が採用した“バイナリ変換の簡易化”の短所は，簡易化による性能劣化であり，目標に対してどこまで性能向上ができるかが重要となる．

[†28] キャッシュメモリの構成や容量などが典型的な例である．

[†29] C 言語で記述したテストセットは各種 ISA ごとの違いもカバーするようなデータとなっているが，レジスタや条件コードの組合せ，アドレスのオフセット範囲，制御命令が弱い部分である．その弱点はインラインアセブルによる ISA 固有テストの追加で対応するが，網羅性はないがプログラム実行で検出すると効率はよい．

5.5. 性能評価シミュレータ における ESPRIT/sim の位置づけ

ESPRIT/sim の位置づけを示すため，表 17に代表的なシミュレータとの差異と相対性能を示す．相対性能は ESPRIT/sim のバイナリ変換アクセラレータの速度を 1 としたときの値である．その値が小さいほど，ESPRIT/sim の方がより高速であることを意味する．比較が困難なものは≒の記号で目安として値を示している．パイプラインシミュレーションを目的としたシミュレータとの性能比較は公正でないが，位置づけを明確にするため敢えて括弧付きで記載している．1 列目の“シミュレーションの粒度”は複数の粒度を持つものでは，性能の比較に使えるレベルのものを代表して記載した．なお，11ページに示した図 2と図 3もこれらの関係を示している．

この表からも分かるように，ESPRIT/sim は異種マルチプロセッサのシミュレーションが可能な数少ないシミュレータであり，その中でもバイナリ変換機能を持ったユニークなシミュレータである．加えて，ESPRIT/sim の実装の個々の手法は独創的ではないが全く独自に研究開発されたものである．

表17 代表的なシミュレータと ESPRIT/sim の比較

シミュレーション粒度	シミュレータ	記載	文献	異種マルチプロセッサ	同種マルチプロセッサ	複数種レガシーISA	複数種ホスト ISA	異種アーキテクチャ間	異種エンディアン間	フルシステムシミュレーション	バイナリ変換	機能切替え	記述言語	実装容易性	可読性・保守性	ホスト並列動作	高速実行	相対性能(命令レベル)	相対性能(ｷｬｯｼｭﾚﾍﾞﾙ)
	ESPRIT/sim	5.3	--	○	○	○	○	○	○	○	○	○	C++	○	○	×	○	1	1
	同 旧版	5.3	[35]	×	×	○	○	○	○	○	○	○	C	△	△	×	○	1.15	0.05
命令／ｷｬｯｼｭﾚ	Shade	5.1.1	[15][46]	×	×	△	×	△	×	×	○	UK	ｱｾﾝﾌﾞﾘ	×	UK	×	○	≒1	--
	SimpleScalar	5.1.3	[7][34]	×	×	○	○	○	×	×	×	△	C	△	×	×	△	0.1	0.1
	WSS	5.1.9	[32]	×	×	×	○	×	×	×	○	×	C	○	UK	×	○	0.9	0.7
	SimCore/Alpha	5.1.5	[5]	×	×	×	○	○	○	×	×	×	C++	○	○	×	△	0.06	0.1
	SimOS+Embra	5.1.2	[14][29]	×	○	UK	UK	UK	UK	○	○	○	UK		UK	△	○	≒0.5	≒0.2
	SimICS	5.1.3	[36]	*1	○	○	*1	*1	*1	○	×	×	gcc	○	○	UK	△	0.1	0.5
	Shaman	(5.2.3)	[48]	×	○	×	UK	UK	UK	×	×	×	C	UK	UK	○	?	0.1	0.2
ﾊﾟｲﾌﾟﾗｲﾝ	ISIS	5.1.6	[44]	×	○	×	○	○	UK	×	×	×	C++	○	○	×	×	(0.002)	(0.003)
	ISIS-SimpleScalar	5.1.7	[45]	×	○	×	○	○	UK	×	×	×	C++	○	△	×	×	(<0.001)	(0.001)
	BurstScalar	(5.6)	[31]	×	×	×	○	×	×	×	×	×	C	UK	UK	×	×	(0.003)	(0.01)
他	Mambo	5.1.8	[49]	×	○	△	○	○	○	○	×	UK	C	UK	UK	○	×	0.05	UK
	SystemC	5.1.10	[42]	○	○	○	○	○	○	○	×	×	C++	○	○	UK	×	<0.01	UK

○：可能，×：不可，△：課題あり，UK：不明，Shaman の速度性能には 16-CPU 構成時の最高性能を使用，*1：文献では異種マルチプロセッサ対応への記載はないが，Virtutech など商用に SimICS が使われて実現されている

5.6. ESPRIT/sim と関連研究との差異

ESPRIT/sim とそれに搭載したバイナリ変換アクセラレータは，5.5で挙げた従来研究を参考にするあるいは技術交流することなく発展してきた．そのため，手法の比較は単純に行えないが，個々の原理や目的には数多くの共通性がある．本節では，ESPRIT/sim の新規性を示すため，関連研究との差異を明確にする．

5.6.1. 異種マルチプロセッサシミュレーション

表 17に示すようにもともと異種マルチプロセッサに対応している性能評価シミュレータはなく，SimICS を受け継いだ Virtutech 社などの商用のシミュレータが対応しているのみである．マルチプロセッサに対応するにはプロセッサやキャッシュメモリのインスタンス化が必要となり，異種マルチプロセッサでは C++の仮想関数などの手段を用いた共通インターフェイスによる呼出しが必要となる．この対応が同種プロセッサと異種プロセッサの大きな違いとなる．ISIS や SystemC も C++を使っているため仮想関数を使っているが，ESPRIT/sim では processor_element，isa という基本クラスの派生したクラスを用いてプロセッサ定義を行ない，proc_*の関数から isa_*の関数を呼び出すなど関数の呼び方に新規性がある．

5.6.2. 同種マルチプロセッサシミュレーション

SimOS，SimICS，Mambo，Shaman，ISIS，ISIS-SimpleScalar はマルチプロセッサに対応しているが，他は対応していない．これらの中でバイナリ変換に対応しているのは SimOS の Embra のみである. Embra はレガシ ISA とホストに制約がある. Embra はコード変換キャッシュを物理アドレス対応で持ち，プロセッサ間で共有をしている．一方，ESPRIT/sim はコード変換キャッシュをプロセッサごとに持ち論理アドレスでアクセスし，プロセッサ間でコード変換キャッシュの影響がないようにして高速化と簡易実装を図っている．これも 2 次キャッシュメモリの容量が大きい最近のマイクロプロセッサの特性を活かした簡易実装であるところに新規性がある．

複数プロセッサのシミュレーションを単一のホスト CPU で行う方法について述べる．ISIS と ISIS-SimpleScalar はクロックごとに評価対象を切り替えている．SimICS の切換え方式は不明である．SimOS と Embra も不明であるが動作原理と parallel Embra の存在からマルチスレッドを使用していると推測している．その他はマルチスレッドである．ESPRIT/sim はシミュレーション結果の再現性を重視しており，単一のホスト CPU 用にはマルチスレッドを用いたマルチプロセッサシミュレーションは行っていないし，将来も行わない予定である．バイナリ変換動作では一定命令数を使用して，解析用シミュレータでは一定クロック数を使用してシミュレーション評価を切り替えている．この方法の新規性については不明である．

5.6.3. 複数種レガシーISA

性能解析を目的としたバイナリ変換では, Shade はレガシー命令セットを Sparc と MIPS に，ホストは Sparc に限定しており適用対象の範囲が狭い．SimOS は Sparc と MIPS に対

応しているようであるがそのほかの ISA については不明である．ESPRIT/sim ではバイナリ変換を行っているがレガシーアーキテクチャに対する制約はなく，M32R，PowerPC と PISA に対応した実装を行っている．バイナリ変換を持たないシミュレータでは，SimICS が複数種に対応している．Mambo は Cell と PowerPC の組合せに限定されている．SimpleScalar は複数種のレガシーに対応しているが排他的であり，複数レガシーに対応した Ver.3.0 では Ver.2.0 に比べて性能劣化している．ESPRIT/sim はほかに MIPS32, MIPS64, SH4，ARM（Thumb 命令と割り込みは未実装）などにも対応し，エンディアンの混在も可能である．このように適用できるレガシーISA に汎用性があり，更にバイナリ変換に対応しているところに新規性がある．

5.6.4. 複数種ホスト ISA，異種アーキテクチャ間，異種エンディアン間

Shade はホストを Sparc に限定し対象範囲が狭い．SimOS も Sparc 以外のホストに関する記述の文献が見当たらず同様と思われる．バイナリ変換機能を持たないシミュレータに限定すると，gcc でコンパイルできるものはホスト依存性が小さく一般的に複数のホストに移植できるといえる．

動的バイナリ変換を行うシミュレータでは，ESPRIT/sim 以外は複数種のホスト ISA に対応してない．

バイナリ変換を行わないものでは SimICS があるが，アセンブリ言語レベルの実装や gcc の方言など制約ある．SimpleScalar の PISA は gcc 以外もサポートしており移植できるホストは多いが，Windows では Cygwin 環境が必須であり Cygwin でも動作できないアプリケーションもある．レガシーISA が PowerPC の場合は AIX のホスト，Alpha の場合は OSF のホストが必要など制約が大きい．また，ホストとレガシーのエンディアンが異なると動作できないという課題がある．

これらに比べて，ESPRIT/sim はレガシーとホストのそれぞれに対して，命令セット，エンディアン，OS の違いを分離し必要に応じた変換を施して解決している．トランスレータに 3 階層の構造を用い，レガシーモデルの記述に C++のクラスを使用したことによりそれらの違いの分離を可能にしている．このようにホストの制約がなく[†30]それを実現している機構に新規性があるといえる．

5.6.5. フルシステムシミュレーション

SimOS，SimICS，Mambo はフルシステムシミュレーションに対応しておりアドレス変換機能を持っている．SimOS はアドレス変換用に全空間を格納する巨大なテーブルを使用しているためワークロードの大きなシミュレーショでは性能が低下する．SimICS は 6 種類，合計 24Kbyte と小さいハッシュテーブルを用いて変換をしている．ESPRIT/sim は OS の評価よりも組込みシステムを対象としているため，リード，ライト，実行の 3 種類に限定してより変換表の切替え処理を簡素化している．SimICS との大きな違いは，命令のアドレス変換がほとんどバイナリ変換用の CCD により処理されることである．

5.6.6. バイナリ変換の簡易実装

バイナリ変換は，開発に膨大なコストが掛かる．特に市販マイクロプロセッサのユーザが性能解析のために開発コストを掛けるのは合理的ではない．ESPRIT/sim は最適化レベルを限定した簡易実装により，開発コストが安く安定動作が可能なバイナリ変換機能を提供している．バイナリ変換の手法を詳細に述べた文献は少ないが，Shade や文献[39]のバイナリ変換は，コード変換キャッシュの構造の複雑化のほかに，レガシー命令間に渡った最適化，ブロック間のチェイン，ブロックの並び替えや分岐の入れ替えを行いアルゴリズムを複雑にして投資効果を悪化させている．ESPRIT/sim は，命令間の最適化は原則的に行わないことにして簡単なテストプログラムで検証カバレッジを確保をしている．更に ESPRIT/sim は，ブロック間のチェインなども行わずに，唯一ユニプロセッサ向けに CCE 内での分岐最適化に留めている．

5.6.7. 複合型シミュレータ

SimpleScalar は，シミュレーション対象の粒度別に sim-fast，sim-cache，sim-outorder のシミュレータを持っている．それらは，目的別にカスタマイズするために個別のメインルーチンを持ち，コンパイル時に不要な変数をコンパイラが除去することを期待した最適化により高速化を実現している．そのため可読性と拡張性に欠ける．また，マルチプロセッサに対応できず，同一プログラムモジュールに複数アーキテクチャや複数シミュレータを統合することができない．またバイナリ変換動作による高速化機能もない．SimpleScalar は, あらかじめ指定した命令数まで sim-fast を使いそこでダンプを行い, 次に sim-outorder を起動してダンプデータをロードをして続きを実行するという使い方をしている．

SimOS は命令実行をするフロントエンド部と詳細シミュレーションを行うバックエンド部に別れている．そのため，フロントエンドとバックエンドは疎結合になりバックエンドからフロントエンドへのフィードバックが難しい．

ESPRIT/sim は 1 つのプログラムモジュールに複数のコンピュータモデルを搭載可能であり，ダンプデータを必要としない．また，粒度の異なるシミュレータ間の行き来が自由である．

5.6.8. C／C++言語実装

SimpleScalar と SimICS は，C 言語実装のインタプリタ方式の命令フロントエンド処理を採用しており，ESPRIT/sim の C++と類似している．しかし，C 言語実装はモデルのバリエーションへの対応に限界がある．

SimCore/Alpha は，シミュレータのモデル記述に C 言語の代わりに C++を用いて可読性向上をねらっている．SimCore/Alpha はデコードの再利用によりオーバヘッド改善をしているがそれでも命令シミュレーションの CPI が 100 を超えており，C 言語に比べてその処理速度はかなり遅い．ESPRIT/sim も C 言語から C++に移植して評価し，間接アクセスと

---

†30 制約はないといっても，16 ビット系のホストや浮動小数命令がないホストは対象としていない．

レジスタ消費による性能低下が 3 割ほどあるという経験例[35]を得ている．C++のクラス化された関数や変数はコンパイル／リンク時ではなくロード／実行時にアドレスが決まるため間接アクセスが多くなる．それに対して ESPRIT/sim のように C++に動的バイナリ変換を適用すると変数や関数への直接アクセスが可能となりオーバヘッドを削減できる．

複数種 ISA のプロセッサを複数個搭載可能なシミュレータでは，プロセッサ，キャッシュメモリ，バス，メモリなどの自在な組合せが必要となり，C 言語による記述方式はモデルの可読性，拡張性，保守性が悪化し C++記述が有効となる．ISIS，ISIS-SimleScalar は異種プロセッサに対応した実績はないが，C++の特性を活かしその潜在性を持っている．しかし，それらの研究対象は主にプロセッサ間のネットワークやスイッチにあり，クラス化の単位などの実装手法と性能に対する配慮のポイントが ESPRIT/sim とは著しく異なっている．

ESPRIT/sim の C++の活用の仕方，その性能低下を緩和する手法などに新規性がある．

5.6.9. インタプリタの構造

C 言語を使って実装されたほとんどのインタプリタはその構造が明らかではないが，そのほとんどが switch～case 文によるデコードと命令処理を実装しているものと考えられる．少なくてもソースコードが公開されている SimpleScalar，SimCore/Alpha は，そのような構造になっている．そのため，インタプリタ用のコードを流用したバイナリ変換の実装も困難である．ESPRIT/sim はインタプリタで2.3.1.3（34ページ）で述べた function 方式を使用しているため，バイナリ変換したバイナリコードからインタプリタの命令処理関数を呼び出すことができ，簡易実装を可能にしている．

5.6.10. コード変換キャッシュの構造

Shade は，FIFO 構造の変換済み命令バッファ（TC）とセットアソシアティブ方式のディレクトリを持つ．ESPRIT/sim は固定長のコードエントリ（CCE）とダイレクトマップのディレクトリを使用している．そのため ESPRIT/sim は，2 次キャッシュ容量が大きくパイプラインが深い最近のプロセッサに適しており，コード変換キャッシュのヒット時の検索が速い特長を持つ，また，この方式は動作を単純化するためデバッグコストが安価となり簡易実装に適している．

マルチプロセッサの動作が可能な Embra は，バイナリ変換されたコードを格納した TC を物理アドレスで管理して CPU 間で TC を共有するため，アドレス変換処理と共有制御が複雑化している．一方，ESPRIT/sim は，コード変換キャッシュの制御を簡単にしつつ高速化が可能なように CPU ごとに論理アドレスに対応したコード変換キャッシュを持たせて，これもキャッシュメモリ容量の大きい最近のホストの特性を活かしている．

これらの簡易実装は，性能向上効果があるもので新規性がある．

5.6.11. 動的バイナリ変換以外の高速化手法

動的バイナリ変換の代わりにシミュレータの C 言語ソースコードを生成する高速化手法を採る WSS [32]は，ホストアーキテクチャの知識が不要かつシミュレータ開発が容易で，特に評価に時間が掛かるプロセッサアーキテクチャの研究分野では高速性能が見込めるため有効である．しかし，静的変換固有の課題があり，それに加えてコンパイラの扱えるプ

ログラムサイズの問題，コンパイル時間の課題，マルチプロセッサ化困難などの課題がある．

SimICS が使用している C 言語とスレッデッドコードによる高速化手法 [36]は，複数のレガシーISA に対応でき簡便な手法である．ホストへの依存性が強く，速度性能面ではバイナリ変換への優位性はない．

関連研究ではないが，計算結果の再利用という手法を用いている BurstScalar [31]は，計算結果の保存と検索に時間を必要とするため，パイプライン構成のシミュレーションなど扱うデータ量が膨大でスローダウンがもともと数千~数万と大きな分野では有効である．しかし，統計対象が命令やキャッシュメモリなど小規模な場合には，テーブル参照のオーバヘッドが小さいバイナリ変換の方が有効である．.

5.6.12. キャッシュシミュレーション

バイナリ変換を使ったキャッシュシミュレーションは Embra も行っているが，巨大なハッシュテーブルを使用しているため，ワークロードが大きいと低速になる．SimICS はバイナリ変換を使っていないが，ラインサイズが固定長 16byte のフィルタによりヒット時のオーバヘッドを削減している．Shaman と WSS もフィルタを使用している．

ESPRIT/sim もフィルタを使用しているが, SimICS のようにエントリ数は固定ではなく，実行時にラインサイズとセット数が合うように再設定しているため，フィルタの更新オーバヘッドが少なくまたキャッシュヒット時のフィルタでの除去効果が大きい．フィルタの更新と検索処理は C++記述されており，2 次キャッシュ用のフィルタ実行は C++関数から呼び出すが，1 次キャッシュ用には変換したバイナリコードからフィルタ配列の参照は行い更新は C++関数を呼び出して，フィルタ処理のオーバヘッドを軽減している．

5.6.13. フロントエンドとバックエンド

SimOS と Shade は，メモリ上に吐き出されたトレースを使って疎結合された別シミュレータが詳細解析を行う．SimpleScalar の各種シミュレータはフロントエンドとバックエンドという区別がないが，単一のシミュレータモジュールに複数種のシミュレータコアは搭載されていない．SimICS を除くそのほかのシミュレータも同様である．それに対し，ESPRIT/sim はすべてオンザフライで解析を行い，各シミュレータコアは密結合され同じリソースを共有している．

5.7. まとめ

本章では，性能評価目的のシミュレータとバイナリ変換の関係を従来研究の視点で述べるとともに，本研究のバイナリ変換アクセラレータの実装対象である ESPRIT/sim について概要と目標を述べた．

5.1では，代表的なシミュレータの特徴を紹介した．5.2では，性能評価目的のシミュレータとバイナリ変換の関係をまとめた．動的バイナリ変換が最も有効な高速化手法であり，エミュレータ用途のバイナリ変換より実装が容易であることを述べた．5.3では，

ESPRIT/sim に関して，その機能，構成，内部構造を簡単に述べた．5.4では今後，必要とされる異種マルチプロセッサ向けの性能評価シミュレータの課題とバイナリ変換アクセラレータの目標を示した．

5.5では，従来研究と ESPRIT/sim の機能や性能の関係を表にまとめた．ESPRIT/sim は複合型のシミュレータで，フルシステムシミュレーションやマルチプロセッサに対応した広範囲の機能を持っている．その実装はホストを限定せず，C++記述を用いて可搬性と拡張性を備えている．更に，異種マルチプロセッサに対応し高速実行が可能な点がユニークである．また，5.5の表 17（84ページ）には，これらの従来研究のシミュレータの速度性能を ESPRIT/sim を 1 とした相対値で表し[†31]，ESPRIT/sim が高速なシミュレータ Shade にも遜色なく他のシミュレータより高速であることを示した．

また，5.6では，関連研究の手法と ESPRIT/sim の手法をその有効性の観点から議論を行い，それらとの差異を明確にした．

[†31] この先, 7.3と8.3で示す ESPRIT/sim のシミュレーション速度性能を先取りしてこの表に反映している

第6章 性能評価シミュレータの命令レベル実行高速化方式の提案

前の第 5 章では本章の導入として，性能評価シミュレータの従来研究，ESPRIT/sim の概要，従来研究に対する位置づけと差異などを述べた．本章では，異種マルチプロセッサシステムのシミュレーションに適したバイナリ変換アクセラレータの方式を提案する．

前章の5.4で掲げた目標達成のため，性能評価シミュレータ ESPRIT/sim にバイナリ変換アクセラレータを搭載するにあたり，6.1では本研究の着眼点について述べる．次に，それらをシミュレータを実現する設計方針として6.2で挙げ，その各々の項目の提案内容を6.3～6.8で説明する．

6. 1. 動的バイナリ変換における本研究の着眼点と新規性

本研究は，性能の評価や解析を目的としたシミュレータの高速化としてバイナリ変換アクセラレータを研究したものである．本節では，まず本研究の着眼点を述べ，次に5.6で個々に述べた関連研究との差異を新規性としてまとめる．

関連研究では，実現しているバイナリ変換で使用した性能を向上させる工夫について述べられているが，それらの投資効果やトレードオフについて言及されていない．また，設計品質を確保するときに設計の簡素化が重要となるが，そのトレードオフも示されていない．本研究では，性能向上への投資効果を上げるため，バイナリ変換の対象とする命令種の効率化，コード生成の最適化，対象とするレガシーISA やホストの種類に依存しない共通化，マイクロプロセッサの進歩による価値観の変化に着目した見直し，マルチプロセッサを対象としたとき固有の工夫，について順に着眼点を述べる．

まず，バイナリ変換の対象とする命令種の効率化について考える．命令の出現頻度は，基本的な命令ほどまた機能が単純な命令ほど実行頻度が高いことが知られている．そこで次のようにブレークダウンする．

- 実行頻度の低い命令のために，バイナリ変換の開発コストを掛けるのは効率が悪いため，バイナリ変換の対象を基本的な命令に限定するのがよい．
- PowerPC の bc*X*命令を例にとると，命令フィールドを組み合わせてより高機能な命令を形成している．しかし，実際によく使われる命令の組合せは限定されている．そのため，複雑な機能を持つ命令でもその一部の機能についてのみを，バイナリ変換の対象を基本的な命令に限定するのがよい．
- 基本的な命令や単純な命令をバイナリ変換の対象とし，それ以外はインタプリタ用の関数を呼び出すようにすればバイナリ変換の開発コストを削減できる．これは，インタプリタを第 3 章で提案した function 方式で実装すれば新たな開発コストの増加要因にはならない．
- バイナリ変換の対象であるレガシーISA を使用しているシミュレーションのユーザが増えるかまたは処理速度向上への要求が増えたら，その時点で変換対象とする命令種

を追加すればよい.

次に，バイナリコード生成の最適化について考える．バイナリ変換はインタプリタに比べて数倍から十倍ほどの性能向上が可能といわれている．その幅は広いため数倍と十倍の途中に丁度よいトレードオフがあると予想でき，次のようなトレードオフを考える.

- バイナリ変換を用いて高い性能を目指す場合は，レガシーISA のレジスタをホストのレジスタに割りつけるのが定石である．この方法を使用するとレガシーレジスタの読出し操作が不要となり，ホスト命令のコード削減と処理実行の高速化が見込める．しかし，CISC から RISC へのアーキテクチャ転換時にレジスタ本数を増やして移植を容易化したときのようにレジスタ数を増やせるわけではない．したがって本質的にはレジスタ本数不足は解消しないため，性能向上に寄与するが十分な性能向上にはならないと考える．また，上記のインタプリタ用の関数を呼び出す手法との両立が困難となる†32．処理速度という点では，メモリ配列からレガシーレジスタの読出しを行わずに，ホストのレジスタ間のコピーを行えばほぼホストのレジスタへの割付けと同等のバイナリ変換の処理性能を実現できると考える．スーパスカラのホストを使用するとその可能性は更に高くなる．そこで，メモリ配列をレガシーレジスタの本籍として結果の書込みはメモリ配列に行うが，読出しにはホストレジスタに残った値を再利用すれば，リードアフターライトと呼ばれるパイプライン干渉を回避できる.
- 1.2.4で既に基本ブロックと拡張ブロックについて説明した．その基本ブロックを跨ぐようなバイナリ変換は行わないようにする簡素化は，開発投資をミニマムにする方法の１つである．しかし，開発経験から拡張ブロックレベルまでの範囲を対象としたバイナリ変換が必要と考える．一方，他の分岐ブロックまたはその分岐ブロック内へ直接に分岐する“チェイニング”を行うと，バイナリ変換されたホストコードの流れに合流が生じてしまい，変換処理が一層複雑化する．したがって，そのようなチェイニングはしない方がよい.
- シミュレータを使用するプロジェクトのニーズに合わせて，バイナリ変換アクセラレータの開発投資を増減するのは自然である．したがって，レガシーISA またはホストごとに変換の適用レベルを設定して，ニーズに合った変換レベルを実装すればよい.

次に，対象とするレガシーISA やホストの種類に依存しない共通化について考える．異種マルチプロセッサの混載シミュレーションを実現するためには，シミュレータのコア部がどのレガシーISA のプロセッサモデルにも依存しないようにインターフェイスを規定する必要がある．また，シミュレータが複数のホストにも対応するためには，レガシーISA とホストの種類の組合せの影響を受けるべきではない．シミュレータに実装すべき処理の種類が，レガシーISA とホストの種類の掛け算にまで増えるとシミュレータの拡張性と保守性に問題を生じる．したがって，レガシーISA およびホストに対応した実装は，シミュレータが標準機能としてサポートする処理に単純にマッピングできるようにして，レガシーISA やホストに依存したカストマイズはできるだけ避けるべきである．そのような観

†32 原理的にはインタプリタ用の関数を呼び出すときに本籍となるメモリ配列に書き戻せばよいが，その制御は複雑となりまた，呼び出しが多いと性能も劣化する.

点から，次の点に着目して 3 階層構造のトランスレータが有効と考えた．

- レガシー命令と別のレガシー命令に渡る最適化を行わなければ，バイナリ変換のレガシーISA への依存部およびホストへの依存部の排除が容易となる．
- レガシー命令を，オペランドの読出し，演算，結果の格納を単位としてその生成方法を記述すれば，用意する機能も単純化できる．加えて，それらの記述がインタプリタの記述と似ていると，テキストエディタを使ってコードの書き換えを省力化できる．これが提案している 3 階層方式のトランスレータの layer-1 の記述となる．
- 上記の機能をホストの 1 個ないし複数個の命令に単純にマッピングできれば，バイナリ変換のロジックを簡単に実装できる．
- バイナリ変換の実装にはホストの機械命令の知識が必要となる．5.1.9で紹介した WSS のように C コンパイラなどの力を借りてバイナリ生成する方法もあるが，デバッグのためにホストの機械命令の知識は必須となる．そのため，デバッガと相性のよい変換処理を実装することが望ましく，表形式で変換ルールを記述する方法は効率がよくないと考える．変換のロジックを記述したソースコードにブレークポイントが設定でき，そのソースコードから生成されたバイナリコードをデバッグできる構造がよい．これが提案している 3 階層方式のトランスレータの layer-2 となる．
- ホスト命令を 1 個生成する処理は，別のホスト命令を生成する処理と独立であれば，ホスト命令同士の組合せ問題がなくなり開発が容易となる．ホスト命令生成に関する簡素化の課題としては，定数を 1 命令で生成できるか，ゼロ値や符号拡張などのために別の命令を使うように変形した方がよいかなどの選択肢である．このような最適化は，生成する頻度が高い命令では最適化実装をすべきであり，そうでない命令では最適化しない方が検証効率が上がる．したがって，ホスト命令種ごとに処理を別けると効果的と考える．これは提案方式の layer-3 に相当する．
- ホスト命令セットには複雑な仕様を持つ命令がある．そのような命令を生成するのは，その仕様の理解も含めて効率が悪い．通常は同等の機能を単純な命令の組み合わせて実現できることが多く，それを利用するのは合理的である．layer-2 レベルでその組合せは記述できる．

次にマイクロプロセッサの進歩により，バイナリ変換の実装方法の得失も変わってきていると考える．特に，パイプラインが深くなったため分岐予測ミスの影響が大きい．また，2 次キャッシュメモリの容量が大きくなりその効果を利用して，条件付分岐を減らす実装がよいと考える．なお，従来研究の Shade や SimOS とは開発時期が異なり同じ評価ベンチマークや同じホストを入手できないため直接に比較はできないが，現在のベンチマークやホストを用いてこの着眼点の効果は確認できる．これは，次のようにブレークダウンする．

- ホストの分岐予測ミスを減らすとともに分岐ペナルティサイクル数を減らすには，バイナリ変換のランタイムの分岐を極力減らすことが効果的である．そのためには，セットアソシアティブ形式の表よりむしろ一回の分岐で判定できるダイレクトマップ形式の表探索が効果的である．ダイレクトマップの欠点はハッシュエントリが競合することであるが，空きエントリを考慮して大きめのエントリ数にし，その大きさを 2 次キャシュメモリより溢れない程度にすればよい．

次に，マルチプロセッサを対象としたとき固有の工夫について考える．ユニプロセッサ

シミュレーションでは気に掛ける必要はないが，マルチプロセッサシステムのシミュレーションにバイナリ変換を適用するときに簡素化すべき次の 2 項目を挙げる．

- マルチプロセッサ動作では，プロセッサ間の同期のためにメモリデータをポーリングすることがある．そのプログラミング手法として後方への分岐を使ったプログラムループが使用されることが多い．したがって，その動作をシミュレーションするバイナリ変換では，分岐をチェイニングする場合に分岐の繰り返し回数を制限するなどの工夫が新たに必要となる．そこでチェイニングを行わない簡素化が有効と考える．
- Embra ではコード変換キャッシュを複数のレガシープロセッサ間で共用しているが，次の理由によりプロセッサごとに専用に設ける方がよい．
    - 異種マルチプロセッサでは，コード変換キャッシュの共用は実行対象のレガシー命令が異なるので意味がない．
    - 同種マルチプロセッサでも，SMP 構成で同一のコードを実行する場合にはコード変換キャッシュの共用の効果があるが，AMP と呼ばれる非対称構成ではその共用効果がない．
    - コード変換キャッシュを共用した場合には，論理アドレスが同じでもアドレス変換後の物理アドレスが異なる場合があるため，Embra では物理アドレスでコード変換キャッシュを管理している．その場合には，コアループ部が論理アドレスから物理アドレスへの変換を行うかまたはプログラムカウンタを 2 個持つ必要があり，いずれも処理オーバヘッドが増える．
    - x86 をホストにした場合には，オペランドに 32 ビットのアドレスフィールドを命令に持つことができ，レガシーレジスタのホスト上のアドレスを計算した結果を格納できる．しかし，共用するとプロセッサごとに異なるレジスタの実体を表現できなり，インデックスレジスタを使ったアクセスが必要になるなど，少ないレジスタ本数で高速にアクセスできる長所が失われるので共用しない方がよい．

    一方，プロセッサ個々に専用のコード変換キャッシュを持たせた場合には，メモリやキャッシュメモリを圧迫する懸念がある．これについては最近の PC では搭載メモリ容量が大きく，また 2 次キャッシュメモリの容量も大きくなっているため課題にはならないと考える．したがって，プロセッサ数が高々数個の組込みシステムを主な対象とする本研究のような分野では，コード変換キャッシュの専用化は長所となっても短所にはならないと考える．

ここからは，視点を変えて設計品質の確保の観点から考えてみる．実装量の増大による品質の劣化，レガシー命令の組合せ，ホスト命令の組合せ，実行履歴に依存した障害，の順に考える．

まず，実装量の増大について考える．レガシーISA とホスト ISA の組合せに応じて検証コストが掛かるような設計をすべきではない．レガシーISA-A に対し α と β の 2 つのホストのデバッグを行ってそれらの検証がパスすれば，別のレガシーISA-B に対してはデバッグと検証は一方のホストのみで済むのが理想である．これは既に述べた 3 階層構造のトランスレータで実現できる．なお，実際には A では使用されない部分などが B で初めて出現

する場合のような試験カバレッジに関する差異のため，若干の例外はある．

次に，生成対象のレガシーISA の命令列を構成するレガシー命令相互の組合せに依存した障害について考える．レガシー命令間の組合せに依存した障害が発生しないような構造を採ると，レガシー命令ごとの単体検証で全体の検証がカバーでき，検証コストは下がる．それができないと複雑な組合せ試験が必要となる．それゆえ，レガシー命令間に渡るバイナリ変換の最適化は行わない方がよいことになり，3 階層構造のトランスレータの着眼点と一致する．

次に，生成結果として現れるホスト命令列内のホスト命令相互の組合せに依存した障害について考える．ホスト命令間の組合せに依存した障害が発生しないような構造を採ると，ホスト命令の種類ごとの検証でカバーでき，検証コストは下がる．それができないと複雑な組合せ試験が必要となる．それゆえ，ホスト命令間に渡るバイナリ変換の最適化は行わない方がよいことになり，これも，3 階層構造のトランスレータの着眼点と一致する．ただし，既に述べたレガシーレジスタの読出の代わりにホストレジスタの値を再利用する実装とも関連するため，統一した手続きを用いてホスト命令間の組合せの問題を軽減する．

次に，実行履歴に依存した障害について考える．コード変換キャッシュの履歴に依存した障害は，その再現と解析に時間が掛かる．そのような障害を撲滅する検証には，ランダムな試験を用いてインタプリタとバイナリ変換の実行結果の比較[3]するなどがあるが，開発コストは増大する．したがって次に示すような簡単化が要る．

- 変換後のバイナリコードの格納場所である CCE を固定長にすれば，CCE の内容はその分岐ブロックのアドレスにのみ依存することになる．また，ディレクトリ部の CCD をダイレクトマップにすることにより，他の CCD の履歴に依存することもなくなる．更に，変換の閾値をゼロに設定し実行するコードは必ずバイナリ生成されるようにする試験モードを使用すると，これも他のプログラムアドレスやプログラムループなどの実行回数への依存もなくなり試験の収束を加速できる．CCE の固定長化は既に述べた分岐予測ミスを減らす着眼点とも一致する．
- マルチプロセッサ構成にて，コード変換キャッシュをプロセッサ間で共用すると，他のプロセッサが実行中のプログラムの影響を間接的に受け，障害の再現がしにくくなる．また，コード変換キャッシュの置換や無効化による障害をなくす明示的な試験も難しい．プロセッサごとに専用のコード変換キャッシュを持てば，マルチプロセッサに固有のバイナリ変換の難しさはほとんどなくなるといえる．これも，既に挙げている着眼点と一致する．

以上のように，簡単な実装を用いて検証コストを抑えつつも，バイナリ変換の長所である高速性能を得ることができると考える．

次に，本研究の新規性についてまとまる．5.6（85ページ）では項目ごとに関連研究を挙げて ESPRIT/sim との差異について述べた．そこで述べたことをまとめて一言で新規性として表現するならば，“異種マルチプロセッサシミュレーションを高速に実現できること”である．その技術要素は，動的バイナリ変換を異種混載のマルチプロセッサに適用できることであり，そのために拡張性と保守容易性を維持したまま，安価な開発コストで，高い速度性能を実現したことである．そのために，次の6.2の方針として述べる施策（1）～（4）と（6）のその具体的な実現方法に新規性がある．

6.2. 動的バイナリ変換アクセラレータの設計方針

前章の5.4で，異種マルチプロセッサ向けのシミュレータの目標として，シミュレーションの速度性能，拡張性と保守性，シミュレータの検証コストを挙げた．これらは相反する要求事項であり，ESPRIT/sim では，それらを実現するバイナリ変換方式を提案する．ESPRIT/sim の設計方針を下記に列挙する．

(1) インタプリタとバイナリ変換の併用： 使用頻度が高い命令のみバイナリ変換対象とし，他はインタプリタ用の関数（InsFnc）を呼び出して使用する．

(2) バイナリ変換のレベル付け: 開発コストに見合った変換レベルの選択が可能なトランスレータにする．

(3) トランスレータの階層化： トランスレータは 3 階層にして，レガシー命令セット数（L）とホスト命令セット数（H）に対して L×H ではなく L+H の組合せで実現する．

(4) 構造の単純化による検証コストの削減： レガシー命令間に渡った変換の最適化は行わない，コード変換キャッシュは固定長にするなどの単純化を行う．

(5) アドレス空間モード： フラットモード，マップドモード，アドレス変換モードの 3 種類のアドレス空間を選択可能として多様な目的に対応する．

(6) マルチプロセッサシミュレーション方式： 複数個の CPU の並列動作に対応した簡易的かつ高速なバイナリ変換動作を実現する．

次節より，それぞれの項目に関して述べる．

6.3. インタプリタとバイナリ変換の併用

動的バイナリ変換は，単純かつ高頻度の命令を変換の対象とし，複雑または低頻度の命令はインタプリタの実装を流用した簡易実装とする．これによる性能への影響を述べる．

まず，トランスレータに全命令分の変換が用意されていると仮定する．インタプリタの 1 命令実行時間 T_{itp} を基準値 1 とし，バイナリ変換後の実行時間を T_{bin}，コード変換の実行時間を T_{tr}，コード変換キャッシュのミス率を P_{miss} とする．全体の処理時間 T_{all} は式（4）に近似できる．

$$Tall = (1 - Pmiss) \times Tbin + Pmiss \times Ttr \qquad \text{式（4）}$$

これに，インタプリタの実行割合を P_{itp}，レガシー命令の分岐成立間隔を L_{br}，CC ミス時の変換起動率を P_{tr}，変換されるレガシー命令列の平均命令長を L_{tr}，CC の検査時間 T_{cc} を加味すると式（5）となる．

$$Tall = (1 - Pmiss) \times Tbin + Pmiss \times Ptr \times \frac{Ltr}{Lbr} \times Ttr + Pmiss \times (1 - Ptr) \times Titp + \frac{1}{Lbr} \times Tcc \qquad \text{式（5）}$$

ここで P_{miss}=0.01%，P_{tr}=10%，T_{bin}=0.1，T_{tr}=10，T_{cc}=0.3, L_{tr}=10, L_{br}=7 の場合には次の計算から T_{itp} / T_{all} ＝ 1 / 0.143 = 7.0 でインタプリタの 7.0 倍の速度性能となる．

$$Tall = 0.9999 \times 0.1 + 0.0001 \times 0.1 \times \frac{10}{7} \times 10 + 0.0001 \times 0.9 \times 1 + \frac{1}{7} \times 0.3$$
$$= 0.1 + 0.00014 + 0.00009 + 0.043 = 0.143$$

次に，インタプリタの専用関数（InsFnc）を併用して生成したバイナリコードから呼出し時の速度性能を計算する. デコード処理なしに InsFnc を呼び出して処理する時間を T_{ifn}, 呼出し確率を P_{ifn} とすると式（5）の第 1 項は 式（6）となる.

$$Tall = (1 - Pmiss) \times (Tbin \times (1 - Pifn) + Tifn \times Pifn) \quad \text{式（6）}$$

P_{ifn}=0.1 とし T_{ifn}=0.5 とするとこの項は 0.14 となり T_{all} は 0.183 となってインタプリタの 5.5 倍になる. P_{ifn}=0.1 は全命令種の 10～20%程度の実装に相当すると, この 5.5 倍は式（5）の例で示した 7.0 倍の約 80%に相当することから，全命令種にバイナリ変換を実装したときの約 80%の効果を期待できることになる.

ESPRIT/sim では，使用頻度の低い命令，システムコール，複雑な命令，制御命令はインタプリタ用の関数 InsFnc を活用し更に，不正命令，ブレークポイント対象アドレスはバイナリ実行の対象とせずインタプリタ動作を行う.

6.4. バイナリ変換のレベル付け

開発コストと性能のバランスはレガシー命令セットアーキテクチャ（レガシーISA）ごとに異なるため，ESPRIT/sim ではバイナリ変換の適用レベルをレガシーISA ごとに選択してより簡易的な実装も可能としている．ここで，いくつかのレベルの例を示す．1 つには，変換単位に着目したレベル設定がある.

- いずれの命令もバイナリ変換を行わずにインタプリタ用の関数 InsFnc のみで構成して，インタプリタのデコードと分岐オーバヘッドのみを削減するレベル
- 次に，バイナリ変換の対象を最初の分岐命令までの基本ブロックに留めるレベル
- 更に条件分岐命令後の命令の生成を継続して拡張ブロックの最後尾となる無条件分岐まで生成を繰り返すレベル

別の観点では次のようなレベルがある.

- プログラムカウンタ（PC）値の更新を各命令ごとに行わず PC が参照される命令まで更新を遅延させるレベル
- ゼロを生成する命令や無条件分岐命令などのようにレガシー命令のフィールドの組合からその動作をより単純なホスト命令列に置き換えるように最適化をするレベル
- レガシーレジスタの実体であるメモリ配列からのデータ読出しを減らすため, ホストレジスタに残ったレガシーレジスタの最終値の結果を再利用する最適化レベル

これらのレベルの実装例は，評価と併せて第 7 章（7.5の表 22）で紹介する.

6.5. トランスレータの階層化

命令のバイナリ生成を行うトランスレータは，図 37に示す 3 階層構造を採る．これによ

り可読性を損なわずにコードの共通化と流用性を高めて，トランスレータの実装を簡易化している．以下，各階層の説明としてレガシーISA に SimpleScalar/PISA[8]を，ホストには x86 と PowerPC[9][12]を用いた実装例を示す．なおコード中の UL は unsigned long の省略表示である．

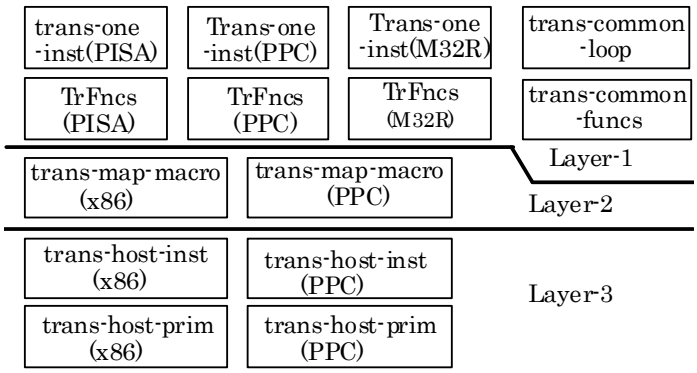


図37 トランスレータの階層

6.5.1. Layer-1

この階層には，レガシーISA に依存しない共通関数として次に示すものがある．

- trans-common-loop：一連の命令列の変換を繰り返す処理
- trans-common-funcs：trans-common-loop に使用する部品としての関数

また，レガシーISA に依存する関数として次に示すものがある．

- trans-one-inst：trans-common-loop に呼び出されるレガシー命令依存の 1 命令を変換する関数
- TrFnc：trans-one-inst から呼び出されるレガシーの各命令を変換する個別関数

trans-common-loop と trans-common-funcs はレガシーISA によらず他のレガシーISA と共通なため，新規レガシーISA の開発コストを削減できる．また，レガシー命令依存の trans-one-inst と TrFnc も他のレガシーISA と記述形式が共通なため他 ISA のソースコードからの流用度が高く，新規レガシーISA 用のこの実装も容易となる．

TrFnc の関数名はレガシーISA に対応し, trans_addu などの名前となる. Layer-1 のコード例を次に示す.

Layer-1: TrFnc の PISA の例

```
bool trans_addu ( ) {
   int rs = (trans_icode >>24) & 0x1F;
   int rd = (trans_icode >> 8) & 0x1F;
   int rt  = (trans_icode >> 16) & 0x1F;
   TR_rGPR(HOST_regA, rs);                  // Read PISA reg
   TR_rGPR(HOST_regB, rt);                  // Read PISA reg
   TR_ADD(HOST_regA, HOST_regB);            // Add
   TR_wGPR(rd, HOST_regA);                  // Write PISA reg
}
```

インタプリタの命令処理関数 InsFnc では rsd = MPC_rGPR(rs)，MPC_wGPR(rd, rslt)などの記述を用いている．そのため，この TrFnc のソースコードはその InsFnc のコードをテ

キスト変換と若干の手修正で改造できる．この記述には，TR_rGPR のような Layer-2 で定義したマクロを用い，レジスタやメモリへのアクセス，各演算などを表現している．このように，インタプリタが呼び出す InsFnc のコードの記述と似ているため，条件判定がなければインタプリタコードの単純な置換で済む．

レガシーISA やホストに依存しないトランスレータにはテーブルドリブンの実装方式が使われることが多い[27]が，本研究の方式は，変換動作のデバッグにデバッガを利用できる点が有利である．なお，命令あたりのレガシーISA 固有の TrFnc のデバッグコストは，インタプリタ方式より一桁から二 2 桁余分に掛かるため，6.3の適用対象命令と6.4の変換レベルに合わせて実装することにより全体のデバッグコストを下げている．

6. 5. 2. Layer-2

この階層で定義されるマクロ trans-map-macro はレガシーISA とホストのどちらにも依存しないマクロ名と引数を持つ．Layer-1 でこのマクロを用いて記述したコードを Layer-3 のホスト依存の関数呼出しに置換するのがこのレイヤーの役目である．このマクロ群はホストの種類数だけ用意される．ホストを追加したときには Layer-3 の tarns-hos-inst とともにこのマクロ群の実装とデバッグが必要であるが，レガシーISA の追加時には不要である．次に，x86 用のマクロと PowerPC 用のマクロの記述例を示す．

Layer-2: trans-map-macro-x86 の例

```
#define TR_aGPR(n)              cpe->regs_ptr+(n)
#define TR_rGPR(r86, reg)       tr86_mov_r_m(r86, TR_aGPR(reg))
#define TR_wGPR(reg, r86)       tr86_mov_m_r(r86,TR_aGPR(reg))
#define TR_ADD(rd, rs)          tr86_add_r(rd, rs)
```

マクロの右辺は，x86 命令に対応した生成関数の呼出しである．cpe はクラス processor_element のポインタである．

Layer-2: trans-map-macro-ppc の例

```
#define TR_rGPR(regPPC, reg) ¥
        trPPC_lwz(regPPC, TR_aGPR(reg), regPTR, (UL *)&cpe->regs)
#define TR_ADD(regPPCd, regPPCs) trPPC_add(regPPCd, regPPCd, regPPCs, 0)
```

マクロの右辺は PowerPC の命令に対応した関数となっている．ただし，trPPC_lwz では，16 ビットのオフセットを指定せずに第 2 引数でアドレス全体を指定し，lwz 命令にはない第 4 引数にてシミュレーション実行時に決まるレジスタポインタの値を渡し，アドレスの範囲チェックを行っている．

次に，レガシーISA とホスト ISA の組合せの課題を回避する例を示す．演算の結果に応じてフラグをセットする命令仕様には，x86 のように符号の有無によらず命令が同じでフラグビットの異なるものと，PowerPC のようにフラグビットが同じで命令が別のものがある．これに対応するために，命令のマクロとフラグ生成のマクロを符号の有無ごとに各々用意している．また，キャリーやオーバフローフラグをレガシーレジスタにセットする命令の生成ではホストのフラグのビットの意味付けやビット位置の違いがあるので，テーブル変換を使って違いを吸収している．

6. 5. 3. Layer-3

この階層には，ホストの各命令に対応した tr86_mov_r_m などの関数（trans-host-inst）

と, それらが呼び出して CC にホストの命令とオペランド定数やディスプレースメントを埋め込む tr86_set_op8 などのプリミティブ関数（trans-host-prim）がある.

Layer-3: trans-host-inst-x86 の例

```
bool tr86_mov_r_m(int reg, UL *addr) {
    if(reg != rEAX ? tr86_set_op16(0x8B05 + reg * 8) : tr86_set_op8(0xA1)
        return tr86_set_data32((UL)addr);
    return false;
}
bool tr86_add_r(int rd, int rs) {
    return tr86_set_op16(0x03C0 + rd * 8 + rs);
}
```

tr86_mov_r_m や tr86_add_r などの x86 命令に対応した関数は，ホストのレジスタ番号や即値範囲に依存したホストコードの最適化も行っており, x86 用には 84 種を用意した. trans-host-prim-funcs には tr86_set_op8/16/24 と tr86_set_data32 を用意し，CC への格納と CC の溢れ検出をしている. 溢れずに CC 格納できたときは変換を継続する. 溢れた場合または無条件分岐命令の変換後は，次のレガシー命令の変換は行わない.

Layer-3: trans-host-inst-ppc の例

```
bool trPPC_lwz(int rd, UL *addr, int ra, UL *ra_val) {
    UL offs = (UL)addr - (UL)ra_val;
    DBT_ASSERT(offs >> 16 != 0xFFFF && offs >> 16 != 0 || offs & 3, "trPPC_lwz");
    return trPPC_op_x_a_imm(32, rd, ra, offs & 0xFFFF);
}
bool trPPC_op_x_a_b_sop_rc(int op, int x, int a, int b, int sop, int rc=0) {
    return trPPC_set_code(
       (UL)op << 26 | x << 21 | a << 16 | b << 11 | sop << 1 | rc);
}
bool trPPC_add(int rd, int ra, int rb, int rc) {
    return trPPC_op_x_a_b_sop_rc(31, rd, ra, rb, 266, rc);
}
```

trans-host-prim には，32 ビットの命令語を格納する trPPC_set_code のほかに trPPC_op_x_a_b_sop_rc など命令形式に対応した関数を 6 個用意した. PowerPC 用の trans-host-inst は 3 行程度の関数を約 100 個で構成した.

6.5.4. 生成されたバイナリ命令列の例

PISA の命令“addu r4,r2,r3”を x86 用と PowerPC 用に変換した例それぞれを示す.

生成されたバイナリ命令列 x86 の例	生成されたバイナリ命令列 PowerPC の例
mov eax, dword ptr [0x07000118] # &cpe->regs	lwz r8, 8(r30) ; rs
mov ecx, dword ptr [0x070011C]	lwz,r9,12(r30) ; rt
add eax,ecx	add r8,r8,r9
mov dword ptr [0x07000120],eax	stw r8,16(r30) ; rd

x86 では変換時に，プロセッサ proc_ss_pisa のインスタンスのメンバであるレジスタ配列の先頭アドレス（&cpe->regs）に該当レジスタのオフセットを足しこんだアドレスを計算しそれを即値アドレスとして命令を生成する. PowerPC では，r30 には&cpe->regs がセットされており，rs と rt の格納されたメモリから読んだ値を足して rd レジスタに書いている.

6.5.5. ホスト依存性の吸収

ホスト依存の課題解決について述べる．ホストのレジスタ割付けは HOST_regA のようにニーモニック記述され Layer-2 の定義により変更ができるが，ホスト x86 にはシフト命令のビット数の間接指定に“CL”レジスタを使用するなど制約がある．Layer-3 はレジスタと演算の組合せによる最適化コードを生成をするが，この例では ECX 以外が割り付けられていると，一旦 ECX を退避する．このような性能低下を防止するため，Layer-3 の関数または Layer-2 で定義した TR_HOST_RESTRICT マクロがホスト制約を検査し，エラーや警告として報告することも可能にしている．

また，ホスト x86 には退避なしに自由に使用できるレジスタ本数が少ないという課題もある．Layer-3 の trans-host-inst 関数はその実行結果を格納するレジスタをワークレジスタにも割り当てるが，複雑なレガシー命令やアドレス変換処理でこれだけではワークレジスタが不足する．そこで Layer-1 で TR_USING と TR_DROP のマクロコールを記述してホストレジスタの保持範囲を明示的に指定可能にしている．Layer-3 の trans-host-inst 関数は，保持対象外の空きレジスタがあればそこから選択し，空きが無い場合は push/pop を使用してワークレジスタを確保し，ホストの空きレジスタを有効活用している．このマクロ記述は機能上は必須なものではなく最適化のヒントであり，レジスタ本数の多いホストへの影響もない．

6.6. 構造の単純化による検証コストの削減

ESPRIT/sim では，レガシー命令 1 個をホストN命令に変換しており，レガシー命令間に渡った個別の最適化はしていない．これにより，レガシー命令間の組合せへの依存性を排除し，検証コストを削減している．

高い速度性能を実現する方法としては，分岐成立時にコアループに戻らずにバイナリ実行の中で CC を探索して次の CCE に分岐する方法が一般的である．しかし，ESPRIT/sim は，分岐情報の記録，間接分岐処理，CC が無効化されたときの対応，障害解析などを単純化するために，5.3の図 36（81ページ）に示すように分岐成立時にはバイナリコードから抜けてコアループ（run_loop）に制御を戻す方式を標準としている．なお，ユニプロセッサ構成時のみ高速化オプションとして, バイナリ実行の中で CC を検査してヒットすれば分岐するように設定できる．また，このオプションの実装の有無は，ホストにより選択できる．

CC は論理アドレスに対応し CPU それぞれ個別に持つ．CC は，ダイレクトマップ方式を採り図 38のようにディレクトリ部 (CCD) とコードエントリ（CCE) 配列からなり, CCE は固定ブロック長にしている．ダイレクトマップにすると CCD がヒット時の分岐が予測ミスが発生せず，高速動作が見込める．CCD は PC 値とコード部へのポインタ（EXE）をメンバとして持つ．CCE の実装には，エントリあたりのバッファを可変長の FIFO 方式としてラップさせる方法 [15]がありそれが性能面では優れているが, CC の管理が複雑化し障害発生時の解析が困難になる．ESPRIT/sim は，ダイレクトマップと固定ブロック長を用いて CC の挙動を単純にし, 障害解析を容易化することで, 動作が複雑な動的バイナリ変換の導入を容易にしている．

なお評価を目的に，2 ウェイの FIFO 方式と 2 ウェイの LRU 方式も実装した．ダイレクトマップ構成では EXE の値は一意に定まる．FIFO 方式では CC の隣接する 2 組をペアとして，新エントリの割付け時には偶数エントリを奇数エントリに退避し偶数に新エントリをセットする．LRU 方式では偶数エントリが“MRU（最新のアクセス)”になるように入れ替えを行う．FIFO と LRU のいずれも CCE の中身は変えずに PC 値と EXE 値を変更している．

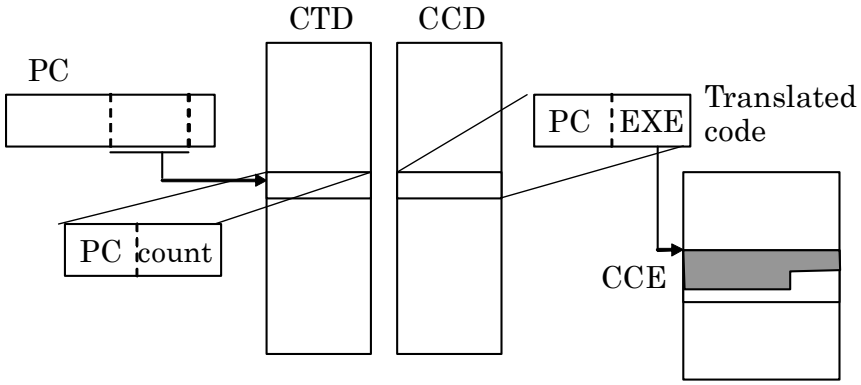


図38 コード変換キャッシュの構造

6.7. アドレス空間モード

ESPRIT/sim は，表 18に示す 3 つのアドレス空間モードを用意して多様なニーズに対応している．アドレス空間は，アドレス変換動作が不要な論理アドレスモードと，アドレス変換をシミュレーションするアドレス変換モードに別れる．論理アドレスモードには更に，連続するメモリ空間を用意したフラットモードと，小さなメモリ空間で 4Gbyte の全空間をアクセス可能なマップドモードを用意している．ESPRIT/sim に搭載した動的バイナリ変換は，各々に対応している．

フラットモードは，ホストメモリの大きさと使い方の制約があるが，アドレス空間の小さいレガシーISA[†33]と評価目的によっては実用的であり，バイナリ変換性能の特性をそのまま表すためには，このモードによる評価を基本とした．

マップドモードのアドレス変換の原理は，SimpleScalar の sim-fast の実装と似ている．バイナリ変換時には，CCE 内のバイナリコードがハッシュテーブルを参照してマッピング済みかどうかの判定を動的に行い未マッピングのときはホストのレジスタを退避後に，C++で記述された関数を呼び出してマッピングを行う．

アドレス変換モードは，SimICS[36]と似たハッシュテーブルによる高速化を行っており，アドレス変換可能かつ保護違反がない場合のみハッシュテーブルに格納する．SimICS は 6

[†33] MIPS や SH4 では 32 ビットのアドレスフィールドに対して上位の 3 ビットをキャッシュ属性に使うなど別目的に使用しており，一般的には 512Mbyte の物理空間しかアクセスできない．

種類のテーブルを持っているが，ESPRIT/sim は命令，リード，ライトの 3 種類と単純化してある．RISC ではプログラム制御により TLB 登録を行う方式が多くそれらの機械命令動作を正確に模擬するため，このモードはシミュレーション対象である TLB がミス状態ではハッシュテーブルと CCD を無効状態に保つように制御している．

マップドモードとアドレス変換モードでは，I/O 領域はハッシュテーブルの登録対象外とし，そこへのアクセス時にはミスを発生させて，バイナリコードからメモリへのアクセスをせずに I/O 領域へのアクセス関数の呼出しを行う．またバイナリ変換で課題となる自己修飾問題[2]には，ライト用と命令用のハッシュエントリを排他的に有効にし，ライト用のハッシュミスをきっかけとして命令用ハッシュエントリの無効化するとともに CCD も無効化して対応している．Shade は，Sparc では命令キャッシュを命令で明示的に無効化していることを利用して，その命令で TC を無効化している．ESPRIT/sim の方法は，レガシーISA やレガシーOS への制約は特に不要である．

表 18アドレス空間モード

	論理アドレス　モード		アドレス変換モード
	フラットモード	マップドモード	
アドレス変換	なし	なし	あり
メモリアドレス計算	不要	ハッシュテーブル変換	ハッシュテーブル＋TLB
メモリアドレス	論理アドレスのみ	論理アドレスのみ	物理アドレス
アクセス可能な領域	シミュレータに実装した容量以下	4 Gbyte 空間	4 Gbyte 空間
用途	基本性能の評価， 組込み機器の小容量メモリ	高性能な CPU を使ったシステム	アドレス変換を使ったシステム

6.8. マルチプロセッサシミュレーション方式

6.8.1. CPU 間のシミュレーションスイッチ

ESPRIT/sim は，processor_element の派生クラスを複数個インスタンス化して，同種マルチプロセッサはもちろん異種マルチプロセッサのモデルを容易に構成できる．シミュレーション対象の CPU のリソース，コード変換キャッシュ（CC）も CPU ごとに独立しているため，ホストのマルチスレッド動作モードを将来追加することも容易である．各 CPU のシミュレーションを N 命令ごとに切り替える方式を採っており，スケジューラ関数が CPU ごとに run_loop を呼び出し，run_loop からは N 命令で抜け，再スケジュールする．その様子を図 39で説明する．この例ではシミュレーショ対象の 2 個の CPU は，図の左側のように並列に動作している．run_loop を切り替えにより，2 個の CPU のシミュレーションが N 命令ごとに交互に行われる．N の数が小さいほど理想状態の動作に近づくが処理速度が低下する．バイナリ変換でこの N の値を確実に守るためには，CC に格納された命令列が動的に命令数を判定するロジックを持つなど，ある程度は性能を犠牲にしないとその

制御は困難となる. そこで, 次の各規則を守ることにより近似的に Nを実現する方式を採った.

- バイナリ生成時に，拡張ブロック中のレガシー命令の命令数が N を超えないように変換量を制限する.
- run_loop 内では実行命令数を加算し，その合計が N以上になったら run_loop から抜ける.
- run_loop 内で 2 回以上バイナリコードを実行すると，N以上の命令を実行することになる．その Nを超えた差分 Kに対し，次回の実行時には N—Kだけ実行するようにスケジュールし，誤差の補正を繰り返す.

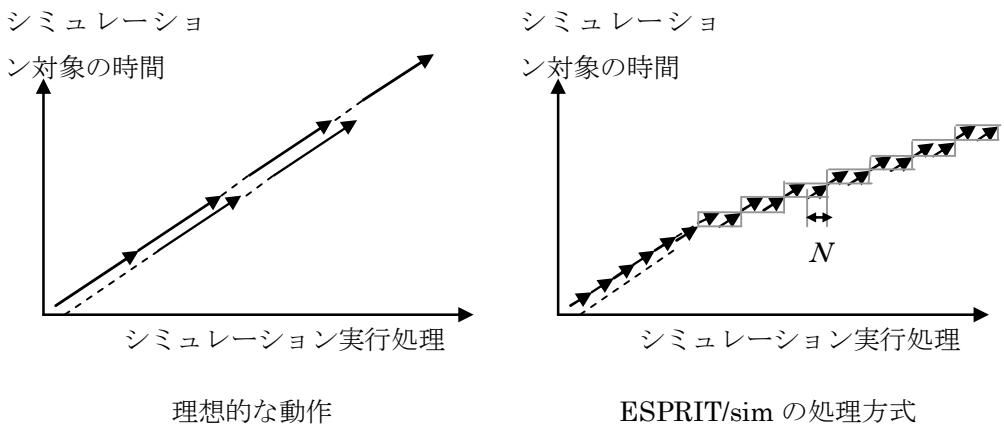


図39 CPU 間のスイッチ方式

6.8.2. コード変換キャッシュメモリの構成

コード変換キャッシュ（CC）は CPU 間で共有すべきか否かという方式の選択がある. ESPRIT/sim は, 異種マルチプロセッサでの共有効果がないこと, x86 のホストでは 32 ビットアドレス値を指定できるためレガシーレジスタなどへのアクセスコードが共通にならないこと，CC の入れ替えが CPU ごとに最適にならないことから CC を非共有とした．非共有の方式はアドレス変換がオンのときでも論理アドレスを索引として CC をハッシュできるため，アドレス変換オンのモードでもオーバヘッドは少ない．短所としてはシミュレーション CPU 数が増えたときにホストのキャッシュミスが増大することであるが, 最近のマイクロプロセッサではキャッシュ容量が増大しているため, 非共有による CC の単純化が有効と判断した.

6.8.3. キャッシュメモリのモデリング

キャッシュメモリには，容量／セット数／ウェイ数の構成，書き戻し方式，置換方式などそのバリエーションが多くモデル化の負荷が高くなる．そのため，モデル作成では特に動作の正当性を保証するのに気を使う．アセンブリ言語記述やバイナリ生成をすればキャッシュメモリのシミュレーションの速度性能は良くなるがモデル作成の効率が悪い. そこで, C++でモデルを記述し, その呼出しを生成したバイナリから行うことにした. また,

試行の結果，C++読出し時のレジスタのセーブ／リストア，C++関数のオーバヘッドが大きいため，ダイレクトマップ型のフィルタを設けてフィルタにミスしたときのみ C++関数を呼び出すようにアクセス方法を改良した．具体的にはこれは，キャッシュに最近ヒットまたはミスしてロードしたラインアドレスをフィルタに記録し，フィルタにミスしたときのみ C++関数を呼び，ヒットしたときは呼出しも記録もしない方式である，この原理は簡単に思いつく手法であり，SimICS [36]と WSS [32]でも使用している．ESPRIT/sim ではキャッシュのセット数 *W* を *W+1* に拡張してセット数と同じエントリ数を持つダイレクトマップのフィルタを用いている．マルチプロセッサ構成のスヌープ動作の結果でキャッシュステータスが変化する場合には，フィルタのエントリも無効化する方式とした．

6.9. まとめ

本章では，性能評価目的のシミュレータの高速化のため，異種マルチプロセッサシステムのシミュレーションに適した動的バイナリ変換アクセラレータの方式提案を行った．異種マルチプロセッサシミュレータ構築上の課題は5.4で述べたように，高いシミュレーション速度性能が要ること，拡張性と保守性の向上が必要なこと，シミュレータの検証コストを安価に抑える必要があることである．それらは相反する内容であるが，6.1で述べた着眼点に基づいて目標として実現するため設計方針を定め，そのうちの 6 項目について6.2で述べた．それらは，インタプリタとバイナリ変換の併用，バイナリ変換のレベル付け，トランスレータの階層化，構造の単純化による検証コストの削減，アドレス空間モード，マルチプロセッサシミュレーション方式であり6.3～6.8でそれぞれ説明した．これらの手法は，異種マルチプロセッサシミュレーションに必要な拡張性を備え，実装コストをミニマムにできかつ投資効果に応じて拡張が可能であり，拡張後も保守コストを抑制できる簡易的な手法でありながら，バイナリ変換による高速性能を得ることができる方式である．

次の第 7 章ではユニプロセッサシミュレーションの速度性能を評価し，第 8 章ではマルチプロセッサシミュレーションの速度性能の評価を行い，これらの提案の有効性を示す．

第7章 ESPRIT/sim のバイナリ変換方式の速度性能評価

本研究では，第 6 章で提案したバイナリ変換方式の有効性を示すため，性能評価シミュレータ ESPRIT/sim にそれらの提案内容を実装して評価を行った．本章では，シミュレーションの速度性能を中心に，ユニプロセッサシミュレーションの評価とその結果を述べる．

まず，7.1では評価方法を，7.2では評価方法の妥当性を述べる．7.3では SPEC CINT95 の速度性能を述べる．次に設計方針として示した内容の効果を具体的に論じる．7.4では実装の共通化の効果を，7.5では変換レベルの選択とその効果，7.6では性能の向上に対する投資効果を述べる．7.7では，バイナリ変換アクセラレータコアの波及効果を述べる．

7.1. 評価方法

バイナリ変換方式はインタプリタ方式に比べてその性能が評価プログラムに大きく依存するため，広い特性を持つ SPEC CINT95[13] （CINT95 と省略）を用いて評価した．CINT95 では多くのシステムコールが使用され，それらを含むライブラリが準備され OS 環境の模擬が容易な SimpleScalar の PISA [7]をレガシー命令セットに選んだ．バイナリモジュールはSimpleScalar 向けに最適化オプション-O2 でリトルエンディアン用にコンパイルされたもの[34]を用いた．124.m88ksim のバイナリには生成誤りがあるため，リトルエンディアンで動作するようにコンパイルし直した．CINT95 の 099.go，126.gcc，130.li，132.ijpeg，134.perl，147.vortex では“train set”を使用した．処理時間が短く計測誤差が大きくなるのを避けるため，124.m88ksim では“test set”を使用した．129.compress にはパラメータとして“400000 e 2231”を使用して，いずれも 100×10^6～3000×10^6 個の命令数となるように処理量を選定した．

ホストには，主な評価用に x86 アーキテクチャの Core 2 Duo を用いた．ホストに依存しない実装であることを示すために PowerPC アーキテクチャの MPC7450 もホストに加えた．プロセッサ周波数，パイプライン段数，キャッシュメモリ，OS，C++コンパイラ，コンパイルオプションを表 19に示す．OS と C++コンパイラはユーザにとってインストールが容易で使いやすいものとして選択した．インタプリタ性能の比較として客観性を増すために SimpleScalar の sim-fast [7][34] と sim-cache [7][34]も両ホストに実装した．なお，VC++ ではこれらはコンパイルできないため Cygwin の gcc 3.3.3 で-O3 オプションを使用してコンパイルした．

アドレス変換モードの速度性能は，シミュレーション対象の TLB の構成とその容量に依存するため，SimpleScalar の sim-cache の標準設定と同じ（4Kbyte ページ，命令 TLB は 16 セット×4 ウェイ，データ TLB は 32 セット×4 ウェイ）とした．PISA にはもともとアドレス変換の仕様がないため，ミス時のアドレス計算の実行と TLB への格納はハードウェアで行う方式を想定した．アドレス変換モードの評価には，動作検証のために PISA の CINT95 のバイナリを論理と物理のページが不一致するようにバイナリモジュールのデー

タを修正して適用した.

表19 ホストマシンの仕様

プロセッサ, コア周波数	パイプライン段数	命令発行数	1次キャッシュ	2次／3次キャッシュ	マシン名	OS, C コンパイラ, 最適化オプション
Core 2 Duo, 1.86GHz	14	4	命令:32Kbyte データ:32Kbyte	2次:2Mbyte	HP xw4400	WindowsXP-SP2, VC++Ver6, /O2
MPC7450, 867MHz	5	3	命令:32Kbyte データ:32Kbyte	2次:256Kbyte 3次:2Mbyte	Power Mac G4	OSX(BSD), gcc3.1, −O3

また, ESPRIT/sim の実装方式が他のレガシーISA にも有効なことを示すため, レガシーISA として PowerPC[9][12]と M32R[11]を追加した. それらの評価プログラムには, システムコールの依存性が低く評価が容易なベンチマークの中から, CCD のヒット率が高い 129.compress (400000 e 2231) とヒット率が低い 099.go (30, 11) を代表例として用いた.

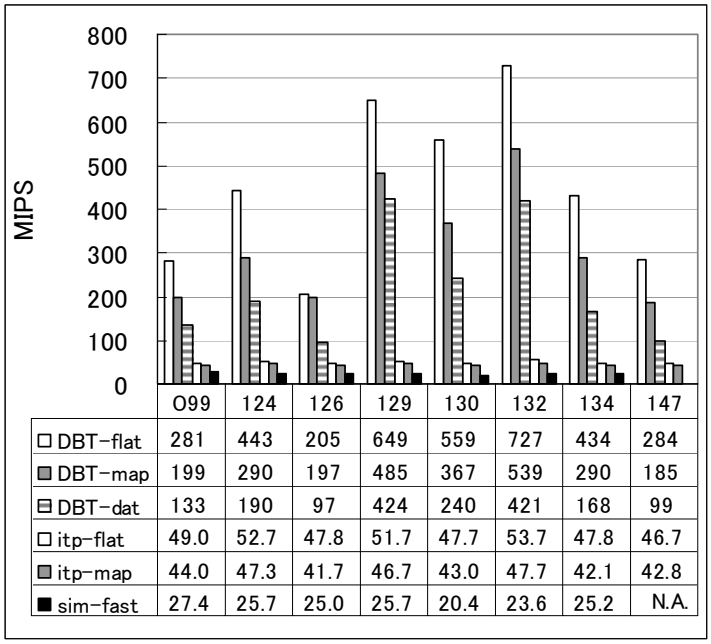
ESPRIT/sim では CTD と CCD のエントリ数および CCE のラインサイズは選択可能であるが, 本測定では, CCE は 800byte, CCD と CTD のエントリ数は 32K とした. CTD や CCD を検索するアドレスの単位は PISA が 8byte, 他は 4byte である. 性能はレガシー命令を 1 秒間に実行できる回数を MIPS 値または相対性能値として使用し 3 回の測定平均を用いた. 複数ベンチマークの平均は, MIPS 値の逆数の算術平均をとりその逆数を使用した. すなわち, 時間平均から求めた MIPS 値である. 以後, これを“平均”は“MIPS 逆数平均”と呼び, 単純に MIPS 値を算術平均したものは“MIPS 値平均”と呼び区別する. またレガシー命令 1 個を実行するホストのサイクル数を CPI とする. ほかのシミュレータとの速度性能を相対比較するときのベンチマークの平均は, これも速度比の逆数の平均の逆数で表し, 時間平均から求めた速度比と同じにする. バイナリ変換レベルは記載がないものは, 既定値として表 22の 8 を使用した.

7.2. バリデーション

インタプリタとバイナリ変換の正当性は, (1) C 言語で記述したテストセット[35]による診断, (2) CINT95 (train set) の実行ログ比較, (3) PISA の命令数は sim-fast と比較, により確認した. なお, sim-fast との命令数の不一致は, システムコール実装 (fstat のブロッキングファクター), とホスト実行環境 (環境変数, ファイルのパス名, 時刻) の違いによるもので, CINT95 の一部のプログラムにてこれらの値の合わせ込み後は一致することを確認した. キャッシュシミュレーションの正当性の確認は, 7.7.で述べるキャッシュメモリのミス回数を sim-cache の 2 ウェイ LRU 構成と比較して行った. ミス回数の誤差は回数で最大 6.7%, ミス率換算では 0.1%以内が発生した. 解析の結果, 命令数の差異と同じ現象であったため, ESPRIT/sim のキャッシュ評価部を sim-cache に追加移植し sim-cache 上でのミス回数を計測し sim-cache の元の回数と同一となることから ESPRIT/sim の正当性を確認した.

7. 3. SPEC CINT95 の性能

SimpleScalar/PISA をレガシーISA に，ホストを Core 2 Duo にして CINT95 を使って評価した速度性能の結果を図 40に示す．この速度性能は変換レベル 9 で測定した結果である[†34]．グラフの縦軸は MIPS 値であり，バイナリ変換モード （DBT-*）として6.7に示したフラットモード（*-flat)，マップドモード（*-map)，アドレス変換モード（*-dat）の各モードの値を示す．比較用のインタプリタには itp-flat と itp-map を，また参考として sim-fast（Ver.3.0）の実測結果も載せている．なお 147.vortex はシステムコールの実装が Cygwin に対応していないため sim-fast は “N.A.” として値を記載していない．また，表20に, 統計機能を活かして採取したバイナリ変換の特性を示す. 6.3で示した Pmiss, Pifn, Lbr のほかに，レガシー命令あたりのホストの平均命令数（Lhost)，全命令中のインタプリタの実行割合（Pitp）も示している．



	O99	124	126	129	130	132	134	147
□ DBT-flat	281	443	205	649	559	727	434	284
▒ DBT-map	199	290	197	485	367	539	290	185
▤ DBT-dat	133	190	97	424	240	421	168	99
□ itp-flat	49.0	52.7	47.8	51.7	47.7	53.7	47.8	46.7
▒ itp-map	44.0	47.3	41.7	46.7	43.0	47.7	42.1	42.8
■ sim-fast	27.4	25.7	25.0	25.7	20.4	23.6	25.2	N.A.

図40 CINT95 の速度性能

[†34]公刊論文 2 では図 5 で変換レベル 8 の値を使い表 5 ではレベル 9 の値を示している．それに対して，ここでは最大性能を示す変換レベル 9 の値を図に使用している．

表20 CINT95 のバイナリ変換特性

program	Pmiss(%)	Pitp(%)	Pifn(%)	Lbr	Lhost
099	0.028	0.181	0.000	10.2	4.2
124	0.001	0.005	0.000	5.8	4.7
126	0.144	0.441	0.030	7.9	4.8
129	0.000	0.001	0.571	6.5	4.9
130	0.003	0.009	0.000	6.7	4.9
132	0.002	0.004	0.027	13.7	4.1
134	0.013	0.013	0.030	7.2	4.6
147	0.037	0.085	0.002	9.6	4.5

フラットモードではベンチマーク間の MIPS 値の差が大きい．これは，1 個の CCE で実行された平均レガシー命令数，分岐が発生して CCD を検索した頻度，CCD がミスしたときのコアループ処理時間とバイナリ変換の時間，インタプリタ関数を使った割合，更にホストのデータキャッシュミス時間が組合わされた結果である．それらの事象を表 20に示した特性で表現すると，Pmiss，Pitp，Pifn が小さく Lbr が大きいほど高い性能となる．132.ijpeg は Pmiss が小さく Lbr が大きいため最も性能が高く，099.go はプログラム領域が広く Pmiss が大きいため性能が低く，126.gcc は Pmiss と Lbr の両要因により性能が最も悪いことが分かる．

マップドモードでは，命令アドレスのマッピングオーバヘッドは小さいが，データアドレスのマッピングのオーバヘッドはロード命令とストア命令の出現頻度が高くなると増える．

アドレス変換モードでは，6.7に示したように命令 TLB のミス時には CC もミスするように制御しているため，TLB ミス率が高いと Pmiss が増大する傾向がある．しかし，評価に用いたような通常の TLB 構成では TLB ミス率は小さく，TLB ミスによる影響は小さい．データのアドレス変換をするデータ TLB もヒット率が高いため，フィルタ動作をしているハッシュテーブル読出しがオーバヘッドの支配要因となっている．この動作はマップドモードの動作と似ているが，解析の結果，ミス検出時には run_loop へ戻るための分岐判定の処理があることと，マップドモードより処理が若干複雑な分レジスタが不足することがオーバヘッドの主要因と分った．

図 40に示した MIPS 値からインタプリタを基準とした相対性能として計算すると，フラットモードでは 4.3～13.5（平均 7.7）倍，マップドモードでは 4.3～11.3（平均 6.3）倍，アドレス変換モードでは 2.7～10.5（平均 4.5）倍となった．sim-fast を基準とすると同等機能のマップドモードで 7.3～22.8（平均 11.8）倍[†35]の速度性能向上が達成でき，その性能向上効果が大きいことが分かる．

図 41は，フラットモードの MIPS 性能（縦棒，左軸）とインタプリタを基準とした相対

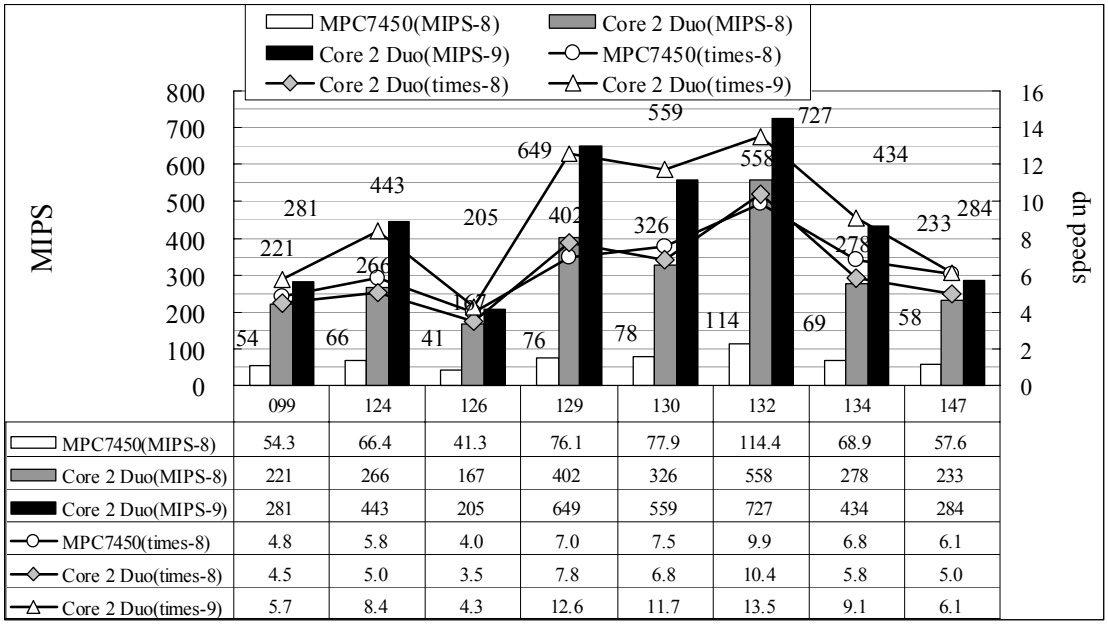
[†35] sim-fast に対するフラットモードでは 8.2～30.8 倍(平均 15.8 倍）である．

性能（折れ線，右軸）を，ホスト Core 2 Duo と MPC7450[†36]の両方を示している．MPC7450 の変換レベルは 8 である．このグラフは，Core 2 Duo の変換レベルについては 8 と 9 の両方を示している．MPC7450 の絶対性能が周波数の差を考慮しても Core 2 Duo に比べて低い主な原因は，バイナリ変換されたコードが関数リタン用のリンクレジスタ（LR）を含む 4 個のレジスタのセーブリストア[†37]を行っていることと，間接分岐が遅いことである．同じレベル 8 同士では 2 つのホストともインタプリタに対する性能向上率は似た傾向を示している．なお，比較に用いた ESPRIT/sim のインタプリタは，アドレス上限チェックなども含んでいるが，その機能を省略している sim-fast より速い．Core 2 Duo の平均ではマップドモードのインタプリタが sim-fast の 1.8 倍，フラットモードのインタプリタが sim-fast の 2.0 倍高速である．

図 42に，レガシーISA の違いによる性能評価の結果を示す．レガシーISA には，PISA，PowerPC（PPC）と M32R を，ホストには Core 2 Duo を，ベンチマークプログラムには 099.go と 129.compress を使用している．変換レベルには 9 を用いた．縦棒と左軸が MIPS 性能を，折れ線と右軸がインタプリタを基準とした相対性能を示している．3 つのレガシーISA ともに，CC のヒット率が高い 129.compress では 10～15 倍，低い 099.go でも 4～8 倍の速度性能向上効果が出ている．PISA に比べると実在のレガシーISA では例外検出が多くなることから，また命令が複雑なために InsFnc の実行率（Pitp）が高くなることから，絶対性能が若干低くなる．PowerPC では，lmw と stmw の命令実行でレジスタ本数を多くロード／ストアする場合もあり，099.go はその典型である．インタプリタも PISA に比べて複雑となるため，結果としてバイナリ変換の速度性能向上効果は PISA と同等かそれ以上となっている．

[†36] MPC7450 では，C／C++言語の実装に PIC（Position Independent Code）が用いられコアループや InsFnc でのグローバル変数のアクセスが遅いという課題があった．これに対しローカル変数のコピーを用いるなどの改善をし，インタプリタで 2～3 割の速度性能.を改善した後の結果をここに示している．

[†37] RISC ホストではバイナリコードが参照するレジスタ構造体やメモリポインタなどが必要である．



	099	124	126	129	130	132	134	147
MPC7450(MIPS-8)	54.3	66.4	41.3	76.1	77.9	114.4	68.9	57.6
Core 2 Duo(MIPS-8)	221	266	167	402	326	558	278	233
Core 2 Duo(MIPS-9)	281	443	205	649	559	727	434	284
MPC7450(times-8)	4.8	5.8	4.0	7.0	7.5	9.9	6.8	6.1
Core 2 Duo(times-8)	4.5	5.0	3.5	7.8	6.8	10.4	5.8	5.0
Core 2 Duo(times-9)	5.7	8.4	4.3	12.6	11.7	13.5	9.1	6.1

図41 ホストの違いによるバイナリ変換性能とインタプリタとの性能差

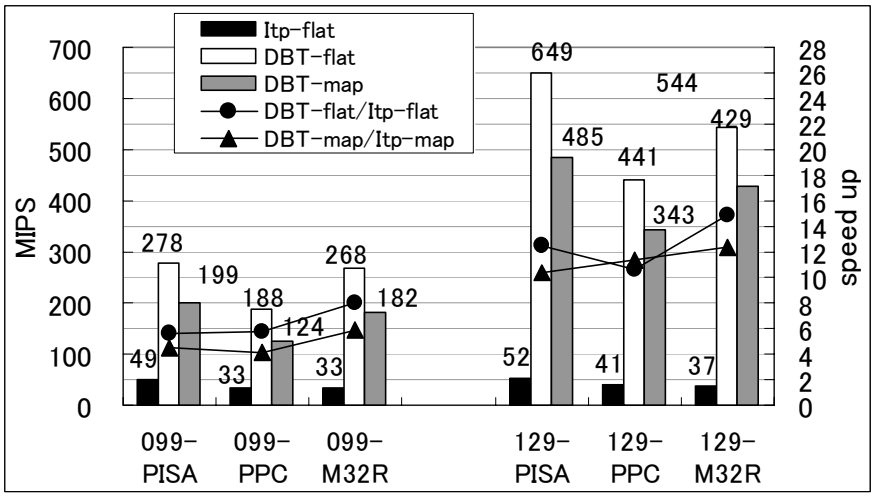


図42 他レガシーのバイナリ変換性能

次にスローダウンを評価する．PISA にはネイティブ動作がないため，PISA のバイナリを Core 2 Duo でシミュレーションした時間を Core 2 Duo のネイティブ実行時間で割った値をスローダウンとして示す．132.ijpeg (penguin) のスローダウンは 3.4，099.go (50, 21) は 5.1，スローダウンが大きな 126.gcc（-O amptjp.i）でも 8.5 であり，目標の 10 と比べると十分小さな値である．実際の組込みシステムの性能は Core 2 Duo の 1/10 以下のものも多いため，これらの値で十分な性能を実現できているといえる．MPC7450 による PISA のシミュレーションでは 099.go（50, 21）が 15，129.compress が 7.1 となる．それに比べて，PowerPC のネイティブ動作に対するシミュレーションのスローダウンは 21 と 11 に拡大する．これからも，PISA に比べて複雑な命令仕様を持つ PowerPC のシミュレーションも複雑となり処理時間を要することが分かる．

7.4. バイナリ変換実装の共通化の効果

表 21にバイナリ変換としてレベル 8 まで実装したソースコード量（行数）を千行単位の KL で示す．TrFnc の実装命令種数は PISA が 68 個，M32R は 46 個で，平均行数はともに 18 行と少ない．一方，複雑な命令を持つ PowerPC（PPC）のバイナリ変換の実装は，インタプリタに実装した命令種の 4 割にあたる 57 種に限定している．しかも，それらの一部の条件（Rc フィールドと OE フィールドともにゼロ）にのみ適用し，そのほかの条件のときには InsFnc を使用している．そのため総コード量は 1.5KL と少ないが命令が複雑な分，平均行数は 27 行と PISA や M32R より若干多めになる．また，この表よりレガシーISA に依存しない Layer-2 と Layer-3 の占める割合が大きく，共通化の効果があることが分かる．Runtime は run_loop を含んでいる．Runtime のレガシーISA 間のコード量に差があるのは，表 20（110ページ）に示した統計などの評価用のオプションを PISA に実装し他は一部しか実装していないことに由来する．

表21 バイナリ変換用のソースコード量（KL）

	Translator			Runtime	Interpreter
Layer	1	2	3	1	--
Common	0.52	--	--	0.05	--
ISA-M32R	0.80	--	--	0.38	1.68
ISA-PPC	1.52	--	--	0.36	2.74
ISA-PISA	1.25	--	--	0.49	1.45
HOST-x86	--	0.13	1.16	--	--
HOST-PPC	--	0.12	0.81	--	--

7.5. バイナリ変換の変換レベルとその効果

変換レベルは表 22に示すようにレベル付けを行い，レガシーISA ごとにレベル選択をできるようにバイナリ変換機能を実装した．変換レベルを上げるにつれ，その実装のコストは増大する．レベルの違いによる性能向上効果をレベル 0～9 について測定評価した結果を図 43に示す．レガシーISA は PISA，ホストは Core 2 Duo である．縦軸はインタプリタに対する相対性能で，グラフの重なりによる見にくさを避けるためプログラム 5 種のみを選択して表示した．

表22 バイナリ変換の変換レベル

レベルとニーモニック	説明
0.interp	高速インタプリタを使用
1.ifnc	InsFnc を呼び出すコードを CC に生成
2.ifnc-pc	PC 更新の最適化を 1 に追加
3.ifnc-cbr	条件分岐命令の後続命令生成を 2 に追加
4.tr-base	各レガシー命令をホスト命令に変換
5.tr-pc	PC 更新の最適化を 4 に追加
6.tr-cbr	条件分岐命令の後続命令生成を 5 に追加
7.tr-code	OP フィールドの特殊条件の最適化（6 に追加）
8.tr-reg	ホストレジスタに残った結果を再利用
9.tr-ccbr	CCE 内から別の CC を検査して分岐

レベル 0 はインタプリタ動作そのものである．レベル 1 から 8 までは6.4で紹介した内容と同じである．レベル 8 の tr-reg の実装をさらにここで説明する．ISA レジスタの実体であるメモリに書き込むとともに別のホストレジスタにも結果を残して後続命令ではそのホストレジスタから読み出せるようにワンパスで簡易実装している．このねらいは，CCE のコード量削減でもホスト命令実行数の削減でもなく，ホストの後続命令によるメモリアクセスパイプラインの遅延削減にある．PowerPC では 3 本のレジスタを，x86 では 2 本のレジスタをこの目的に割り付けている．この方式では，CCE 内で最初にそれらのレジスタのセーブ行い，CCE から抜けるときにリストアする必要があり，メモリアクセスパイプラインの遅延削減の代償としてこのオーバヘッドが発生する．そのため，132.ijpeg や 129.compress のように Lbr が大きい場合はこのオーバヘッドが薄まり効果が大きい. 処理の遅いメモリアクセス命令が少なくレジスタ演算命令が相対的に多いプログラムはその効果が大きく見える．逆にメモリアクセスが多いプログラムではこれによる性能向上効果が薄まる．そのような理由により，他のプログラムでは差異が小さく，わずかに性能が低下しているものもある.

レベル 1～3 の InsFnc を使用しただけの実装（ifnc-*）でもインタプリタの約 1.5 倍の速度性能を確保できている. 市販のプロセッサでは命令デコードが PISA より複雑なため, ifnc の効果は更に増大する. それを数値で示す. 変換レベル ifnc-cbr で 099.go の実行では PISA はインタプリタの 1.3 倍しか向上していない．しかし，グラフや表には示していないが PowerPC は 1.6 倍，M32R が 2.4 倍に速度性能が向上した．129.compress では PISA が 1.9 倍に対し，PowerPC は 2.1 倍，M32R は 2.4 倍に向上した.

レベル 4 から 8 に至る改良では，コード量の増加はレガシーISA ごとに高々50～100 行であるが，安定動作に至るデバッグ期間は行数の増加以上に延びる．このようにバイナリ変換を部分適用しただけでもインタプリタの 3～8 倍向上し, 変換関数 TrFnc を用意せずにインタプリタ用の InsFnc を流用しても 1.2 倍から 2 倍程度の速度性能の向上を実現できる.

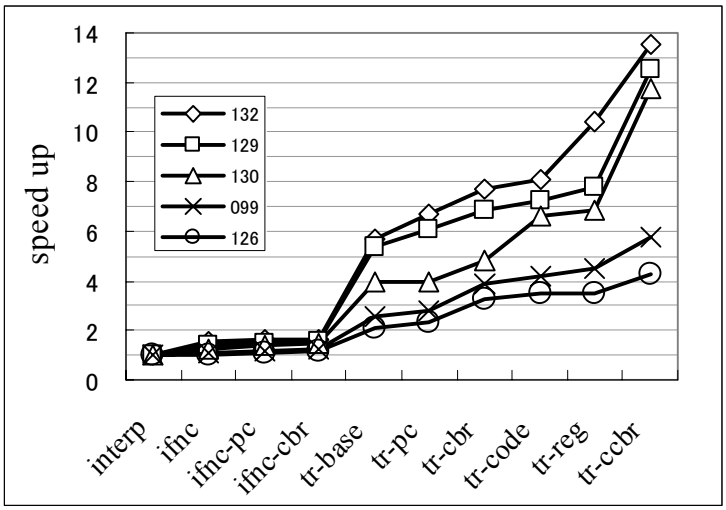


図43 バイナリ変換の変換レベルと性能向上

次に，変換対象の PISA のレガシー命令の種類を限定したときの Core 2 Duo の速度性能の評価結果を図 44に示す．ここでは CINT95 のプログラム 8 個の命令実行頻度統計をまず計った. その平均のトップ 18 について 3 個ずつ変換対象として増やした変換ルールを tr-3, tr-6，tr-9, . . , tr-18 と定義した．変換レベルは 7 を使用して，順次 TrFnc を増やして性能向上を評価した．折れ線の縦軸は，実装した 68 種の命令のバイナリ変換を生かしたときの速度性能を 100%としている. 棒グラフはプログラム 8 個の各命令の出現頻度を平均値で示している．各プログラムの 90%の頻度をカバーする命令種はそれぞれに異なるが，8～19 種の範囲であった．全部の命令種数の 15%にあたるトップ 18 の命令種のみで 48～95%の速度性能を実現できている．CINT95 の平均では 79%の性能である．なお，TrFnc として実装した 68 命令種のうち CINT95 で実行されたものは 55 命令種で残りの 13 命令は実行されていない.

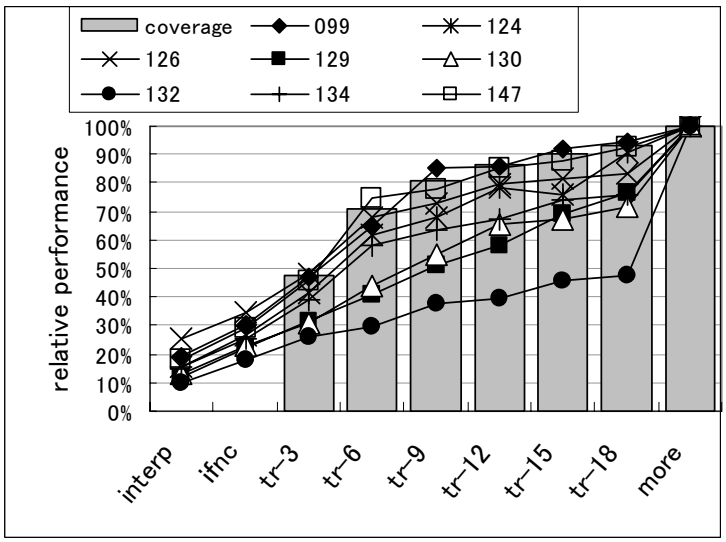


図44 バイナリ変換対象の命令種の拡大と性能

7.6. バイナリ変換の高速化手法とトレードオフに関する議論

本節では，一般的に用いられているバイナリ変換の高速化手法に対して，ESPRIT/sim で採用した高速化手法の効果を議論する．

7.6.1. コード変換キャッシュの単純化

最初に，CC の 2 ウェイ構成の効果について，変換レベル 7 で比較評価した結果を述べる．ヒット率はダイレクトマップ，FIFO，LRU の順に向上するが，ヒット時のオーバヘッドもこの順に増大する．124.m88ksim，129.compress，130.li はコード領域が狭くもともとミス率が低く 2 ウェイ化によるヒット率の向上効果がないため，2 個目のウェイにヒットしたときのオーバヘッドが増え，結果的に 2 ウェイの速度性能はダイレクトマップの速度性能の 87～92%に低下している．132.ijpeg は CCD の最初のウェイへのヒット率が高くかつ Lbr が大きい．そのため，2 ウェイ化によるオーバヘッドの影響は薄まり FIFO と LRU ともに速度性能は 97%に留まり性能低下は小さい．CCD のミス率の大きい 099.go と 126.gcc の速度性能は，FIFO が 95%と 98%に低下しているのに対し，LRU では 97%と 102%と向上している．したがって，CCD のセット数が小さい場合の 2 ウェイの効果がここに一部に見られるが，オーバヘッドの小さいダイレクトマップを用いて CCD のセット数を多くする方が性能に有利になることが分かる．

7.6.2. 分岐処理の単純化

バイナリ変換には，条件付分岐命令を含む全部の分岐命令を出口とする命令列を基本ブロックと呼びそれを変換単位とする方式と，条件付分岐命令以降も変換対象として複数の出口を持つ拡張ブロックを変換単位とする方式がある[15]. 変換レベル 3 と変換レベル 6 以降は後者に相当する．分岐発生時の高速化の方式として，CCE から run_loop に戻らずに変換済みの他ブロックに直接に分岐する方法[15][39]もある．ESPRIT/sim ではマルチプロセッサ動作時に，CCE から他ブロックの CCE に直接に分岐をせずに run_loop に戻す方法を採り，各 CPU の命令数の間隔を平均で一定に，最大時でも 2 倍以内に揃うように制御を簡素化している．また，ESPRIT/sim のユニプロセッサ向けのバイナリ変換には，CCD を検査して他の CCE に間接分岐する機能と同一 CCE 内の他アドレスへ直接に分岐する機能がありともにワンパスで実現している．そのうち，前者を適用したものが変換レベル 9 である．図 40（109ページ）に示した性能はレベル 9 で，この機能をオフにしたレベル 8 と比較して表 23に示す．*-8 はレベル 8，*-9 はレベル 9 を示す．これらの単位は MIPS である．フラットモードのレベル 8 に対する相対性能（speedup）は 22～72%（平均で 39%）向上した．レベル 8 のスローダウン値 5.5～10.6 は，7.3で示したように 3.4～8.5 と改善効果は大きいが，プログラム間の速度性能差も更に拡大している．他レガシーISA でもレベル 8 からレベル 9 の改良で 15～34%の性能向上効果があった．レベル 9 の実装は，Layer-2 の変更が 1 行，追加コードは ISA 共通関数が 45 行と Layer-3 の x86 用が 52 行と少ないが，命令数の計数誤りや CC の比較誤りによる性能向上不足などのデバッグと評価に計 2 日掛かった．なお同一 CCE 内への前方分岐と後方分岐能も実装して評価したが，CCD と CCE は余分に必要となるがレベル 9 でもこの部分的な効果を持つため，新たな性能向上は 0.7～

7.4%に過ぎないことが分った．またこの機能とレベル 8 の機能との両立をワンパスで実装するのは困難なため ESPRIT/sim への正式適用から除外している．

表23 CCE 内から分岐したときの速度性能

program	execution speed（MIPS）						speedup
	flat-8	map-8	dat-8	flat-9	map-9	dat-9	
099	221	176	124	281	199	133	1.27
124	266	237	165	443	290	190	1.67
126	167	136	94	205	197	97	1.23
129	402	341	301	649	485	424	1.61
130	326	267	202	559	367	240	1.72
132	558	432	361	727	539	421	1.30
134	278	223	150	434	290	168	1.56
147	233	167	95	284	185	99	1.22
Ave.	271	218	151	377	276	166	1.39

7.6.3. レガシーレジスタアクセスの最適化

バイナリ変換の高速化の手法として，メモリに置かれたレガシーレジスタのホストレジスタへの読出しと書き戻しの回数を減らすなどのレガシー命令間に渡る最適化[15]がある．この方法は命令生成をツーパスにするか，ワンパスでもコードの再変換が必要となり，特に基本ブロックを跨った処理が複雑になる．命令間に渡った最適化の効果を評価した文献は少なく性能比較ができるものはない．そこで，その効果算定を目的として拡張ブロック内で同一の整数レジスタを 2 回以上の多重書込みをした回数と書込み後のレジスタを読出した回数を計測し，それらとホストのキャッシュメモリのアクセスペナルティを削減できる時間から理想的な性能向上効果を推定した．CINT95 の 8 種では，ホスト Core 2 Duo の変換レベル 8 では CPI が 3～11 であるのに対し CPI の削減値は多重書込み分が 0.1～0.4（平均 0.1），読出し分が同 0.3～0.8（平均 0.5）で，速度性能の向上換算では 7～52%（平均 13%）の向上と計算した．変換レベル 8 は既に Layer-3 でホスト x 86 用に 90 行，PowerPC 用に 107 行のコードにより実装済であり，このレジスタ読出し分の最適化対象回数の 65～96%（平均 82%）分は改善済みである．これらのソース行数は少ないがデバッグに計 3 日掛かり，レガシーISA を 1 個追加するときのデバッグ期間（1～2 週間）に比べて効率が悪い．この経験より文献にあるような理想的な最適化の実装を適用した場合には，更に投資効果が低くなることが予想される．

7.6.4. 提案手法のトレードオフ

これら分岐の最適化とレジスタの最適化の両方を行っている Shade のスローダウンは，レガシーISA とホスト ISA が同じならば 3～6，異なれば 8～15 である[15]．ホストの違いもあり直接の速度性能の比較は難しいが，ESPRIT/sim の方式も同程度の速度性能を実現していると考える．ESPRIT/sim は固定長の CCE を用いており Shade のように FIFO による CCE 間の分岐制約もなく，またダイレクトマップを用いた効果により CCE から他 CCE へのヒット検索の高速化と単純化も実現している．

更に最適化をする手法には，文献[39]に紹介されているように，レガシー関数からの戻りを速くするための間接分岐先の予測バッファ，無条件分岐命令を最適化するブロックの直列化や再配置化がある．また，実行時の統計より分岐側と非分岐側を入れ替える方法[3]も

ある．これらの手法には，コンパイラの最適化手法同様に複雑で設計工数はもとより多大な検証工数が掛かると考えられる．工数と品質に関する文献は見当たらないが，試験用の命令列をランダム生成しインタプリタとバイナリ変換の実行結果の比較[3]をするなどの大規模な開発が必須である．そのような高度な実装との直接の比較は難しいので，コストが安いバイナリ変換としてソース行数を記載した手法[27]と比較してみる．その共通部のソース行数は 33K 行と ESPRIT/sim の数倍多い．また，その定義データの行数は ESPRIT/sim の Layer-1 と同程度であるが，記述密度が濃い．したがって，本方式は開発の難易度が低く簡易的な実装であるといえる．

7.7. 動的バイナリ変換の高速化性能の効用

7.6までは，統計機能を命令数に限定したシミュレータコアの高速化に焦点を絞り議論と評価をしてきた．本章では，性能評価シミュレーション全体への効用を述べる．

まず，命令レベルのシミュレーション高速化により，評価プログラム全体の動作ができる環境の準備，プログラムの全貌の把握，ハードウェア構成などのパラメータ調整，詳細評価を行うプログラム箇所までのセットアップなどの処理時間の大幅な短縮が実現できる．

また，ある特定機能に着目したシミュレーションを行うときにシミュレータコアの高速化が必要といわれている．分岐予測やキャッシュシミュレーションでは高速化したインタプリタコアを使用しても 71～84%の時間をコアが占めているという報告もある[5]．本研究でもそれを，7.7.1でプロファイル機能，7.7.2でキャッシュシミュレーション，7.7.3でワンパス型のマルチサイズキャッシュシミュレータにバイナリ変換を適用した例を用いて効果を示す. いずれもレガシーISA は PISA, ホストは Core 2 Duo, 評価プログラムは CINT95, フラットモード動作に対して行った評価結果である．

7.7.1. プロファイル

バイナリ変換のコアに，命令コードごとの頻度を集計する命令種プロファイル機能と，4Kbyte ページ単位のメモリアクセス数の集計をするメモリページのプロファイル機能を変換レベル 7 に追加して速度性能を評価した．命令統計シミュレーションは，表 24に示すように 146～342（平均 212）MIPS で，同等機能の sim-profile の 13 倍の速度性能である．ページ統計シミュレーションは，表 25に示すように 132～361（平均 199）MIPS で，統計なしの ESPRIT/sim インタプリタの平均 4.1 倍の速度性能である．このような機能の追加は容易であり，統計なしのバイナリ変換動作からの速度性能の低下は約 20%にすぎない．

表24 命令種のプロファイルへの応用例

性能	0 99	124	126	129	130	132	134	147
命令統計（MIPS）	175	206	146	299	261	342	226	183
DBT-flat 比	0.84	0.78	0.86	0.77	0.82	0.75	0.78	0.82
インタプリタ比	3.6	3.9	3.0	5.8	5.5	6.4	4.7	3.9
sim-profile 比	11.0	12.9	9.1	18.7	16.3	21.4	14.1	11.4

表25 メモリページのプロファイルへの応用例

性能	0 99	124	126	129	130	132	134	147
ページ統計（MIPS）	170	211	132	297	235	361	196	155
DBT-flat 比	0.82	0.80	0.78	0.76	0.74	0.79	0.68	0.69
インタプリタ比	3.5	4.0	2.8	5.7	4.9	6.7	4.1	3.3

7.7.2. キャッシュシミュレーション

2 ウェイセットアソシアティブ LRU 置換方式の一次キャッシュシミュレーション機能を変換レベル 7 に追加して速度性能を評価した．評価対象のキャッシュメモリは，データ（L1D）と命令（L1I）ともにラインサイズは 32byte で容量は 16Kbyte に設定した．キャッシュシミュレーション部は，ラインサイズ，セット数を定数とした平易な C++言語記述の関数をバイナリコードから呼び出している．このモデルではキャッシュのフィルタリングは行っていない．表 26に PISA の CINT95 を Core 2 Duo 1.86GHz でシミュレーションしたときの速度性能を MIPS 値で示す．

命令キャッシュシミュレーションは 89～203（平均 128）MIPS，データキャッシュは 79～162（平均 107）MIPS，両方のシミュレーションは 52～103（平均 67）MIPS となった．統計なしのバイナリ変換動作に比べると性能低下は著しいが，同一構成で実行した sim-cache の 5.3～10.1 倍（MIPS 値平均 7.9 倍，MIPS 逆数平均 7.5 倍）の速度性能である．

表26 キャッシュメモリのシミュレーションへの応用例 1

性能	0 99	124	126	129	130	132	134	147
(A)L1I cache (MIPS)	99.4	119	89	199	180	203	123	111
(B)L1D cache (MIPS)	93.0	143	79	141	108	162	105	79
(C) L1I+L1D (MIPS)	64	86	52	103	81	110	69	55
(D) sim-cache (MIPS)	10.2	10.6	9.7	10.6	9.2	10.9	9.6	N.A.
(E) C/D	6.3	8.1	5.3	9.8	8.8	10.1	7.2	N.A.

次に，2 次キャッシュメモリも含むキャッシュシミュレーションに応用した結果を表 27 に示す[†38]．このキャッシュモデルは C++で記述され，キャッシュの各パラメータは定数ではなく変数として記述している．1 次キャッシュのパラメータは前の例と同じである．2 次キャッシュ（L2）のパラメータには，ラインサイズ 256byte，4 ウェイの LRU ライトバック方式で 2Mbyte を設定した．この評価は変換レベル 9 の構成をもとに実施した．ここでは，各キャッシュのセット専用のアレイをフィルタとして設けて最近のヒットしたラインアドレスを記憶しそれと同じアクセスではキャッシュメモリモデルを参照しないようにしている．1 次キャッシュのフィルタ引きの動作をバイナリ生成したコードより実行することにより，速度の遅い C++ライブラリの速度性能を緩和できている．特に 132.ijpeg や

[†38] 評価を，キャッシュへの応用，マルチプロセッサの評価，レベル 9 の追加の順に評価を進めた経緯がありそのため比較の基準に選んだ変換レベルが異なっている．

129.compress のようにキャッシュメモリにヒットしやすい場合はその効果が大きい．その結果，sim-cahce に比べて，MIPS 値平均で 15.6 倍，MIPS 逆数平均で 11.4 倍の速度性能を実現している．

表27 キャッシュメモリのシミュレーションへの応用例 2

性能	0 99	124	126	129	130	132	134	147
(F) L1i+L1D+L2+filter(MIPS)	81	79	61	269	170	291	122	80
(G) sim-cache L1i+L1D+L2(MIPS)	9.8	10.2	9.2	10.1	9.2	10.8	9.6	N.A.
(H) F/G	8.3	7.8	6.6	26.8	18.6	27.0	12.7	N.A.

7.7.3. ワンパス型マルチサイズキャッシュシミュレータ

複数のキャッシュ容量構成に対応して，キャッシュミス率の変動を時系列グラフにワンパスで出力できるマルチサイズキャッシュシミュレータは，有用である．特に，アプリケーション全体の動作を把握し，アプリケーション間の挙動の違いを短時間に調査するのに効果がある．ここでは，キャッシュメモリの構成を 2 ウェイセットアソシアティブ LRU 置換方式とし，キャッシュメモリの容量を命令とデータについて各々4Kbyte，8Kbyte と 2 倍ずつ 1Mbyte までの計 18 種類を 1 回でシミュレーションできる機能を ESPRIT/sim に追加して評価した．実装と評価は変換レベル 7 をベースに行った．図 45に命令キャッシュメモリのミス率の表示結果の例を示す．横軸は命令数で単位は百万命令，縦軸はミス率を示している．印刷の都合上グラフカーブの本数は間引いてある．

全部で 18 種類のキャッシュメモリのシミュレーションを行ったときの速度性能は表 28 に示すように 30～100MIPS で平均では 43MIPS である．この統計機能がないバイナリ変換動作時の速度性能に比べると 13～24%に低下する．しかし，この結果は sim-cache の 3.1～9.3 倍（平均 4.5 倍）の高速性を示しており，しかも sim-cache で同等の計測を行うのに必要な 5 回の実行時間と比較すると sim-cache の 22 倍相当の効果があり，バイナリ変換の高速性を活かした例といえる．なお，この実装にあたり，命令とデータの各キャッシュアクセスのためにそれぞれ大きさの異なる 2 種フィルタを使用し性能低下を軽減している．

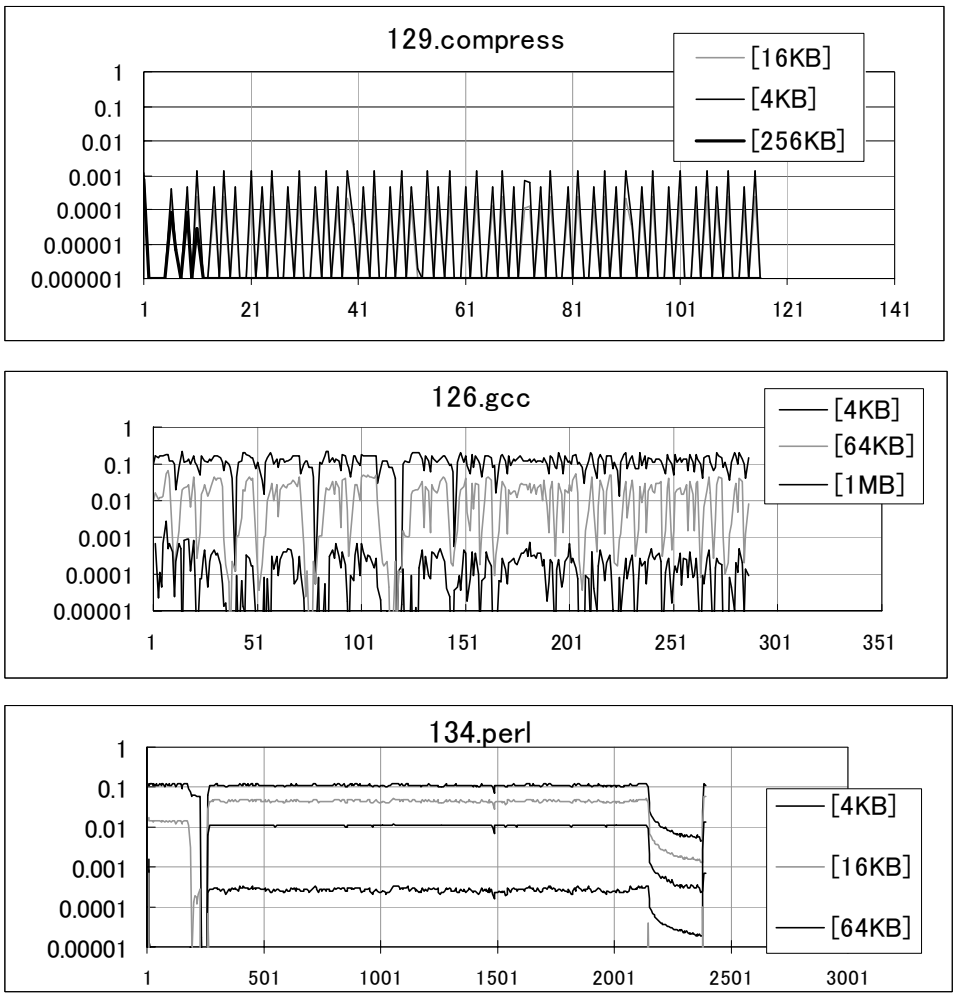


図45 ワンパス型マルチサイズキャッシュシミュレータの表示例

表28 ワンパス型マルチサイズキャッシュシミュレータの速度性能

	0 99	124	126	129	130	132	134	147
one-pass cache sim(MIPS)	30	41	31	71	45	100	39	29
sim-cache 比	3.1	4.0	3.4	7.0	4.9	9.3	4.1	N.A.

7.8. まとめ

第 7 章では，前章の6.2で述べた（1）～（6）の提案のうち，ユニプロセッサに焦点を当てて（1）～（5）の手法についてその効果を評価した．

（1）ホスト命令を生成する対象は使用頻度の高いレガシー命令に限定し，その他のレガシー命令用にはインタプリタのソースコードを併用したバイナリ変換を行う．

（2）トランスレータの実装は，開発コストに見合った効率的な変換レベルの選択ができるようにする．

（3）トランスレータの構造は，3 階層にしてレガシー命令セットとホストの命令セットを独立にし，組合せを不要にする．また，トランスレータのコードの共通部を

増やす.

(4) レガシー命令間に渡った複雑な最適化は行わず，また変換後の命令列を格納するコード変換キャッシュは固定長の構造とするなどの単純化をする.

(5) 多様な使い方に対応し 3 種類のアドレス空間モードを設ける.

これらの手法を用いて，開発量が小さくスモールスタートかつ品質の確保が容易で，簡易的なバイナリ変換の実装を実現している.

バイナリ変換はシミュレーション対象のプログラムによりその性能が大きく異なるため，広い特性を持つ SPEC CINT95 を用いて総合評価を行った．その結果は，7.3に示したようにレガシーISA は PISA でホストが Core 2 Duo では 205～727MIPS, 最小の CPI 値は 2.6, インタプリタの 3～14 倍[†39]，著名なシミュレータである SimpleScalar/PISA の sim-fast の 8～30 倍の速度性能を達成している．また，ホストを MPC7450 に替えた評価及びレガシーISA を PowerPC と M32R に替えた評価も行い，異なるホストまたは異なるレガシーISA でも同様な効果があることを示した.

7.4では，バイナリ変換のために費やしたソースコード量のデータから，手法 (3) の効果により，レガシーISA 依存部，ホスト ISA 依存部，共通部とソースコードが分離され，レガシーとホストの種類を増やしてもその影響が小さいことを示した.

7.5では，(1) の変換対象とする命令種と (2) の変換レベルをそれぞれを変えて評価を行い，ごく一部の命令のみ変換対象としてもかなりの速度性能が得られ，また，変換レベルもレガシーISA やホスト ISA ごとに投資効果を考慮してトレードオフを見つけられることを示した.

7.6では，Shade やエミュレータ用などのバイナリ変換で使用されている高速化技法に対してトレードオフを議論し，本研究で提案している簡易実装が有効であることを示した.

7.7では，バイナリ変換のシミュレータコアを応用したプロファイリング，キャッシュシミュレーション，ワンパス型マルチサイズキャッシュシミュレータの試作と性能評価を行った．いずれも，シミュレータコア単体に比べると性能低下はあるが，統計機能のないインタプリタより高速であり更にインタプリタに該当機能を付けたものより一桁高速である.

以上のように，拡張性が高く設計品質の確保が容易かつ開発コストが安い性能評価シミュレータの高速化手法として，ESPRIT/sim に搭載した簡易的なバイナリ変換の実装方式を述べ，その評価により有効なことを述べた．なお，マルチプロセッサの評価は第 8 章で行う.

[†39] 性能向上率が最も低いのは 126.gcc であり，小数以下を丸めると変換レベル 8 では 3，変換レベル 9 では 4 となる．結論として幅を持たせる意味で低い値の 3 を使用している.

第8章 ESPRIT/sim のマルチプロセッサ対応のシミュレーション速度性能評価

本研究では，第 6 章で提案したバイナリ変換方式の有効性を示すため，性能評価シミュレータ ESPRIT/sim にマルチプロセッサ向けのバイナリ変換トランスレータを実装して評価を行った．第 7 章では，そのベースとしてユニプロセッサについて評価を行った．本章では，マルチプロセッサシミュレーションの評価をする．

まず，8.1で評価の方法を，8.2で評価方法の妥当性を述べる．8.3では SPLASH-2 による同種マルチプロセッサシミュレーションの速度性能の評価結果を，8.4では異種マルチプロセッサシミュレーションの評価結果を述べる．

なお，第 7 章では性能値を表現するのに MIPS 値を主に用いてきたが，本章では 1 命令をシミュレーショするのに掛かるホストのクロック数である CPI を使用する．それは，マルチプロセッサシミュレーションに関する研究では MIPS 値は余り使われず，スローダウン値や CPI が使われていることに習っている．なお，その理由は CPU 数に比例してシミュレーションの速度性能がダウンするため比例関係にあるかどうかを分かりやすく表現しているためと考える．

8.1. 評価方法

同種マルチプロセッサ動作の評価対象に，図 46に示すような最大 16 個のプロセッサコアがバス結合された UMA 方式のシステムを想定した．各 CPU[†40]はプロセッサコアがそれぞれ専用の一次キャッシュを持ち，L1 データキャッシュ（D$）と L1 命令キャッシュ（I$）ともに 16Kbyte でラインサイズ 32byte，セットアソシアティブ 2 ウェイの LRU 置換，データはライトバック，命令はライトスルーに構成パラメータを指定をした．それぞれの L1 データは MESI プロトコル[†41]でコヒーレンシーを保っている．2 次キャッシュメモリ（S$）は，ラインサイズ 256byte，4 ウェイ LRU 置換で容量が 2Mbyte のライトバック方式を指定した．レガシーISA に SimpelScalar/PISA（リトルエンディアン）を用いた．動作プログラムには SPLASH-2 ベンチマーク[33]より，FFT（1M 点），LU（contiguous block, 1024x1024），LU（noncontiguous block, 1024x1024），RADIX（1 千万件）を用いた．評価プログラムのこれらのパラメータは既定設定値を基本とし，CPU 数のほかに問題サイズを大きい値に変更した．既定の問題サイズは高速なシミュレータにとっては命令数が少なすぎ測定誤差が大きくなるためで，総命令数が 10 億～100 億個になるように変更している．CPU 間の同

[†40] 用語”CPU”はプロセッサコアにキャッシュメモリを含めたものとしてここでは使用している．CPU 一個に複数のプロセッサコアを持つ場合にはこの定義は当てはまらないが，本章では CPU には 1 個のプロセッサコアがあるとして表記している．

[†41] Modified, Exclusive, Shared, Invalid の 4 ステートでキャッシュ状態を示すプロトコルで市販 CPU では主流の方式である．

期を取る null.macro [33] には，バリア同期（BARRIER）とスレッド生成（CREATE）などを専用関数としてレガシーISA 用向けのソースコードを記述した．新たに実装した CPUID 取得，fetch&dec，CPU 起動，CPU ウェイトの各システムコールをそれらの関数から呼び出すようにした. SPLASH-2 とこれらの関数のコンパイルは, SimpleScalar/PISA 用の gcc2.9.5 にオプションは-O2 を用いて行った．なお，SPLASH-2 が呼び出している malloc 関数は，SimpleScalar のライブラリ中ではマルチスレッド対応になっておらず誤動作するため，排他的に malloc を呼ぶように SPLASH-2 のソースコードを変更した．

評価に用いたホストに，ユニプロセッサの評価と同じで Core 2 Duo を用いた．その仕様は表 19（108ページ）と同一で PC に搭載したメモリ容量は 1Gbyte である．Core 2 Duo は 2 個の CPU コアを搭載しているが，ESPRIT/sim はシングルスレッドで動作している．シミュレーション時間は測定誤差が小さくなるように，クロックサイクル相当の精度がある RDTSC 命令で計測した．性能の比較対象である SimpleScalar の Ver.3.0 の sim-fast と sim-cache も，Cygwin の gcc 3.3.3 で-03 オプションでコンパイルして実装した．本測定では，CCE は 800byte，CCD と CTD のエントリ数は 32K 個とした．CPU 間のスイッチ間隔 Nは 100 に設定した．

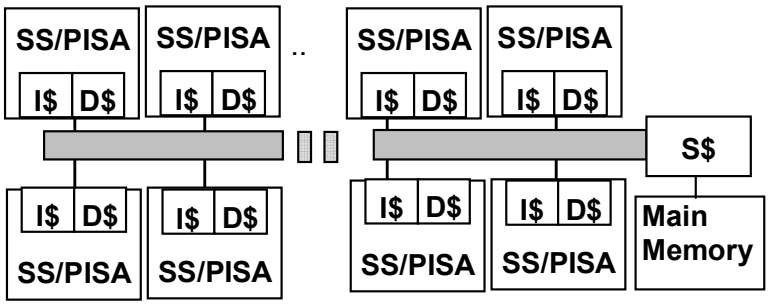


図46 同種マルチプロセッサの評価対象の構成

異種マルチプロセッサ用の標準的な評価プログラムは存在しないため，SPLASH-2 の RADIX-sort を改造したプログラムを用いて PowerPC と M32R を混載した動作の実行をした. CPU 数は 2 から 16 とし PowerPC と M32R が半数ずつの図 47に示す構成で評価した．エンディアンはともにビッグである．グローバルデータは構造体にまとめてポインタからアクセスができるようにし，PowerPC は OSX に搭載された gcc3.1 の-O2 オプションで，M32R は gcc3.4.5 の-O2 オプションで別々にコンパイルして，メモリアドレスが重ならないようにリンカーに異なる開始アドレスを指定した．RADIX-sort はソート前に浮動小数の乱数計算を用いてデータ生成をしている．M32R は固定小数命令で浮動小数演算をエミュレートするため処理に多くの命令を消費し, その分 PowerPC は待ちループが多い. そこで, M32R と PowerPC の乱数生成のソースコードをともに整数の倍長演算を使う計算に書き換えて是正した．メイン CPU は M32R にし，PowerPC 用のグローバルデータ領域を示すポインタ定数が M32R 用のデータ領域を参照するようにパッチを適用した．また，*N-1* 個の子スレッド生成をする親スレッドの for ループを 2 つに別け，後半のループが起動するスレッド関数 slave_sort のアドレスを PowerPC 用に変更した．なお，命令数の合計は約 14 億個である．

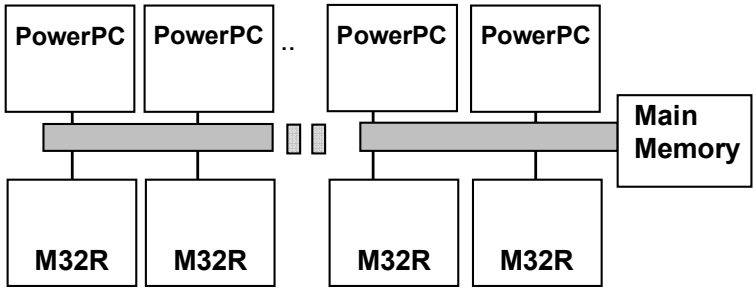


図47 異種マルチプロセッサの評価対象の構成

8.2. バリデーション

インタプリタとバイナリ変換の正当性は，基本的に7.2と同じ方法で確認した．マルチプロセッサ動作では命令数とキャッシュミス数が変化するため，正確な期待値を求めるのが困難となる．そこで，単一 CPU 動作同士の結果を比較するとともに，動作のトレース結果を部分的に目視チェックで確認を行った．SPLASH-2 には演算結果を確認するテストモード（-t オプション）があるため，それを使い機能的に正しい動作をしていることも検証した．

8.3. 同種マルチプロセッサのシミュレーション速度性能

8.3.1. 命令実行のシミュレーションの速度性能

表 29に命令シミュレーション速度性能を示す．そのグラフは図 48（126ページ）の左側に示す. 速度性能はレガシー1 命令を実行するのに掛かったホストのクロックサイクル数を CPI として示している．評価した CPU 数は 1，2，4，8，16 個である．表の列“1”から“16”は CPU 数を示し，各セルは CPI 値を示す．“ss”の列は sim-fast のユニプロセッサ動作時の CPI を示している．列 “speed up”は sim-fast の CPI 値を 1-CPU 動作時の CPI で割ったもので, sim-fast に対して何倍速いかを比で示している．CPI は値が小さいほど，“speed up”は値が大きいほど性能向上効果が大きい．グラフの線が水平であれば，CPU が増えてもオーバヘッドがないことを意味し，傾きが大きいとオーバヘッドが多いことを示す．CPU 数が増えても傾きはわずかであり処理時間はほぼ比例の範囲に留まっている．16-CPU 構成でも 1-CPU 構成の 76～80%の速度性能を実現しているためオーバヘッドが少ないといえる．なお，ユニプロセッサ構成の sim-fast と比べると 1-CPU は 8～14 倍，2～16-CPU は 6～12 倍の速度性能を示している．

表29 SPLASH-2 の命令シミュレーション速度性能（CPI）

CPU counts	1	2	4	8	16	ss	speed up
FFT　1M points	5.7	6.6	6.7	7.0	7.5	78	14x
LUC (cont.blk)　1024x1024	6.4	7.3	7.5	7.7	8.1	71	11x
LUN (noncont.blk)　1024x1024	6.3	7.3	7.6	7.5	8.1	71	11x
RADIX　key=10,000,000	7.6	8.5	8.7	9.0	9.5	61	8x

8.3.2. キャッシュシミュレーションの速度性能

表 30と図 48の右側にキャッシュシミュレーションの速度性能を示す．評価した CPU 数は 1，2，4，8，16 個である．表の列“1”から“16”は CPU 数を示し，各セルは CPI 値を示す．“ss”の列は sim-cache のユニプロセッサ動作時の CPI を示す．列“speed up”は sim-cache の CPI 値を 1-CPU 時の CPI 値で割ったもので, sim-cache に対する速度性能比を示す. この値が大きいほど性能向上率が高いことを示している. 16-CPU 構成でも 1-CPU 構成の 72～90%の速度性能を実現し，これもオーバヘッドが少ないといえる．なお，ユニプロセッサ構成の sim-cache と比べると 1-CPU では 10～19 倍，2～16-CPU でも 8～17 倍の性能を示している．

このように，キャッシュシミュレーションを行ってもバイナリ変換コアの 26～80%（平均 52%）の速度性能を達成しており，バイナリ変換の効果は大きいといえる．

表30 SPLASH-2 のキャッシュシミュレーション速度性能（CPI）

CPU counts	1	2	4	8	16	ss	speed up
FFT　1M points	11.3	12.5	12.5	13.3	15.1	185	16x
LUC (cont.blk)　1024x1024	10.6	11.6	11.8	12.7	14.7	197	19x
LUN (noncont.blk)　1024x1024	21.0	21.6	21.2	21.5	23.4	200	10x
RADIX　key=10,000,000	11.1	12.1	12.4	13.2	15.8	153	14x

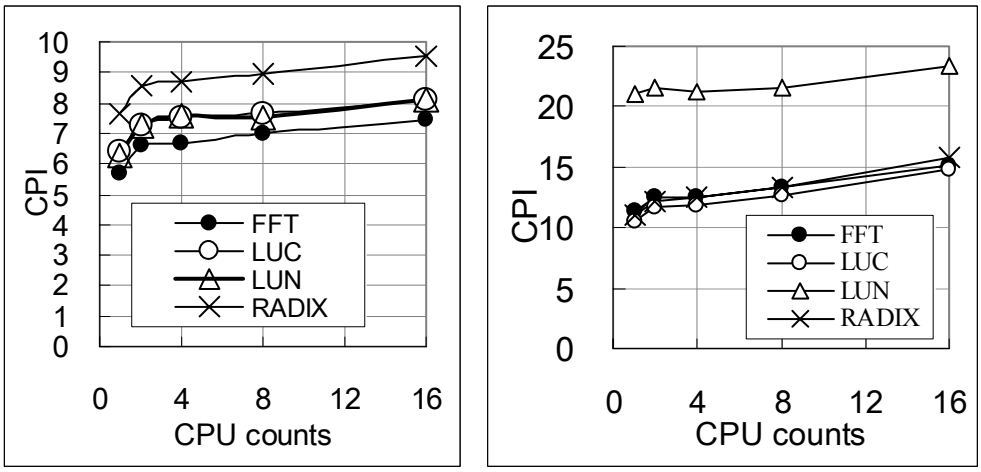


図48 SPLASH-2 のシミュレーション速度性能（CPI）

8.3.3. キャッシュシミュレーションの高速化手法の効果

次に図 49を用いてキャッシュシミュレーションの高速化の効果を述べる. ここでは, CPU 数が 4 のときの速度性能を MIPS 値で示す．この評価は次の 4 種類のモデルを使い速度性能を求めている．このグラフに比較のために sim-cache の性能も載せている．

NF :　C++にて記述したキャッシュモデルをバイナリコードから単純に呼ぶもの

CF :　C++記述内でフィルタを掛けるもの

BF :　バイナリコード内でフィルタを掛けるもの（表 30の測定条件）

BF-NC : CPU 間のキャッシュコヒーレンシーをチェックしないもの(参考用)

C++記述のキャッシュモデルは本来は低速であるが, グラフに示すようにフィルタの併用に

より性能低下を緩和し，3～4 倍に性能を高めていることが分かる．

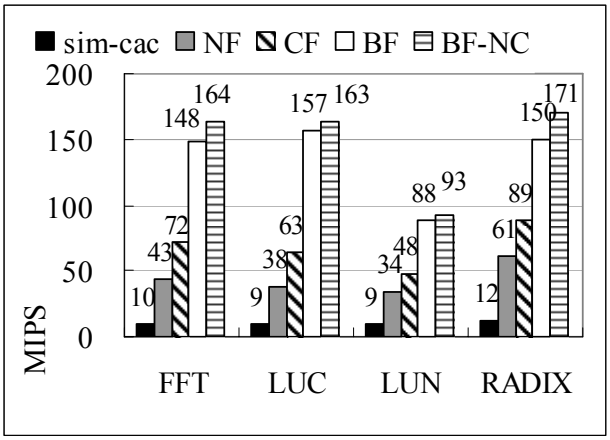


図49 キャッシュシミュレーション速度（CPU 数=4）

8.3.4. CPU 間のスイッチ間隔に関する評価

マルチプロセッサでは6.8.1で提案したシミュレーション対象の CPU を切り替える方式を採っており，そのシミュレーションをスイッチする間隔について評価した．スイッチ間隔 Nの単位は命令数であり，本節までの評価結果は Nを既定値の 100 に設定したときの値である．一般に間隔を短くすると精度は向上し性能は低下する．図 50に Nを変えたときのキャッシュシミュレーションの速度性能を示す. N=1000 のときの速度性能を 100%として，N を小さくしたときの速度の低下の様子を示すために速度性能を相対値で表している．図 51には，N を変えたときのデータキャッシュのアクセス数の誤差率を左側のグラフに，ミス回数の誤差率を右側のグラフに，ともに単位は百分率を用いて示す．なお，この評価では $N=1$ のときの値を正しいものとしてそれらとの差を誤差と定義している．

Nを変えて評価した結果, Nが 3200 以下ならば命令数の誤差では 0.1%未満, キャッシュアクセス回数では 0.03%，ミス回数でも 0.5%未満の誤差に収まることが分かった．また，図 50に示すように $N \geqq 50$ であればスイッチ間隔 1000 と同様な性能が得られる．このことから，SPLASH-2 では $1000 \geqq N \geqq 50$ となるように Nを選べば，精度と性能の両方を満足できるシミュレーションができ, また6.8.1で提案した CPU 間のスイッチによるマルチプロセッサシミュレーションの手法が有効であるといえる．

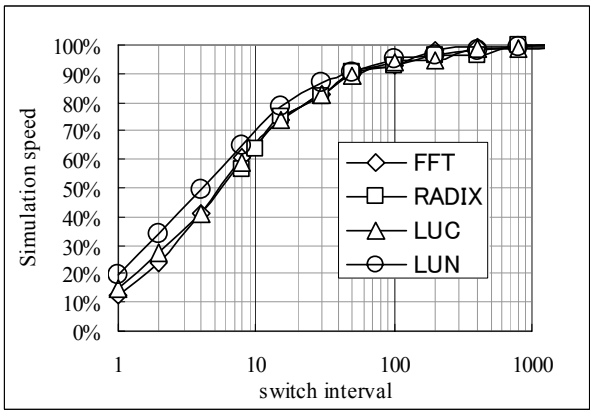


図50 スイッチ間隔によるシミュレーション速度性能

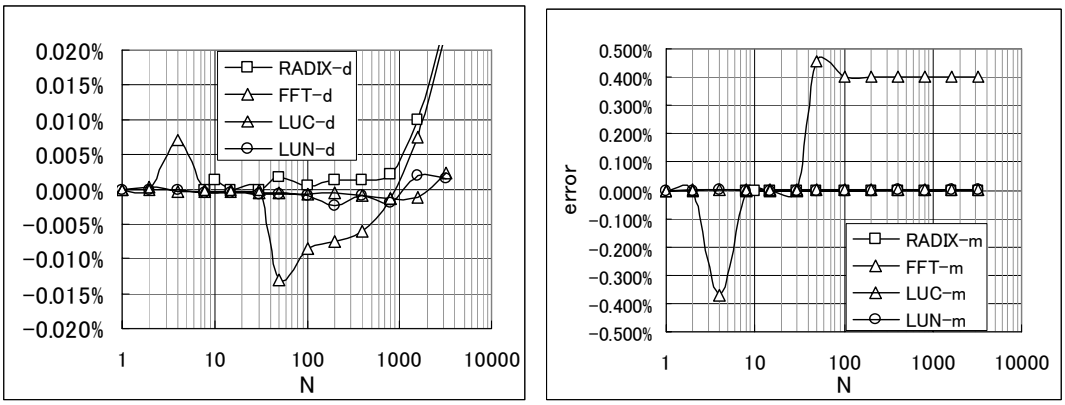


図51 スイッチ間隔によるデータキャッシュアクセス回数とミス回数の誤差率

8.4. 異種マルチプロセッサのシミュレーション速度性能

異種マルチプロセッサ構成で RADIX-sort をシミュレーションしたときの速度性能の結果を示す．図 52の左側には参考として PowerPC のみで構成した同種マルチプロセッサの速度性能を示す．棒グラフは CPI 値で，折れ線グラフは 1-CPU を基準とした相対性能である．右側のグラフは，PowerPC と M32R を混載した異種マルチプロセッサの速度性能である．こちらの相対性能は 2-CPU を基準としている．PowerPC のみの構成（左側）では8.3の評価と同様に乱数生成に浮動小数命令を使用したのに対して，混載した結果（右側）はPowerPC と M32R の両レガシーISA ともに整数演算になったことと M32R は単純な命令が多いことから，異種混載モデルの方が命令あたりの速度性能は向上した．異種マルチプロセッサの CPU は半数ずつ割り付けているが, 合計 CPU 数が 4, 8, 16 個と増えても 2-CPU 時の 96%，92%，85%と性能がほぼ均一である傾向を示すことを確認した．

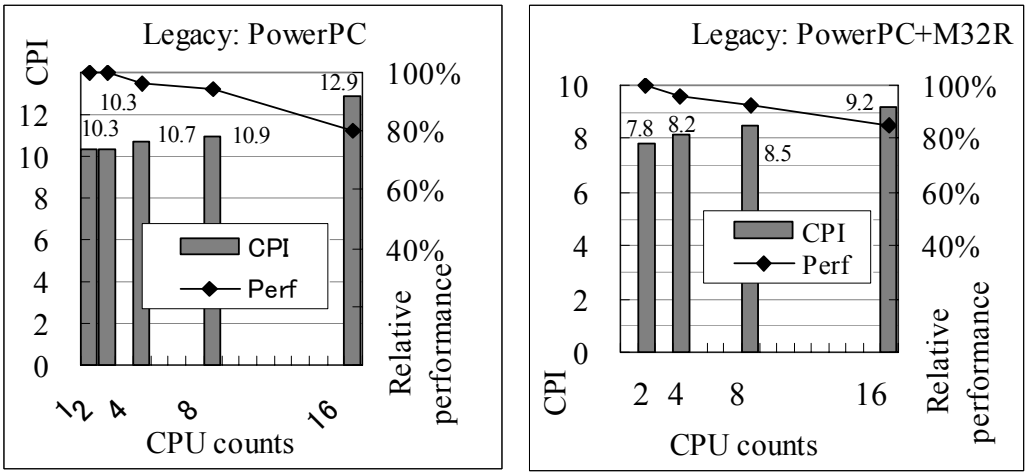


図52 同種マルチプロセッサと異種マルチプロセッサシミュレーションの速度性能

8.5. まとめ

前章（第 7 章）では6.2で提案した（1）～（5）についてその効果を評価した．それに対し，本章では（6）マルチプロセッサシミュレーション方式に焦点を当てて評価を行った．その結果，提案方式を実装した ESPRIT/sim は，7.8で述べたユニプロセッサの簡易実装かつ高性能という特長に加えて，その提案手法が有効で同種及び異種マルチプロセッサシステムにも性能向上効果が高いことを示した．

具体的には，(6-1)各プロセッサのシミュレーションは指定したスイッチ間隔で切り替えながら順次評価を行う，(6-2) 変換後の命令列を格納するコード変換キャッシュはプロセッサごとに持つ，(6-3）多様な構成を採りうるキャッシュメモリの評価には C／C++実装のキャッシュモデルを用いてバイナリ生成したフィルタ機能によりオーバヘッドを削減する手法を提案した．これらの手法は（1）～（5）の手法の特性を活かしており，構造を単純化して開発量が小さくスモールスタートかつ品質の確保が容易で簡易的なバイナリ変換の特長を，異種マルチプロセッサまで拡大している．

CPU 数が 2～16 個の同種マルチプロセッサ構成で SPLASH-2 を評価し，その命令シミュレーションに掛かる CPI 値が 6.6～9.5 と小さくその速度性能は高いことを示した．著名な命令シミュレータ sim-fast より 1-CPU 構成の比較で 8～14 倍速い．CPU 数を増やしてもオーバヘッドは小さく 16-CPU でも 1-CPU の 76～80%の速度性能がある．

また，異種マルチプロセッサ動作として M32R と PowerPC の混載の 2～16-CPU のモデルで RADIX-sort の速度性能を評価した．その速度性能も同種マルチプロセッサと同様に高く，CPU を増やしたときのオーバヘッドも少なく 16-CPU では 2-CPU の 85%の性能を示した．

キャッシュシミュレーションの速度性能も評価した．CPU 数が 2～16 個の同種マルチプロセッサ構成では，CPI 値が 11.6～23.4 と小さく高性能である．ここでも，CPU 数を増やしても速度性能の低下は小さく 16-CPU でも 1-CPU の 72～90%の速度性能を達成している．SPLASH-2 のベンチマークの速度性能は著名なキャッシュシミュレータ sim-cache より 1-CPU 構成の比較で 10～19 倍速く，CPI 値 11～21 を達成している．

以上のように，異種マルチプロセッサを含む組込み機器向けの性能評価シミュレータの高速化手法として，ESPRIT/sim に搭載した簡易的なバイナリ変換の実装方式を述べ，その評価結果よりその手法が有効なことを述べた．

第9章 結言

本章では，これまで述べてきた研究内容をまとめるとともに，今後の展開について述べる．

9.1. インタプリタに関する本研究のまとめ

まず，インタプリタに関する研究についてまとめる．第 2 章では，インタプリタ方式のエミュレータの有効性と課題を述べた．2.3.2で C 言語による実装方式の事例として SimpleScalar の PISA の sim-safe を取上げ，2.3.4ではその評価を行った．その結果，PISA のアセンブリ言語記述の理想的なインタプリタコードに対し sim-safe は 0.39～0.55 倍の速度性能に留まっていること，SPEC CPU95 の 13 本のプログラムの CPI を測定しその値がプログラムに依存せずに同様な値であることと，sim-safe のオーバヘッドの原因を明らかにした．また，PISA の命令はデコードは 1 回で済む特殊なものであり市販 CPU とは異なるため，MIPS-IV ライクな命令語体系に改造した sim-MIPSlike も試作して評価した．更に sim-safe と sim-MIPSlike の評価よりそれらのコアループの処理時間が全体の 73～79%の時間を占め，コアループの速度性能の改善が効率的であることを示した．sim-safe に見られるような switch 方式のインタプリタは，そのコードの最適化がコンパイラ任せとなりインタプリタの性能向上への課題を解決できる着実な設計手法になり得ないなどの問題を指摘した．

これらの結果を踏まえ，第 3 章では switch 方式の対極にある function 方式を設計手法として提案した．まず function 方式の原型 emu-PISA-base の試作を行いその問題点を挙げた．次に，それを解決する改良 function 方式を提案し，その実装例 emu-MIPSlike-opt を説明した．

第 4 章では，emu-PISA と emu-MIPSlike の速度性能を評価した．改良 function 方式は理想インタプリタコードに対し I-LOOP の速度性能では，emu-PISA-opt が 0.81～0.89 倍，emu-MIPSlike-opt では 0.81～0.85 倍を実現した．SPEC CPU95 では特にホストが RISC のときの速度性能の向上は大きく，switch 方式を基準とすると PISA 用のインタプリタは 1.63～1.75 倍，MIPSlike 用は 1.48～1.77 倍の速度性能を達成した．更に，別のレガシーISA の評価のため，PowerPC，SH4 及び M32R のインタプリタを switch 方式と改良 function 方式の両方について試作を行った．ここでも，改良 function 方式が switch 方式の 1.33～2.25 倍の高速であり，方式として優位性があることを示した．これらの結果より，レガシーアーキテクチャによらず改良 function 方式が C 言語実装のインタプリタ方式のエミュレータ実装に有効であるといえる．参考としてホストを x86 とした評価を実施し，レジスタ本数が制約となり性能向上は困難であるが，実用レベルのレガシーISA では switch 方式より優れていることを確認した．また，コアループの占める割合が約 70～80%と高いことから，コアループのみの試作と評価を行うことにより性能値の概略見積りが可能となるといえる．

9.2. バイナリ変換に関する本研究のまとめ

次に，バイナリ変換に関する研究についてまとめる．ESPRIT/sim は，市販のプロセッサを使ったシステムの性能解析を目的としたシミュレータである．特に，異種マルチプロセッサシステムのシミュレーションに適用可能な数少ない性能評価シミュレータであり，その速度性能を高めるバイナリ変換の手法について第 6 章で提案した．(1) ホスト命令を生成する対象は使用頻度の高いレガシー命令に限定し，その他のレガシー命令用にはインタプリタのソースコードを併用したバイナリ変換を行う，(2) トランスレータの実装は，開発コストに見合った効率的な変換レベルの選択ができるようにする，(3) トランスレータの構造は 3 階層にして，レガシー命令セットとホストの命令セットを独立にし，組合せを不要にする，(4) レガシー命令間に渡った複雑な最適化は行わず，また変換後の命令列を格納するコード変換キャッシュは固定長の構造を採るなどの単純化をする，(5) 多様な使い方に対応し 3 種類のアドレス空間モードを設ける，(6) 異種マルチプロセッサに適したコード変換キャシュやバイナリ変換に適したシミュレーションスイッチ方式を使う，などである．それらの特長により，開発量が小さくスモールスタートかつ品質の確保が容易で簡易的なバイナリ変換を実現している．

第 7 章では，評価の結果，このような実装によりレガシーISA が PISA，ホストが Core 2 Duo のときの CINT95 のシミュレーション実行では，インタプリタの 3～14 倍，sim-fast の 8～30 倍の速度性能を達成していることを示した．また，ホストを MPC7450 に替えた評価，レガシーISA に PowerPC と M32R を加えた評価も行った．それらの結果，ホスト，レガシーISA によらず同様な速度性能の向上効果があることも示した．

また，第 8 章では，SPLASH-2 のマルチプロセッサ動作の評価例にて，sim-fast の 8～14 倍，sim-cache の 10～19 倍の速度性能があり，CPU 数を増やしてもオーバヘッドは小さく 16-CPU でも 1-CPU の 76～90%の速度性能を確保できることを示した．また，異種マルチプロセッサ動作の評価例も示し，これも 16-CPU で 2-CPU の 85%の速度性能があり同様に高速であることを示した．以上のように，異種マルチプロセッサを含む組込み機器向けの性能評価シミュレータの高速化手法として，ESPRIT/sim に搭載した簡易的なバイナリ変換の実装方式を述べ，その評価により有効なことを述べた．

9.3. 今後の取り組むべき課題

本研究では，命令仕様が RISC 系の CPU をホストとしたインタプリタ方式のエミュレーション性能の向上に焦点を当て研究してきた．それは，x86 系の CPU は PC を中心としたコンシューマ志向の製品であり，長期間に渡る製品供給を必須条件とする産業用途には馴染まない側面があったからである．しかしながら，インテル社も 2008 年 4 月から ATOM プロセッサを正式に発表して出荷し始めたことから，これまで安定供給されてきた RISC 系の組込み用途の CPU の淘汰が加速する懸念もある．したがって，x86 をホストとしたインタプリタの高性能化や，同 64 ビットモードを活用して明示的に利用可能なレジスタ本数を拡大してそのインタプリタ性能を向上させる研究も必要になると思われる．

本研究のモチベーションとなった異種マルチプロセッサシステムは組込み機器の一般的な構成であり，今後，そのシミュレーションのニーズが高まる．シミュレーション対象の命令セットアーキテクチャに x86 を加えた拡張や，マルチスレッド機能がある CPU コアの高速シミュレーションのニーズも増えてくる．そして，まだ研究するべきことには，現時点ではモデリングに手間が掛かる I/O を簡易的にモデル化してかつ高速にシミュレーションする技術などがある．また，本研究で扱った性能評価向けのバイナリ変換アクセラレータの研究も，ホスト CPU 単体の性能向上が飽和しつつあることに対応し，今後はマルチコア構成のホストを意識した性能向上にも取り組むべきと考える．シミュレーション対象が複数個の CPU という好条件下でも，キャッシュメモリの並列シミュレーションによりホスト間のオーバヘッドが大きくなり性能上の課題となることが SimOS など過去の研究結果から，予見できている．マルチコアのホストでは 2 次キャッシュや 3 次キャッシュが共有され細粒度の並列化も可能となるため，従来のデータ並列性に着目した手法ではなく機能並列を含めたシミュレータの高速化も新たな研究課題になると考える．

本研究の成果を，命令レベルシミュレーションの高速化の更なる研究と応用に活用いただき，組込み機器の発展に寄与いただければ幸いである．

謝辞

本研究において長年に渡りご指導とご鞭撻を頂いた慶応義塾大学情報工学科教授 天野英晴博士に深く感謝申し上げます．また，本論文の執筆にあたり，慶応義塾大学情報工学科教授 山本喜一博士，情報工学科准教授 山崎信行博士，システムデザイン工学科准教授 西宏章博士には，ご多忙中にも関わらず懇切丁寧な助言を頂き論文の完成ができ深く感謝致します．

慶応義塾大学理工学研究科博士課程への社会人入学の機会を与えて下さった，三菱電機（株）情報技術総合研究所元所長 肥塚裕至氏に深く感謝します．博士課程派遣に推挙し励まして下さった元同副所長 風間成介氏に深く感謝します．また，研究を進めるにあたりご指導ご鞭撻を頂いた西井龍五氏，勝山光太郎氏，菅隆志氏，中川路哲男氏，白井健治氏，高畑泰志氏，武田保孝氏，小島泰三氏に御礼申し上げます．評価にあたり，マシンを提供して協力下さったリアルタイムプラットフォーム技術部の方々に感謝致します．

また，慶応義塾大学天野研究室での環境設定，ベンチマーク SPLASH-2 向けの同期マクロ作成に助言を頂き，論文の初稿の査読にコメントを頂いた田辺靖貴氏に感謝の意を表します．親子ほど年の離れた私に温かく接してくださった天野研究室の学生の方々に感謝いたします．

論文目録

本研究に関する論文

【公刊論文】

1. 近江谷康人，天野英晴，“C 言語実装を用いたインタプリタ方式の命令エミュレーション性能の向上”，電子情報通信学会論文誌 和文 D vol. J91-D No.2 pp.413-434, 2008 年 2 月
2. 近江谷康人, 天野英晴，“性能評価シミュレータ ESPRIT/sim における動的バイナリ変換アクセラレータの簡易実装”，電子情報通信学会論文誌 和文 D vol. J91-D No.10　pp.2449-2465, 2008 年 10 月
3. 平岡誠一，近江谷康人，西川浩司，山崎弘巳，“ソフトウェア資産活用に有効なバイナリトランスレーション技術”，三菱電機技報 vol.77, no.7, pp.487-490, 2003 年 7 月

【国際会議】

1. Y.Ohmiya and H.Amano “ESPRIT/sim: A HIGH SPEED PERFORMANCE-SIMULATOR FOR HETEROGENEOUS EMBEDDED MULTIPROCESSORS”, IASTED International Conference Parallel and Distributed Computer and Systems, pp.252-257, Nov. 2008

【その他】

1. Y.Ohmiya and H.Amano, “Dynamic Power Analysis of Embedded Systems Based on Instruction Execution Driven Simulation”, Coolchips XI poster vol.2008 pp.128, Apr. 2008
2. 近江谷康人，天野英晴，“インタプリタ方式による命令エミュレーション処理性能”，情報処理学会研究報告 vol.2005, no.164, pp.85-90, 2005 年 8 月
3. 近江谷康人，天野英晴，“命令シミュレーション手法を用いた性能シミュレータ”，情報処理学会研究報告 vol.2006，no.169, pp.133-138, 2006 年 8 月

参考文献

[1] C. Cifuents, V. Malhotra, “Binary Translation: Static, Dynamic, Retargetable?”, pp.340-349, Software Maintenance 1996, Proceedings, International Conf. on 4-8 Nov. 1996.

[2] E. Altman, D. Kaeri, and Y. Sheffer, “Welcome to the Opportunities of Binary Translation”, IEEE computer, 33, pp.40-45, 2000.

[3] C. Zheng, and C. Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation”, IEEE computer, vol.33, no.3, pp.47-52, 2000

[4] R.A. Uhlig and T.N. Mudge, “Trace-Driven Memory Simulation: A Survey”, ACM Computing Surveys, vol.29, no.2, pp.128-170, 1997.

[5] 吉瀬謙二，片桐孝弘，本多弘樹，弓場敏嗣，“SimCore/Alpha Functional Simulator の設計と実装”，信学論（D-I），vol.J8-D-I no.2, pp.143-154, Feb. 2005.

[6] 近江谷康人, 天野英晴 “インタプリタ方式による命令エミュレーション性能”，情処学研報，vol.2005, no.164, pp.85-90, Aug 2005.

[7] D. Burger, T.M. Austine, “The SimpleScalar tool set, version 2.0”, Tech. Report 1342, Computer Science Department, University of Wisconsin-Madison, June 1997.

[8] http://www.mips.com/content/PressRoom/TeachLibrary/RseriesDocs/

[9] Motorola,“PowerPC™ Microprocessor Family: The Programming Environments for 32-bit Microprocessors”, Motorola, 1997.

[10] ルネサステクノロジ（株），“SuperH RISC engine ファミリ”，http://japan.renesas.com/media/products/mpumcu/child_folder/03_sh.pdf

[11] ルネサステクノロジ（株），“M32R ファミリ M32R/ECU シリーズ”，http://japan.renesas.com/media/products/mpumcu/child_folder/04_m32r.pdf

[12] Freescale semiconductor, “MPC7450 RISC Microprocessor Family Reference Manual, http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM.pdf

[13] SPEC CPU95, http://www.spec.org/cpu95/

[14] M. Rosenblum, S. Herrod, E. Witchel, A. Guputa, “Complete Computer Simulation: The SimOS Approach”, vol.03, no.4, pp.34-43, IEEE Parallel and Distributed Technology, 1995.

[15] B. Cmelik, D. Keppel, “Shade: A Fast Instruction Set Simulator for Execution Profiling”, pp.128-137, ACM SIGMETRICS, Nashville, TN, 1994 (also available from http://portal.acm.org)

[16] “命令エミュレーション方法”，近江谷康人，特開 2002-182928

[17] 平岡精一，近江谷康人，西川浩司，山崎弘巳，“ソフトウェア資産活用に有効なバイナリトランスレーション技術”，三菱電機技報，vol.77, No7，pp.59-62，July 2003.

[18] 中田尚，大野和彦，中島浩，“高性能マイクロプロセッサの高速シミュレーション”，先進的計算基盤システムシンポジウム SACSIS2003 論文集，pp.89-96, 2003

[19] 小谷田重則，平井義郎，和田美加代，“GRANPOWER6000 シリーズの高速化技術”，FUJITSU, vol.49, no.1, pp.21-25, Jan. 1998

[20] FLEX-ES, http://www.funsoft.com/

[21] R. Yung, “Evaluation of a Commercial Microprocessor”, TR-98-65, pp.5-6, Sun Microsystems Inc, USA, 1998. (http://research.sun.com/techrep/1998/smli_tr-98-65.pdf)

[22] “System/370 Architecture Reference Summary”, pp.14-17, NY, USA, 1986.

[23] H. Lowery, B. Mitchell, “Mission computer replacement prototype for special operations force aircraft an application of commercial technology to avionics”, Digital Avionics Systems Conf. 2000. Proc. DASC. The 19th, vol.1, pp.7-13, Oct. 2000.

[24] Transitive corporation, http://www.transitive.com/

[25] 萩原 宏，“マイクロプログラミング”，産業図書，1977

[26] S.S. Mukherjee, S.V. Adve, T.Austin, J.Emer, and P.S. Magnusson: “Performance simulation tools”, Computer, vol.35, no.2, pp.38-39, 2002.

[27] C. Cifuentes, and M.V. Emmerik, “UQBT: Adaptable Binary Translation at Low Cost”, IEEE computer, vol.33, no.3, pp.60-66, 2000.

[28] IBM corporation, ”DAISY: Dynamically Architected Instruction Set from Yorktown”, http://www.research.ibm.com/daisy/

[29] E. Witchel and M. Rosenblum, “Embra: Fast and Flexible Machine Simulation”, Proc. of the 1996 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, pp.68-79, Philadelphia, May 1996.

[30] M. Wakabayashi, H. Amano, “Environment for multi-processor simulator development”, Parallel Architectures, Algorithms and Networks, 2000. I-SPAN 2000 Proc. pp.64-71, International Symp. Dec. 2000.

[31] T. Nakada, H. Nakashima, “Design and Implementation of a High Speed Microprocessor simulator BurstScalar”, Proc. 12th IEEE/ACM International Symp. Modeling, Analysis and Simulation of Computer and Telecommun. Systems, pp. 364-372, October 2004.

[32] T. Nakada, T. Tsumura, H. Nakashima, “Design and Implementation of a Workload Specific Simulator”, Proc. of 39th Annual Simulation Symp. pp.230-243, 2005.

[33] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A.Gupta, “The SPLASH-2 programs: Characterization and methodological considerations”, Proc. 22nd International Symp. on Computer Architecture, pp.24-36, Jun. 1995.

[34] SimpleScalar LLC, http://www.simplescalar.com/

[35] 近江谷康人，天野英晴，“命令シミュレーション手法を用いた性能シミュレータ”，情

処学研報，vol. 2006, no.169, pp.133-138, Aug. 2006.

[36] P.S. Magnusson, “Efficient Instruction Cache Simulation with a Threaded-Code Interpreter”, Proc. of the 97 Winter Simulation Conf. pp.1093-1100, 1997.

[37] P.S. Magnusson, F. Dahlgren, “SimICS/sun4m: A Virtual Workstation”, Proc. of USENIX Annual Technical Conf. New Orleans, pp.1-10,1998.

[38] 近江谷康人, 天野英晴, “C 言語実装を用いたインタプリタ方式の命令エミュレーション性能の向上”，信学論（D)，vol.191D no.2, pp.413-434, Feb. 2008 .

[39] C. Cifuentes, B. Lewis and D. Ung, “Walkabout – A Retargetable Dynamic Binary Translation Framework”, http://research.sun.com/techrep/2002/smli_tr-2002-106.pdf

[40] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, “Dynamic and Transparent Binary Translation”, IEEE computer, 33, pp.54-59, 2000.

[41] Crusoe CMS, http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf

[42] SystemC, open systemc initiative, http://www.systemc.org/

[43] 新舎隆夫，伊藤徳義，小野裕幸，大西洋一，“ハードウェア/ソフトウェア協調検証の高速化技術の開発と製品化”，Embedded Technology 2005 最先端 SoC 設計技術セミナー，Nov. 2005（http://www.starc.jp/download/et2005/02_shinsha.pdf)

[44] 若林正樹, 天野英晴，“並列計算機シミュレータの構築支援環境”, 信学論 (D-1), no.3, pp.247-256, Mar. 2001.

[45] 田辺靖貴，塙敏博，天野英晴，“マルチプロセッサシステム構築支援スーパスカラプロセッサモデルの開発”，信学論（D-I)，vol.J90-D, no.6, pp.1428-1444, Jun. 2007.

[46] R.F. Cmelik and D. Keppel, “Shade: A Fast Instruction Set Simulator for Execution Profiling”, SUN Microsystems Technical Report TR-93-12, July 1993.

[47] F.G. Soltis, 日本アイビーエム，“Inside the AS/400 (second edition) ”，インフォクリエイツ, 1998.

[48] H. Matsuo, S. Imafuku, K. Ohno, H. Nakashima, “Shaman: A Distributed Simulator for Shared Memory Multiprocessors”, In Proc. 10th IEEE/ACM International Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 347–355, Oct. 2002.

[49] Patrick Bohrer, James Peterson, Hazim Shafi, “Advances in PowerPC System Simulation”, pp.1-74, ISPASS Workshop, March, 2003 (http://www.power.org/resources/devcorner/cellcorner/MamboTutorial-2003-03.pdf)

[50] T. Lindholm and F. Yelli, Sun Microsystems Inc., ”Java Virtual Machine Specification 2nd Edition”, 2000.

[51] T. Suganuma, T.Ogasawara, M. Takeuchi, T.Yasue, M. Kawahito, K. Ishizaki, H. Komatsu and T. Nakatani, ”Overview of the IBM Java Just-in-Time compiler”, IBM System J. vol.39, 1, pp.175-193, 2000.

[52] Sun Microsystems Inc., ”The CLDC HotSpot(tm) Implementation Virtual

Machine”,
http://java.sun.com/producs/hotspot/dosc/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v.1.4.1_1002_1.html

[53] http://www.vmware.com/jp/

[54] F. BELLARD, ”QEMU a Fast and Portable Dynamic Translator”, USENIX Association, FREENIX Track: USENIX Annual Conf. pp.41-46, 2005. (http://bellard.org/qemu)

[55] R.D. Nielsen, “DOS on the DOC”, NeXTWorld, pp.50-51, Mar. 1991.

[56] Sparc, http://www.sun.com/

[57] Cygwin, http://www..cygwin.com/

[58] Rosseta, http://www.apple.com/jp/rosetta/

[59] V. Bara, E. Duesterwald and S. Banerjia, “Dynamo: a transparent dynamic optimization system”, Proc. of the 2000 ACM SIGPLAN Conf. on Prgramming Language Design and Implementation (PLDI), pp.1-12, 2000.

[60] 矢野聖宗，中田尚，津邑公暁，中島浩，“時間軸分割並列化による高速マイクロプロセッサシミュレーション”，情処学研報，vol. 2006, no.169, pp.139-144, Aug. 2006.

付録

バイナリ変換の run_loop の実装例

（フラットモードとマップドモードのソースに対応）

```
for(;;) {

large_loop:if(pe->nia_effective) {
      pe->pc = (pc_copy = pe->nia);
      pe->nia_effective = 0;
   }
   ix = (pc_copy & CC_IX_MASK) >>CC_IX_SHIFT;
   if(ccd[ix].pc == pc_copy) { // CCD ヒット?
      //- バイナリ実行 -//
      pe->instrcnt += (*ccd[ix].exe)();
      pc_copy = pe->pc;
      if(pe->expcond)  goto exp;
      else if(pe->instrcnt < stop_cnt)
         goto large_loop;
      else goto exit_loop;
   }
   if(ctd[ix].pc != pc_copy) { //CTD ミス?
      ctd[ix].pc=pc_copy; ctd[ix].cnt=0;
   }
   if(++ctd[ix].cnt > threthold) {
      ctd[ix].pc = ccd[ix].pc;
      ctd[ix].cnt = 0;
      //- ISA 専用の totanslator を呼ぶ
      pe->trans_common(pc_copy);
      //- 例外かﾌﾞﾚｰｸﾎﾟｲﾝﾄ発生?
      if(ccd[ix].pc == 0xFFFFFFFF)
           goto interp;
      //- バイナリ実行 -//
      pe->instrcnt+=(*ccd[ix].exe)();
      pc_copy = pe->pc;
      if(pe->instrcnt < stop_cnt)
           goto large_loop;
      else goto exit_loop;
   }
```

```
// ここからはインタプリタ動作
interp:  if (pe->expcond ||
    !(pc_copy <= max_mem_copy))goto exp;

interp_loop://インタプリタ内の繰返し
  icode_u =
    *(mema = GET_MEM_ADR_4B(pc_copy));
  icode = *(++mema);
  // InsFn の呼出し
  (*(insfnc_ptr_copy+(icode_u & 0xFF)))();
  pe->instrcnt++;
  pe->pc = (pc_copy += 8);
  if(pe->nia_effective) {
     pe->pc = (pc_copy = pe->nia);
     pe->nia_effective = 0;
     if(pe->instrcnt < stop_cnt)
        goto large_loop;
     goto exit_loop;
  } else {
     if(pe->expcond) goto exp;
     else if(pe->instrcnt >= stop_cnt)
        goto exit_loop;
     else if (!(pc_copy <= max_mem_copy))
        goto exp;
     else goto interp_loop;
  }

} // end for
// ここからは例外処理
exp:   // 例外フラグを検査をするコードと
       // 分岐処理の記載は省略.
      goto exit_loop;
exit_loop: return;
```