博 士 論 文

帰納的なシミュレーション・ポイント選出手法に関する研究

(A Study on Inductive Methods to Select Simulation Points)

指 導 教 員

坂井 修一 教授

崔 珉誠

(Choi Minseong)

概 要

プロセッサの研究・開発においては，アーキテクチャやデザイン上の工夫の効果を測ったり，キャッシュ・サイズなどの各種パラメタを決定したりするために，個々のアプリケーション・プログラムに対する対象プロセッサの実行性能を知る必要がある．そしてそのためには，実際にプロセッサを製造することなく実行性能を知ることが重要であり，個々のプログラムを実行するプロセッサの実行性能をソフトウェアによって再現するシミュレーションが一般的に行われる．

しかし，特に最近の Out-of-Order なプロセッサの場合，そのシミュレーションには極めて長い時間がかかるという問題がある．その根本的な原因は，対象アプリケーション・プログラムの命令数が桁違いに多いことであり，シミュレータを数倍程度高速化することは，重要なことではあるが，抜本的な解決にはならない．

そこで，対象アプリケーション・プログラムのごく一部のみをシミュレートした結果から全体の性能を推定するサンプリング・シミュレーションが行われる．シミュレートする部分をシミュレーション・ポイントと呼ぶ．

シミュレーション・ポイントの選出手法としては，SimPoint がよく知られている．この手法はプログラムを事前エミュレーションして得られる PC の系列に基づいて，シミュレーション・ポイントを選出する．この手法は，同じ静的区間を実行する動的区間が同じフェーズであり，プロセッサは同じような振る舞いを示すという仮定に基づいている．

しかし，この仮定には反例がある．たとえば，同じコードであっても，入力が異なると，処理するデータ量に応じてキャッシュ・ヒット率が変化し，IPC に大きな影響を及ぼす．キャッシュ・ヒット率は，キャッシュ・サイズの影響を大きく受ける．キャッシュ・サイズは，ターゲット・プロセッサのマイクロアーキテクチャの

パラメタである．したがって，この問題を克服するためには，マイクロアーキテクチャに関する情報を考慮する必要がある．

本論文では，特徴的なプロセッサすべてが同じ Instruction Per Cycle（IPC）を示す区間が同じフェーズであるという仮定に基づいて，特徴的なマイクロアーキテクチャを持つ複数のプロセッサを事前にシミュレートすることによって得られた IPC の系列からシミュレーション・ポイントを選択する帰納的な手法を提案する．

また，基本的な手法に対する以下の 3 つの改良手法を提案する．

1. キャッシュ容量によって生じるフェーズに対応する超多階層キャッシュを持つ基底モデル．
2. 事前シミュレーションにかかる時間を削減するエミュレーションをベースとする基底モデル．
3. ベンチマークに含まれるすべてのプログラムの実行を単一のワークロードとみなして，すべてのプログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する集合的な手法．

キャッシュの容量によってフェーズが生じる可能性がある．それに対して，キャッシュ容量固有のフェーズを持つ多階層キャッシュ基底モデルに追加することで，帰納的手法をもっとより正確な推定を達成した．

シミュレーション・ポイントを選出するためにいくつかの基底モデルを事前シミュレーションする．事前シミュレーションは，原則的には，ベンチマークに対して一度だけ実行すればよい．それにも関わらず，事前シミュレーションにかかる時間は膨大である．そのため，事前シミュレーションにかかる時間を削減するエミュレーションをベースとする基底モデルに取り替えすることによって，帰納的手法の短所を補完した．

異なるプログラムにも，プロセッサが同様に振る舞う区間が存在する特徴を利用して，ベンチマークに含まれるすべてのプログラムの実行を一つのワークロードとみなして，すべてのプログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する集合的な帰納的選出手法を提案した．

SPEC CPU 2006 の ref 入力における先頭 100G 命令の IPC 推定を行った結果，提案手法は，100G 命令の約 0.1% をシミュレートした場合，平均 0.4% の IPC 推定誤差を達成した．

Abstract

In research and development of processors, it is necessary to know the execution performance of the target processor for each application program in order to evaluate improvement in architecture and/or in design, or to decide the parameters such as the cache sizes. To this end, it is important to know the execution performance without actually manufacturing the processor but with simulation which reproduces the execution performance by simulator software.

However, particularly in the case of recent out-of-order processors, this simulation takes a considerably long time. The essential reason of this long simulation time is that the number of instructions in the benchmark programs is several orders of magnitude larger, and speeding-up the simulator several times is important but cannot be a fundamental solution.

Thus, sampling simulation, which inspect overall performance from simulation result of small portion of the target program, is widely used. This potion is called a simulation point.

Existing methods to select simulation points, such as SimPoint, is based on an assumption that dynamic sections executing the same static section of the program are of the same phase and the processor shows similar behavior in these dynamic sections.

However, there are counterexamples for this idea. The target processor executing the same static section of the program behaves similarly. Even when the target processor is executing the same loop or function, the cache hit rates will differ depending on the amount of data it refers, and the IPC can be drastically different.

This paper proposes an inductive method that selects simulation points from the results obtained by pre-simulating several processors with distinctive microarchitectures, based on an assumption that sections in which all the distinctive processors have similar instructions per cycle (IPC) values are of the same phase.

In addition, this paper proposes the following three improving method to the inductive method:

1. Emulation-based basis models that drastically reduces the time for pre-simulation.
2. A basis model with highly-hierarchical caches that deals with phases caused by fluctuating working set sizes.
3. A collective inductive method that selects simulation points from all benchmark programs regarding all of the benchmark programs as one workload.

This paper shows evaluation result of the first 100G instructions of SPEC 2006 programs with ref input. Our inductive method achieved an IPC estimation error of approximately 0.4% by simulating approximately 0.1% of the 100G instructions.

The emulation-based basis models achieved more accurate estimation with less simulation points than SimPoint, but it does not exceed the case of adding the simin advance.

Although the estimation error became worse for some programs, collective inductive method achieved reduction of sampling rate to 61.93% while limiting the degradation of error to 0.5% on average.

目 次

図 目 次

表 目 次

第1章

序 論

プロセッサの研究・開発においては，アーキテクチャやデザイン上の工夫の効果を測ったり，キャッシュ・サイズなどの各種パラメタを決定したりするために，個々のアプリケーション・プログラムに対する対象プロセッサの実行性能を知る必要がある．そしてそのためには，実際にプロセッサを製造することなく実行性能を知ることが重要であり，個々のプログラムを実行するプロセッサの実行性能をソフトウェアによって再現するシミュレーションが一般的に行われる．

しかし，特に最近の Out-of-Order なプロセッサの場合，そのシミュレーションには極めて長い時間がかかるという問題がある．その根本的な原因は，対象アプリケーション・プログラムの命令数が桁違いに多いことであり，シミュレータを数倍程度高速化することは，重要なことではあるが，抜本的な解決にはならない．

そこで，対象アプリケーション・プログラムのごく一部のみをシミュレートした結果から全体の性能を推定するサンプリング・シミュレーションが行われる．シミュレートする部分をシミュレーション・ポイントと呼ぶ．

シミュレーション・ポイントの選出手法としては，SimPoint がよく知られている．この手法はプログラムを事前エミュレーションして得られる PC の系列に基づく演繹的な手法であると言える．本論文では，特徴的なアーキテクチャを持つ複数のモデルを事前シミュレーションして得られる IPC の系列に基づく帰納的な手法を提案する．

1.1 プロセッサのシミュレーション

エミュレーションとシミュレーション ソフトウェアによってプロセッサの動作を再現する場合，この分野では一般に，各命令の実行結果のみを再現することを**エミュレーション**，実行結果以外のターゲット・プロセッサの振る舞いまでも再現することを**シミュレーション**と呼んで区別する．

このことは，プロセッサの定義と平行な関係にある．プロセッサの定義は，一般に，**命令セット・アーキテクチャ** (Instruction-Set Architecture: **ISA**) と**マイクロアーキテクチャ**の 2 階層に分けられる．

エミュレータ エミュレータは，ISA の情報のみを用い，その結果は個々のマイクロアーキテクチャには依存しない．

1 命令のエミュレーションは，典型的には，以下のような処理になる：

1. PC を表す変数を添え字として主記憶を表す配列から命令を取り出す．
2. ビット・マスクとシフトにより，命令のオプコード，オペランドを取り出す．
3. 2 で得られたオプコードにより case 分岐する．
4. 2 で得られたオペランドを添え字としてレジスタ・ファイルを表す配列にアクセスし，命令を実行する．

この処理は，最近の out-of-order スーパスカラ・プロセッサでは 10〜20 サイクル程度で実行できる．

シミュレータ 一方，シミュレータは，同時実行可能な命令数，キャッシュや分岐予測器の構成など，ターゲットとするプロセッサのマイクロアーキテクチャの情報をも用いる．そして，キャッシュ・ヒット率や分岐予測ヒット率などの個々の性能評価指標，そして，総合的な速度指標である **IPC** (Instructions Per Cycle) を得る．

これらの性能指標を正確に求めるため，シミュレータは，ターゲット・プロセッサのサイクルごとの振る舞いをある程度正確に再現する必要がある．特に out-of-order なプロセッサのシミュレーションでは，待ち行列のシミュレーションのように，各命令がサイクルごとにプロセッサのどのユニットにからどのユニットに移動するかを再現する必要がある．フェッチされてからコミットされるまでの in-flight な命令が対象となり，その数は最大 100 程度にもなる．

Cycle-Accurate なシミュレータ このようなサイクルごとの振る舞いを完全に再現するシミュレータは，特に，**cycle-accurate** なシミュレータと呼ばれる．

それに対して，cycle-accurate ではないシミュレータは，典型的には，各命令の実行結果の再現をエミュレーションによって行うものである．その場合，高速化が可能であるものの，予測ミスによって実行されてしまった命令による性能の変化などの再現が不可能になる．また，このようなシミュレータには，得られた性能評価指標に影響を与えるバグの発見が困難になるという問題がある．性能評価指標が正しく再現できないバグがあったとしても，エミュレーションによるプログラムの実行結果には影響を与えないから，シミュレータ自体は正常終了してしまう．そのため，得られた性能評価指標が正しいかどうかの判断は，評価者の直感に頼ることになる．広く用いられているシミュレータであっても，このようなバグが事後に報告されることは珍しくなく，この問題は看過できない．

このような問題を避けるため，第 5 章の評価で用いるシミュレータ鬼斬弐は，完全に cycle-accurate なシミュレータとなっている [1]．鬼斬弐は，命令の実行ステージにおいて命令を実行し，コミット・ステージにおいてオンライン・エミュレータの実行結果によって検証を行う．したがって，予測ミス後の振る舞いも正確に再現される．また，振る舞いを正しく再現できないバグがあった場合には，その部分の再現の accuracy にも依存するが，オンライン・エミュレータによる検証において発見することができる．

長いシミュレーション時間 シミュレーションには，極めて長い時間がかかる．その理由は，以下の 2 つに分けられる：

1. シミュレータの実行速度が遅い．
2. 評価に用いるベンチマーク・プログラムの命令数が非常に多い．

以下，それぞれについて詳しく述べる．

シミュレータの実行速度の遅さ 実機による，いわゆるネイティブ実行の速度に対して，エミュレータやシミュレータによる実行速度の低下の比を Speed-Down (**SD**) と呼ぶ．一般に，エミュレータの SD は 10 程度である．一方，シミュレータの SD は 1,000 以上にもなる．cycle-accurate なシミュレータの SD は更に大きく，10,000 程度にもなることがある．

エミュレータのSDは，1命令の処理を行うために必要なサイクル数で与えられ，上述の通り10〜20程度となる．

シミュレータのSDが1,000以上にもなるのは，やはり上述の通り，最大100程度にもなるin-flightな命令のそれぞれがプロセッサのどのユニットからどのユニットに移動するかを再現する必要があるためである．

SDが1,000としても，実機で10分かかるプログラムには，10,000分，すなわち，7日以上かかる計算になる．

ベンチマーク・プログラムの命令数の多さ 例えば，プロセッサ性能を評価するための最も典型的なベンチマークであるSPEC 2006 [2]の場合，含まれる29個のプログラムのうち最長のものは100T命令程度にもなる [3]．このプログラムをシミュレータで実行するには，年単位の時間がかかる．

その上，後述するように，性能評価のためには，通常，何十通りもパラメタを変えてシミュレーションを行う必要がある．

1.2 サンプリング・シミュレーション

したがって，シミュレータ自体を高速化することは，重要なことではあるが，この文脈では根本的な解決にはならない．たとえばcycle-accuracyを捨てることなどによって数倍程度の高速化が可能であったとしても，年単位のものが月単位に短縮されるだけで，評価に用いるには依然として非実用的であるからである．

そこで実際には，ベンチマーク・プログラムのごく一部のみをシミュレーションする**サンプリング・シミュレーション**が行われる．サンプリング・シミュレーションにおいて，実際にシミュレートされる（動的な）箇所を，**シミュレーション・ポイント**と呼ぶ．

スキップ後シミュレーション 前節で述べたスキップ後シミュレーションは，広く用いられてはいるものの，シミュレーションした部分がベンチマーク・プログラムの特徴を確かに反映したものであるかどうか疑問が残る．実際，SPEC 2006のastar, mcf, omnetppなどでは，1G命令では初期化部分をスキップするには不十分

であることが分かっている [3]．参考のため，Appendix A に，SPEC 2006 の全 29 プログラムの最初の 100G 命令の IPC の遷移を示す．

図 1.1 は，誤差—抽出率グラフを表す．サンプリング・シミュレーション手法の基本的な性能は，このような二次元のグラフによって表すことができる．誤差—抽出率グラフにおいて，y 軸は，性能指標 (同図では IPC) の推定の**誤差** (error)，すなわち，ターゲット・プログラムの全命令をシミュレートすることによって得られた**真の** IPC に対する，選出されたシミュレーション・ポイントのみをシミュレートして推定された IPC の，差の比率 (%) を表す．x 軸は，サンプリング・シミュレーションにおける**抽出率** (sampling rate)，すなわち，ターゲット・プログラムの全実行命令に対する，選出されたシミュレーション・ポイントに含まれる命令の比率 (%) を表す．

一般に，誤差と抽出率との間にはトレードオフがあるので，一つの手法に対しては，同図のような反比例的な曲線がプロットされる．この曲線が原点に近いほど，シミュレーション・ポイント選出手法のとしての性能が高い，すなわち，より少ないシミュレーション・ポイントにより，より正確な IPC 推定が行えることを意味する．実際には，要求された推定誤差 —— 例えば，1% 以下 —— を，より低い抽出率で実現することが期待される．

同図は，1G 命令スキップ後の 100M 命令をシミュレーションした場合の誤差—抽出率グラフである．同図では，三角が後述する SimPoint による誤差—抽出率を，ダイヤモンドがスキップ後シミュレーションの誤差—抽出率を，それぞれ示している．スキップ後シミュレーションでは，全 100G 命令のうちの 100M 命令をシミュレートしたので，抽出率はプログラムによらず 0.1% になる．

同図に示されているように誤差が 160% にも上るプログラムも見られ，ベンチマーク・プログラムの特徴的な部分をまったく実行できていないことが分かる．

ランダム・サンプリング　ランダム・サンプリングを用いれば，統計的な信頼性を確保できる [4–8]．

ランダム・サンプリングでは，全インターバルの中からランダムに一定数を選び，これをシミュレーション・ポイントとする．それぞれのシミュレーション・ポイントを実行した結果得られた評価指標について平均をとることで，プログラム全体の評価指標を推定する．

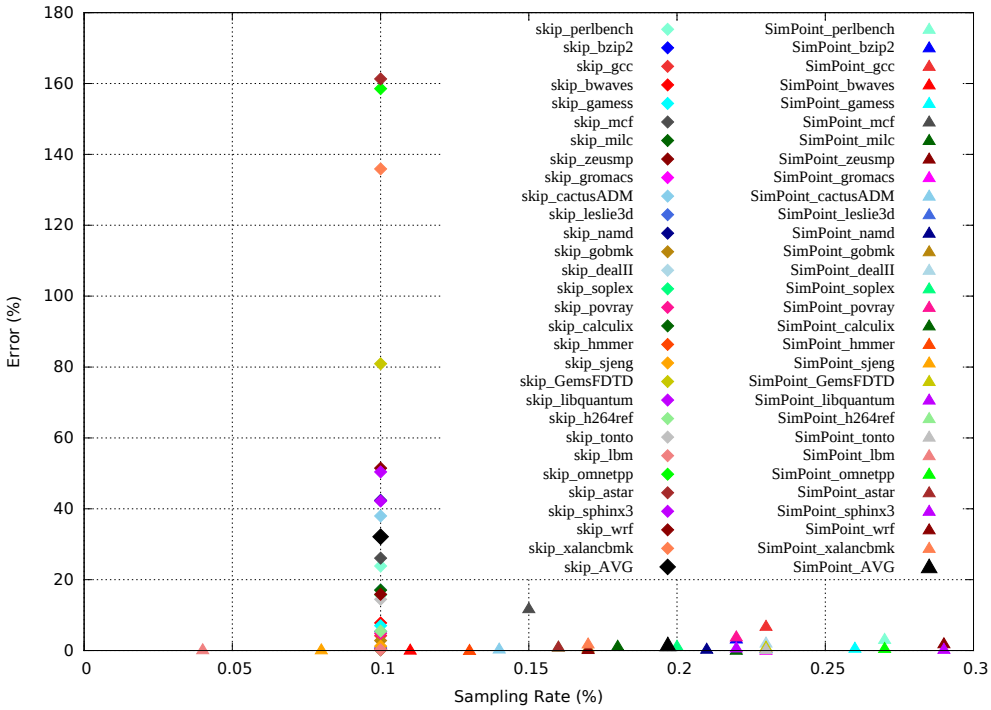


図 1.1: SimPoint とスキップ後シミュレーションの誤差—抽出率グラフ

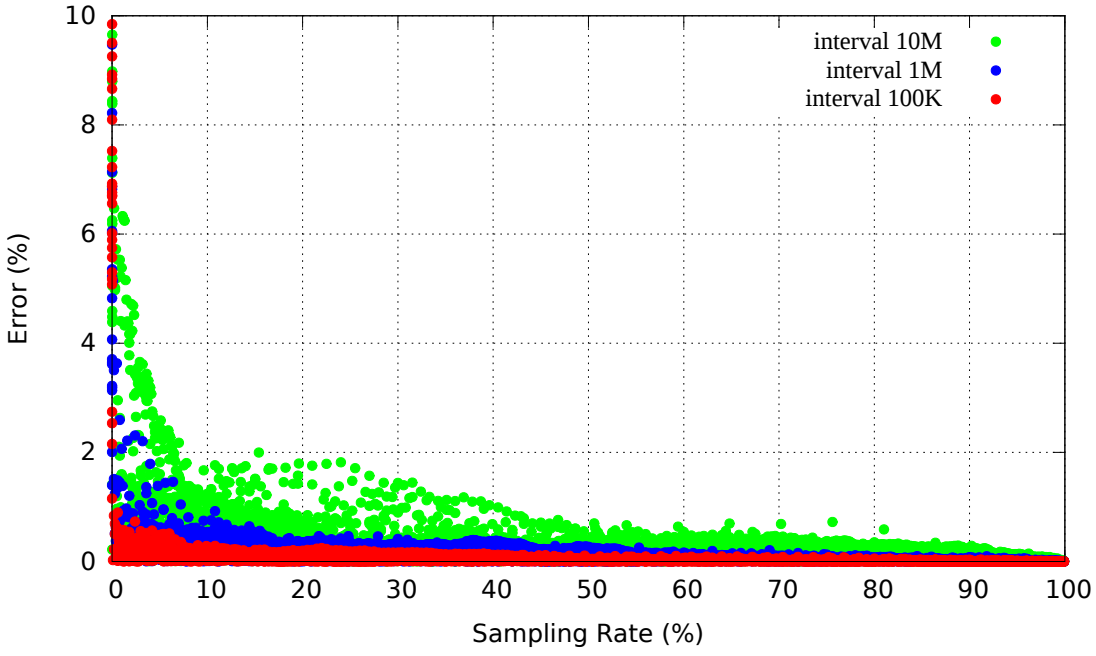


図 1.2: ランダム・サンプリングの誤差—抽出率グラフ

図 1.2 は perlbench の *ref* 入力のランダム・サンプリングの誤差—抽出率グラフである．同図では，選出されるシミュレーション・ポイントの割合を 100% まで徐々に増やし，得られた誤差をプロットしてある．

図 1.1 に示されているように，SimPoint は，perlbench に対しては，わずか 0.26% の抽出率で 0.1% の誤差を達成している．ランダム・サンプリングは，スキップ後シミュレーションに比べればはるかによいものの，SimPoint ほどの誤差—抽出率を達成することはできない．これは，Appendix A に示されているように，IPC の分布は一様ではなく，何らかの指針に基づいてシミュレーション・ポイントを選んだ方がよいことを示している．

1.3 プロセッサのシミュレーションの実際

プロセッサの研究・開発においては，アーキテクチャやデザイン上の工夫の効果を測ったり，キャッシュ・サイズなどの各種パラメタを決定したりするために，対象とするアプリケーション・プログラムに対する対象プロセッサの実行性能を知る必要がある．

スキップ後シミュレーション　典型的には，プログラムの最初の数 G 命令程度をエミュレーションによってスキップし，その後の数百 M 命令程度のみをシミュレーションすることが多い．数 G 命令をスキップするのは，初期化部分を評価に含めないためである．ベンチマークには様々な特徴を持つプログラムが選定されているはずであるが，各プログラムの初期化部分ばかりを評価したのでは，その意図を十分に汲むことができない．

実例　実際に，5.1.4 節で詳しく説明してある regcache [9] は，非レイテンシ指向レジスタ・キャッシュ・システム (Non- Latency-Oriented Register Cache System) で，IPC を保ったまま，レジスタ・ファイルの複雑さを下げる仕組みである．このプロセッサの性能を評価するには，一つの条件で，10G スキップ後 1G シミュレーションを行う場合，6 日かかってしまうデータがある．その上，50 種類のパラメタ条件を変更しながら評価を行うべきである．それを概算したら，1 年だという時間になる．もちろん，研究室では高性能のサーバを利用して，同時実行で性能評価を行っ

ているが，それは膨大な資金がかかっているのを意味する．その場合であっても，新しいアイデアの性能評価には，基本的に一週間ぐらいの時間を必要とする．さらに，細かな修正まで重なった場合，より多くの時間を費やす必要がある．図 1.3 は regcache の性能評価の結果を示す．図からもわかるように，相対 IPC で性能を確認するには，regcache サイズとポート数の組み合わせて 29 個のベンチマーク・プログラムをシミュレーションしなければならない．

その他のプロセンサ カーネル最適化を求めている GPGPU のアーキテクチャ研究でもシミュレータを使用してカーネル動作を確認する．ところが，GPGPU アーキテクチャには多くのコアがシミュレートされるため，Cycle-Accurate なシミュレータには時間がかかる．実際に，実機で 30 分程度実行する GPU が実行するシミュレータによって最大 4 週間当たりもかかる [10]．

スマートフォンなどのアプリケーションでは大体 6 つのワークロードについて評価する．これらの実機実行には，平均 50 時間かかる．また，スマートフォン・

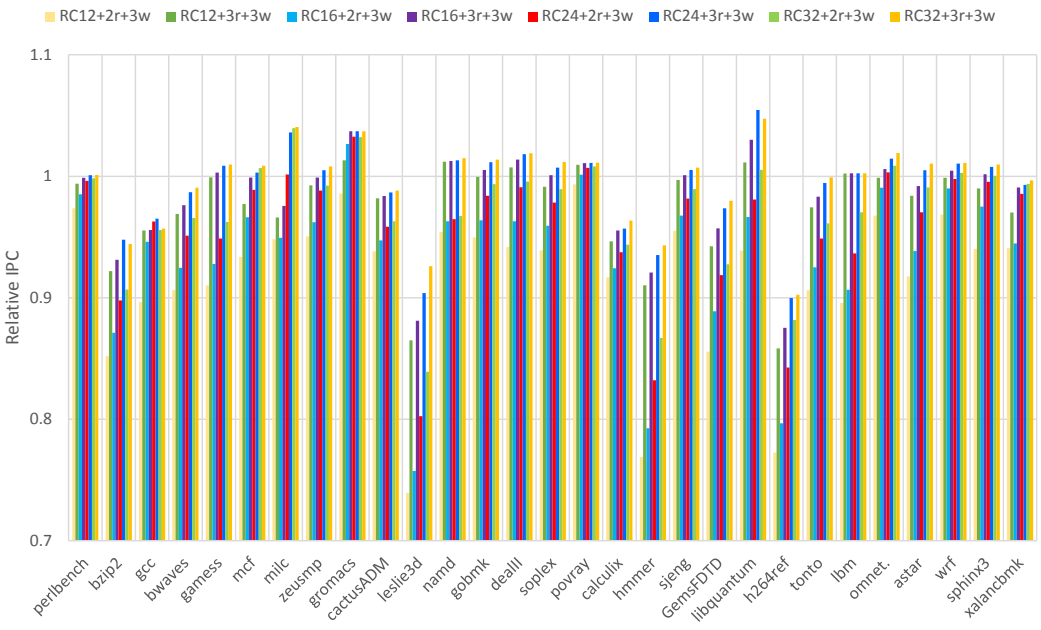


図 1.3: regcache 性能評価とパラメタの相関関係

アプリケーションベンチマーク・プログラム全体で，合計 28 個のプロセスと 158 個のスレッドを変えながら実行する必要がある [11].

1.4 フェーズ検出に基づく方法

よりよいシミュレーション・ポイントを選出するには，**フェーズ検出** [12, 13] の考え方を用いればよい.

フェーズ プログラムの繰り返し構造に起因して，プログラムの動的な区間の中にはプロセッサが同様の振る舞いを示すものが多くある．区間のうち，同様な振る舞いを示す区間は同じフェーズ，異なる振る舞いを示す区間は異なるフェーズと呼ぶことができる．ここで，同様な振る舞いとは，合目的的に定義すれば，IPC，および，その他の性能指標がお互いに近い値を示すことを意味する.

このような考え方に基づけば，プログラム実行の全区間を，例えば数十〜数百種類のフェーズに分類し，同じフェーズに属する区間から 1 つずつを選んでシミュレーション・ポイントとすればよい.

SimPoint フェーズ検出に基づいてシミュレーション・ポイントを選出する代表的な手法として **SimPoint** が挙げられる [14–17]．SimPoint をはじめとするほとんどの手法の特徴は，フェーズ検出（とシミュレーション・ポイントの選出）に**プログラム・カウンタ** (**PC**) の系列を用いることにある.

SimPoint は，まずターゲット・プログラムのエミュレーションを行い,（プログラム実行の全長にわたる）PC の系列を得る．そして，その PC 列を固定長の**インターバル**に分割する．それぞれのインターバルに含まれる PC の種類を基に，**クラスタリング**を行う．すなわち，もし，2 つのインターバルがほぼ同じ PC を含む場合，それらは同じクラスタに分類されることになる．最後に，それぞれのクラスタの代表点をシミュレーション・ポイントの 1 つとして選出する.

IPC，および，その他の性能指標は，選出された各シミュレーション・ポイントの性能指標を，各クラスタのインターバルの数で重み付けした平均として推定される.

既存手法の仮定 SimPointをはじめ，PCを基礎とする手法は，以下を仮定していることになる；すなわち，プログラムの同じ静的部分を実行する動的部分は同じフェーズであり，ターゲット・プロセッサはこれらの部分で同様に振る舞う．

しかしこの仮定には明らかな反例がある．たとえ同じコードであっても，入力が異なれば，プロセッサが同じ振る舞いを示すとは限らない．典型的には，処理されるデータの量が異なれば，キャッシュ・ヒット率は大きく異なり，IPCは大きく変化し得る．実際，SPEC 2006の一部のベンチマークでは，同じ部分が異なるサイズのデータに対して繰り返し実行されており，SimPointの精度を大きく悪化させている．

帰納的手法 SimPointは，プログラムを事前エミュレーションして得られるPCの系列に基づいてシミュレーション・ポイントを選出する手法である．本論文では，特徴的なアーキテクチャを持つ複数のモデルによってプログラムを事前シミュレーションすることによって得られるIPCの系列に基づいてシミュレーション・ポイントを選出する手法を提案する．

SimPointなどの既存手法は**演繹的**であるのに対して，提案された方法は**帰納的**であると言える．

1.5 本論文の構成

次章以降の本項の内容は，以下の通りである：

第2章 シミュレーション・ポイント選出手法

本章では，背景知識として, シミュレーション・ポイント選出の代表的な手法であるSimPointや，その他の手法についてまとめる．SimPointをはじめとするこれらの手法は，実行された命令のPCを基礎にシミュレーション・ポイントを選出する手法である．すなわち，これらの手法は,「プログラムの同じような静的区間を実行している動的区間は同じフェーズである」という仮定に基づく．また，マルチスレッドCPUとGPUプロセンサ，フルシステムシミュレーション手法に関して紹介する．

第3章 帰納的なシミュレーション・ポイント選出手法

本章では，提案手法を理解するためのたとえ話から帰納的なシミュレーション・ポイント選出手法について詳しく述べる．帰納的手法は，特徴的なマイクロアーキテクチャを持つモデルがすべて同じ IPC を持つインターバルが同じフェーズであるという仮定に基づいて，特徴的なマイクロアーキテクチャを持つ複数のモデルを事前にシミュレートしておくことによって得られたインターバルの IPC からシミュレーション・ポイントを選出する．

第 4 章　帰納的な選出手法の改良

本章では，第 3 章で述べた基本的な手法に対する 3 つの改良手法について述べる．すなわち，

1. キャッシュ容量によって生じるフェーズに対応する超多階層キャッシュを持つ基底モデル，
2. 事前シミュレーションにかかる時間を削減するエミュレーション・ベースとする基底モデル，
3. ベンチマークに含まれるすべてのプログラムの実行を一つのワークロードとみなして，すべてのプログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する集合的な帰納的選出手法

の 3 つである．

第 5 章　基本的な手法の性能評価

第 3 章で述べた基本的な帰納手法について，SPEC CPU 2006 ベンチマークの全 29 プログラムの 100G 命令に対して，抽出率, 推定誤差を評価する．その結果，100G 命令の約 0.1% をシミュレーションすることによって，平均約 0.4% の誤差で IPC を推定できることを示している．既存手法である SimPoint に対して，IPC の推定誤差は 1/4 程度になる．

第 6 章　改良手法の性能評価

第 4 章で述べた 3 つの改良手法の評価の結果を示す.

1. 超多階層キャッシュを持つ基底モデルを追加することによって，特定のモデル，特定のプログラムに対してより正確な推定が達成される．

2. エミュレーションをベースとする基底モデルによって推定誤差を SimPoint 改善しながら，事前シミュレーション時間をなくす.
3. 集合的な手法によって，抽出率を 1/3 程度に削減することができる.

第7章 結論

本論文の内容をまとめ，展望を示して, 本論文を結ぶ.

第2章

シミュレーション・ポイント選出手法

本章ではまず，2.1 節において，シミュレーション・ポイント選出の最も代表的な手法である SimPoint [14–17] について原理から工程，クラスタリング方法まで詳しく説明する．SimPoint は，同じ静的区間を実行する動的区間が同じフェーズである仮定に基づいて，命令セット・アーキテクチャの情報のみを用いて，シミュレーション・ポイントを選択する．すなわち，マイクロアーキテクチャの情報は用いない．

2.2 節では，その他の手法として，Software Phase Marker，SimFlex について大まかにまとめる．

これらの手法は，SimPoint をさらに進化した手法でも関わらず，同じ静的区間を実行する動的区間が同じフェーズである仮定には変わらない．Software Phase Marker では，ソフトウェアのレイヤーからフェーズの切れ目を探し，過不足なくフェーズの切れ目の候補ごとに区間に切ることが可能である．一方で，SimFlex では，許容できる誤差から信頼区間を設定し，ランダム・サンプリングした区間をシミュレーション・ポイントとする．

SimPoint をはじめとするこれらの手法は，実行された命令の PC を基礎にシミュレーション・ポイントを選出する手法である．すなわち，プログラムの静的な性質に基づいてシミュレーション・ポイントを選出する．最後に 2.3 節では，これらの手法が基づいている土台の全般的問題点についてまとめる．

2.1 SimPoint

フェーズ検出の考えに基づいてシミュレーション・ポイントを選出する代表的な手法として **SimPoint** が挙げられる [14–17].

2.1.1 SimPoint の原理

第 1 章でも述べたように，SimPoint は，事前エミュレーションによって得られた PC の系列のみを基に，シミュレーション・ポイント選出を行う．

SimPoint は，まずプログラムの実行の全長を固定長の**インターバル**に分割する．そして，それぞれのインターバルに含まれる命令のプログラム・カウンタ (**PC**) を基にインターバルをクラスタリングする．すなわち，2 つのインターバルがほぼ同じ PC を集合として含む場合，それらは同じクラスタに分類される．最後に，それぞれのクラスタの代表点をシミュレーション・ポイントとして選出する．

シミュレーションは，こうして選出されたシミュレーション・ポイントのみに対して行われる．プログラム全体の IPC，およびその他の性能指標は，シミュレーションによって得られた各シミュレーション・ポイントの性能指標を，各クラスタのインターバルの数で重み付けして平均することによって推定される．

このように，SimPoint のシミュレーション・ポイント選出は，PC，すなわち，静的命令に基づいて行われる．したがって SimPoint は，以下の仮定に従ってシミュレーション・ポイント選出を行っていると言える，すなわち,「プログラムの同じような静的区間を実行している動的区間は同じフェーズである」.

2.1.2 SimPoint の工程

事前エミュレーション　まず，SimPoint はターゲット・プログラムを事前エミュレーションすることによって，実行された命令の PC の系列を取得する．

この PC 列は，固定長の**インターバル**に分割される．インターバルの長さは，通常 1M〜100M 命令である（第 5 章参照）.

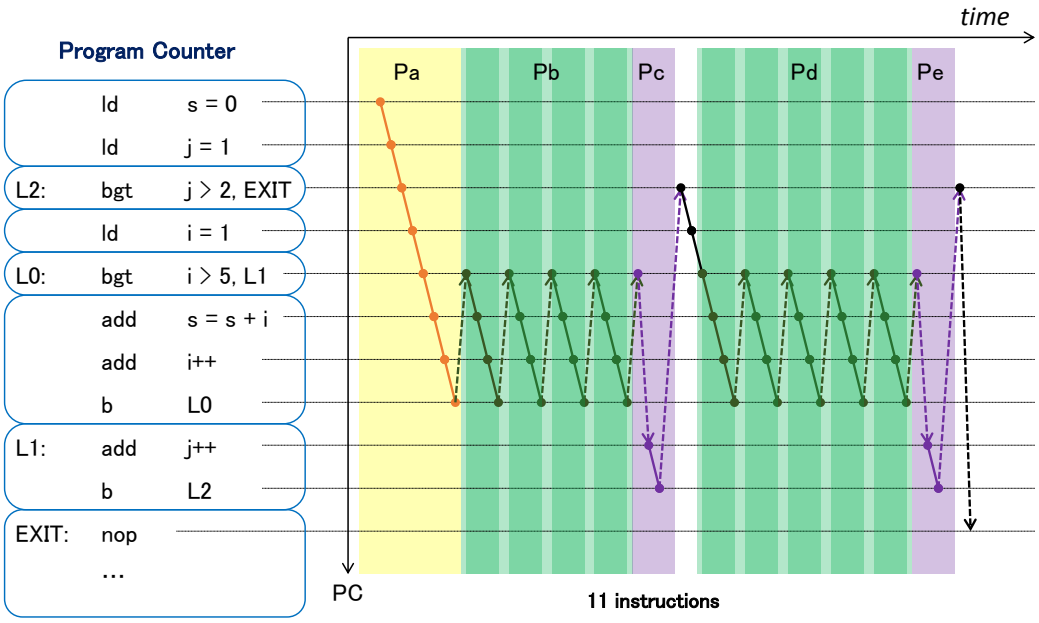


図 2.1: プログラムとフェーズ

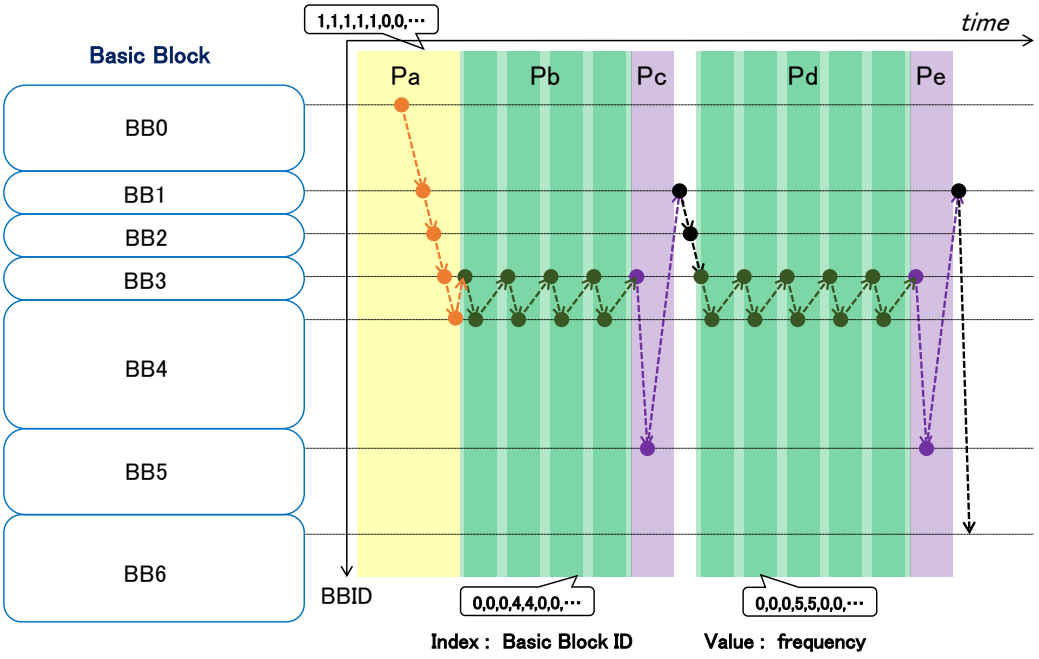


図 2.2: BB ベクトルとフェーズ

PC 列とフェーズ 図 2.1 に，1 から 5 までの和を 2 回求めるプログラムを示す．同図中，Pb と Pd と記された領域では，bgt から b までの 4 命令が繰り返し実行されており，同一のフェーズと見なすことができる．同様に，Pc と Pe も，同一のフェーズと見なすことができる．結果，このプログラムの実行は，{Pa}，{Pb, Pd}，{Pc, Pe} の 3 つのフェーズに分類することができる．

基本ブロック 実際には，データ量の削減のため，PC 列を**基本ブロック**列に変換して，以降の処理を行う．基本ブロックとは分岐や合流を含まない命令の列である．基本ブロック内の命令は順に実行されるため，PC 列を基本ブロック列に変換すれば，情報量を減らすことなく，データ量を削減することができる．図 2.1 の 11 種の命令は，図 2.2 では 7 種の基本ブロックで表されている．

基本ブロック・ベクトル SimPoint では，インターバルの特徴量として**基本ブロック・ベクトル** (Basic Block Vector: **BB ベクトル**) を用いる．BB ベクトルはインターバルごとに定義され，その要素はそのインターバルにおける各基本ブロックの出現回数である．図 2.2 の例では，Pa では，BB0 から BB4 がそれぞれ 1 回ずつ出現しているため，その BB ベクトルは $(1, 1, 1, 1, 1, 0, \ldots, 0)$ となる．また，Pb では，BB3 と BB4 がそれぞれ 4 回ずつ出現しているため，その BB ベクトルは，$(0, 0, 0, 4, 4, 0, \ldots, 0)$ となる．

クラスタリング インターバルは，BB ベクトルに基づいてクラスタリングされる．同一のクラスタに分類されたインターバルは同じフェーズであるとみなされる．したがって，各クラスタから重心に最も近いインターバルを取り出し，これをシミュレーション・ポイントの 1 つとして選出する．

シミュレーションと性能指標の推定 シミュレーションは，こうして選出されたシミュレーション・ポイントのみに対して実行する．

得られたシミュレーション・ポイントの性能指標を，各クラスタに属するインターバルの数で重みを付けて平均し，プログラム全体の性能指標を推定する．

2.1.3 基本ブロック・ベクトルの k-means 法によるクラスタリング

BB ベクトルの次元数はターゲット・プログラムの実行中に現れた基本ブロックの種類の数に等しい．第 5 章で扱う SPEC CPU 2006 の gcc では，最初の 100G 命令に対して，基本ブロックは 44,224 種類も出現する．

一方，1 つのインターバル中には，基本ブロックは高々数十〜数千種程度の基本ブロックしか出現しない．したがって，それ以外の要素はすべて 0 となる．

その結果 BB ベクトルは，高次元の疎なベクトルとなる．

SimPoint は，クラスタリングに **k-means 法**を用いることをクレームの一つとしている．これは，k-means 法は，BB ベクトルなどの疎なベクトルをクラスタリングに適してからである．

2.2 その他の手法

ここでは，Software Phase Marker [18]，SimFlex [4–8] などの関連する手法を紹介する．

2.2.1 Software Phase Marker

ソフトウェアのレイヤーからのアプローチとしてフェーズの切れ目を探す Software Phase Marker （以下，SPM）を紹介する．SPM はハードウェアと独立に，プログラムへの様々な入力によるフェーズを検出するために使用される．

Call-Loop グラフとマーカ　SPM では一般的なプログラムの関数呼び出しの構造を表す Call グラフにループの実行ノードを追記した階層的な Call-Loop グラフを用いる．プログラムの実行時のグラフのエッジでの実行命令数のばらつき（最大，平均，および，標準偏差）を追跡して，ばらつきの少ないものエッジにマーカをつける．すなわち，マーカのついているエッジは毎回似たような命令が実行されているエッジということになる．

区間の切り分けと推定 マーカのついているエッジを毎回通る際に実行される命令列は類似している可能性が高い．したがって同じフェーズである可能性が高いため，1 つの区間として扱うべきである．そこで SPM では区間の切り分け方として：

- マーカの命令列
- 2 つのマーカに挟まれた命令列

という 2 つの部分に分け，それぞれに 1 つの区間を割り当てる．これにより，可変長の区間が得られる．得られた区間は SimPoint と同様の方法でクラスタリングし，シミュレーション・ポイントを選ぶ．

SPM の可変長区間の切り分け方に従えば，過不足なくフェーズの切れ目の候補ごとに区間に切ることが可能であり，SimPoint の問題に対処できるとしている．

一方で，ベンチマーク・プログラムの評価結果は，SimPoint よりも悪い結果となっているが非常に細かな区間が多く発生してしまうためである．SPM は SimPoint より精度の向上やシミュレーション時間の短縮することができないが，シミュレーション・ポイントをプログラムにマッピングすることによって，シミュレーション・ポイント再利用可能である．

2.2.2 SimFlex

ランダム・サンプリング SimFlex [4–7] ではランダム・サンプリングした区間をシミュレーション・ポイントとする．許容できる誤差から信頼区間を設定し，サンプリング・サイズが決定される．

SimPoint はランダム・サンプリングよりも少ないシミュレーション・ポイントで正確に IPC 推定が可能とされている [15] が，統計的な信頼を確保できるのがランダム・サンプリングのメリットだとしている．

Live-Points キャッシュや分岐予測器が初期状態のままシミュレーション・ポイントだけをシミュレーションすると IPC の誤差が生じる可能性がある．これは，全長実行の途中でその区間をシミュレーションされる場合とキャッシュや分岐予測器の状態が異なっているためである．そこで一般にはシミュレーション・ポイントをシミュレーションする際にはキャッシュや分岐予測器のシミュレーションを一定命

令数分 前から行い，全長実行のときと状態を揃えることで誤差を縮められる．

この，キャッシュや分岐予測器の予備シミュレーション時間を短縮するために，SimFlex ではチェックポイントを設けではキャッシュや分岐予測器の状態をあらかじめストレージに保存しておく．これにより予備シミュレーションの時間を短縮することができる．

2.3 既存手法の問題点

ランダム・サンプリングを行う SimFlex は除いて，SimPoint や Software Phase Marker などは，PC や call グラフなど，プログラムの静的な性質に基づいてシミュレーション・ポイントを選出する．すなわち，これらの手法は,「プログラムの同じような静的区間を実行している動的区間は同じフェーズである」という仮定に基づくものである．

しかし，第 1 章でも述べたように，この仮定には反例がある．たとえば，同じコードであっても，入力が異なると，処理するデータ量に応じてキャッシュ・ヒット率が変化し，IPC に大きな影響を及ぼす．実際，5.3 節で示されるように，メモリ・インテンシブないくつかのプログラムにおいて，SimPoint の精度が大きく低下している．

キャッシュ・ヒット率は，キャッシュ・サイズの影響を大きく受ける．キャッシュ・サイズは，ターゲット・プロセッサのマイクロアーキテクチャのパラメータである．したがって，この問題を克服するためには，マイクロアーキテクチャに関する情報を考慮する必要がある．

2.4 関連研究

ここでは，マルチスレッド CPU [19] と GPGPU プロセッサ [10]，フル・システム [11] などに関連するシミュレーション手法を紹介する．

2.4.1 マルチスレッド・プロセッサシミュレーション

プログラムを逐次的処理するシングルスレッドと同様に，マルチスレッドもシミュレーションによる性能評価が欠かせない．ところが，マルチスレッドは同期並列実行を行うため，シングルスレッドのシミュレータをそのまま適用するには問題がある．例えば，マルチスレッドにとって，IPC と実行命令の数で，実行時間を表すには不十分である．

シングルとマルチスレッドの違い マルチスレッドはプログラムの同期並列処理が可能であるが，マルチスレッドの間にメモリ共有および，同期イベントを通じて相互作用し，隣接するスレッドのタイミングに影響を与える．そのため，単にシングルスレッド倍数の性能向上だと考えるわけにはいかない．例えば，スレッド同期により，あるスレッドが休眠状態またはスピン状態になったとしても，他のスレッドは休眠時間だとカウントされるまま，実際には実行を続けている．

[19] では，スレッドごとの性能を追跡しながら, 高速転送 (fast-forwarding) によって accurate なアプリケーション同期実行を行う．その結果に基づいて，アプリケーションフェーズ検出を行い，シミュレーション・ポイントを選択する手法を提案した．

性能 (IPC) 推定シミュレーション・ポイント 高速転送で性能とタイミングを完全に分離するのは難しいが，同期イベント・タイミングを維持することができる．すなわち，スレッドごとに異なるコードの実行と異なる実行時間 (run-time) を accurate に再現する．例えば，マルチスレッドの異なるコード実行，または個別のメモリアクセスパターンによって，スレッドが異なるキャッシュミス率を持つ．すなわち，データ依存によって性能に大きな影響を及ぼす．

高速転送中は，現在の命令カウントとともに非休眠実行の IPC を使って，各スレッドの経過時間を追跡する．最も重要なのは，同期イベントは通常通りシミュレートされる．例えば，スレッドがスリープ状態になると，そのスレッドは停止され；スレッドが起動すると，現在の時間が提供され，シミュレーションが継続する．また，全長なベンチマーク・プログラムについてシミュレーションを行うことによって非休眠期間での IPC を計算することが可能になる．これは，スレッド間の同期と実行時間に及ぼす影響を最善に保つ方法だと [19] の筆者らは主張する．

実行時間推定シミュレーション・ポイント シングルスレッド・アプリケーションでは，IPC の正および負のエラーが平均化されることによって実行時間を推測できた．ところが，実質的な同期化を伴うマルチスレッド・アプリケーションの場合，早いスレッドも遅いスレッドもあるし，非クリティカル・スレッドもある．そこで，アプリケーションの周期性を考慮して，サンプリングパラメータをアプリケーションごとに決定する．すなわち，異なる繰り返し命令の数を比較することで，アプリケーションの不規則性のレベルがわかる．これは SPM [18] の原理と似ている．

結論として，高速転送を使ってスレッドごとの IPC とスレッド間の同期イベントを再現することで，推定精度が大幅に向上すると筆者らは主張する．

2.4.2 GPU シミュレーション

カーネル最適化を求めている GPGPU のアーキテクチャ研究でもシミュレータを使用してカーネル動作を確認する．ところが，GPGPU アーキテクチャには多くのコアがシミュレートされるため，Cycle-Accurate なシミュレータには時間がかかる．

GPU と CPU の違い GPGPU カーネルと CPU マルチスレッドアプリケーションは同じく，スレッドという概念が使われている．CPU ではそのコア数とほぼ同数のスレッドが動作するのに対し，GPU ではコアに対し数千〜数万のスレッドが動作する．スレッドブロックサイズは，スレッドブロック内のスレッド命令の数として定義される．また，カーネル起動は通常，複数のスレッドブロックを持ち，異なるスレッドブロックは異なる数の命令を持つこともある．例えば，カーネル 1 がスレッド命令 100 と 100 の数をそれぞれ 2 つのスレッド・ブロックを持ち，カーネル 2 がスレッド命令 160 と 40 の数をそれぞれ 2 つのスレッド・ブロックをもつと仮定する．両方のカーネル起動が同じサイズ（200 スレッド命令）を持つ場合だとしても，異なるスレッドブロック・インターリーブのため，異なる実行を行うことになる．すなわち，GPU の性能は違う結果をもたらす．

また，streaming multiprocessor (SM) 上のウォープ (warp) 数や SM 数などによって，GPU の性能に大きな影響を及ぼす．たとえ，2 つのカーネルが同じ数のスレッド命令を持つとしても，制御・フローの違いによって，1 つのウォープ命令で 32 ス

レッド命令を実行する時も，32 ウォープ命令で 32 スレッド命令を実行することもあり，それらは異なる IPC を有する可能性がある．さらに，32 スレッド命令を含むウォープ命令は，いずれのアクセスも合体できない場合に，少なくとも 1 つおよび最大 32 のメモリ要求を発行することができる．

2.4.3 スレッドブロック・ポイント

CPU シミュレーションの時間を短縮するために，サンプリング技術が [4,14–19] 広く使用されているが，GPGPU シミュレーションでは研究が少ない．GPGPU カーネルは CPU マルチスレッドアプリケーションに似ているように見えるが，マルチスレッドアプリケーション用のプロファイリングベースのサンプリングを適用するには問題がある．例えば，定期的にサンプリングを行うシステマティック・サンプリング [19] は，特に，規則的 (regular) 実行パターンを持つカーネルでは，必要以上に大きなサンプリング・サイズにつながる可能性がある．これは，アプリケーションの周期性によってシミュレーション・ポイントを選出するからである．さらに，プロファイリングベースのサンプリング [14] は，シミュレートされたプラットフォームと同様のプロファイリングプラットフォームの構成を必要とし，SM 上のウォープ数や SM 数など，シミュレートされた構成が変更されると，プロファイリングをやり直す必要が生じる．

[10] では，GPGPU カーネルのプロファイリング・ベースのサンプリング手法に基づくスレッドブロック・ポイント (TB Point) を提案した．

GPGPU アプリケーションは，複数のカーネルを持つ． そこで，インター起動・サンプリング (inter-launch) とイントラ起動・サンプリング (intra-launch) を使用して，カーネル単位で GPGPU シミュレーション時間を短縮する．すなわち，カーネル起動回数を減らすために，前者はシミュレーションが必要なカーネル起動を選択し，異なるカーネル起動に同質の振る舞いがある場合は，それらのうちの 1 つだけをシミュレートする．

インター起動・サンプリング インター起動・サンプリングでは，プロファイリングを通じて，似ている性能を持つスレッドブロックを選出する．これらのスレッドブロックを見つけるために， ウォープ・スケジューリング効果を考慮した数学モ

デルを開発した.

異なる数のウォープおよびSMなどのハードウェア構成に対して，スレッドブロック・ポイントは，オーバーヘッドを減らすため，プロファイリング結果を再利用しながらクラスタリングを再実行する． また，マルコフ連鎖モデルを使って，ウォープ・スケジューリング効果をモデル化する.

インター起動・サンプリングでは，カーネル起動・サイズ，制御・フロー，メモリ・サイズ，スレッドブロック・サイズの特徴ベクトルを用いて階層的クラスタリングを行う．各特徴ベクトルはカーネル起動を表すので，同じクラスタ内で分類されたカーネルは同質の性能（IPC）を持つとする．そして， 各クラスタについて，クラスタの中心に最も近い特徴間ベクトルを持つカーネルがシミュレーション・ポイントとして選択される.

イントラ起動・サンプリング イントラ起動・サンプリングでは，リソースの競合および/または待ち行列の遅延による可変メモリ待ち時間によって引き起こされるIPC変動を定量化する．ここでは，エポック (epoch) という概念を導入して，各エポックのクラスタIDを生成する．エポックのサイズはシステム占有量に等しい．同じストール確率 (stall probability) と平均ストールサイクル (average stall cycles) を持つスレッドブロックはフェーズ分類を行い．それらの各クラスタについて，クラスタの中心に最も近い特徴ベクトルを持つカーネルがシミュレーション・ポイントとして選択される．また，異常値スレッドブロック (outlier thread blocks) は，後処理を行い，元分類されていたクラスから除去され，それ自身のクラスターに割り当てる．そうすることによって，他よりもかなり多くの命令を持つ異常値スレッドブロックで引き起すスレッドレベルの並列性（TLP）の変化を検出する.

結果として，スレッドブロック・ポイントは，SMやウォープ数が異なるなど，システムの占有率が変化するハードウェア構成に対応できたと筆者らは主張する.

2.4.4 フルシステムシミュレーション

フルシステム・シミュレータは，完全なアプリケーションをシミュレートし，オペレーティング・システムの動作を確認するためのツールである．ハイパーバイザ，同時実行するアプリケーション間の干渉などを理解するのに役立つ．ところが，調

査中のシステムとテスト中のアプリケーションはますます複雑になり，長いシミュレーション時間がかかる．これは，システム設計パラメータとワークロードを変えながら，システムの確実性とシステム全体の影響を確認する必要があるからである．

[11] では，プロセッサ設計空間を探索し，フルシステム・シミュレーション環境でアプリケーションを特徴付ける手法を提案した．

Android アプリケーションを分析する際の大きな問題は，手動によるユーザー入力なしで再現可能な結果を生成することである．ここで，[11] らは，AutoGUI というユーザー・インターフェイスの自動化ツールを提案した．また，SimPoint，Principal Component Analysis (PCA)，一部実施要因実験・デザイン (Fractional Factorial experimental designs) を組み合わせ，ワークロードの特性と解析に必要なシミュレーション時間を大幅に削減した．

AutoGUI Android アプリケーションを分析する際の大きな問題は，ユーザー入力なしで手動による再現可能な結果を生成することである．たとえば，一般的に，ゲームのロード画面はシステムの速さによってより早めに現れる．ところが，タイム同期化された入力でリプレイすると，ロード画面後，ゲームとの相互作用が，遅いマシンのほうが早めにリプレイされる．この問題に対処するために，AutoGUI では，Virtual Network Computing（VNC）を介して Remote Framebuffer Protocol 上で動作するため，ソフトウェアプラットフォームに依存しない．また，主にキャプチャとリプレイの 2 つの部分から構成され，キャプチャの特別なイベントで時間遅延の任意の挿入をサポートする．それは，実際のアプリケーションで一般的な実行間変動を分析するためである．

SimPoint オペレーティング・システムや 対話型アプリケーションを使用するより複雑なフルシステム環境で，SimPoint の検証を行う．通常，SimPoint は少ないシミュレーション・ポイント数でより正確な全長の特性を推定するため使われているが，ワークロードのフェーズ検出にも使われる．すなわち，選出されたシミュレーション・ポイントが一つのワークロード内ではなく，いくつかのワークロード全体でどのように似ているかを検証した．

PCA PCA [20] は，主な性能指標を抽出するために巨大なデータセットを使用する数学的手順である．SimPoint を利用して各スマートフォンワークロードについて選出したシミュレーション・ポイントに PCA を適用した．

一部実施要因実験・デザイン マイクロアーキテクチャーで一般的に，3 つ以上のパラメーター間の高次効果は実際には小さい．そこで，level- 2 キャッシュ・サイズと主記憶へのレイテンシ，発行キュー・サイズにかなり異なる性能を持つ BBench と Caffeinemark のベンチマーク・プログラムを選んで，分岐予測器の性能を左右するサイズパラメータを加えて一部実施要因実験・デザインを行う．

結果として，AutoGUI ツールで詳細なシミュレータにユーザの入力をリプレイし，Android で対話型ワークロードを実行した実際のマルチスレッドアプリケーションの SimPoint による選出されたシミュレーション・ポイントで性能評価が行う場合，うまく行ったと筆者らは主張する．

2.4.5 関連研究のまとめ

マルチスレッド CPU [19] や GPGPU プロセッサ [10] または，フル・システム [11] でシミュレーション・ポイントを選出することで，シミュレーション時間を短縮する様々な研究が行っていた．

マルチスレッド CPU はプログラムの同期並列処理の長所があり，スレッドの間にメモリ共有および，同期イベントによる相互作用に着目して，その特徴を表すシミュレーション・ポイントを選出するのに主眼が置かれていた．

GPGPU プロセッサはマルチスレッド CPU と似ているが，CPU では，そのコア数とほぼ同数のスレッドが動作する．一方，GPU では，コアに対し数千〜数万のスレッドが並列に動作することができる．そこで，GPU に対し，スレッドブロック単位でフェーズを分類を行い，シミュレーション・ポイントを選出していた．

フルシステム・シミュレータはオペレーティング・システムの動作を確認するため使われていた．例えば，ハイパーバイザ，スマートフォンなどの性能を確認するのに使われていた．ここでは，ユーザーインターフェイスの自動化ツールで対話型アプリケーションの結果を再現していた．

これらのプロセッサでもプロファイリングを通じて，似ている性能を持つフェーズを検出するが，プログラムの静的な性質に基づいて行う問題点はSimPointと同じである．

そこで次章からは，特徴的なマイクロアーキテクチャを持ついくつかのモデルによってプログラムのシミュレーションを行い，その結果から汎用的なシミュレーション・ポイントを選出する手法を提案する．

第3章

帰納的な選出手法

SimPoint は，プログラム・カウンタを基に演繹的にシミュレーション・ポイントを選出する手法であった．コードの同じ区間を実行しているなら同じフェーズとみなす．すなわち，マイクロアーキテクチャの情報は無視して，命令セット・アーキテクチャの情報のみを用いる．

それに対し提案手法は，特徴的なマイクロアーキテクチャを持つ複数のモデルでプログラムを事前シミュレーションし，得られた IPC を基に帰納的にシミュレーション・ポイントを選出する [21,22]．すなわち，特徴的なマイクロアーキテクチャを持つモデルがすべて同じ IPC を持つインターバルが同じフェーズとみなす．ここで，特徴的なマイクロアーキテクチャというのは，互いに十分異なり，特徴的な IPC の振る舞いを示すものが望ましい．

本章ではまず，3.1 節において，SimPoint と提案手法わかりやすくするためのサーキットの喩えを例として説明する．そして，サーキットの喩えから同じ類推で，提案手法の原理を導出する．また，提案手法の工程，クラスタリングアルゴリズムまで詳しく述べ，SimPoint より優れている部分を原理的に説明する．特に，3.2 節において，提案手法の特徴量として使われている IPC ベクトルについて SimPoint の Basic Block Vector と比較しながら原理から作り方，利用先まで一つ一つ確かめていこうとする．

3.1　帰納的手法への導入

3.1.1　マイクロアーキテクチャの IPC 遷移の特性

図 3.1 は，異なるマイクロアーキテクチャを持つ 4 種のモデルで，SPEC CPU 2006 [2] の perlbench を実行した際の IPC の遷移を示している．インターバルは，細かすぎるとグラフが塗りつぶされてしまうため，やや粗く，10M 命令とした（第 5 章の評価では，SimPoint は 1M 命令，提案手法は 10K 命令程度が最適）．同図は，*ref* 入力に対する実行の最初の 12G 命令を示しているが，以降 100G 命令まで，10G〜12G 命令と同様の振る舞いが続く．

モデルは，3.4 節で詳しく述べる以下の 4 種である：

sim　一般的な 6-issue out-of-order スーパスカラ・プロセッサ．

sim-cache　キャッシュのヒット率を 100%にした sim．

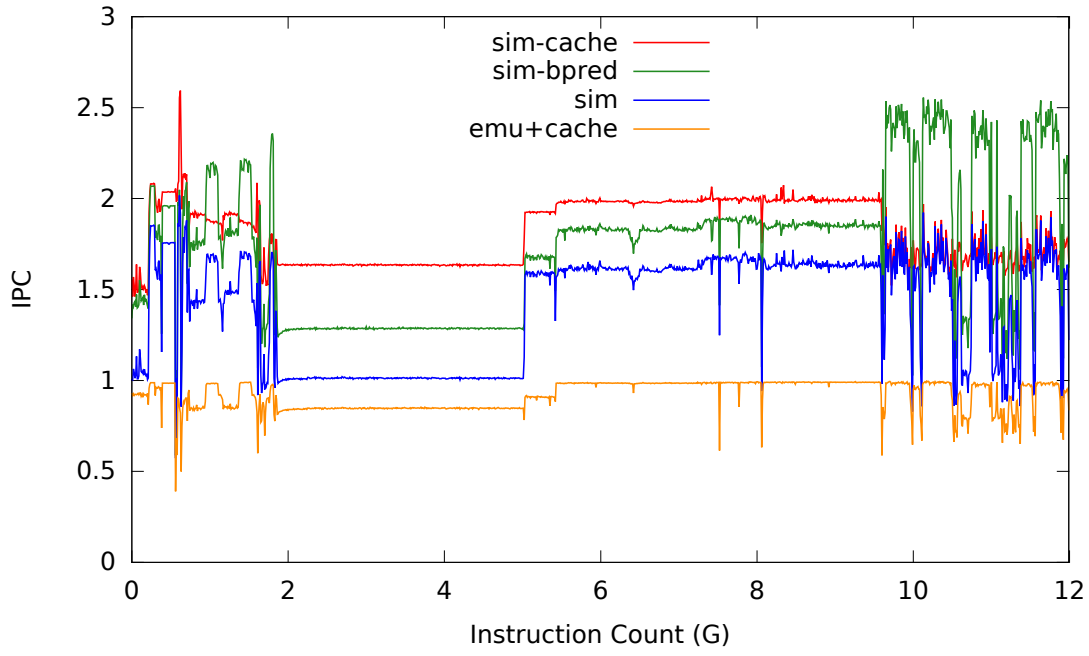


図 3.1: 400.perlbench の *ref* 入力の全長実行

sim-bpred 分岐予測のヒット率を 100%にした sim.

emu+cache キャッシュはあるが分岐予測は行わないスカラ・プロセッサ.

IPC は，基本的には異なるモデル間で互いに並行的に変化するものの，モデルによって高いところ，低いところがある．このことを利用して，未知のマイクロアーキテクチャについても IPC 推定を行うことができると期待できる．

3.1.2 サーキットの喩え

提案手法の理解に役立つよう，サーキットの喩えを例として説明する．ある長大なサーキットを走行する移動手段 X のタイムを，サーキットの全長を走行することなく求めるにはどうしたらよいかを考える．手法は以下の通りである．

まず，あらかじめ，色々なタイプの移動手段を用意してサーキットの全長を走行させ，タイムを測定する．サーキットはスタートからゴールまでを S1 から S7 の区間に区切ってあり，それぞれの区間での速度が図 3.2 のように計測されたとする．例えば，レーシングカーは直線が二度続いた場合と，下りの場合に速度が上がり，

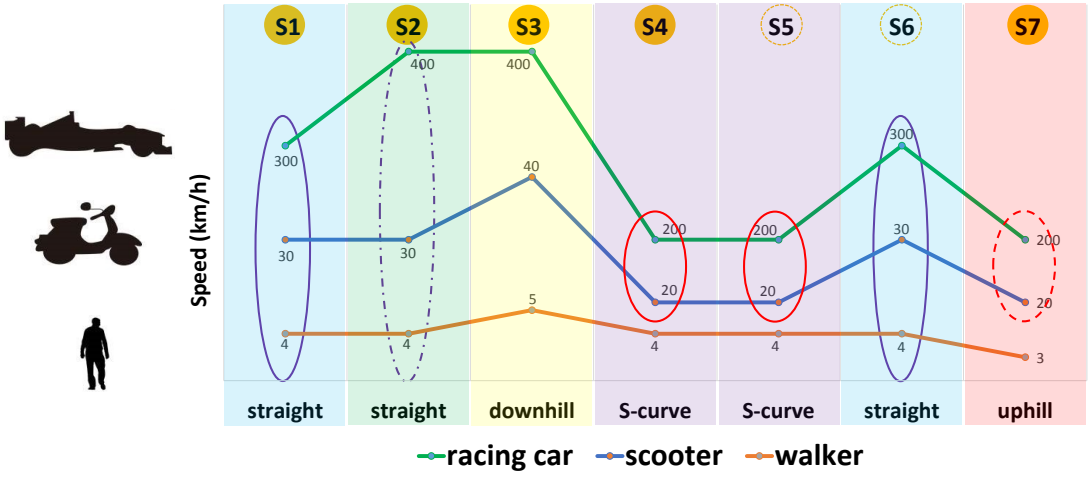


図 3.2: サーキットを様々な移動手段で走行した際の区間ごとの速度

S 字と上りの場合には速度が下がる．他の移動手段でも，その特徴に応じて速度が上下している．

なお，速度の変化の原因を示唆するために，区間を直線，下り，上り，S 字と記載してあるが，手法はそれぞれの区間が本当にどのようなものであるかを判断する訳ではない．手法に必要なのは各区間の速度だけである．

S1 と S6 の両方で，3 つの移動手段すべての速度は $(300, 30, 4)$ である．したがって，S1 と S6 は似たような区間であると考えることができる．S4 と S5 も同様である．したがって，S5 と S6 では，実際に走行する必要はなく，S1 と S4 の速度でそれぞれ代用することができるであろう．

今，未知の移動手段 X について，サーキットのタイムを推定するには，図 3.2 でオレンジ色の丸で印をつけた区間のみを実際に走行すればよい．この印の区間はタイムを求めるのに走行すべき最低限の区間であり，ベンチマークで言うところの，シミュレーション・ポイントの役割を担っている．

この方法では，事前に用意する移動手段が，互いに十分異なる性質をもったものであることが重要である．例えば，図の例で，もしスクータとレーシングカーの結果のみから，「走行すべき最低限の区間」を求めた場合，本来異なる性質を持つ区間である S7 と S4，S5 が同じ性質の区間に分類されてしまい，移動手段 X についての推定の精度が下がってしまう可能性がある．

この喩えと同様のことがベンチマークを用いたプロセッサの性能評価についてもいえる．複数の特徴的なマイクロアーキテクチャを持つプロセッサでプログラムを全長実行した結果があれば，区間ごとの IPC を比較することで「シミュレーションすべき最低限の区間」，すなわち，シミュレーション・ポイントを得ることができるであろう．

3.1.3　SimPoint との相違点

SimPoint の PC に着目するシミュレーション・ポイント選出手法は，前節のサーキットの喩えを用いると，区間の静的な性質から演繹する方法であると言える．すなわち，図の直線，上り/下り，S 字といった，区間の静的な性質の類似性から，走行すべき最低限の区間を知る方法である．

しかしこの方法では，レーシングカーが異なるタイムを示しているはずの，S1

と S2 が同じ性質の道として分類されてしまい，X についての推定の精度が下がってしまう可能性がある．これは区間の静的な性質が同じでも，レーシングカーが直線が続くと速くなるように，その区間に移動手段が差し掛かったときの状態によってタイムが変化してしまうためである．

提案手法は，実際の走行結果からサーキットのフェーズを知るため，このような区間に差し掛かった時の移動手段の状態が自動的に考慮されることになる．

3.2 帰納的手法

帰納的手法は，特徴的なマイクロアーキテクチャを持つモデルがすべて同じ IPC を持つインターバルが同じフェーズであるという仮定に基づいて，特徴的なマイクロアーキテクチャを持つ複数のモデルを事前にシミュレートしておくことによって得られたインターバルの IPC からシミュレーション・ポイントを選出する．

本節では提案手法の工程について説明する．提案手法の流れは以下の通りである：

1. 特徴的なマイクロアーキテクチャをもつ複数のモデルによる事前シミュレーションによって，インターバルごとの IPC ベクトルを得る．
2. インターバルごとの IPC ベクトルによるクラスタリングを行う．

以下ではそれぞれの工程を順に見ていく．

基底モデルと事前シミュレーション　提案手法では，先ず，シミュレーション対象プログラムをいくつかの特徴的なマイクロアーキテクチャを持つモデルでシミュレーションする．これらのモデルを**基底モデル** (basis model) と呼ぶ．3.1.2 項のサーキットの喩えからも分かるように，基底モデルは互いに十分異なり，特徴的な IPC の振る舞いを示すものが望ましい．基底モデルの選定については，3.4 節で詳しく述べる．

IPC ベクトル　基底モデルによる事前シミュレーションによって，インターバルごとの IPC ベクトルの系列を得る．

n 種の基底モデル $m_1, m_2, \ldots, m_n$ で，2 つのインターバルを事前シミュレーションすることで，2 個の n 次元の **IPC ベクトル** $\boldsymbol{v}_1 = (i_1^1, i_2^1, \ldots, i_n^1)$ と $\boldsymbol{v}_2 = (i_1^2, i_2^2, \ldots, i_n^2)$

を得る．ここで i^i_j はインターバル i におけるモデル m_j の IPC を表す．

これらに対して，$\boldsymbol{v}_1 \simeq \boldsymbol{v}_2$，すなわち，$i^1_1 \simeq i^2_1, i^1_2 \simeq i^2_2, \ldots, i^1_n \simeq i^2_n$ が成り立つとする．$m_1, m_2, \ldots, m_n$ のマイクロアーキテクチャが互いに十分異なっていれば，2 つのインターバルは同じフェーズであり，未知のモデル m_{n+1} に対しても $i^1_{n+1} \simeq i^2_{n+1}$ となることが期待できる．そして，シミュレーションによって i^1_{n+1} だけを得ることによって，$i^2_{n+1} (\simeq i^1_{n+1})$ を推定することができる．

IPC ベクトルのクラスタリング　次に，得られた IPC ベクトルに対しクラスタリングを行う．すなわち，距離の近い IPC ベクトルを同じフェーズに，離れているものを別のフェーズに分類する．

今回の評価では，k-means 法ではなく，以下のような単純なアルゴリズムを用いてクラスタリングを行った．ある IPC ベクトル $\boldsymbol{v}$ に対して：

1. 既にある全てのクラスタに対し，それらの重心と $\boldsymbol{v}$ との距離（ユークリッド距離，または，チェビシェフ距離）を比較する．
2. $\boldsymbol{v}$ からの距離が閾値以内であれば，$\boldsymbol{v}$ を最も近いクラスタに追加する．重心を再計算しておく．
3. 閾値以内にクラスタがなければ，新しいクラスタを作成し，そのクラスタの最初のメンバとして $\boldsymbol{v}$ を追加する．

すべてのインターバルに対して，1 から 3 を繰り返す．

このクラスタリング方法では，閾値の大小が，シミュレーション・ポイントの数と誤差との間のトレードオフを決定する．閾値は，ベクトル空間内の各クラスタの最大の「半径」を与える．閾値を小さくすると，より小さいクラスタが数多く生じる．その結果，より多くのシミュレーション・ポイントが選出され，誤差が減少することになる．

IPC ベクトルに性質については，3.3 でまとめる．

シミュレーション・ポイントの選出と評価指標の推定　クラスタリングの後は，SimPoint と同様である．

シミュレーション・ポイントの選出では，それぞれクラスタの重心に最も近いインターバルを取り出し，これをシミュレーション・ポイントの 1 つとする．

IPC，および，その他の性能評価指標の推定では，選出されたシミュレーション・ポイントのみをシミュレートし，各クラスタのインターバルの数で重み付けした平均をもって，プログラム全体に対する評価指標を推定することができる．

3.3 IPC ベクトル

2.1.2 項で説明したように，SimPoint における BB ベクトルは，高次元の疎なベクトルである．それに対して，提案手法における IPC ベクトルは低次元の密なベクトルとなる．表 3.1 に，BB ベクトルと IPC ベクトルの特徴をまとめる．

この違いが，いくつかの点で提案手法に有利に働く．

3.3.1 IPC ベクトルのクラスタリング

SimPoint では，BB ベクトルは高次元の疎なベクトルであるから，k-means 法のような複雑なアルゴリズムを用いざるを得ない．前節で述べた単純なクラスタリング・アルゴリズムは IPC ベクトルが低次元の密なベクトルであるために機能する．

IPC ベクトルの分布　図 3.3 に，図 3.1 と同じ perlbench の *ref* 入力に対する，3 つの基底モデル，sim，sim-cache，sim-bpred に対する IPC ベクトルの分布を示す．二

表 3.1: BB ベクトルと IPC ベクトル

	BB ベクトル	IPC ベクトル
次元	BB の種類	基底モデルの種類
次元数	数百〜数万	4 程度
要素	BB の出現回数	IPC
要素の値	0 以上の整数	0.0〜4.0 程度の浮動小数点数
非 0 要素数	数十〜数千	（すべて）

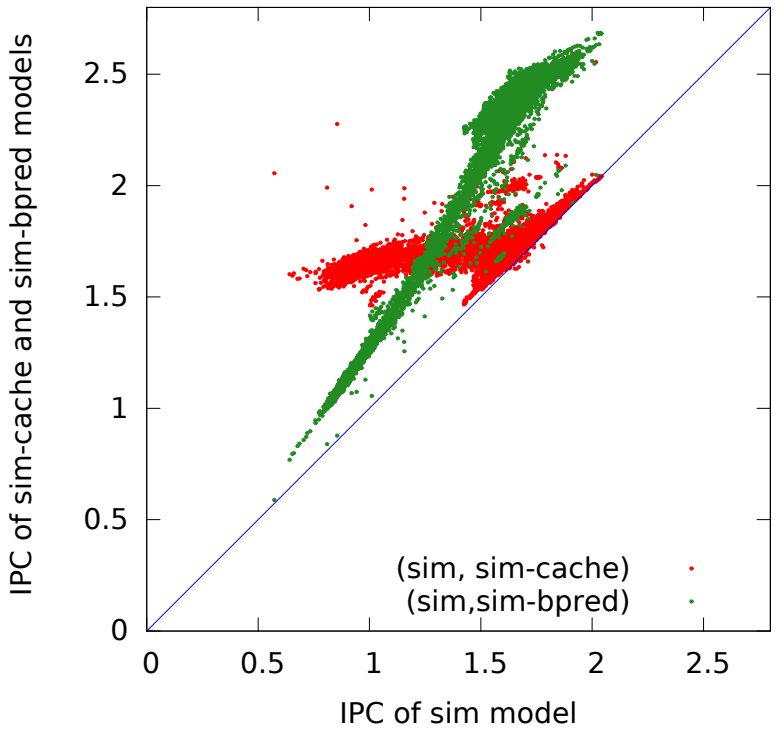


図 3.3: IPC ベクトルの分布

次元のグラフにプロットするため，(sim, sim-cache) と (sim, sim-bpred) の 2 つのベクトルに分けて示している．インターバル長は，10M 命令とした．

IPC ベクトルの分布はそれぞれ直線 $y = cx$，$y = bx$ 上に集中している．c，b は，sim と sim-cache，sim と sim-bpred の基本的な性能比を表す．これは，図 3.1 で見た通り，異なるマイクロアーキテクチャの IPC は基本的には並行的に変化することによる．そして，$y = cx$，$y = bx$ からのずれが，個々のマイクロアーキテクチャの特徴を表している．

クラスタリングするとは言っても，IPC ベクトルは，いわゆる「クラスタ」として複数，離散的に存在する訳ではなく，$y = cx$，$y = bx$ 付近にそれぞれ 1 つの「クラスタ」として存在している．

IPC ベクトルのクラスタリングの要諦は，各次元を十分に細かく，たとえば 0.01 刻みに，分類することにある．したがって，ベクトル空間を 0.01 刻みの超立方体

に等分割するような方法であっても問題なく機能する．前項で述べたアルゴリズムは，無駄な超立方体を管理しなくてもよいように，クラスタを動的に生成しているに過ぎない．

3.3.2 IPC ベクトルのクラスタリングにおける閾値

提案のクラスタリング・アルゴリズムでは，閾値を小さくすると，より小さいクラスタが数多く生じ，より多くのシミュレーション・ポイントが選出され，推定精度が向上することになる．

このように，提案のクラスタリング・アルゴリズムにおいては，推定精度の要請から閾値が決定され，クラスタの数は閾値によって従属的に決定される．

一方，SimPoint で用いられている k-means 法においては，最適なクラスタ数 k を決定することは一般に難しい [23]．

3.3.3 IPC ベクトルとインターバル長

提案手法も，SimPoint 同様に，プログラム実行を区間に切り分ける必要がある．今回は SimPoint 同様に固定長のインターバルでプログラム実行を区切ることとした．

ただし，固定長で区切った場合，図 3.4 のように，フェーズの切れ目を含んだインターバルが生じる．このようなインターバルは，クラスタリングにおいてノイズとして働く [18, 24, 25]．

しかし提案手法では，より短いインターバルを用いることでこの問題に対処することができる．BB ベクトルに比べて IPC ベクトルは，インターバルを細かくしてその数が増えたとしても，効率的にクラスタリングすることができるからである．

一方 BB ベクトルでは，インターバルを細かくすると，インターバル数が増えることによってかえってクラスタリングの性能が低下してしまう．

5 の評価では，インターバル長は，SimPoint が $1M$ 命令程度が最適であったのに対して，提案手法では，それより 2 桁も小さい 10K 命令程度が最適となった．

3.4　基底モデルの選定

3.1.2 項のサーキットの喩えからもわかる通り，基底モデルは互いに十分異なっていることが望ましい．ここで，どのような性質のマイクロアーキテクチャを選べばよいか考察する．

現代のスーパスカラ・プロセッサにおける IPC 変動要因は，以下の 3 つである：

1. キャッシュ・ミス
2. 分岐予測ミス
3. 命令の依存関係

プロセッサの研究・開発では，これらの影響を減らして性能を向上（あるいは維持）させることに主眼が置かれている．

そこで初期の評価（第 5 章）に際しては，基底モデルは以下の 4 種とした：

sim　一般的な 4-way out-of-order スーパスカラ・プロセッサ．表 5.2 にその構成をまとめる．ターゲット・モデルの評価と同じシミュレータにより IPC を求める．

sim-cache　キャッシュのヒット率を 100%にした sim．すなわち，キャッシュ・ミ

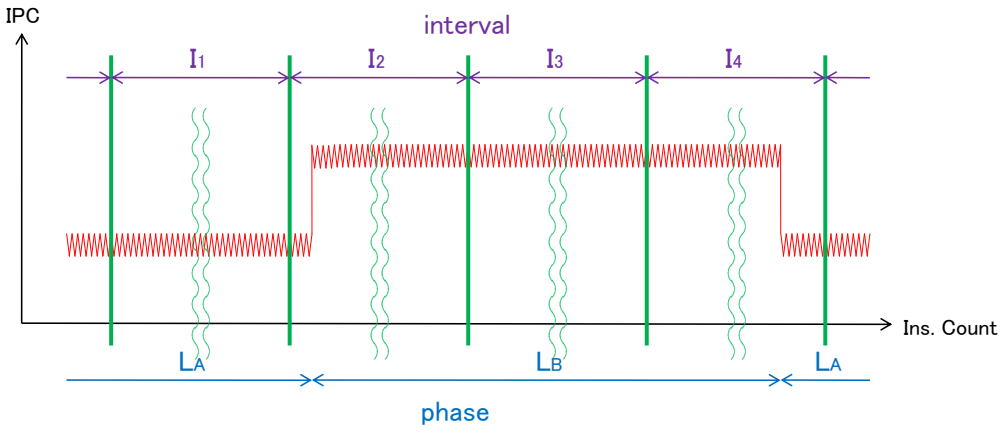


図 3.4: 固定長で区切った場合のフェーズの切れ目

スの影響がない.

sim-bpred 分岐予測のヒット率を 100%にした sim. すなわち，分岐予測ミスの影響がない.

emu+cache いわゆるキャッシュ・シミュレーションによるモデル. 以下で詳しく述べる.

基底モデル emu+cache キャッシュなどの特定の分野の研究においては，キャッシュ・ヒット率のみを知ればよく，プロセッサの総合的な速度 (IPC) を求める必要がない場合がある. そのような場合には，**キャッシュ・シミュレーション**と呼ばれる方法が採られる. これは，プロセッサ本体の動作の再現はシミュレーションではなくエミュレーションで済ませ，キャッシュのみをシミュレーションするものである. この場合，プロセッサ本体のエミュレーションは，キャッシュに対する入力を生成する役割を果たす.

モデル emu+cache は，このキャッシュ・シミュレーションをベースとするものである.

ただし，キャッシュ・ヒット率ではなく IPC を必要とするため，キャッシュ・ミスのペナルティ（サイクル数）の総和を実行命令数に加えて実行サイクル数とする.

このモデルは，キャッシュはあるが分岐予測がないスカラ・プロセッサと等価である. 分岐予測がないとは，命令フェッチから分岐命令の実行までを 1 サイクルで実行するような，非常に浅いパイプラインを想定すればよい.

表 3.2: IPC 変動要因と基底モデル

基底モデル	1) cache miss	2) bpred miss	3) inst. dep.
sim	✓	✓	✓
sim-cache		✓	✓
sim-bpred	✓		✓
emu+cache	✓		

基底モデルと IPC 変動要因　表 3.2 は，これらの基底モデルが 3 つの IPC 変動要因のどれに影響を受けるかを示す．sim 以外は，いずれも IPC 変動要因のうち，いくつかの影響を全く受けない,「理想的な」マイクロアーキテクチャである．これらの「理想的な」モデルは，性能の上限を与え，これら以上に特徴的な IPC の変動はないと考えられる．

また，これら 3 つの「理想的な」マイクロアーキテクチャに対する比較の意味で，一般的なスーパスカラ・プロセッサのモデル sim を追加している．

現在ある，あるいは，将来開発される「現実的な」プロセッサの性能はこれらの「理想的な」マイクロアーキテクチャと一般的なスーパスカラ・プロセッサの「間」にあると考えられる．すなわち，これらのモデルは,「現実的な」モデルを表現するための**基底ベクトル** (basis vector) の役割を果たすと言える．

第4章

帰納的な選出手法の改良

本章では，第 3 章で述べた帰納的選出手法 [21, 22] に対する，3 つの改良について述べる．

4.1 節では，キャッシュ容量によって生じるフェーズに対応するために，超多階層キャッシュを持つ基底モデルを紹介する．この基底モデルによってキャッシュ容量の異なる多くのマイクロアーキテクチャの性能評価も可能になる．超多階層キャッシュはワーキング・セット・サイズは減少し，それが上位の階層のキャッシュに収まった段階で，IPC が階段状に向上するモデルである．

4.2 節では，事前シミュレーションにかかる時間を削減するエミュレーションをベースとする基底モデルについて述べる．基本的な提案手法では，いくつかの基底モデルを事前シミュレーションする必要がある．そこで，キャッシュ・ミス，分岐予測ミス，命令の依存関係対応する理想的なモデルを導入することによって，事前シミュレーションにかかる時間を無くして，基本的な帰納手法の短所が補完される．

最後に 4.3 節では，基本的な帰納手法は，一つのプログラム内において，PC が異なっても同様に振る舞う部分をフェーズとして検出してシミュレーション・ポイントを選出した．逆に，異なるプログラムにもプロセッサが同様に振る舞う区間が存在する．そこで，ベンチマークに含まれるすべてのプログラムの実行を一つのワークロードとみなして，すべてのプログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する集合的な帰納的選出手法について紹介する．

4.1　超多階層キャッシュを持つ基底モデル

キャッシュの容量によってフェーズが生じる可能性がある．このようなフェーズは，3.4 節で述べた他の 4 つの基底モデルでは補足できない可能性がある．

4.1.1　キャッシュ容量によるフェーズ

図 4.1 と 4.2 に，ワーキング・セット・サイズが徐々に小さくなるようなコードの例と，そのようなコードを実行した時の IPC の変化の様子を示す．実行が進むにしたがって，ワーキング・セット・サイズは減少し，それが上位の階層のキャッシュ

```
enum { N = 1024*1024*1024 };
int array[N];

for (i = N-1; i >= 0; i--)
    for (j = 0; j <= i; j++)
        do_something_for(array[j]);
```

図 4.1: ワーキング・セット・サイズが小さくなるコード

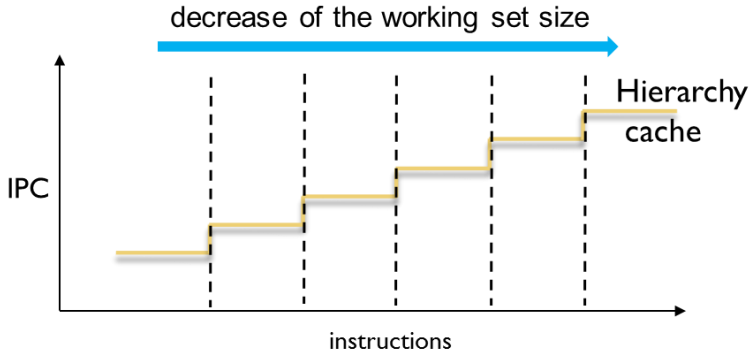


図 4.2: 階層キャッシュの容量によって生じるフェーズ

に収まった段階で，IPC が階段状に向上することになる．

3.4 節で示した 4 種の基底モデルのうち，キャッシュ・パーフェクトのモデルでは，フェーズは生じない．一方，基本構成のモデル sim は，level-1/2 キャッシュと主記憶の，計 3 つの記憶階層に合わせた 3 つのフェーズを生じる．その結果，提案手法では，sim のキャッシュ容量に合わせた 3 つのシミュレーション・ポイントを選出することになる．

ところが，たとえばキャッシュ容量を半分にしたマイクロアーキテクチャでは，IPC の向上が基本構成に比べて遅く生じることになる．すなわち，基本構成のような特定のキャッシュの構成を持つモデルによって選出したシミュレーション・ポイントでは，キャッシュ容量に対して敏感なワークロードに対しては，一般性が十分ではない．

4.1.2 超多階層キャッシュ

このような，キャッシュ容量によって生じるフェーズに対応するために，キャッシュ容量の異なる多くのマイクロアーキテクチャを基底モデルに追加するのは現実的ではない．そこで，超多階層のキャッシュを持つ単一のモデルを基底モデルに加えることを考える．

第 6 章の評価では，基底モデル emu+cache のキャッシュを，

8KB, 16KB, 32KB, . . . , 256KB, 1MB, . . . , 8MB, 16MB

の，12 階層のキャッシュに変更したモデルを用いた．隣接する 2 つの階層間で，容量比は 2 倍，レイテンシは図 4.3 で示す通り，2 サイクルずつ増やした．この場合，図 4.2 の例では，12 箇所のシミュレーション・ポイントが選出されることになる．

この基底モデルを，**emu+hhcache** (highly-hierarchical cache) と呼ぶ．

4.2 エミュレーション・ベースの基底モデル

事前シミュレーションにかかる時間 提案手法では，シミュレーション・ポイントを選出するためにいくつかの基底モデルを事前シミュレーションする．事前シミュレーションは，原則的には，ベンチマークに対して一度だけ実行すればよい．例え

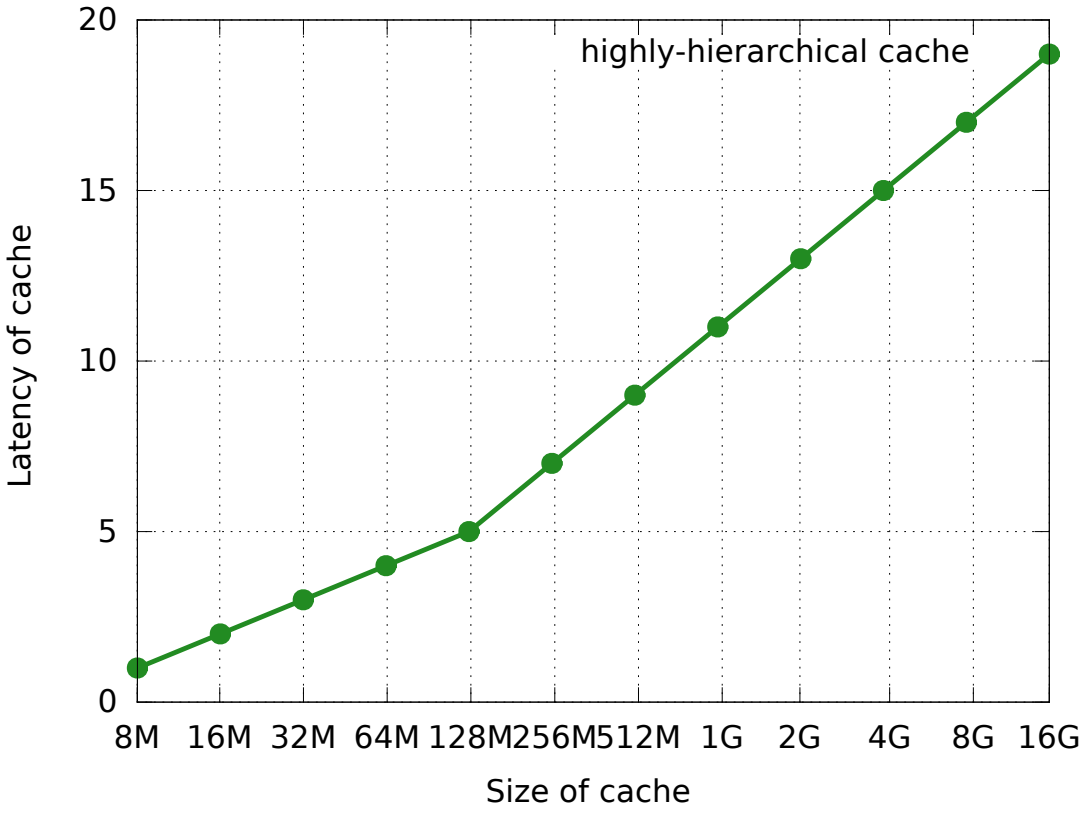


図 4.3: 超多階層キャッシュサイズごとのレィテンシ

ば，SPEC のような著名なベンチマークに対してであれば，世界で誰かが一度だけ実行し，得られたシミュレーション・ポイントを公開すればよい．

それでもなお，第 1 章で述べたように，事前シミュレーションにかかる時間は膨大である．第 5 章，第 6 章の評価では，全長ではなく，各プログラムの 100G 命令に対してのみの評価に止まっている．100G 命令であっても，プログラムあたり月単位の時間がかかっている [1]．

事前シミュレーション時間を削減する基底モデル　そこで，3.4 節で述べた 3 つの IPC 変動要因，すなわち，キャッシュ・ミス，分岐予測ミス，命令の依存関係に着

[1] シミュレーションにかかる時間は，おおよそ実行サイクル数に比例する．したがって，同じ 100G 命令でも，IPC の低いプログラムのシミュレーションにはより長い時間がかかる．

目して，それぞれの要因に対応する理想的なモデルを導入することによって，事前シミュレーションにかかる時間の短縮を試みる．

1.1 節 で述べたように，シミュレーションに時間がかかるのは，特に out-of-order なプロセッサの場合，各命令がサイクルごとにプロセッサのどのユニットにからどのユニットに移動するかを再現する必要があるためである．

4.2.1 分岐予測ミス

第 3 章で述べた基底モデル emu+cache は，キャッシュ・シミュレーションをベースとするモデルであった．すなわち，プロセッサ本体の動作の再現は，シミュレーションではなくエミュレーションで済ませ，キャッシュのみをシミュレーションする．そして，キャッシュ・ミスのペナルティ（サイクル数）の和を，実行命令数に加えて実行サイクル数とする．

この手法では，時間のかかるプロセッサ本体のシミュレーションをエミュレーションに置き換えることができる．

分岐予測に対してもこの手法を応用することができる．すなわち，**分岐予測シミュレーション**を行い，プロセッサ本体の動作の再現はエミュレーションで済ませ，分岐予測器のみをシミュレーションする．そして，分岐予測ミスのペナルティ（サイクル数）の総和を，実行命令数に加えて実行サイクル数とする．

この基底モデルを **emu+bpred** と呼ぶ．

基底モデル emu+cache は，分岐予測のないスカラ・プロセッサと等価であった．対照的に，emu+bpred は，キャッシュ・パーフェクトであるスカラ・プロセッサと等価である．

4.2.2 Instruction-Level Parallelism

命令間の依存に関しては，プロセッサ本体がその影響を受けるため，キャッシュ・ミスと分岐予測ミスの場合のようにエミュレーションに置き換える訳にはいかない．

そこで，資源量無限大の理想的な out-of-order プロセッサをモデル化することによって，Instruction-Level Parallelism (ILP) を測り，それをもってプロセッサのシミュレーションを省略することを考える．キャッシュ・パーフェクト，分岐予測パー

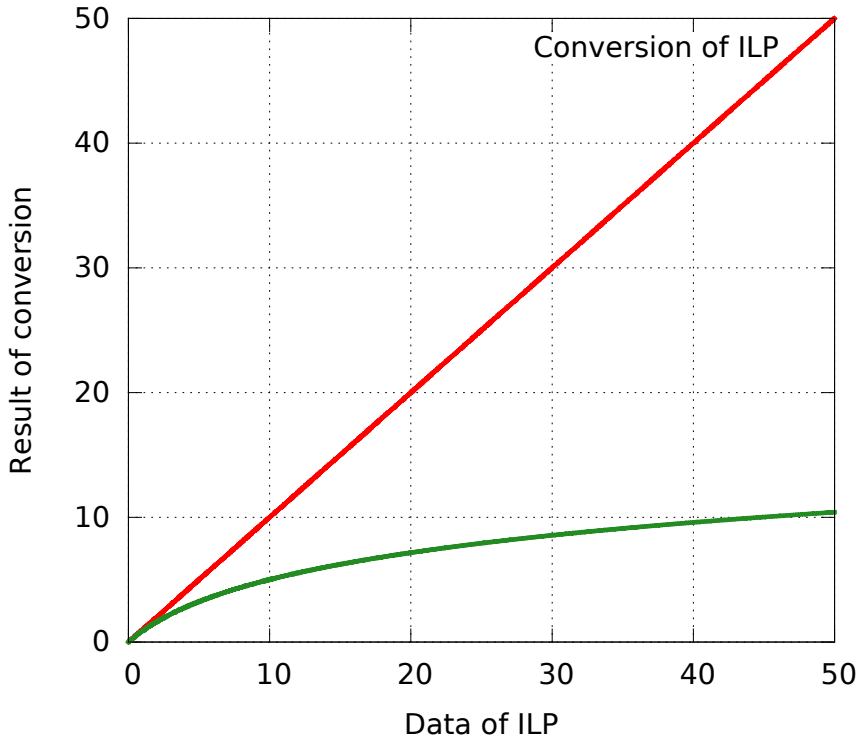


図 4.4: ILP の変換結果

フェクトとすると，このモデルの実行サイクル数は，命令間の依存関係を表すグラフの高さ等しい．このグラフの高さは，以下のようにしてエミュレーション時に容易に求めることができる．

レジスタとメモリの各ワードに，プログラムの最初からその時までのグラフの暫定高さを記録する．そして，命令の実行ごとに，ソース・オペランドのうちで最も暫定高さの高いものを選び，それに命令のレイテンシを加えて，デスティネーション・オペランドの暫定高さを更新する．この場合，命令のレイテンシは，1 に固定してよい．

ただし ILP は，50〜100 といった，IPC と考えるには大きすぎる値になり得る．この値をそのままクラスタリングに用いた場合，1 と 2 の違いと 100 と 101 の違いが同じ重みをもつことになり，都合が悪い．

そこで，関数 $f(x) = a\log(x/a + 1)\,(a > 0)$ によって，大きすぎる値を抑えるこ

表 4.1: IPC 変動要因と基底モデル (2)

基底モデル	1) cache miss	2) bpred miss	3) ILP	4) MLP
sim	✓	✓	✓	✓
sim-cache		✓	✓	
sim-bpred	✓		✓	✓
emu+cache	✓			
emu+cache	✓			
emu+bpred		✓		
emu+ilp			✓	
emu+mlp				✓

とにする．この関数は，$f'(x) = 1/(x/a + 1)$，$f'(0) = 1$ であり，$x = 0$ 付近では線形に近い．$a = 4$ の場合，図 4.4 に示すように，$f(4) \simeq 2.77$，$f(50) \simeq 10.41$ となる．

この基底モデルを，**emu+ilp** と呼ぶ．

4.2.3 Memory-Level Parallelism

Out-of-order プロセッサでは，MLP (Memory-Level Parallelism) を抽出することによって，すなわち，キャッシュ・ミスの処理中に後続の命令を実行することによって，区間によっては in-order プロセッサに比べて何十倍もの高速化を達成することがある．

この効果は，emu+cache と emu+ilp の組み合わせでは表現することができない．

そこで，新たな基底モデル，**emu+mlp** を追加する．このモデルの IPC は，emu+ilp において，ロード命令のレイテンシを emu+cache から得られるキャッシュ（主記憶）のレイテンシに置き換えることで得られる．

4.2.4　4 つの基底モデルの組み合わせ

表 4.1（下）に，以上で述べた 4 つの基底モデルをまとめる．これらを用いれば，事前シミュレーションにおいて時間かかるプロセッサ本体のシミュレーションを完全に省略することができる．

また，これらの基底モデルの IPC は，一回のエミュレーションにおいて同時に求められることに注意されたい．

4.3　集合的な選出手法

本節では，**集合的な** (collective) 帰納的選出方法を提案する．

集合的な帰納的選出手法　異なるプログラムにも，プロセッサが同様に振る舞う区間が存在する．たとえば，配列要素の総和を求めるコードなどは多くのプログラムに存在する．配列のサイズによるキャッシュ・ヒット率の差は生じるであろうが，逆に言えば，配列のサイズが同程度であれば，プロセッサの振る舞いはほぼ完全に同一になる．そうした区間は，プログラムの垣根を越えて同一のフェーズとみなすことができる．

この，プログラムにまたがるフェーズを利用して，図 4.5 に示すように，複数のプログラムから，それらのプログラム全体に対するシミュレーション・ポイントを選出することが考えられる．このことによって，頻出するパタンを重複して選ぶことが避けられ，誤差を維持しながら，全体としてシミュレーション・ポイントが削減されると期待できる．

図 4.5 では，縦に並べている各クラスタは閾値によって分類され，横に並んでいる四角 C_j^i は各クラスタ j の中に含まれている n 種のベンチマーク・プログラムの $A, B, \ldots, Z$ の IPC ベクトル v_i のグループである．様々なクラスタに表示された同じ色同士は，同じベンチマーク・プログラムのベクトルであることを意味する．また，赤い文字で表示された四角は，そのベンチマーク・プログラムのベクトルグループの中に，シミュレーションポイントに選出されるベクトルがあることを示している．

3.2 節で述べたアルゴリズムでは，個々のプログラムの IPC ベクトル列を求め，それに対してクラスタリングを施して，個々のプログラムのシミュレーション・ポイントを選出していた．それに対して集合的な選出方法では，すべてのプログラムの IPC ベクトルを連結して，クラスタリングを施すだけで，すべてのプログラムに対するシミュレーション・ポイントを選出することができる．

集合的な演繹的選出手法 PC をベースとする SimPoint などでは，このような集合的な手法は考えづらい．プログラム間でまったく同一の振る舞いをする部分コードがあったとしても，PC はそれぞれ異なるからである．

このことは，一つのプログラム内であっても同様に成り立つ．PC をベースとする手法では，たとえ一つのプログラム内であっても，同様の振る舞いをする異なる部分コードを同一のフェーズと見なすことができない．

逆に言えば，従来の提案手法は，一つのプログラム内においては，PC は異なるが同様に振る舞う部分からは単一のシミュレーション・ポイントを選出していたことになる．

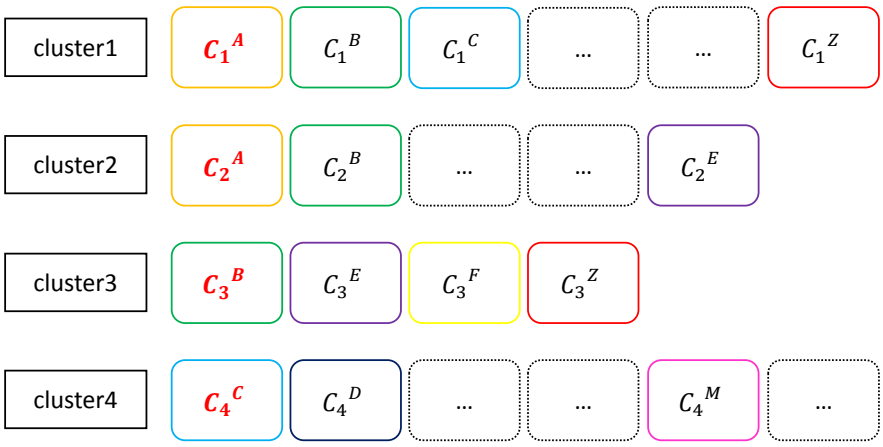


図 4.5: 集合的な選出手法のクラスタの様子

第5章

基本的な手法の性能評価

この章では，第3章で述べた提案手法の評価結果を示す．第4章で述べた改良については，第6章で示す．

以下，5.1節では，評価方法として使われたシミュレータは，我々の研究室で開発した鬼斬弐である．これは，cycle-accurateなプロセッサ・シミュレータである．

帰納手法に用いた基底モデルは，IPC 変動の主な要因であるキャッシュミス，分岐予測ミス，命令の依存関係に基づいて，互いに十分異なっているモデルを選択し，事前シミュレーションを行った．その後，前述した基底モデルを使ってシミュレーション・ポイントの選出を行い，そのシミュレーション・ポイントでターゲットモデルのIPCを推定することによって評価をした．また，ターゲットモデル4つのうち，最初の3 つが，IPC の変動要因のフェーズを提案手法が反映できているか評価するためのアーキテクチャであり，4 つ目は，実際のシミュレーションによるマイクロアーキテクチャ評価に即したものである．

5.2節でSPEC 2006 の29 種類ベンチマーク・プログラムについて提案手法の2つのパラメタであるインターバル長と閾値を導出し，総合的な結果を示す．そして，5.3節では，特徴的ないくつかのベンチマーク・プログラムについて提案手法の優れる性能を詳細な結果で示す．

5.1 評価方法

この評価では，提案手法と SimPoint で，シミュレーション・ポイントを選出し，それを用いて推定された IPC の推定誤差を算出した．

5.1.1 シミュレータ

シミュレータには cycle-accurate なプロセッサ・シミュレータである**鬼斬弐** [1] を用いた．鬼斬弐は，命令の実行ステージにおいて命令を実行し，コミット・ステージにおいてオンライン・エミュレータの実行結果によって検証を行う．したがって，予測ミス後の振る舞いも正確に再現される．このシミュレータは，今回評価のターゲットとしたレジスタ・キャッシュ・システム [9,26] を含む多くのマイクロアーキテクチャの評価でも用いられている [27,28]．

5.1.2 ベンチマーク

ベンチマークには，後述する例外を除いて，表 5.1 に示す SPEC CPU 2006 [2] の全 29 本のプログラムを用いた．入力は，*ref* を用いた．

時間制約のため，各プログラム実行の最初の 100G 命令に対して，シミュレーション・ポイントの選出と，それを用いた IPC の推定を行った．通常，マイクロアーキテクチャに関する研究では高々数百 M 命令程度しかシミュレーションしない [9,26–28] から，全長ではないとは言え，100G 命令は通常の研究より 3 桁近く多い．

5.1.3 基底モデル

基底モデルには，3.4 節で述べた以下の 4 つを用いた：

sim 一般的な 4-way スーパスカラ・プロセッサ．表 5.2 に詳細を示す．以下で述べるターゲット・モデルと同じシミュレータにより IPC を求める．

表 5.1: SPEC CPU 2006 のプログラム

INT	FP
400.perlbench	410.bwaves
401.bzip2	416.gamess
483.xalancbmk	482.sphinx3
471.omnetpp	434.zeusmp
445.gobmk	435.gromacs
464.h264ref	436.cactusADM
458.sjeng	437.leslie3d
462.libquantum	444.namd
473.astar	447.dealII
403.gcc	450.soplex
429.mcf	481.wrf
456.hmmer	465.tonto
	459.GemsFDTD
	470.lbm
	454.calculix
	453.povray
	433.milc

表 5.2: モデル sim の構成

ISA	Alpha w/ byte/word ext.
pipeline stages	Fetch:3, Rename:2, Dispatch:2, Issue:4
fetch width	4 inst.
issue width	Int:2, FP:2, Mem:2
inst. window	Int:32, FP:16, Mem:16
branch pred.	8KB g-share
BTB	2K entries, 4way
RAS	8 entries
L1C	32KB, 4way, 3cycles, 64B/line
L2C	4MB, 8way, 15cycles, 128B/line
main memory	200cycles

sim-cache sim のキャッシュ・ヒット率を 100%としたもの.

sim-bpred sim の分岐予測ヒット率を 100%としたもの.

emu+cache キャッシュあり，分岐予測なしのスカラ・プロセッサ.

5.1.4 ターゲット・モデル

IPC の推定は，以下の 4 つターゲット・モデルに対して行った：

cache-half sim から，L1 および L2 キャッシュのウェイを半分に削減したもの. それぞれの容量も半分となっている.

pht-single sim から，分岐予測器の PHT のカウンタを 2 から 1 ビットに変更したもの.

eight-way Intel Haswell [29]，IBM POWER8 [30] プロセッサ相当の 8-way のスーパスカラ・プロセッサ.

regcache レジスタ・キャッシュを持つスーパスカラ・プロセッサ. 詳しくは後述.

表 5.3 に，ターゲット・モデルの sim に対する相対 IPC をまとめる.

最初の 3 つのモデルは，基底モデルの単純なバリエーションで，提案手法が IPC の IPC 変動要因をどのように反映するかを確かめるためのものである．それに対して regcache は，以下で述べるように，実際のシミュレーションによるマイクロアーキテクチャ評価に即したものである.

表 5.3: 各ターゲット・モデルの sim に対する相対 IPC（SPEC CPU 2006 の 29 プログラムの平均）

ターゲット・モデル	Relative IPC
cache-half	−7.6%
pht-single	−2.5%
eight-way	11.1%
regcache	−9.2%

レジスタ・キャッシュ・システム モデル regcache は，**非レイテンシ指向レジスタ・キャッシュ・システム** [9, 26] を採用している．このレジスタ・キャッシュ・システムは，レジスタ・キャッシュのポート数以上のレジスタ・キャッシュ・ミスが1サイクル内に発生すると，バックエンド・パイプラインが1サイクル，ストールする．

このシステムは，レジスタ・キャッシュ・サイズが十分に大きい（例えば8エントリ）場合，ほとんどIPC低下を起こさない，すなわち，ベースとなる sim とほぼ同じIPCを示す．したがって regcache では，評価のために敢えてレジスタ・キャッシュサイズを2エントリにした．その結果，IPCは sim より9.2%低下している（表5.3）．マイクロアーキテクチャの評価に際しては，推奨とは離れたパラメタでの評価も当然必要になる．

モデル regcache の IPC 変動要因はレジスタ・キャッシュ・ミスであり，これは3.4節で説明した3つのIPC変動要因とは異なる．したがって，このモデルの結果は最も重要である；なぜなら，これら3つ以外のIPC変動要因を持つ一般のターゲット・モデルに対しても，提案手法が機能するかどうかを示すからである．

なお，SPEC CPU 2006 の 453.povray では，およそ 70G 命令を実行したところでシミュレータが異常終了してしまった．これは，2エントリという想定外に少ないレジスタ・キャッシュ・サイズに，稀な条件が重なったためと考えられる．4エントリの場合には，100G 命令を完走する．そのため，regcache に対しては，残りの28本のプログラムでの評価となる．

5.2 総合的な結果

5.2.1 誤差—抽出率グラフ

図5.1と5.2は，誤差—抽出率グラフを表す．サンプリング・シミュレーション手法の基本的な性能は，このような二次元のグラフによって表すことができる．誤差—抽出率グラフにおいて，y軸は，性能指標 (同図ではIPC) の推定の**誤差** (error)，すなわち，ターゲット・プログラムの全命令をシミュレートすることによって得られた**真の** IPC に対する，選出されたシミュレーション・ポイントのみをシミュレートして推定されたIPCの，差の比率 (%) を表す．x軸は，サンプリング・シミュレーションにおける**抽出率** (sampling rate)，すなわち，ターゲット・プログラムの全実

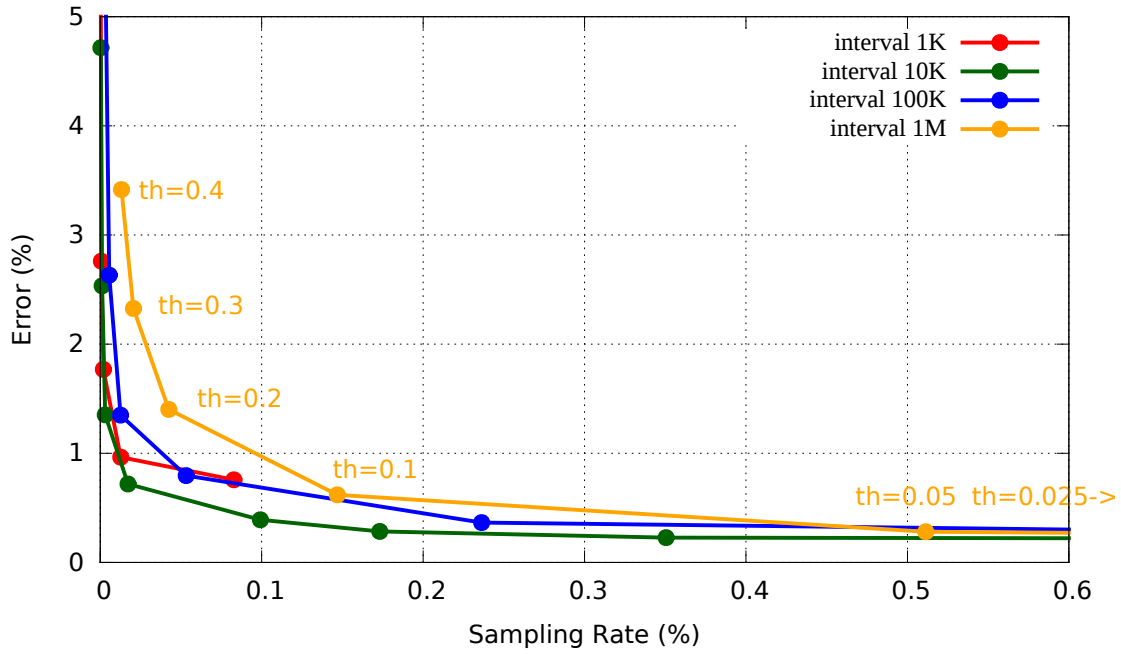


図 5.1: 異なるインターバル長に対する提案手法の誤差—抽出率グラフ

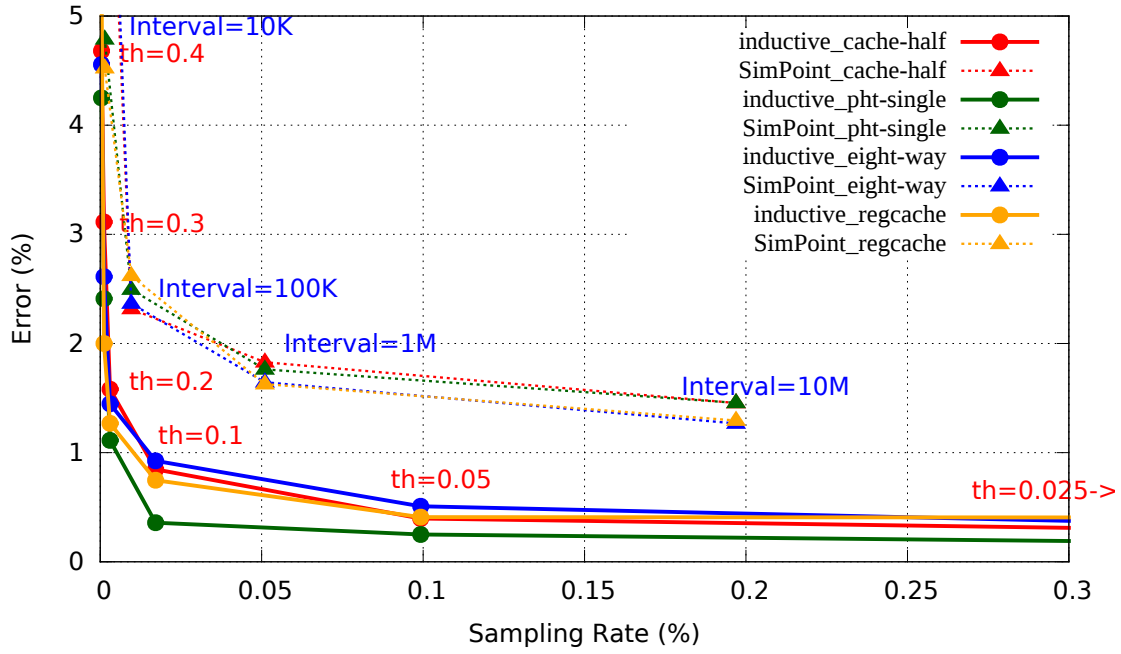


図 5.2: SimPoint と提案手法のターゲット・モデルごとの誤差—抽出率グラフ

行命令に対する，選出されたシミュレーション・ポイントに含まれる命令の比率 (%) を表す．

一般に，誤差と抽出率との間にはトレードオフがあるので，一つの手法に対しては，同図のような反比例的な曲線がプロットされる．この曲線が原点に近いほど，シミュレーション・ポイント選出手法のとしての性能が高い，すなわち，より少ないシミュレーション・ポイントにより，より正確な IPC 推定が行えることを意味する．実際には，要求された推定誤差 —— 例えば，1% 以下 —— を，より低い抽出率で実現することが期待される．

5.2.2　インターバル長

SimPoint は，単一のパラメタ，すなわちインターバル長を持つ．一方提案手法は，2 つのパラメタ，インターバル長とクラスタリング閾値を持つ．そこで最初に，提案手法に最適なインターバル長を探索することにする．

図 5.1 は，インターバル長を変化させた時の提案手法の誤差—抽出率グラフを示す．このグラフには，1K，10K，100K，1M 命令の 4 種のインターバル長に対して 4 本の曲線がプロットされている．各曲線は，0.025，0.05，0.1，0.2，0.3，0.4 の 6 つの閾値に対してプロットされている．

あるインターバル長と閾値の対に対して，4 種のターゲット・モデルと 29 種のベンチマーク・プログラムに対応する $4 \times 29 = 116$ の誤差—抽出率対が得られる．曲線上の各点は，これら 116 の誤差—抽出率対の平均値である．

グラフより，インターバル長 10K 命令の曲線が最も原点に近いところにあり，どの閾値に対しても 10K 命令が最適であることが分かる．次項で述べるように，SimPoint の適切なインターバル長は 1M 以上の命令である．提案手法のインターバル長は，その 1/100 程度と，非常に短い．

したがって以後，提案手法のインターバル長は，特に指定のない限り，10K 命令に固定する．

5.2.3　ターゲット・モデルごとの比較

次に，4 つのターゲット・モデルごとに，SimPoint と提案手法との比較を行う．

図 5.2 は，4 つのターゲット・モデルごとの誤差—抽出率グラフを表す．同図中，SimPoint は破線で，提案手法は実線で表す．1 つの閾値，1 つのターゲット・モデルに対して，29 本のプログラムに対応する 29 の誤差—抽出率対が得られる．曲線上の各点は，この 29 の誤差—抽出率を平均したものである．なお，抽出率は異なるターゲット・モデルでも変わらないので，同じ閾値の 4 つのターゲット・モデルに対応する 4 つのマーカーが垂直に並ぶことになる．

SimPoint の曲線は，インターバル長を 10K，100K，1M，および 10M 命令と変更することによってプロットされている．それに対して提案手法の曲線は，インターバル長を 10K 命令に固定したうえで，クラスタリング閾値を 0.025，0.05，0.1，0.2，0.3，0.4 と変更することによってプロットされている．つまり，前出の図 5.1 のインターバル長 10K 命令の曲線は，この図 5.2 における，4 つのターゲットモデルに対する 4 本の曲線の平均である．

図 5.2 では，4 本の曲線が互いにまとまっており，モデルごとの結果には，提案手法にも SimPoint にも目立ったばらつきは見られない．

そして，提案手法の曲線のグループが SimPoint のグループよりも原点に近く，提案手法の方がよりよいシミュレーション・ポイントを選出できていることが分かる．例えば，シミュレーション・ポイントの割合が約 0.1% の場合，誤差を約 1.6% から 0.4% に改善することができる．これは，約 1% の誤差を許容するとしたら，抽出率は 0.02% 以下に削減されることになる．すなわち，100G の命令の中で $100\text{G} \times 0.02\% = 20\text{M}$ だけをシミュレートすることによって 1% の誤差を達成することができる．

さらに，SimPoint の誤差は，より大きな抽出率で飽和している．SimPoint の誤差は，1M インターバル命令 [14–17] の時，誤差が飽和状態になる．

結論として，我々の提案手法は，クラスタリング閾値を変更することによって SimPoint よりもシミュレーション・ポイント割合と誤差間のトレードオフ点をより自由に選出することができる．

5.2.4 インターバルによるフェーズ検出

SimPoint は膨大な数の疎ベクトルをクラスタリングすることは本質的に困難であり，この限界から 1M 命令より長いインターバルを必要とする．

対照的に，我々の提案手法は 1K インターバル命令ではうまく機能してない． そ

れは，IPC を安定して測定するには 1K インターバル命令が本質的に短すぎるからである．例え，主記憶メモリのレイテンシは数百サイクルなので，1K インターバル命令の IPC は，数十%の最終レベルのキャッシュミスによって変動する．

ここで，上の推測を証明する方法はないことに注意されたい． 2 つのインターバルのうち 1 つのインターバルの IPC が最終レベルキャッシュミスによってのみ低下する場合，それらは定義によって異なるフェーズとみなされるべきである． 言い換えると，フェーズは，1K 命令のような短いインターバルでその意味を失う．

5.2.5　選出されたインターバルのシミュレーション・ポイント

前述ようにクラスタリング閾値は，シミュレーション・ポイントの数と推定の精度との間のトレードオフに影響を及ぼす． より小さい閾値より，より多くのシミュレーション・ポイントが選ばれ，より推定精度が向上する．

一方で，小さい閾値より推定の精度は上がりるが，クラスタリング時間が長くな

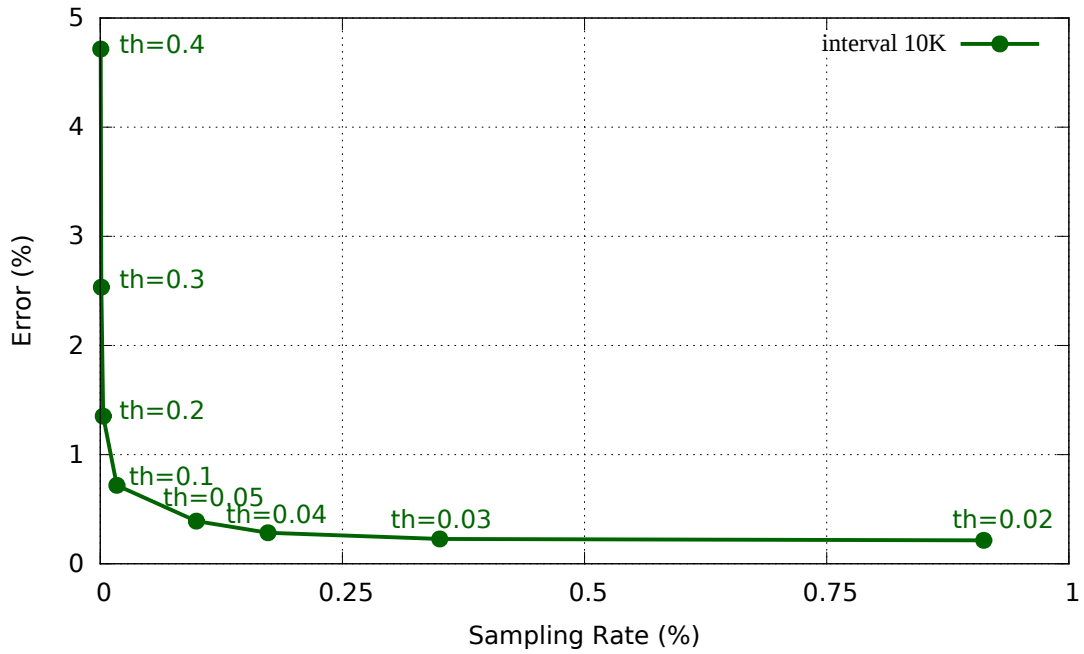


図 5.3: 異なる閾値に対する誤差―抽出率グラフ

る．それによって最適な閾値を選ぶ必要がある．この図 5.3 より，提案手法の閾値の飽和ポイントが 0.05 であることを示す．曲線上の各点は，合計 29 種類のプログラムの 4 つの推定アーキテクチャのそれぞれ合計 116 通り (プログラム 29 種類×モデル 4 種類）の組み合わせで推定を行った際の，推定結果を幾何平均したものである．

図 5.4 は提案手法によって選出された 483.xalancbmk のシミュレーションを示す．この図では，x 軸は 0 から 100G の命令で測定された動的インターバル数を示し，y 軸は IPC を示す． 各ポイントはスーパスカラ・プロセッサモデルの 3 つの閾値：0.05，0.2，0.4 によってシミュレーション・ポイントとして選出されたインターバルの IPC である．これらの 3 つの閾値に対して，選出された 10K インターバル命令のシミュレーション・ポイントの数は，9924，304，41 である． 4 つの評価モデルに対する誤差の平均はそれぞれ 0.04%，1.35%，4.7% である． ここで，5.3 節の図 5.10, 5.11, 5.12, 5.13 より，IPC のばらつきが広かっているのは，インターバル長さが違うためである．図 5.4 では，10K インターバル命令，図 5.10, 5.11, 5.12, 5.13 では

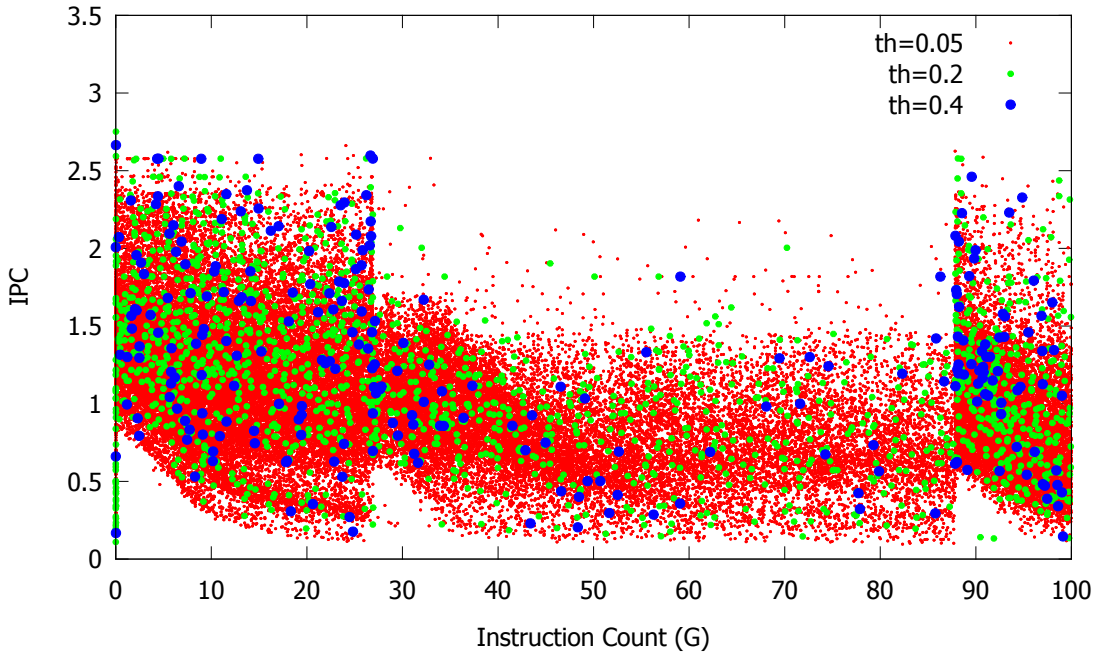


図 5.4: 483.xalancbmk の選出されたシミュレーション・ポイント．

インターバル長は 10M 命令である．より短いインターバルで観測されるより高いまたはより低い IPC 値は，より長いインターバルで平均化される．

5.2.6　モデル regcache のプログラムごとの結果

前に述べたように regcache モデルは一般的な IPC 変動要因によるフェーズを提案手法が検出できる確かめるに適切である．それで，regcache モデルを通じて，個々のプログラムの概観をを見ていく．ここでは，regcache について，パラメタは提案手法と SimPoint のインターバルはそれぞれが 10K 命令，1M 命令であり，前の結果から得られた各手法の適切値である．提案手法のクラスタリングの閾値は 0.05 であり，これはシミュレーション・ポイントの割合がより高い精度で選出されたものである．図 5.5 と 図 5.4 は 29 種類のプログラムについて，先ほど同様，シミュレーション・ポイントの割合と IPC 推定の誤差率の関係をプロットしたもので, この図で SimPoint が三角で，提案手法が丸の凡例で示されている．

個個ベンチマーク・プログラムの対応関係をもっと概観的見るため，図中の下のグラフは矢印で示す．同図中，1 本の矢印は 29 種中の 1 種のプログラムに対応し，矢印の始点 → 終点が SimPoint→ 提案手法の誤差—抽出率をそれぞれ表す．

明らかに，提案手法のプロットは原点の周りに集中しているが，SimPoint は原点からの遠いところにプロットされる．これは，ほとんどのプログラムで提案手法が SimPoint より優れる性能を示す．すなわち，29 例中 21 例において，誤差の削減を達成している．

提案手法は，3.4 節で説明されているものとは異なる IPC 変動要因を持つ regcache でもうまくいく．これは，部分的には，regcache モデルの IPC 低下がサイクルあたりのレジスタ・アクセス数に起因するため，命令レベルの並列性と強く相関しているからと考えられる．これは，スカラ・プロセッサとスーパスカラ・プロセッサとの間の IPC の違いによって区別することができる．逆に，ターゲット・モデルに命令レベルの並列性に相関しない別の IPC 変動要因がある場合には，その方法を別の基底モデルで区別する必要がある．

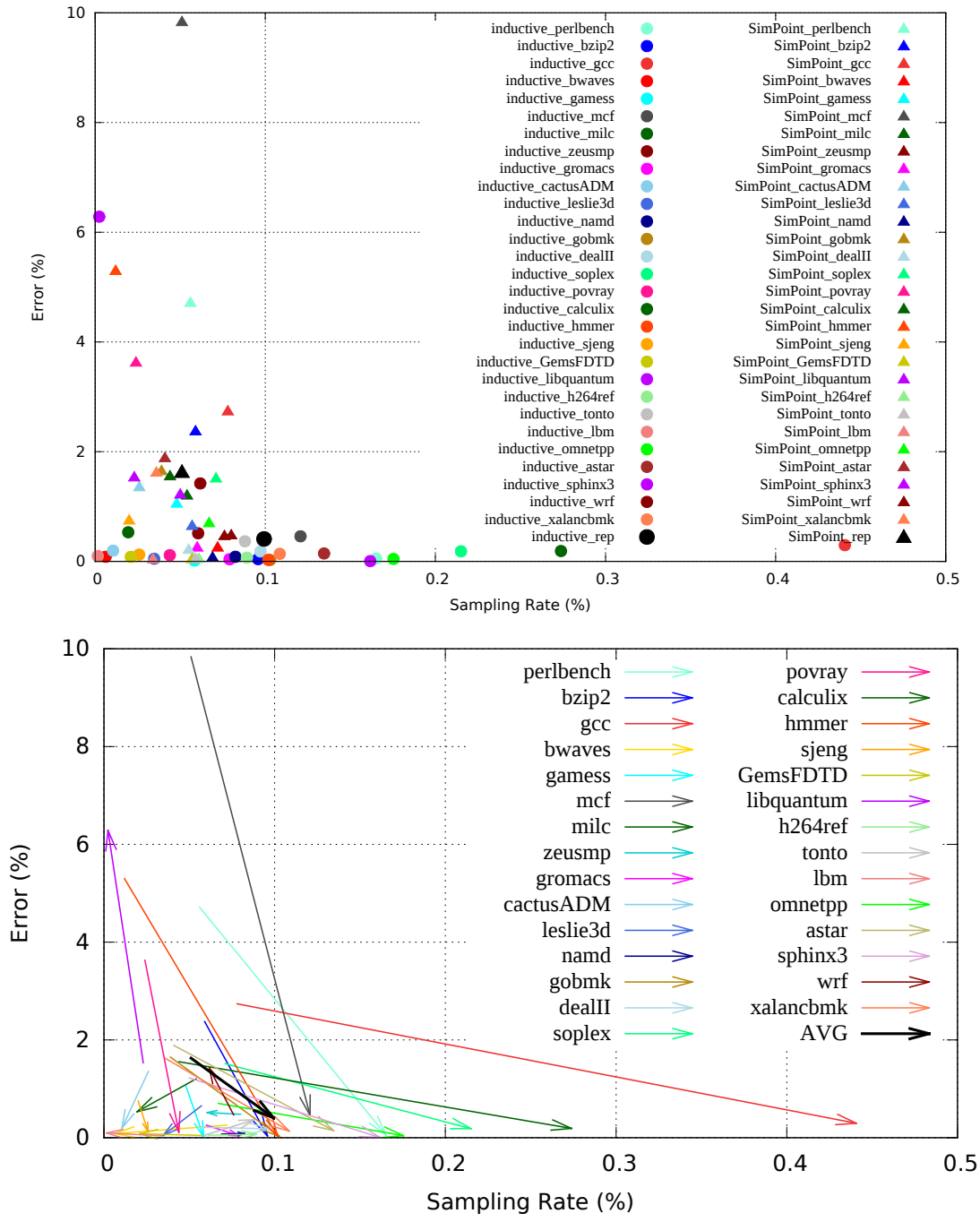


図 5.5: モデル regcache の誤差—抽出率グラフ

表 5.4: モデル regcache の誤差 (Estimation Error: EE) – 抽出率 (Sampling Rate: SPR).

	Proposal		SimPoint	
program	SPR (%)	EE (%)	SPR (%)	EE (%)
400.perlbench	0.16531	0.0569819	0.064	2.400915332
401.bzip2	0.096	0.0317895	0.059	2.375155834
403.gcc	0.44067	0.294711	0.057	3.175515964
410.bwaves	0.00615	0.0682408	0.075	0.120894075
416.gamess	0.05866	0.00288012	0.048	1.043201498
429.mcf	0.12072	0.457023	0.041	2.02422783
433.milc	0.01947	0.530655	0.061	1.074328027
434.zeusmp	0.0605	0.510476	0.08	0.678597196
435.gromacs	0.07905	0.0105306	0.061	0.35012265
436.cactusADM	0.01052	0.217292	0.029	1.663994503
437.leslie3d	0.0347	0.0450978	0.063	1.063727122
444.namd	0.08252	0.0691879	0.069	0.445028262
445.gobmk	0.10289	0.00573879	0.032	2.429978395
447.dealII	0.09733	0.179495	0.055	0.687564914
450.soplex	0.21462	0.205719	0.071	1.613389053
453.povray	0.04385	0.113594	0.076	0.099344258
454.calculix	0.27383	0.189064	0.051	1.582828304
456.hmmer	0.10176	0.0275611	0.01	0.049125903
458.sjeng	0.02577	0.106415	0.02	0.749368112
459.GemsFDTD	0.02092	0.0913519	0.058	0.162416971
462.libquantum	0.00236	6.28389	0.022	1.55448812
464.h264ref	0.08956	0.128633	0.057	0.115202151
465.tonto	0.08833	0.307492	0.058	0.640419459
470.lbm	0.00156	0.0978773	0.007	0.898131423
471.omnetpp	0.17556	0.0948222	0.068	0.71854975
473.astar	0.13454	0.144677	0.053	0.941293766
481.wrf	0.06178	1.4277	0.063	0.68934959
482.sphinx3	0.16146	0.0034065	0.055	4.020708492
483.xalancbmk	0.10864	0.180485	0.034	5.032397794
AVERAGE	0.099277	0.409751	0.051621	1.324147

5.2.7 その他のプログラム

本評価では29種類のプログラム×４種類のモデルの合計116通りの推定を行ったが，おおむね提案手法の方がSimPointより良い結果を示していたといえる．提案手法よりSimPointの方が良い結果を示してたのは現在確認されている中では8例である．

この中，cache-halfの435.gromacsと447.dealII，eight-wayの486.wrfと403.gcc，regcacheの462.libquantumと486.wrf計6例は，パラメタによっては，SimPointの方が提案手法よりよい結果を示した．の理由は，以下の節でいくつかの試験を通じて部分的に議論される．

5.3 詳細な結果

この節では，結果の中からいくつかの特徴的だったものについて，クラスタリング結果を色分けして示す．

5.3.1 クラスタ色分けグラフ

図5.6，5.7，5.8，および，5.9は**クラスタ色分けグラフ**を示している．

グラフ中，点1つがインターバル1つに相当する．x軸はそのインターバルがプログラムの開始から何命令目かを表し，y軸はそのインターバルの区間IPCを表す．各手法によって同じクラスタに分類されたインターバルには同じ色が割り当てられている．すべてのクラスタをプロットするとグラフが塗りつぶされてしまうため，クラスタの大きさ（含まれるインターバルの数）の順に上位10クラスタ（に属するインターバル）のみをプロットしている．

その上位10クラスタには，gnuplotの“classic” sequence (red, green, magenta, turquoise, orange, light-green, pink, blue, purple, gray)を順に割り当てている．したがって，異なるグラフ間では同じ色に意味はないことに留意されたい．また，黒い点はシミュレーション・ポイントとして選出されたインターバルを示している．

なお，細かいとやはり塗り潰されてしまうため，両手法ともインターバル長は10M命令とした．提案手法のクラスタリング閾値は0.05とした．

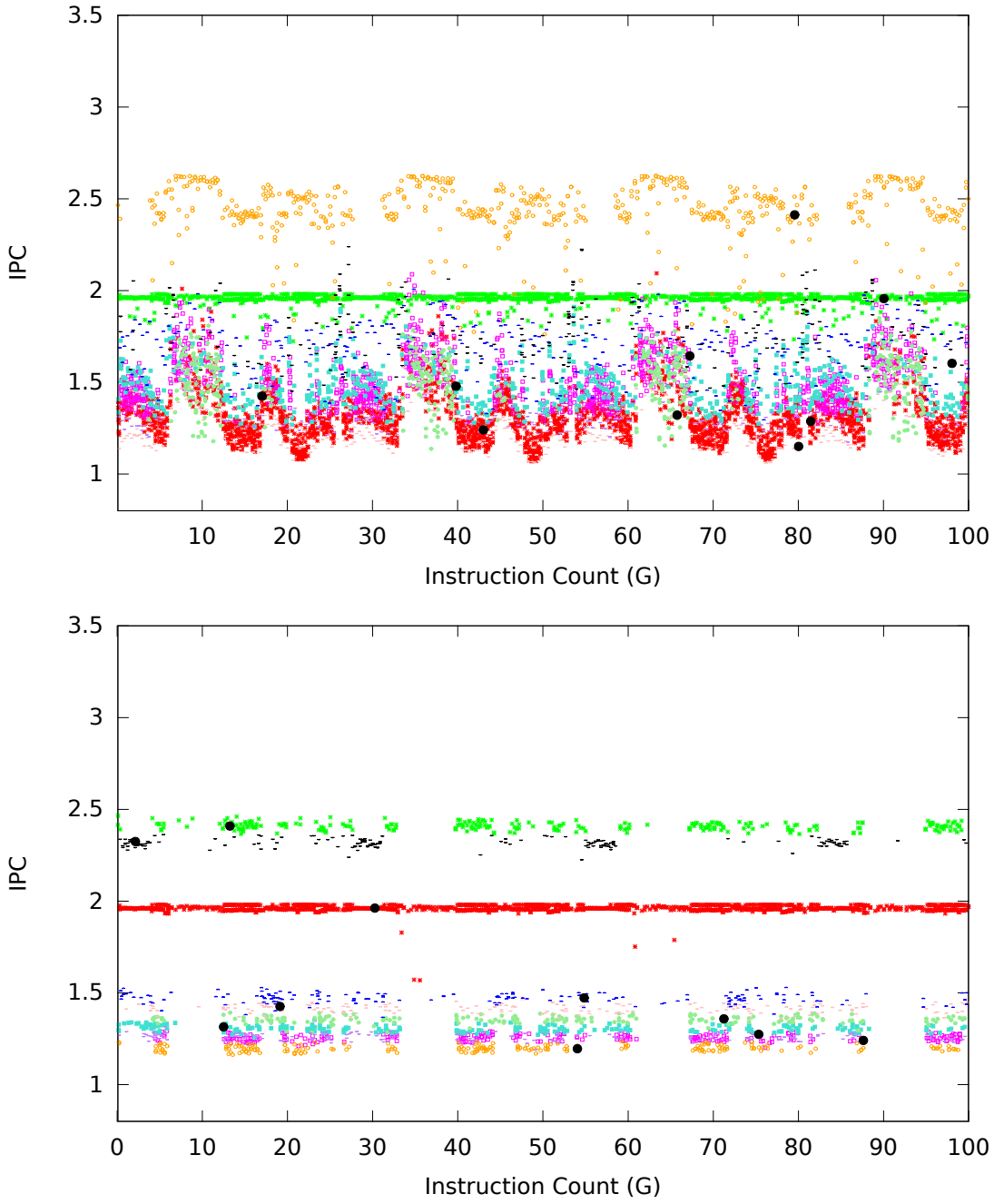


図 5.6: cache-half 上の bzip2 のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

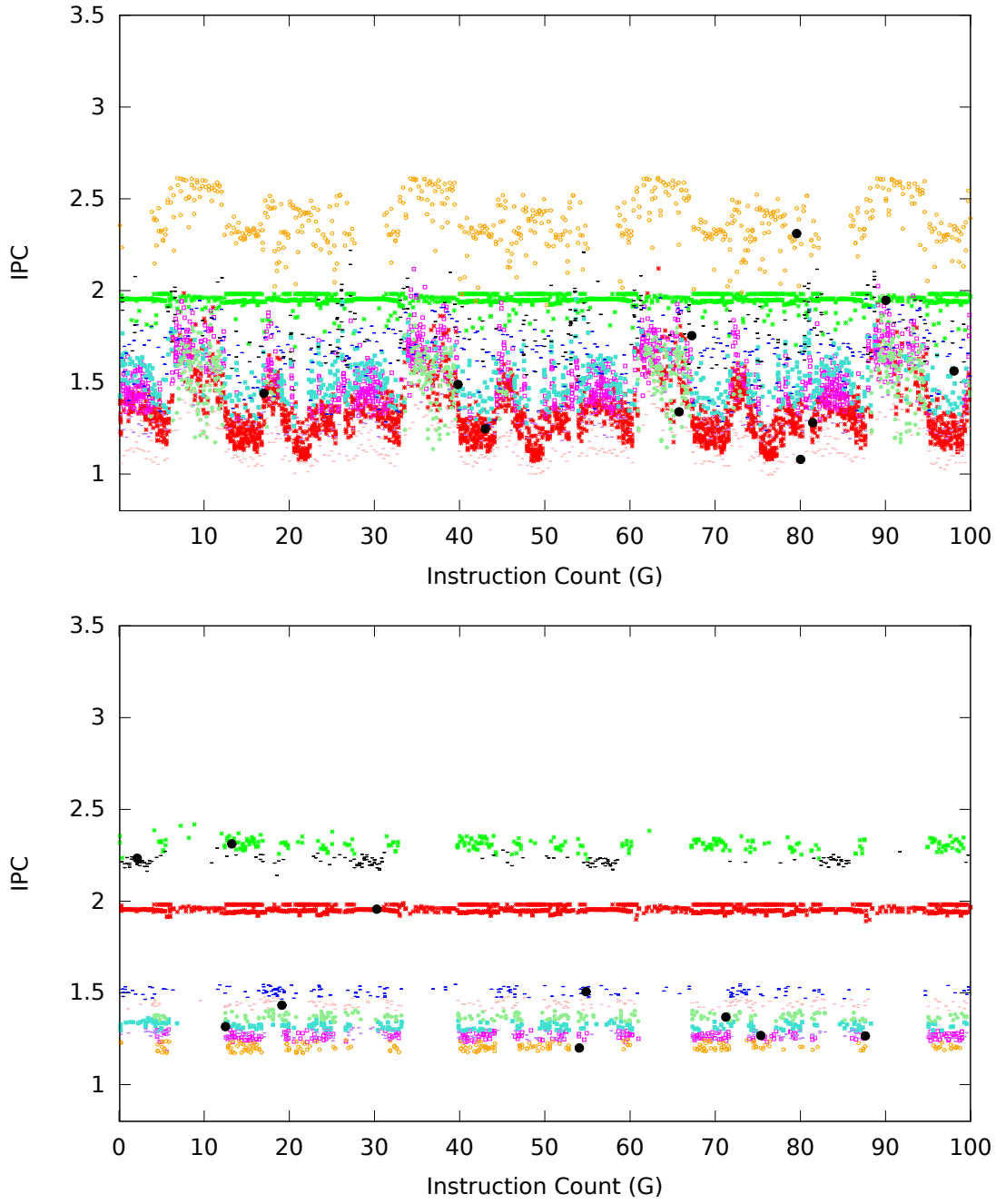


図 5.7: pht-single 上の bzip2 のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

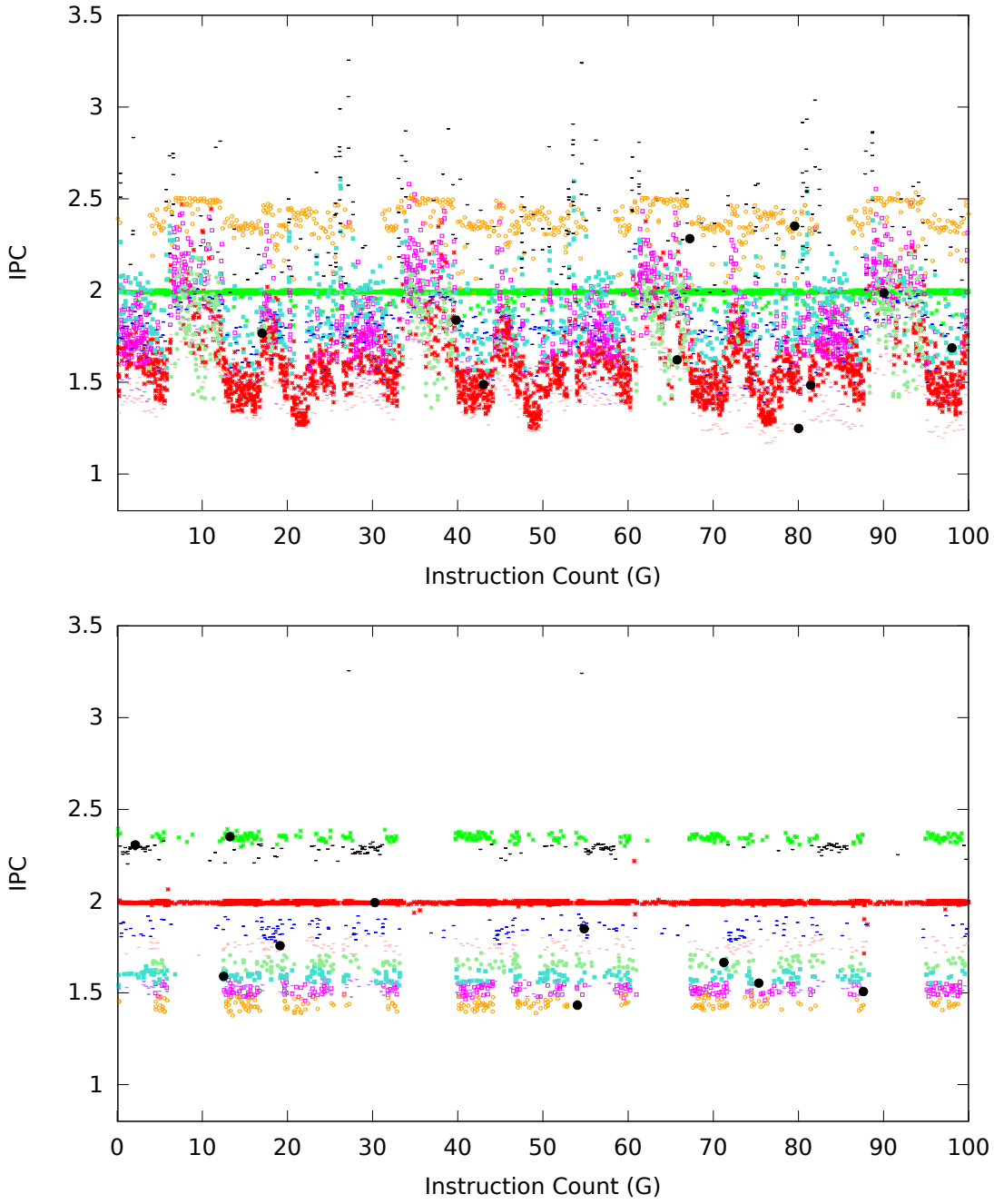


図 5.8: eight-way 上の bizp2 のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

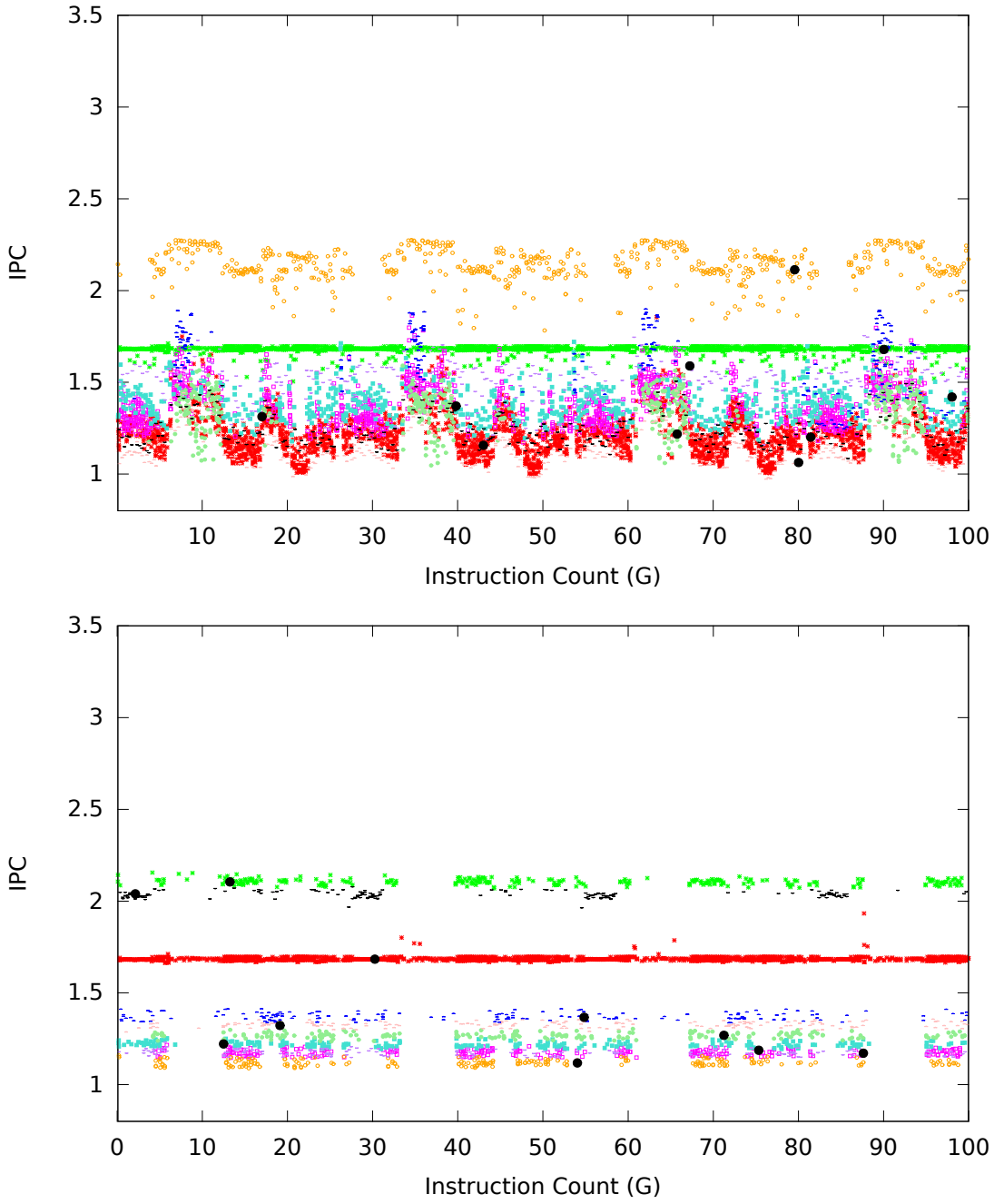


図 5.9: regcache 上の bizp2 のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

5.3.2　401.bzip2

401.bzip2 は提案手法と SimPoint の比較の中では特に提案手法でよい結果が出たプログラムの一つである．図 5.6，5.7，5.8，および，5.9 はそれぞれ，cache-half，pht-single，eight-way，および regcache 上で 401.bzip2 を評価した際のクラスタ色分けグラフである．

提案手法では，きれいな水平の帯が認められる．これは，ほぼ同一の IPC のインターバルが同じクラスタに分類されていることを示している．

一方で SimPoint では，異なる色の点が混じっており，同じ色，すなわち，同じクラスタに分類されたインターバルの IPC に大きなばらつきがあることが分かる．クラスタからのシミュレーション・ポイントとしての選出されたインターバルは，少なくとも IPC に関しては，そのクラスタの代表点とは言い難い．その結果，IPC 推定が悪くなる．

5.3.3　483.xalancbmk

次に 483.xalancbmk について同様の比較を行う．

前述したように，483.xalancbmk は実行に従ってワーキング・セット・サイズが増加するループを持つプログラムである．その結果，キャッシュ・ヒット率と IPC は徐々に悪化する．

図 5.10，5.11，5.12，5.13 はそれぞれ，cache-half，pht-single，eight-way，regcache 上で xalancbmk をシミュレートした際の IPC を示している．ただし提案手法のクラスタリング閾値は 0.3 とした．

両方のグラフには異なる色の帯が見られるが，特に SimPoint の帯は大きく湾曲している．SimPoint のクラスタリングは PC に基づいており，静的区間の同じ集合が繰り返し実行されている．帯の曲線は，これらの静的区間の IPC が，徐々に増加するワーキング・セット・サイズのために，互いに同じ速度で徐々に劣化することを示している．

一方で，提案手法の水平な帯は，PC とは無関係にほとんど同じ IPC の区間が同じクラスタに分類されていることを示している．ところが，図 5.10 に示されているように，cache-half に対しては，提案手法でも IPC の分布が大きく広がってしまっ

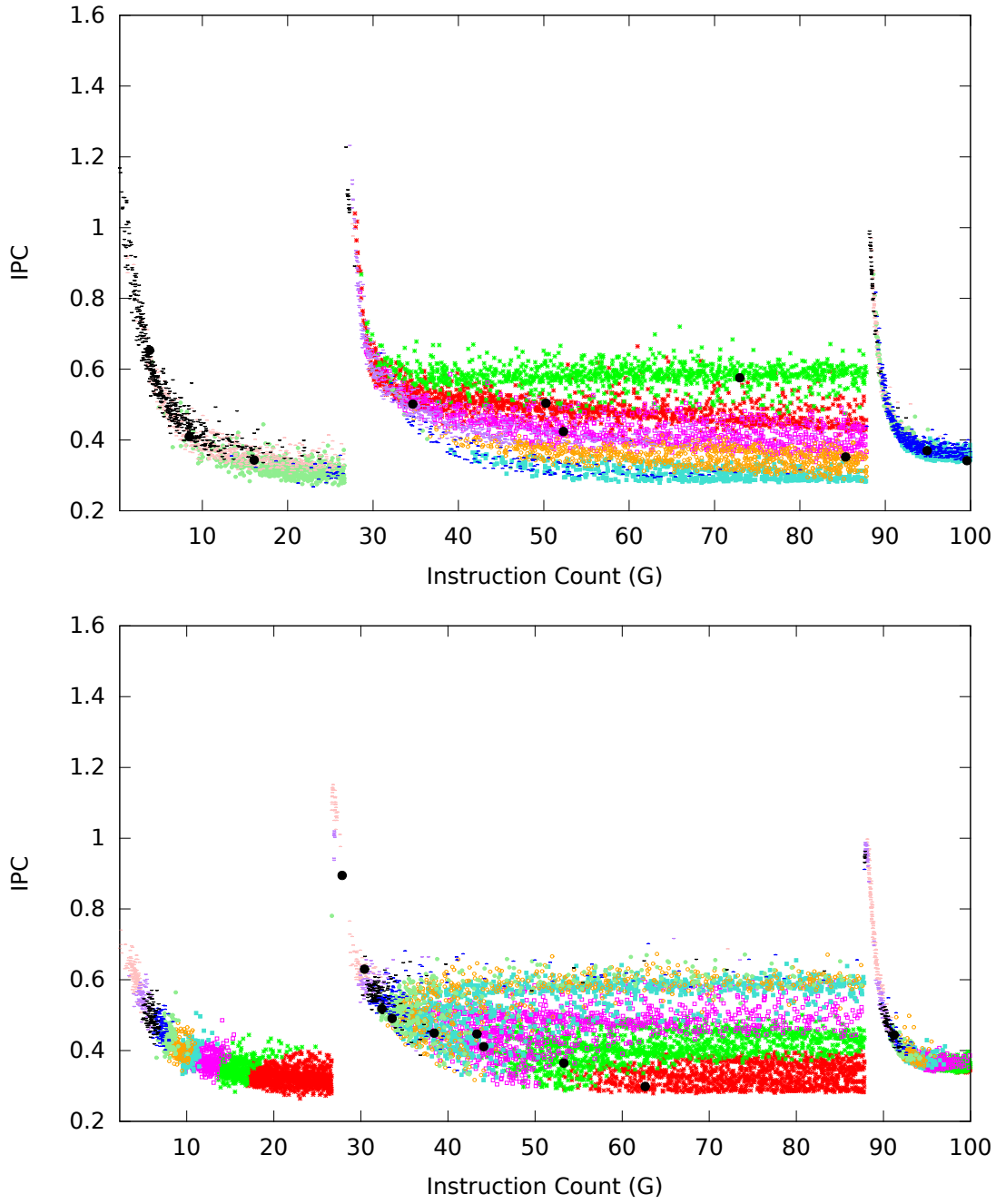


図 5.10: cache-half 上の xalancbmk のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

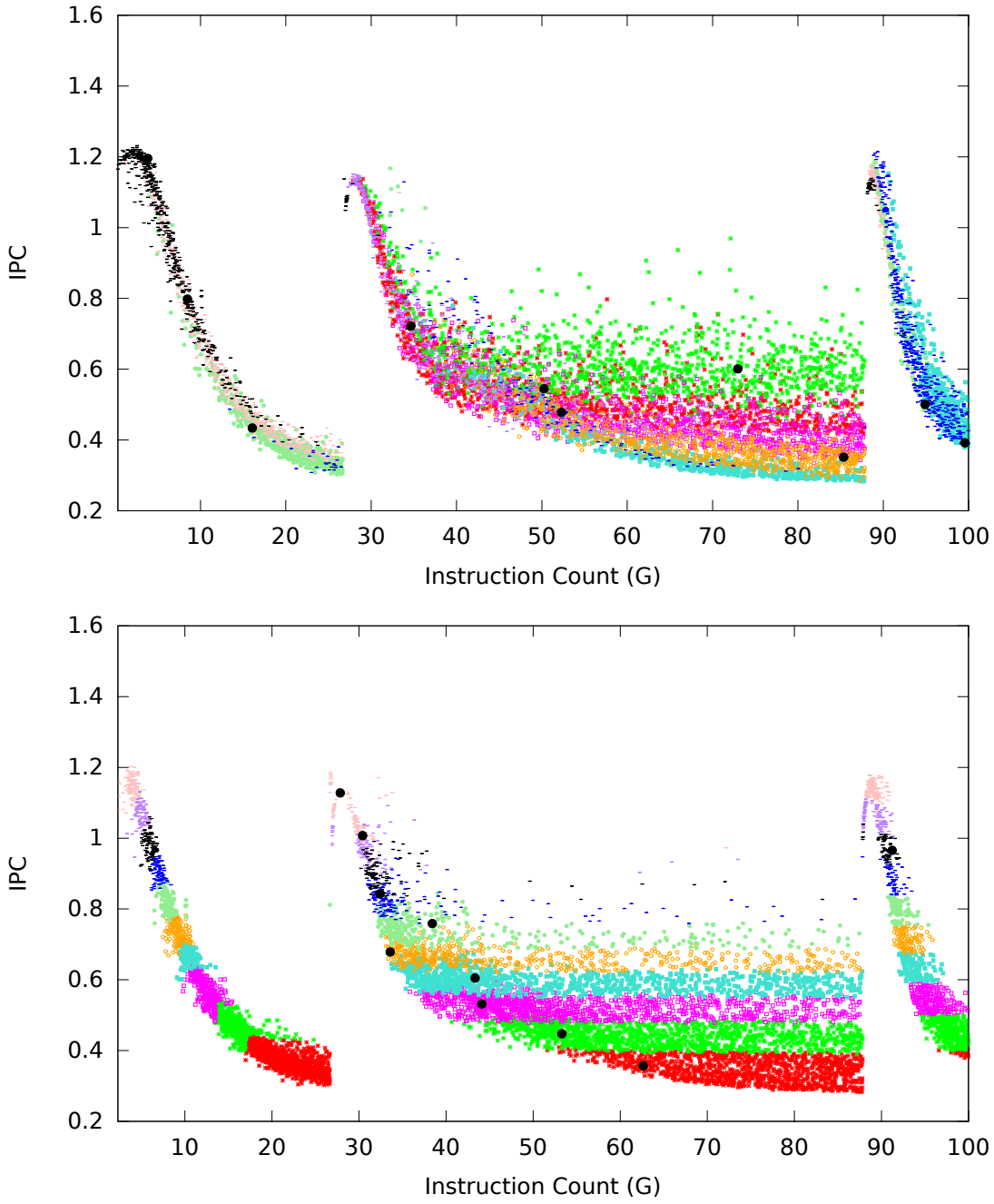


図 5.11: pht-single 上の xalancbmk のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

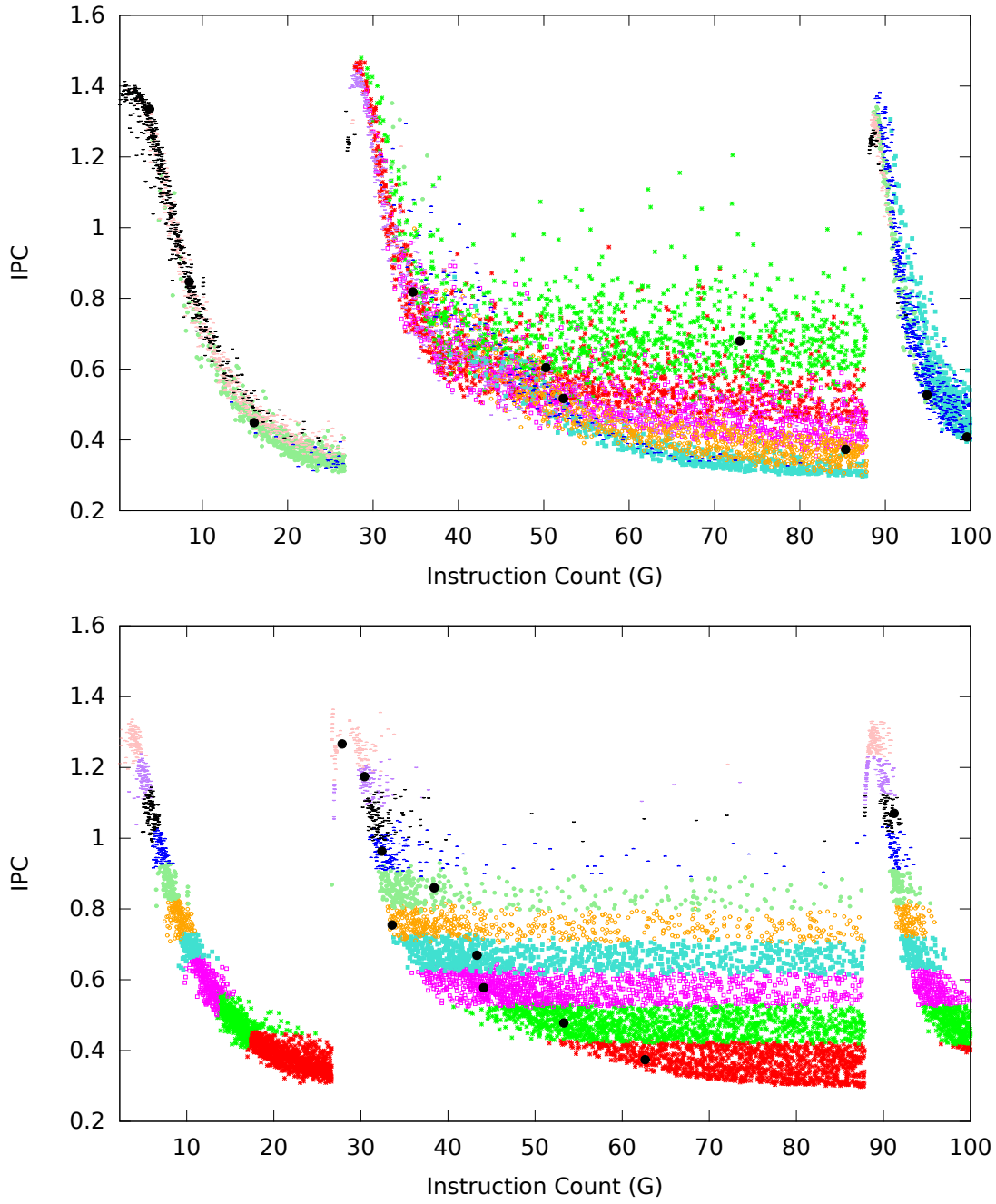


図 5.12: eight-way 上の xalancbmk のクラスタ色分けグラフ．SimPoint（上）と提案手法（下）．

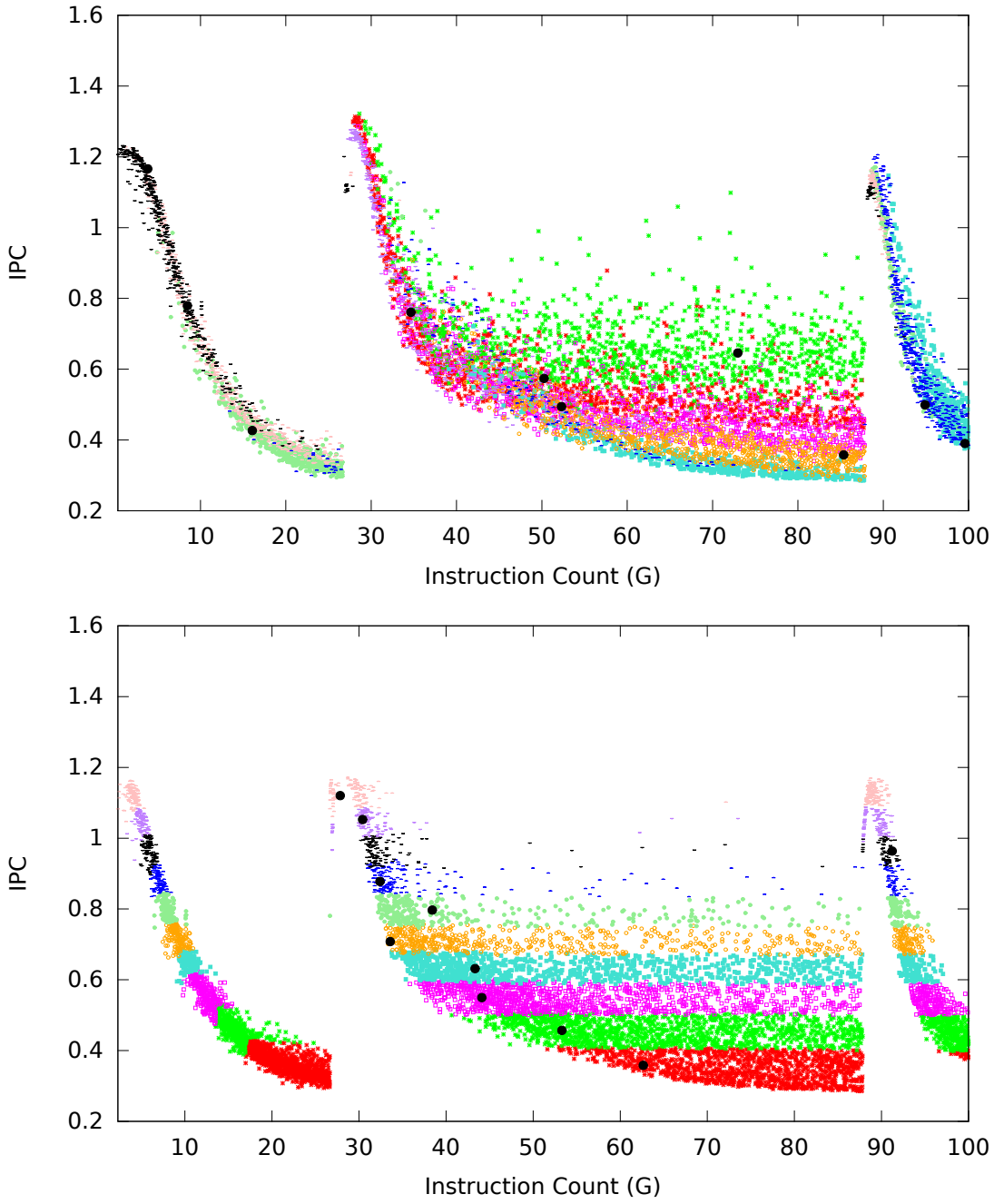


図 5.13: regcache 上の xalancbmk のクラスタ色分けグラフ．SimPoint （上）と提案手法（下）.

ている．これは，キャッシュ容量を半分にしたために生じた新しいフェーズを提案手法が正しく分類できていないために起こると考えられる．

第6章

改良手法の性能評価

この章では，第4章で述べた3つの改良点の評価を行う．6.3，6.2および，6.4節，のそれぞれにおいて，

1. 超多階層キャッシュ，すなわち，超多階層キャッシュを持つ emu+hhcache を emu+cache の代わりに基底モデルに用いて評価する．超多階層キャッシュは，キャッシュ容量によって生じるフェーズに対応するために，キャッシュ容量の異なる多のキャッシュを持つ単一のモデルである．
2. 事前シミュレーション時間を削減するエミュレーション・ベースの基底モデル，すなわち，キャッシュ・ミス，分岐予測ミス，命令の依存関係に着目したそれぞれの要因に対応する理想的な基底モデルである．そういうことによって事前シミュレーションにかかる時間を無くして，基本的な帰納手法の短所が補完される．これらを用いてシミュレーション・ポイントを選出し，そのシミュレーション・ポイントを使って評価する．
3. 集合的手法，すなわち，ベンチマークに含まれるすべてのプログラムの実行を単一のワークロードとみなして，すべてのベンチマーク・プログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する手法の結果を示す．この手法は，PCではなく，IPCベクトルを用いてフェーズ検出を行い，シミュレーション・ポイントを選出するから可能である．

6.1 評価方法

第 5 章の結果に従い，本章でも，特に断りのない限り，インターバル長は 10K 命令に固定する．一方，クラスタリング閾値は，0.05，0.1，0.2，0.3 と変更して評価するが，特に断りのない場合には 0.05 とする．

6.2 超多階層キャッシュ

この節では，超多階層キャッシュを持つ emu+hhcache を emu+cache の代わりに基底モデルに用いて評価する．

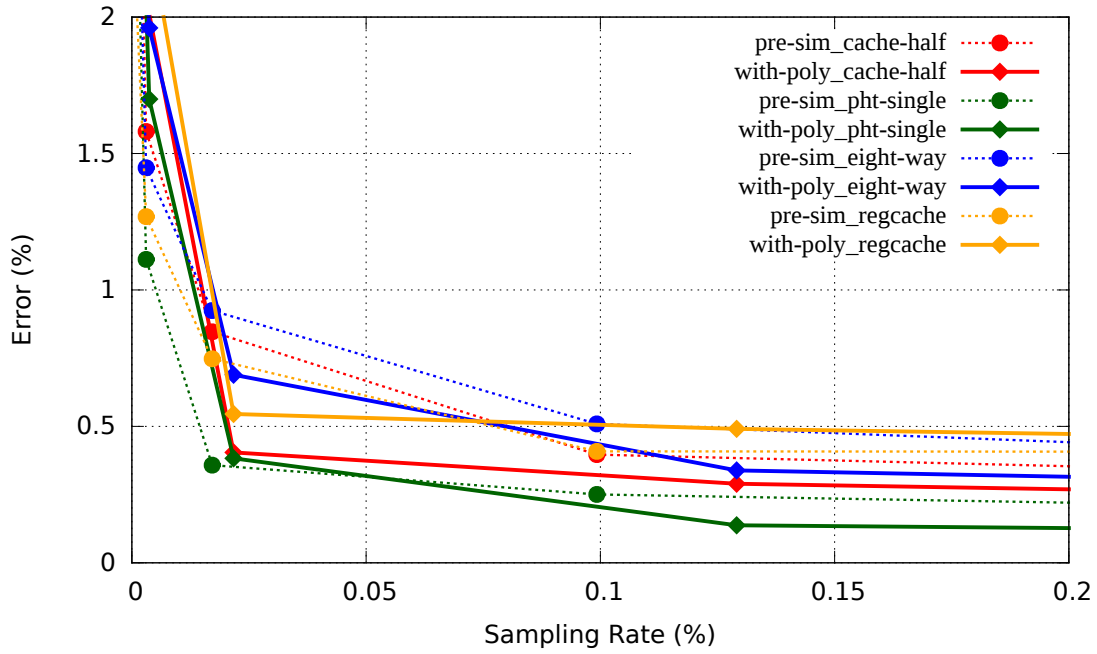


図 6.1: emu+cache（破線）と emu+hhcache（実線）を用いた手法の誤差—抽出率グラフ

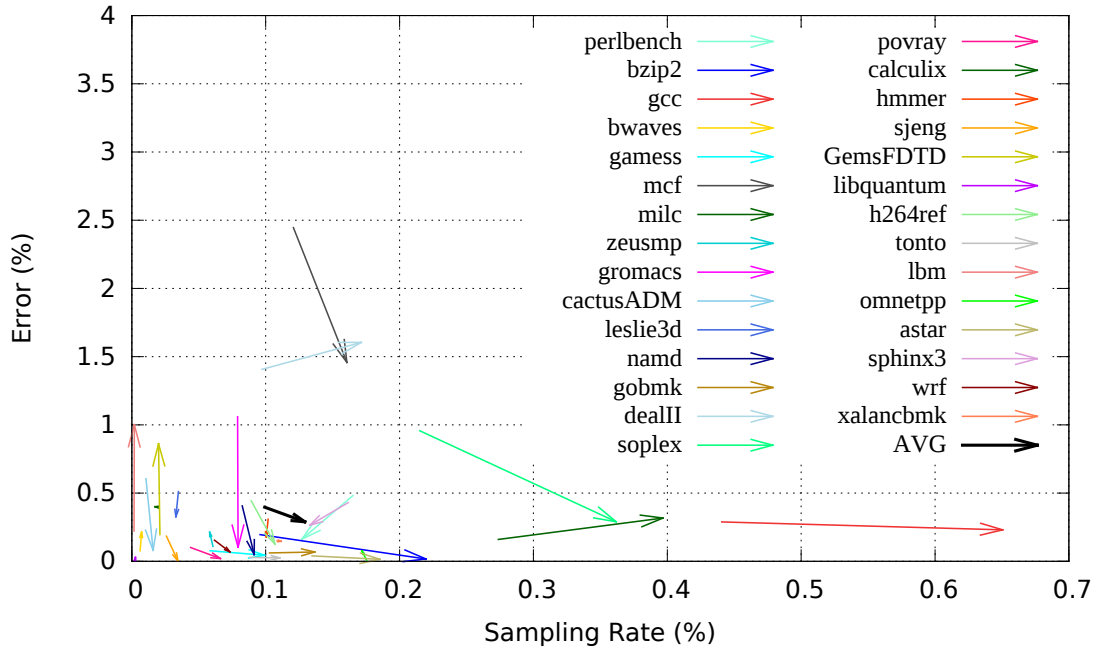


図 6.2: 超多階層キャッシュを追加したシミュレーション・ベースの手法の cache-half の誤差—抽出率グラフ

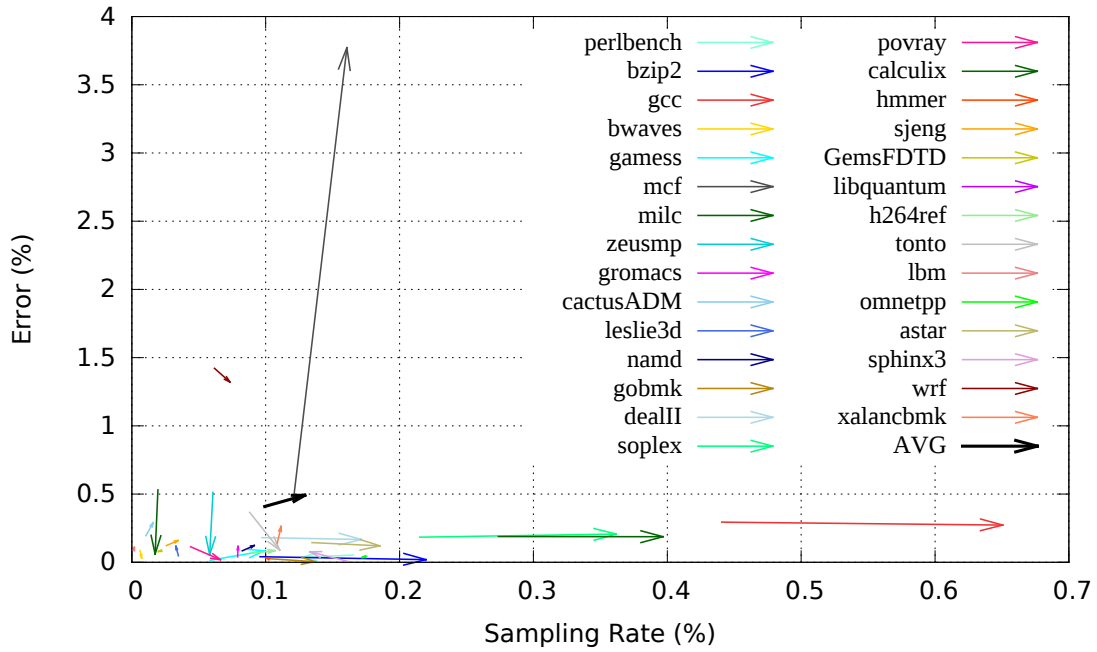


図 6.3: 超多階層キャッシュを追加したシミュレーション・ベースの手法の regcache の誤差—抽出率グラフ

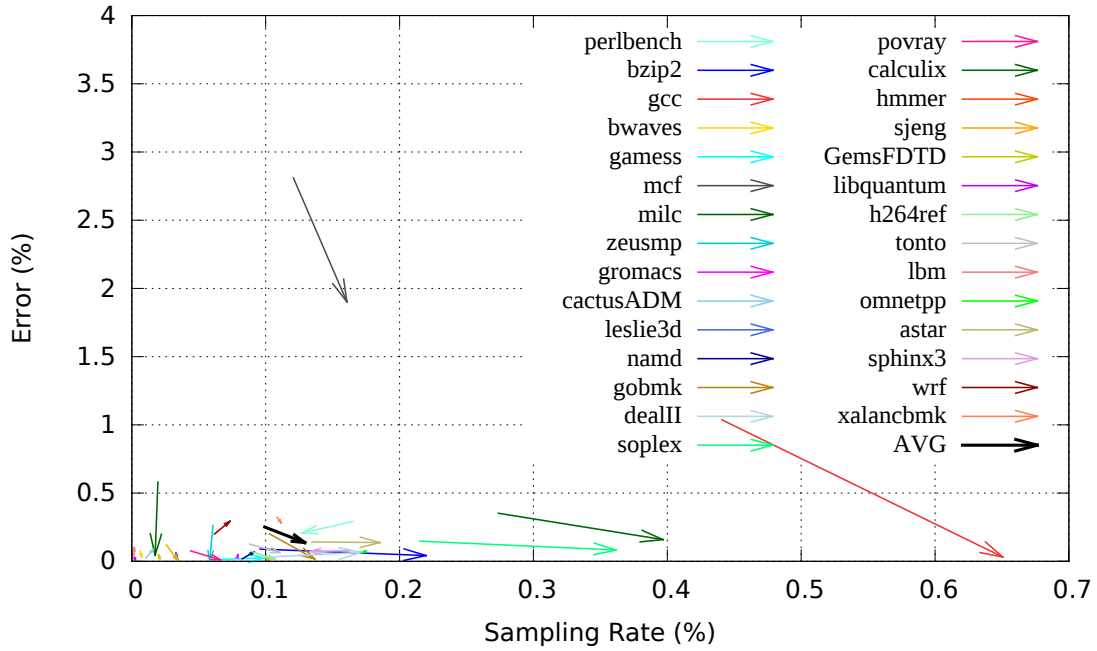


図 6.4: 超多階層キャッシュを追加したシミュレーション・ベースの手法の pht-single の誤差—抽出率グラフ

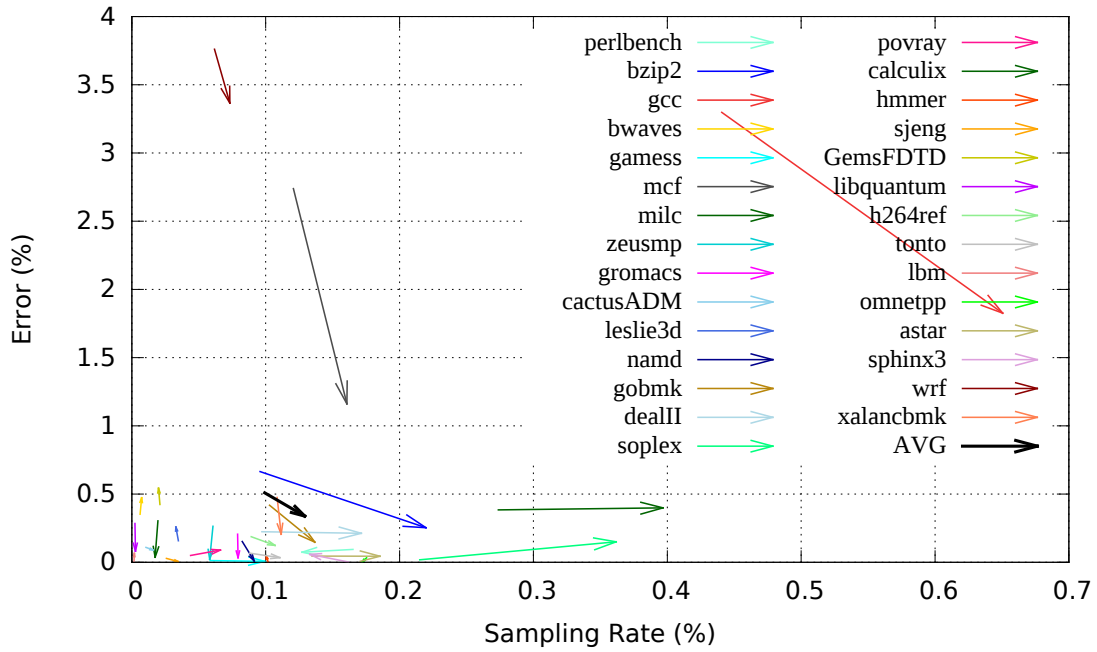


図 6.5: 超多階層キャッシュを追加したシミュレーション・ベースの手法の eight-way の誤差—抽出率グラフ

6.2.1　ターゲット・モデルごとの比較

4 つのターゲット・モデルごとの超多階層キャッシュを持つ基底モデルを用いる手法とシミュレーション・ベースの基底モデルを用いる手法との比較を行う．

図 6.1 は，4 つのターゲット・モデルについて，超多階層キャッシュを持つ基底モデルを用いる手法は実線で，シミュレーション・ベースの基底モデルを用いる手法は破線で表す．曲線上の各点は，29 種類のプログラムの結果を平均したものである．

図 6.1 は，4 つの曲線がインターバルごとにまとまっていることを表している．これは，ターゲット・モデルごとの結果の平均には，超多階層キャッシュを持つ基底モデルを用いる手法も目立ったばらつきは見られない．

そして，超多階層キャッシュを持つ基底モデルを用いる手法の曲線のグループがシミュレーション・ベースの基底モデルを用いる手法のグループよりも原点に近いので，超多階層キャッシュを持つ基底モデルを用いる手法がより良いシミュレーション・ポイントを選択できることがわかる．例え，シミュレーション・ポイントの割合が約 0.1% の場合，推定誤差は約 0.4% から 0.35% に改善する．

その他のプログラム　ところが，図 6.2, 図 6.3, 図 6.4, 図 6.5 において，470.lbm のプログラムの推定結果は 4 つのターゲット・モデルで見てみると，すべての場合について，特に cache-half の時，emu+cache の方がよい結果になっている．一方，429.mcf の時，regcache 以外のターゲット・モデルで超多階層キャッシュを追加した場合，さらなる推定性能向上の効果があった．

6.2.2　cache-half 上の 483.xalancbmk

図 6.6 に，cache-half 上の 483.xalancbmk のクラスタ色分けグラフを示す．図 6.6（上）では，桃色，緑色，淡桃色のクラスタに分類された点が上下に散らばっている．一方で，同図（下）では，この傾向が抑えられている．これは，今まで同じクラスタに誤分類されていたインターバルが別々のクラスタに分割されたためである．すなわち，超多階層キャッシュがキャッシュの容量に起因するフェーズをより正確に捉えられていることを示す．

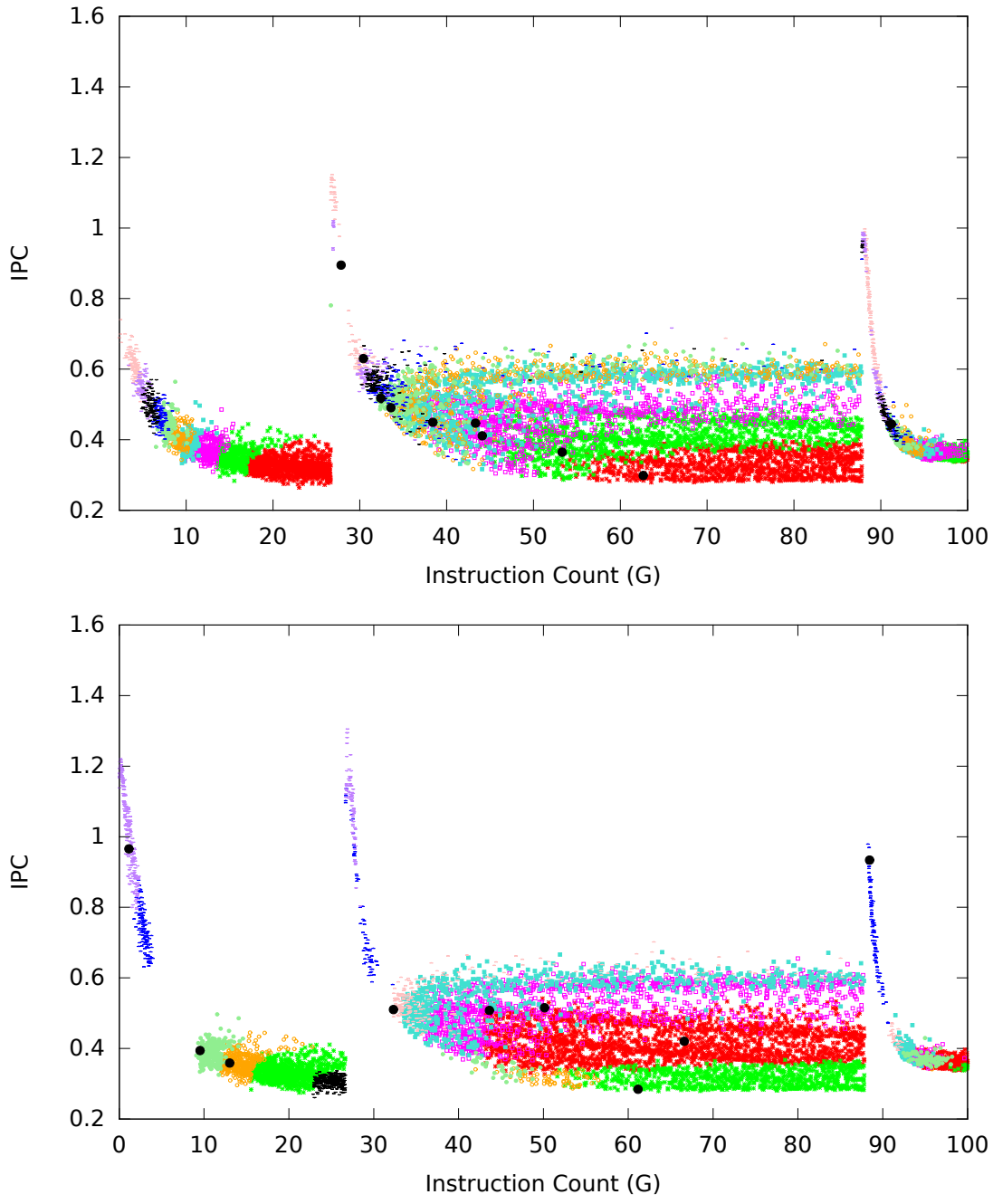


図 6.6: cache-half 上の xalancbmk のクラスタ色分けグラフ．emu+cache（上）と emu+hhcache（下）．

6.2.3　cache-half 上の 435.gromacs

435.gromacs は基本的な帰納手法と超多階層キャッシュを追加した場合の比較の中では特に超多階層キャッシュを追加した手法でよい結果が出たプログラムの一つである．また，超多階層キャッシュを追加することで，誤差を SimPoint より削減することができた．図 6.7（上）では, 58G 程度までに赤色クラスタに分類された点が，その以後緑色に分類された点と同じ模様をしているにもかかわらず，別のクラスタに分類されていた．一方で，同図（下）では，そんな現象がなく，同じ赤色のクラスタに分類されている．また，同じ閾値で，IPC が集中している 0.8∼1.4 の領域ではなく，IPC が希少である 1.4∼2.2 の領域も正確にクラスタ分類ができているのが確認できる．すなわち，超多階層キャッシュを持つ基底モデルを用いる手法がより良いクラスタを分類することを示す．

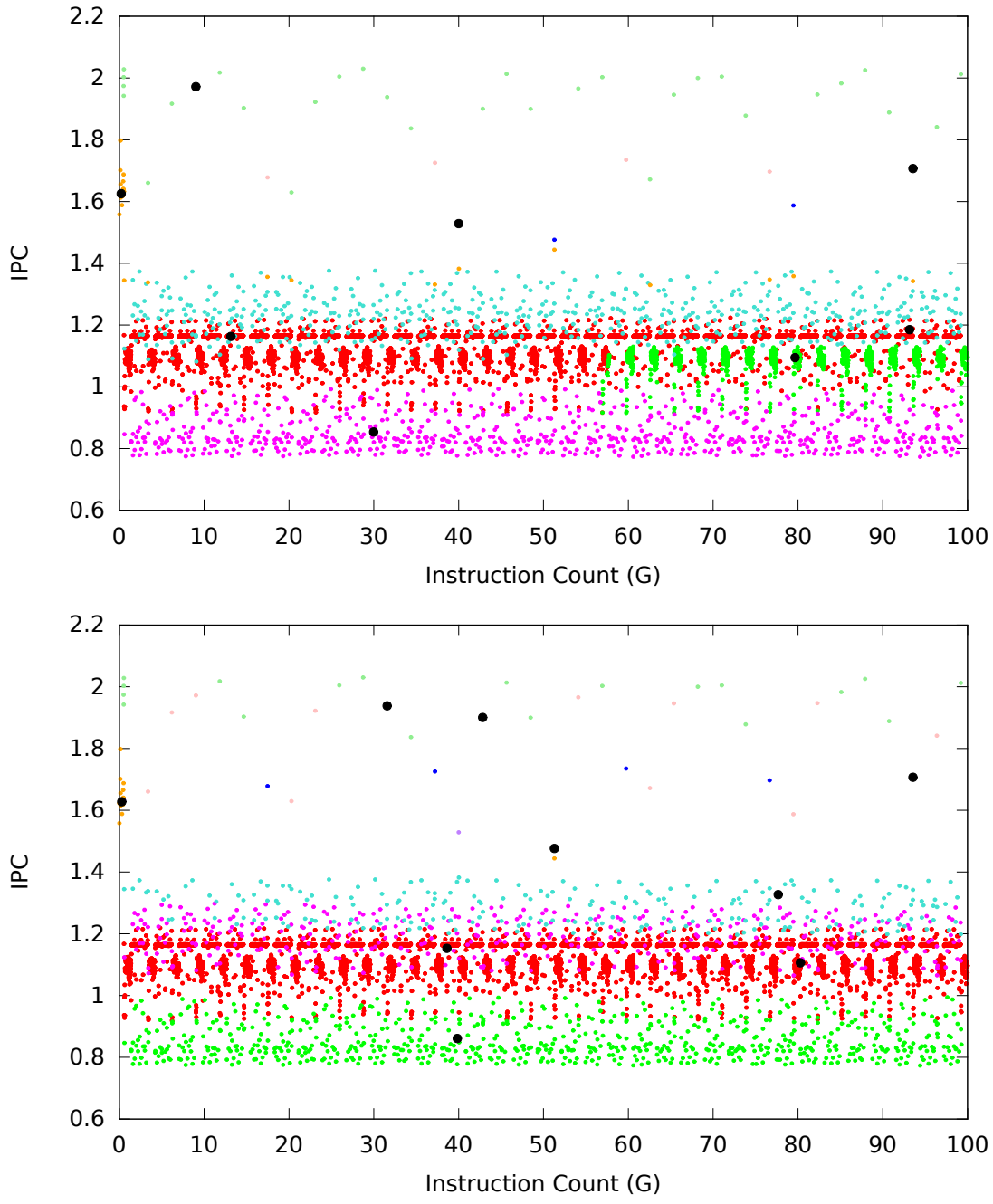


図 6.7: cache-half 上の gromacs のクラスタ色分けグラフ．emu+cache（上）と emu+hhcache（下）．

6.3　エミュレーション・ベースの基底モデル

この節では，4.2 節で述べたエミュレーション・ベースの基底モデルについて評価する．また，6.2 節で，より良い推定評価を達成している超多階層キャッシュを持つ emu+hhcache も emu+cache の代わりに基底モデルに用いる．

結果は，29 例中 12 例において，エミュレーション・ベースの手法の誤差がシミュレーション・ベースの手法より悪くなった．そこで，図 6.8 に，regcache の誤差—抽出率グラフを示す．同図中，1 本の矢印は 29 種中の 1 種のプログラムに対応し，矢印の始点 → 終点がシミュレーション → エミュレーション・ベースの手法の誤差—抽出率をそれぞれ表す．

グラフに示されているように，エミュレーション・ベースの手法では，平均的には誤差—抽出率が若干悪化している．

この場合，事前シミュレーションを必要とする基底モデルは全部事前エミュレーションで代用することによって帰納手法の短所が補完される．

この結果は，4 つの基底モデルでは表現できない IPC 変動要因がまだ存在することを示唆している．

それにも関わらず，SimPoint より優れる誤差を達成する．図 6.9 では，4 本の曲線が互いにまとまっており，モデルごとの結果には，提案手法にも SimPoint にも目立ったばらつきは見られない．

そして，エミュレーション・ベースの手法の曲線のグループが SimPoint のグループよりも原点に近く，エミュレーション・ベースの手法の方がよりよいシミュレーション・ポイントを選択できていることが分かる．例えば，シミュレーション・ポイントの割合が約 0.1% の場合，誤差を約 1.6% から 0.9% に改善することができる．

また，cache-half の 447.dealII で 1.4%から 0.5%に，regcache の 481.wrf で 1.4%から 0.3%に，eight-way の 481.wrf で 3.7%から 1.4%に，403.gcc で 3.3%から 1.2%の誤差性能向上を確認した．これは，emu+ilp によってスーパスカラ・プロセッサの幅が一般化された効果によると考えられる．

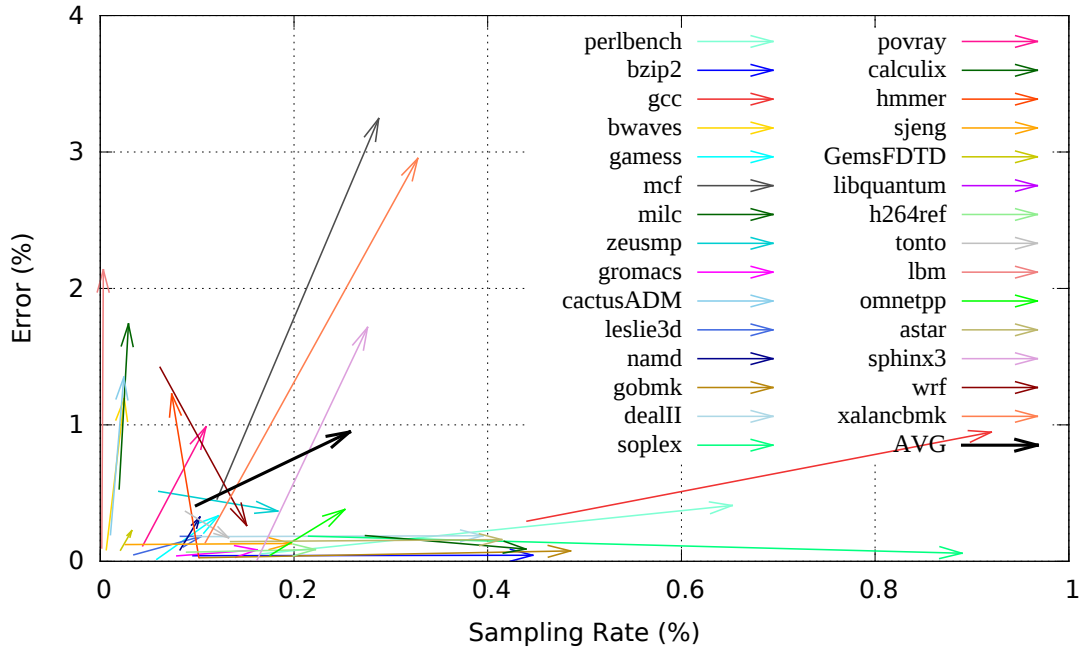


図 6.8: シミュレーション → エミュレーション・ベースの手法の regcache の誤差—抽出率グラフ

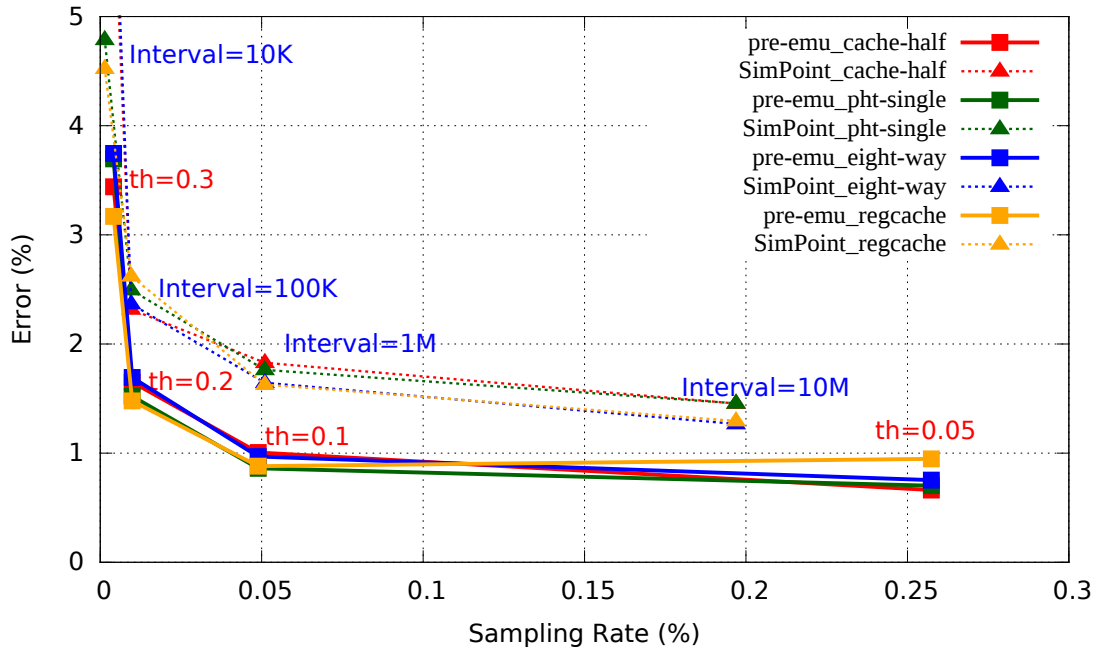


図 6.9: SimPoint とエミュレーション・ベースの手法のターゲット・モデルごとの誤差—抽出率グラフ

6.4　集合的な手法

図 6.10 と図 6.11 に，集合的手法の regcache の結果を示す．グラフ中，各矢印の，始点が非集合的，終点が集合的手法の誤差—抽出率を表す．AVG は，29 プログラムの平均を表す．また，図 6.10 はシミュレーション・ベースの手法で選出したシミュレーション・ポイントの結果であり，図 6.11 はエミュレーション・ベースの手法で選出したシミュレーション・ポイントの結果である．

シミュレーション・ベースの場合，一部誤差が悪化するプログラムも見られるものの，平均では誤差の悪化は 0.5% に抑えつつ，全体で 61.93% の抽出率の削減率が達成されている．この誤差は 0.9%で，SimPoint の 1.6%よりも優れている性能を示す．

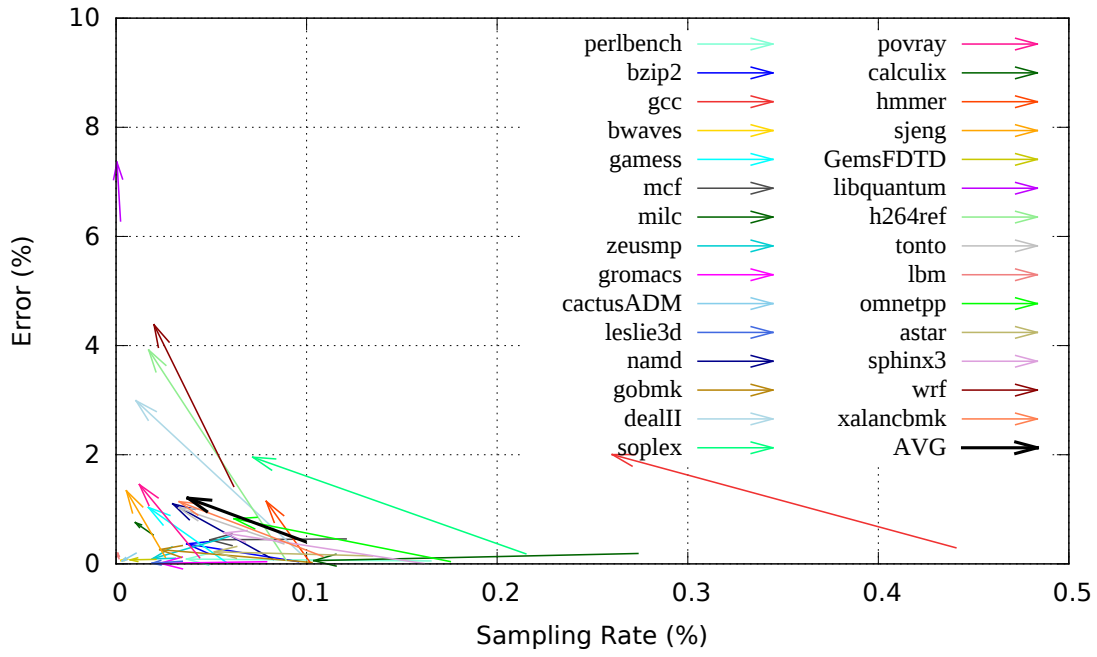


図 6.10: シミュレーション・ベース集合的手法の regcache の誤差—抽出率グラフ

ところが，エミュレーション・ベースの場合，シミュレーション・ベースと同じく全体で 59.68% の抽出率の削減率が達成されているが，平均では誤差の悪化は 1.8% で，SimPoint より悪くなる．

ここで，regcache の 462.libquantum で，誤差が 6.4%から 3.1% まで大幅に改善している．この原因を現在調査中である．

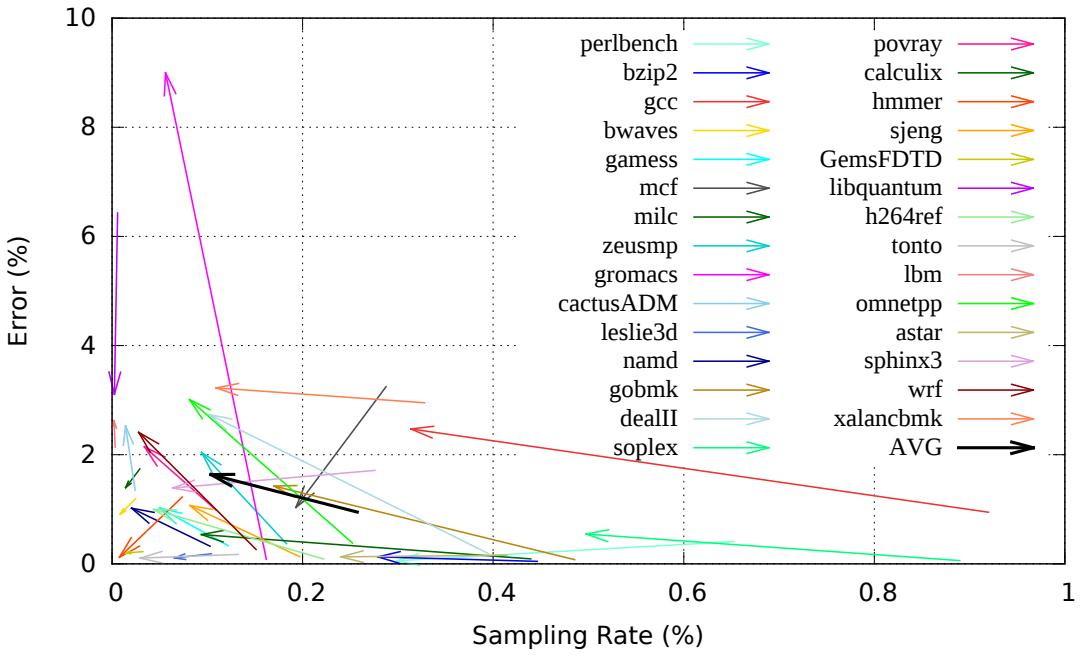


図 6.11: エミュレーション・ベース集合的手法の regcache の誤差—抽出率グラフ

第7章

結 論

プロセッサの研究・開発においては，シミュレーションが重要な役割を果たす．しかし，特に最近の Out-of-Order なプロセッサの場合，そのシミュレーションには極めて長い時間がかかるという問題がある．その根本的な原因は，対象アプリケーション・プログラムの命令数が桁違いに多いことであり，シミュレータを数倍程度高速化することは，重要なことではあるが，抜本的な解決にはならない．そこで，対象アプリケーション・プログラムのごく一部のみをシミュレートした結果から全体の性能を推定するサンプリング・シミュレーションが行われる．サンプリング・シミュレーションにおいて，実際にシミュレートする箇所をシミュレーション・ポイントと呼ぶ．

SimPoint をはじめとする既存のシミュレーション・ポイントの選出手法は，プログラムを事前エミュレーションして得られる PC の系列に基づく演繹的な手法である．それに対して本論文では，特徴的なアーキテクチャを持つ複数のモデルを事前シミュレーションして得られる IPC の系列に基づく帰納的な手法を提案し，評価した．

7.1 帰納的手法

提案手法は，複数の特徴的な基底モデルがすべて同じIPCを示す区間は同じフェーズであるという仮定に基づく．特徴的なマイクロアーキテクチャを持ついくつかの基底モデルの事前シミュレーションの結果得られるIPCベクトルの系列を得，それをクラスタリングすることで，帰納的にシミュレーション・ポイントを選択する．IPCベクトルは低次元の密なベクトルであり，そのクラスタリングは容易である．

我々はSPEC CPU 2006ベンチマークプログラムのシミュレーション・ポイントを選択し，最新の研究から高度なマイクロアーキテクチャを含むいくつかのターゲット・モデルのIPCを推定した．評価結果は，手法がSimPointより少ないシミュレーション・ポイントでより正確な推定を達成することを示している．具体的には，すべてのベンチマーク・プログラムの平均で, シミュレーション・ポイントの割合が約0.1%の場合，推定誤差は約1.6%から0.4%に改善した．

提案手法の改良としてキャッシュ容量固有のフェーズを持つ超多階層キャッシュを持つ単一のモデル を基底モデルに追加することで，帰納的手法をもっとより正確な推定を達成した．超多階層キャッシュはワーキング・セット・サイズは減少し，それが上位の階層のキャッシュに収まった段階で，IPCが階段状に向上するモデルである．

評価結果は，すべてのベンチマーク・プログラムの平均で, シミュレーション・ポイントの割合が約0.1%の場合，推定誤差は約0.4%から0.35%に改善した．この場合，キャッシュ容量固有のフェーズの追加するため，基本的な帰納手法よりシミュレーション・ポイントの割合が若干増える．

また，事前シミュレーションにかかる時間を削減するキャッシュ・ミス，分岐予測ミス，命令の依存関係対応する理想的なモデルをエミュレーション・ベースとする基底モデルを使ってフェーズ検出を行い，シミュレーション・ポイントを選出する場合，SimPointより正確な推定を達成した．

評価結果は，すべてのベンチマーク・プログラムの平均で, シミュレーション・ポイントの割合が約0.1%の場合，推定誤差は約1.6%から0.9%に改善した．これで，事前シミュレーションにかかる時間を無くことによって，基本的な帰納手法の短所が補完される．ところが，基本的な帰納手法の0.4%よりは悪くなる結果になった．これは，エミュレーション・ベースのうち，命令並列性が最大限であり，ターゲッ

トモデルの幅に比べてより多くのフェーズを分類して，余分なシミュレーション・ポイントを選んでいるからであると推定する．

基本的な帰納手法は，一つのプログラム内において，PC が異なっても同様に振る舞う部分をフェーズとして検出してシミュレーション・ポイントを選出した．逆に，異なるプログラムにもプロセッサが同様に振る舞う区間が存在する．そこで，ベンチマークに含まれるすべてのプログラムの実行を一つのワークロードとみなして，すべてのプログラムからすべてのプログラムに対するシミュレーション・ポイントを選出する集合的な帰納的選出手法を提案した．

評価結果は，誤差の悪化を 0.5% おさえつつ，全体の 61.93% 抽出率の削減が達成した．ところが，エミュレーション・ベースとする帰納手法の場合，同じく全体で 59.68% 抽出率の削減が達成したが，平均誤差が 2.7% である結果となった．

7.2 今後の課題

今後検討すべき課題としては，マルチスレッド・プロセッサや GPU・プロセッサへの適用が考えられる．

提案手法は，SimPoint の問題点を解決しながら，SimPoint より優れる性能を確認したので，SimPoint を使ってプロファイリングしたところを提案手法が十分にカーバできると思う．

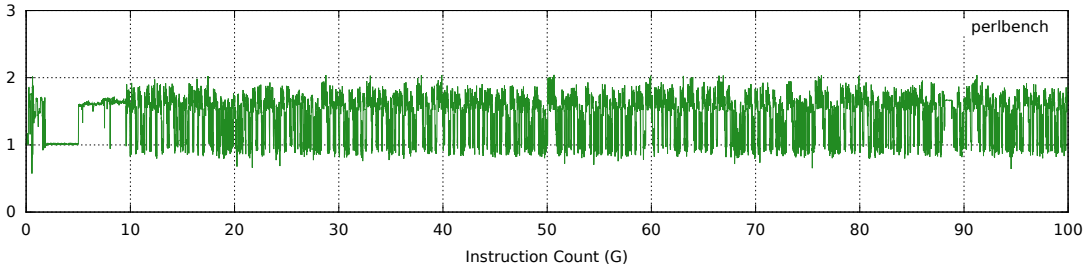
特徴的なマイクロアーキテクチャを持ついくつかのモデルによってプログラムのシミュレーションを行い，その結果から汎用的なシミュレーション・ポイントを選出することで，いわゆる広い範囲でのマイクロアーキテクチャの性能のフェーズ検出が可能になる．例えば，ＧＰＵの場合，多くの命令を持つ異常値スレッドブロックで引き起すスレッドレベルの並列性（TLP）を基底モデルに追加することで，マルチスレッドによるフェーズ検出にも適用することが考えられる．

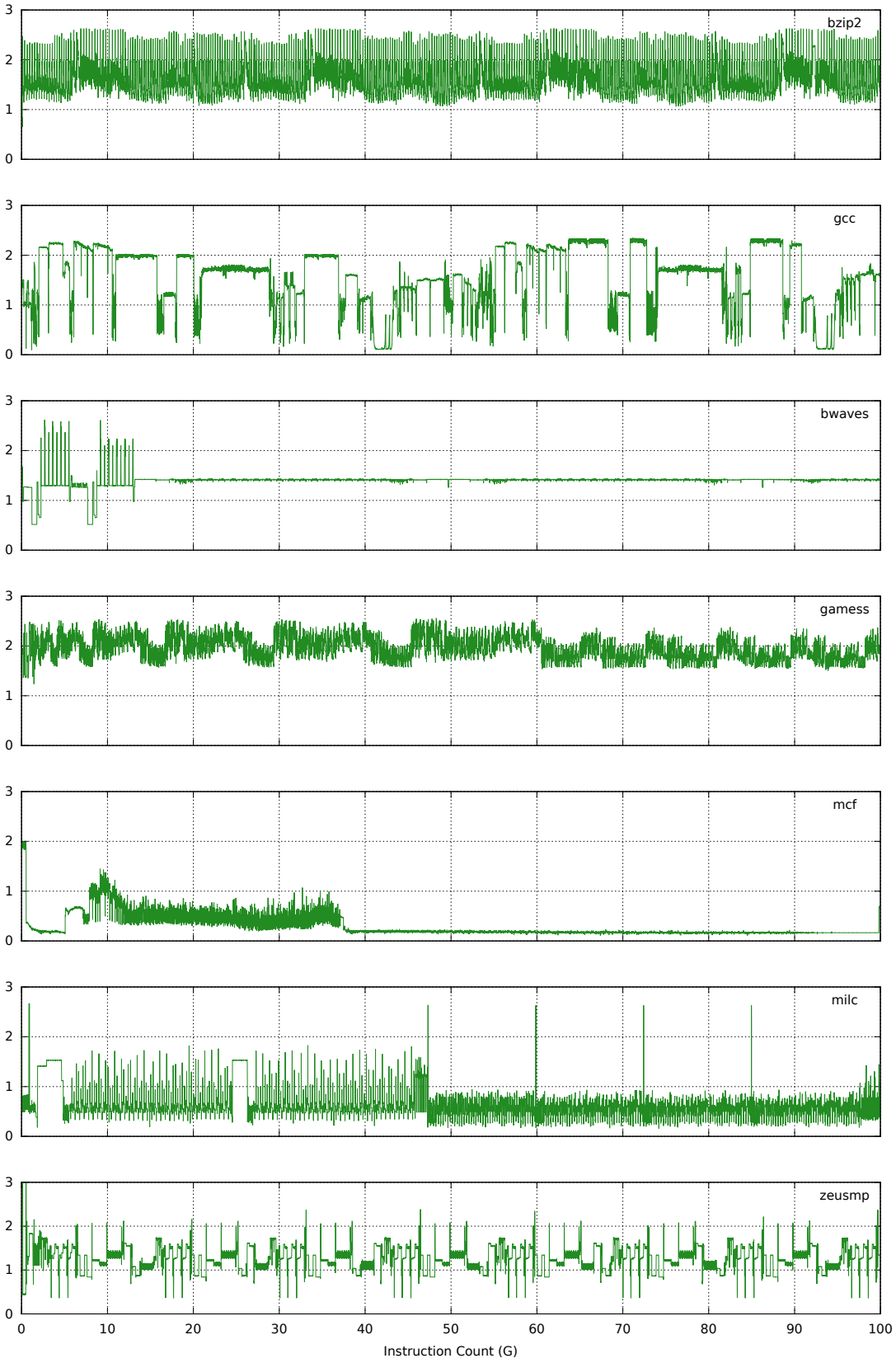
超多階層キャッシュはキャッシュ容量によって生じるフェーズに対応するための単一モデルであり，こういう特徴は，マルチスレッドでメモリ共有にによるフェーズ検出に役割を果たすと期待する．

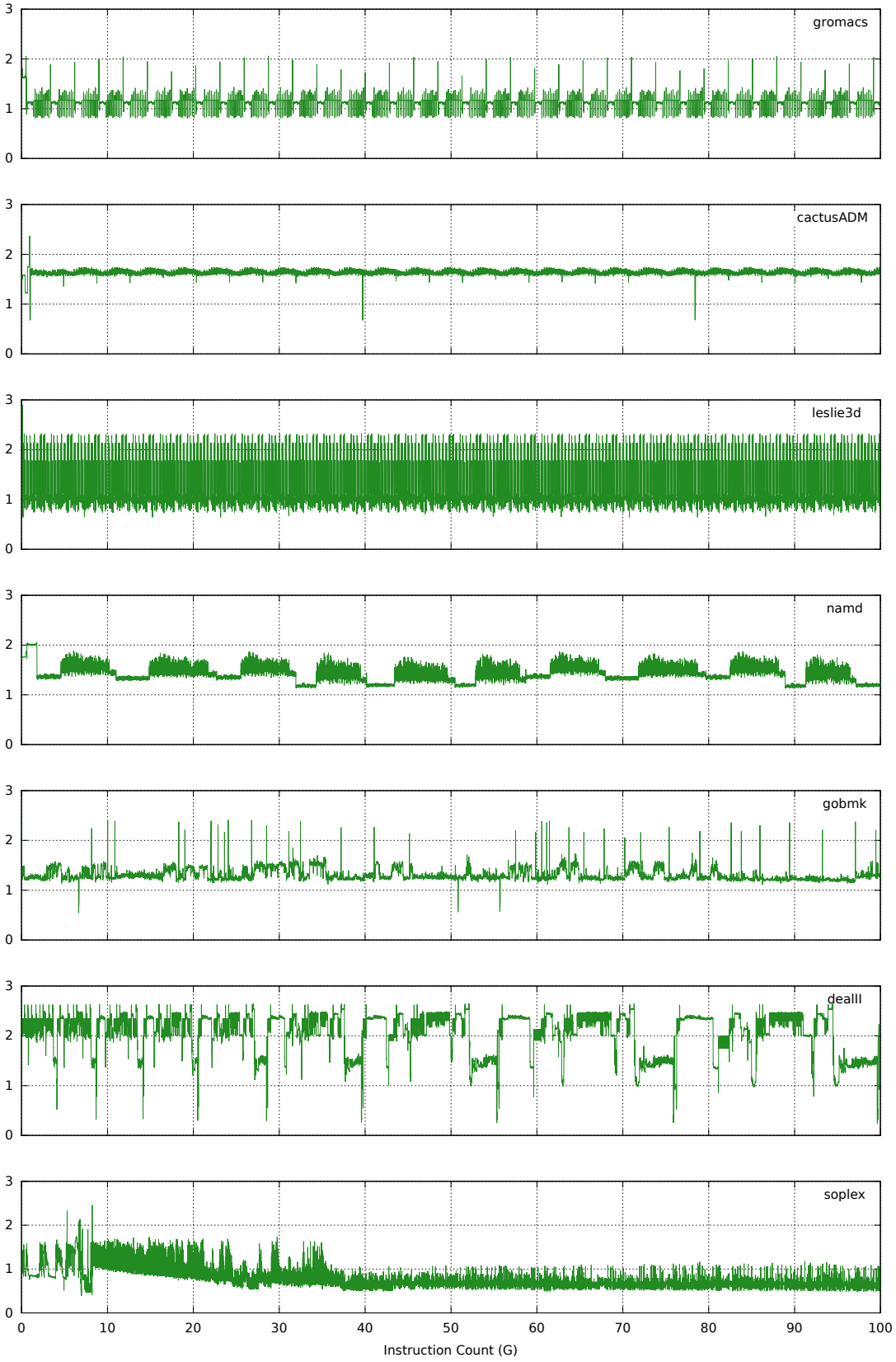
付 録A

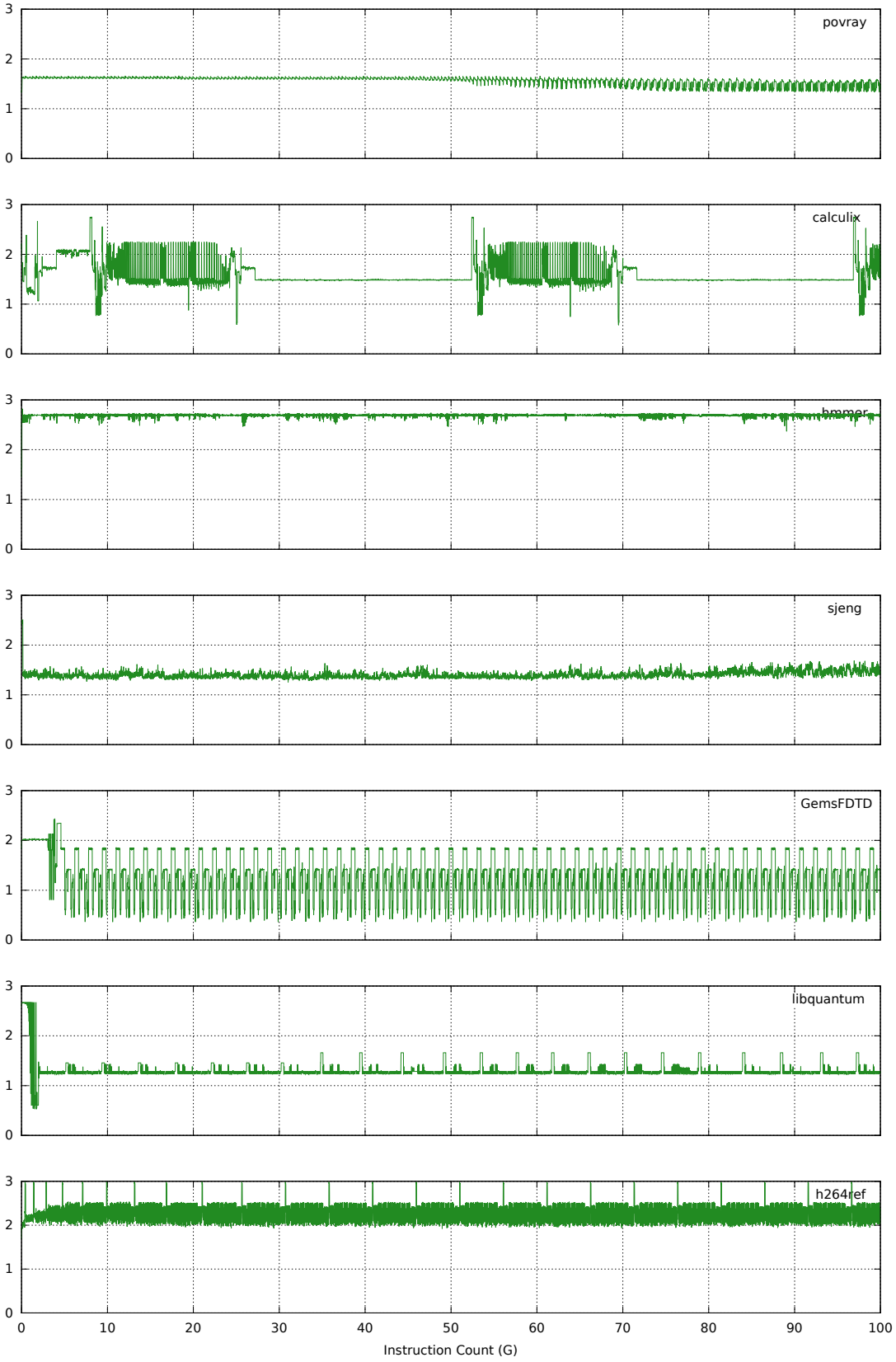
SPEC CPU 2006 [2]のIPCの遷移

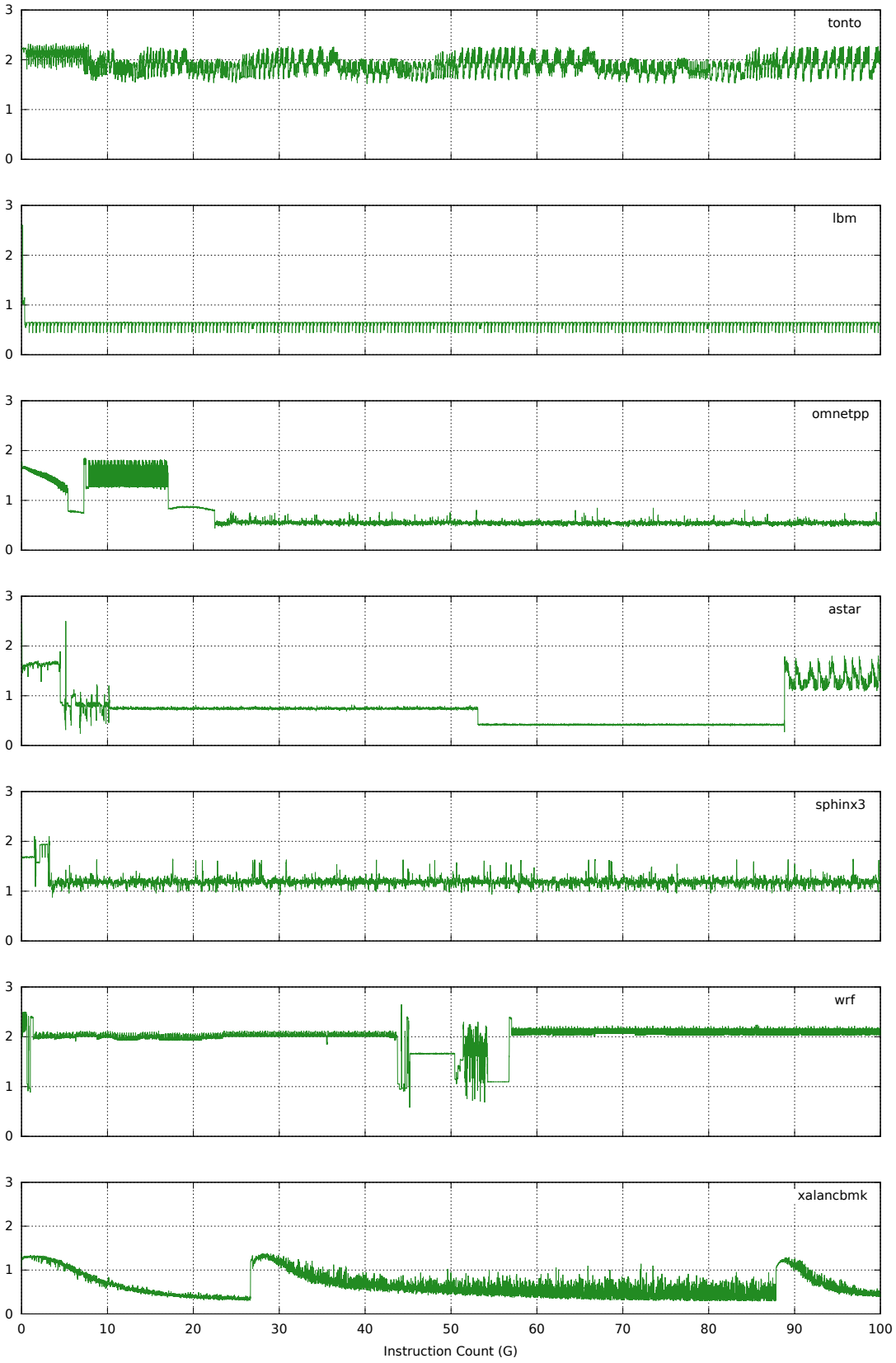
SPEC CPU 2006 の 29 本のプログラムの最初の 100G 命令の IPC の遷移を示す. モデルは sim である（第 5 章，表 5.2）．グラフ上の 1 点は，10M 命令のインターバルに対する区間 IPC を示す.




bzip2
gcc
bwaves
gamess
mcf
milc
zeusmp
0
1
2
3
0
10
20
30
40
50
60
70
80
90
100
Instruction Count (G)


gromacs
cactusADM
leslie3d
namd
gobmk
dealII
soplex
0
1
2
3
0
10
20
30
40
50
60
70
80
90
100
Instruction Count (G)


povray
calculix
hmmer
sjeng
GemsFDTD
libquantum
h264ref
0
1
2
3
0
10
20
30
40
50
60
70
80
90
100
Instruction Count (G)


tonto
lbm
omnetpp
astar
sphinx3
wrf
xalancbmk
0
1
2
3
0
10
20
30
40
50
60
70
80
90
100
Instruction Count (G)

謝辞

本研究を進めるにあたり，指導教員である坂井修一教授には，博士課程間の長きにわたり，御指導，御鞭撻を頂きました．ここに深く感謝の意を表します．

国立情報学研究所の五島正裕教授には研究の方針から細部に至るまで数多くの有益なご助言をいただき，また研究以外に関しても様々な機会に幅広くご相談させていただきました．本当にありがとうございました．

浅見徹教授，喜連川優教授，田浦健次朗教授，入江英嗣准教授には，審査において大変有益なご助言を頂きました．

当時研究室の学生であった山田淳二氏，神保潮氏，宮永瑞紀氏，福田隆氏には，本研究の基礎となるアイディアに関する議論を通じて，多くのご協力をいただきました．

名古屋大学の塩谷亮太准教授には，研究に関する数多くのご協力，ご助言のほか，研究室での生活において多くのご指導をいただきました．

秘書の八木原晴水氏，長谷部環氏，勝紀子氏，樫村純子氏には，研究室での生活や事務手続きに関する数多くのサポートをしていただきました．

その他，研究室に在籍中多くの皆様に，研究生活を通じて様々なご協力，ご支援を頂きました．

ここに深甚なる謝意を表します．

参考文献

[1] Onikiri 2. https://github.com/onikiri/onikiri2/.

[2] The Standard Performance Evaluation Corporation. SPEC CPU 2006. http://www.spec.org/cpu2006/.

[3] Arun A. Nair and Lizy K. John. Simulation points for SPEC CPU 2006. In *Proceedings of Internatinal Conference on Computer Design (ICCD)*, pp. 397–403, October 2008.

[4] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. SimFlex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31, No. 4, pp. 31–35, March 2004.

[5] Thomas F. Wenisch and Roland E. Wunderlich. SimFlex: Fast, accurate and flexible simulation of computer systems. In *Tutorial in the Internatinal Symposium on Microarchitecture (MICRO)*, November 2005.

[6] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsaf, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro, special issue on Computer Architecture Simulation*, Vol. 26, pp. 18 – 31, August 2006.

[7] SimFlex: Fast, accurate & flexible computer architecture simulation. http://parsa.epfl.ch/simflex/.

[8] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. Simulation sampling with live-points. In *Proceedings of Internatinal Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2006.

[9] Junji Yamada, Ushio Jimbo, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai. Design of a register cache system with an open source process design kit for 45nm technology. *IEICE Transactions on Electronics*, Vol. E100-D, No. 3, March 2017. (accepted).

[10] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S. Lee. Tbpoint: Reducing simulation time for large-scale gpgpu kernels. *Proceedings of Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 437–446, May 2014.

[11] Dam Sunwoo, William Wang, Mrinmoy Ghosh, Geoffrey Blake Chander Sudanthi, Christopher D. Emmons, and Nigel C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 113–122, September 2013.

[12] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. In *Proceedings of Internatinal Solid-State Circuits Conf. (ISSCC)*, 2002.

[13] M. Hind, V. Rajan, and P. F. Sweeney. Phase detection: A problem classication. Technical report, IBM Research, 2003.

[14] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *Proceedings of Internatinal Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2003.

[15] Greg Hamerly, Erez Perelman, and Brad Calder. How to use SimPoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31, No. 4, March 2004.

[16] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calde. SimPoint 3.0: Faster and more flexible program analysis. *Journal of Instruction-Level Parallelism (JILP)*, Vol. 7, No. 4, pp. 1–28, September 2005.

[17] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research (JMLR)*, Vol. 7, No. 2, pp. 343–378, February 2006.

[18] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *Proceedings of Internatinal Symposium on Code Generation and Optimization (CGO)*, pp. 135–146, 2006.

[19] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multithreaded applications. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 2–12, April 2013.

[20] G. H. Dunteman. Principal components analysis. *Sage Publications*, 1989.

[21] MinSeong Choi, Takashi Fukuda, Masahiro Goshima, and Shuichi Sakai. An inductive method to select simulation points. In *Proceedings of The 3rd. International Workshop on Computer Systems and Architectures (CSA)*, December 2015. (poster presentation with full paper review).

[22] MinSeong Choi, Takashi Fukuda, Masahiro Goshima, and Shuichi Sakai. An inductive method to select simulation points. *IEICE Transactions on Information and Systems*, Vol. E99-D, No. 12, December 2016.

[23] G. Hamerly and C. Elkan. Learning the k in k-means. Technical Report CS2002-0716, University of California, May 2002.

[24] Yuichi Akamatsu, Masahiro Goshima, and Shuichi Sakai. A phase detection technique not using fixed-length intervals. In *Proceedings of Symposium on Advanced Computing Systems and Infrastructures (SACSIS)*, May 2011. (in Japanese).

[25] Kaoru Hayakawa, Naruonore Kurata, and Shuichi Sakai. A phase detection technique with variable-length segments. *IPSJ SIG Report 2013-ARC-206 (SWoPP)*, No. 26, pp. 1 – 9, August 2013. (in Japanese).

[26] Ryota Shioya, Masahiro Goshima, and Shuichi Sakai. Register cache system not for latency reduction purpose. In *Proceedings of Internatinal Symposium on Microarchitecture (MICRO)*, December 2010.

[27] Ryota Shioya and Hideki Ando. Energy efficiency improvement of renamed trace cache through the reduction of dependent path length. In *Proceedings of 32nd IEEE Internatinal Conference on Computer Design (ICCD)*, pp. 416–423, October 2014.

[28] Ryota Shioya, Masahiro Goshima, and Hideki Ando. A front-end execution architecture for high energy efficiency. In *Proceedings of 47th Annual IEEE/ACM Internatinal Symposium on Microarchitecture (MICRO)*, pp. 419–431, December 2014.

[29] Kevin Krewell. Intel's Haswell cuts core power. *Microprocessor Report*, September 2012.

[30] B. Sinharoy, et al. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, Vol. 59, No. 1, pp. 2:1–2:21, 2015.

発表文献

雑誌論文

[1] MinSeong Choi, Takashi Fukuda, Masahiro Goshima, and Shuichi Sakai: An Inductive Method to Select Simulation Points, *IEICE Transactions on Information and Systems*, E99-D(12) pp. 2891-2990 (2016).

国際会議

[2] MinSeong Choi, Takashi Fukuda, Masahiro Goshima, and Shuichi Sakai: An Inductive Method to Select Simulation Points, *3rd. International Workshop on Computer Systems and Architectures (CSA)* pp.392-395 (2015). (poster presentation with full paper review).

査読付き国内会議

[3] 崔 珉誠, 福田 隆, 神保 潮, 五島 正裕, 坂井 修一: 帰納的なシミュレーション・ポイント選出手法の改良, *The 1st. cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming (xSIG)* (2017).

口頭発表

以下，Cui XiangHua は旧名.

[4] Cui XiangHua, Gawk GyeDal: Design of Low-Voltage Bandgap Reference with offset Compensation, 電子情報通信学術大会論文集 *(CEIC)*, 電子工学会, pp. 245–248 (2004). (in Korean).

[5] Choe DongGwon, Sin YunJae, Cui XiangHua, Gawk GyeDal: Design of charge pump with high pumping gain, *2004* 年度夏季総合学術大会論文集 *(2)(IEIE)*, 電子工学会, pp. 473–476 (2004). (in Korean).

特　許

[6] Cui XiangHua, Lee JeongWoo, Shin SangHoon: Repair Circuit and Repair Method of Semiconductor Apparatus, No:20110156034 (US). (2011).

[7] Lee JeongWoo, Lee HyungDong, Choi JunGi, Shin SangHoon, Cui XiangHua: Semiconductor Integrated Circuit, No:20110024743 (US). (2011).

[8] Choi YoungGyeong, Cui XiangHua: Circuit for Generating Pumping Voltage of Semiconductor Memory Apparatus, No:1020080094477 (KR). (2010).

[9] Choi HyangHwa (Cui XiangHua): High Voltage Pumping Device, No:07498866(US). (2009).

[10] Cui XiangHua: Apparatus Generating Release Enable Signal And Apparatus Generating Core Voltage Including the Same, No:1020080013687 (KR). (2009).

[11] Cui XiangHua, Choi YoungGyeong: Core Voltage Generation Circuit, No:1020080001588 (KR). (2009).

[12] Cui XiangHua: Circuit for Controlling Voltage of Semiconductor Memory Apparatus, No:1020070127591 (KR). (2009).

[13] Cui XiangHua: VBB Detector And VBB Generation Circuit Including the Same, No:1020070114117 (KR). (2009).

[14] Cui XiangHua: Internal Voltage Generating Circuit, No:1020070114117 (KR). (2009).

[15] Cui XiangHua: Oscillator Circuit and Method for Controlling the Same, No:1020070014583 (KR). (2008).

[16] Cui XiangHua: Internal Voltage Generator For Semiconductor Device, No:1020060096615 (KR). (2008).

[17] Cui XiangHua: Circuit for Controlling Internal Voltage in Semiconductor Memory Apparatus, No:1020050129743 (KR). (2007).

[18] Cui XiangHua: Apparatus for Controlling Voltage Generator of Semiconductor Memory, No:1020060053454 (KR). (2007).