修士論文

帰納的なシミュレーション・ポイント選出手法

Inductive Simulation Points Selection Technique

2015年2月5日提出

指導教員　坂井 修一 教授

東京大学大学院 情報理工学系研究科

電子情報学専攻

48-136433 福田 隆

概要

プロセッサのシミュレーションには極めて長い時間がかかるが，シミュレーション・ポイントを選出することでこれを短くできる．シミュレーション・ポイントとはプログラムの中で，その箇所を実行すればプログラム実行の全体の性能が推定できる箇所の事である．このような箇所の検出には，従来の手法ではプロセッサの命令セットアーキテクチャの情報のみを用いていた．本研究では特徴的なマイクロアーキテクチャを持ついくつかのプロセッサ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，他のあらゆるアーキテクチャに適用できるシミュレーション・ポイントを検出する手法を提案する．また，評価としてて SPEC2006 で ref 入力における先頭 100G 命令の IPC 推定を行った結果を示す．

目 次

図 目 次

第1章　はじめに

現代のプロセッサの研究・開発には，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，プロセッサの性能評価には極めて長い時間がかかってしまうという問題点がある．

長いシミュレーション時間　この問題の原因は以下の 2 つに分けられる．まず第 1 に，シミュレータの実行速度の遅さが挙げられる．実機による，いわゆる native 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．エミュレータの SD は 10 程度であるが，cycle-accurate なシミュレータの SD は 1,000〜10,000 程度にもなる．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．

第 2 に，評価に用いるベンチマークの命令数が非常に多いことがあげられる．例えば，プロセッサの評価に用いられる代表的なベンチマークである SPEC 2006 の場合，長いプログラムでは百 T 命令程度にもなる [13]．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．また，このような問題を抱えながら性能評価のためには，何十通りもパラメータを変えてシミュレーションを行う必要がある．

プロセッサのシミュレーション　ここで，プロセッサのシミュレーションについて詳しく述べる．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，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**と呼ぶ．プロセッサの定義は，**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．エミュレーションでは，ISA の情報のみを用いる．一方シミュレーションは，同時実行可能命令数，キャッシュや分岐予測器の構成など，マイクロアーキテクチャの情報をも用いることになる．プロセッサの性能評価では，当然，シミュレーションを行うことになる．

プロセッサのシミュレーションによる性能評価では，キャッシュ・ヒット率や分岐予測ヒット率などの個々の指標に加えて，総合的な速度指標である **IPC** (Instruction Per Cycle) を得る必要があることが多い．そのために，サイクルごとの振る舞いを正確に再現するシミュレータは，**cycle-accurate** なシミュレータと呼ばれる．

シミュレーションによる評価の現状　シミュレーションに長時間を要するという問題への対処法としてはまず実行するプログラム —— シミュレータの高速化は重要なことであるが，この文脈では根本的な解決にはならない．たとえ数倍程度の高速化が可能であったとしても，年単位が月単位に削減されるだけで，評価に用いるには依然として非現実的であるからである．そのため実際には，ベンチマークのごく一部のみをシミュレートする方法がとられる．典型的には，各ベンチマーク・プログラムのごく一部，例えば，プログラムの最初の 1G 命令をエミュレーションによってスキップし，その後の 100M 命令のみをシミュレートする．1G 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，その意図を蔑ろにすることになる．

このような方法は，一般に認められてはいるものの，実行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．

シミュレーション・ポイントとフェーズ検出　このような問題に対処するために，**シミュレーション・ポイント**を選定することが考えられる．シミュレーション・ポイントとは，そこだけを実行することによってプログラム全長にわたるプロセッサの振る舞いが推定できるようなプログラムの動的な一部のことである．例えば，前出の「1G 命令をスキップした後の 100M 命令」は（単純すぎる）シミュレーション・ポイントの一種と考えることができる．

よりよいシミュレーション・ポイントを選定するには，実行の**フェーズ** [8][9] の考え方を用いればよい．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．このような考え方に基づけば，プログラムの実行の全区間を，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・ポイントを選定する代表的な手法として **SimPoint** が挙げられる [4]．SimPoint をはじめとするほとんどの手法の特徴は，ISA の情報のみを用い，マイクロアーキテクチャの情報を用いないことにある．すなわち，エミュレーションによって得られたプログラム・カウンタの系列のみを基に，フェーズ検出（とシミュレーション・ポイント選定）を行うのである．

SimPoint は，プログラムの実行の全長を固定長の区間 —— **インターバル**に分割

図 1.1: 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

し，それぞれのインターバルに含まれるプログラム・カウンタ (正確には基本ブロック．2.1 章で詳述する．) の種類を基に，k-means 法によってクラスタリングを行う．[3] 各クラスタに属するインターバルが同じフェーズとみなされる．これは，以下の仮定に基づく；すなわち，プログラムの同じような（静的）部分を実行している（動的）区間は同じフェーズである．

しかしこの仮定には明らかな反例がある．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，CPI にも大きな影響を及ぼす．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 の予測精度を大きく悪化させている．

提案手法　しかし，既存の手法が ISA の情報のみを用い，マイクロアーキテクチャの情報を用いないことは，ある意味 当然である．特定のマイクロアーキテクチャの情報を用いてシミュレーション・ポイントを選定したとして，それを別のマイクロアーキテクチャの評価に用いられる保証がないからである．

そこで本研究では，特徴的なマイクロアーキテクチャを持ついくつかのプロセッサによってプログラムのシミュレーションを行うことで，その結果から汎用的なシミュレーション・ポイントを選定する手法を提案する．

サーキットのたとえ話　提案手法はサーキットのたとえ話を用いるとわかりやすい．ある長大なサーキットを走る車のタイムを，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．

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る．サーキットはスタートからゴールまでを R1 から R7 の区間で区切られており，それぞれの区間での速度が図 1.1 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、S 字と上りの場合には速度が下がっている．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．

ここで R1 と R6，R4 と R5 はどの移動手段を使っても速度に差がないので，似たような区間であると考えることができる．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 1.1 でオレンジ色の丸で印をつけた区間のみを実際走行することでサーキットのタイムが推定できる．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．この方法では複数の移動手段の走行結果をもとに「走行すべき最低限の区間」を求める手法であるいえる．

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．上の例で，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 R7 と R4，R5 が同じ性質の区間に分類されてしまい，車 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．

このようなサーキットのたとえ話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でベンチマークを全長実行した結果が得られれば，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．

SimPoint との相違点　SimPoint の PC に着目するフェーズ検出手法は，前節のサーキットのたとえ話をもちいると，区間の静的な性質からサーキットのフェーズを知る方法であると言える．すなわち，図の直線，くだり，S 字といった，区間の静的な性質から走行すべき最低限の区間を知る方法であるといえる．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，R1 と R2 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二度続くと速くなるように）その区間に意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，このような区間にさしかかったときの移動手段の状態が考慮されている．

本稿の構成　本稿の構成は以下のとおりである．まず 2 章ではフェーズ検出の代表的な手法である SimPoint やその関連研究について述べる．3 章で本稿の提案手法について述べる．4 章では今回の評価のための提案種の具体的な実装について説明を行い，5 章で評価を行う．最後に 6 章でまとめと今後の課題について述べる．

第2章　先行研究

本章ではまずフェーズ検出の最も代表的な手法である SimPoint[5] について説明する．SimPoint は PC の振る舞いに着目してフェーズを検出する手法である．SimPoint の問題を踏まえた上で，これに類する先行研究にも触れる．最後に，SimPoint に代表されるこれまでのフェーズ検出手法全般の問題点について触れる．

2.1　SimPoint

SimPoint の原理　SimPoint は以下の仮定に従ってフェーズ検出を行う；すなわち「プログラムの同じような（静的）部分を実行している（動的）区間は同じフェーズである」という仮定である．これに則り，SimPoint ではプログラムの動的命令列を一定の長さの区間に区切り，それぞれの区間がプログラムのどの部分を含んでいるのかを調べる．この,「プログラムの部分」の単位となるのが基本ブロックである．この動的命令列を固定長で分割した区間のことを **interval** と呼ぶ．SimPoint ではインターバルに含まれる基本ブロックの種類・頻度から特徴量（後述の基本ブロックベクトル）を定義し，これを比較することでインターバルをフェーズに分類する．

基本ブロック　基本ブロックとは分岐や合流を含まない命令のまとまりである．SimPoint は PC の振る舞いをもとにフェーズを検出する．一般に PC 上でフェーズの変わり目となるのは分岐や合流が発生する命令である．よって，分岐と合流のない命令についてはひとまとめに扱ってよいため，PC 列を基本ブロック (basic blok) 列にまとめてしまう．基本ブロック列は PC 列と同じ情報量を持っているがデータサイズは小さくなっており，扱いやすい．

基本ブロックベクトル　SimPoint では節で述べた，フェーズの性質を示す特徴量として**基本ブロックベクトル**を用いる．基本ブロックベクトルはインターバルごとに定義される．その要素数はプログラム全体に含まれる基本ブロックベクトルの種類数に等しい．従って基本ブロックベクトルは一般に高次元のベクトルとなる．基本ブロックベクトルの要素は，その区間における，それぞれの基本ブロックの出現回数が入る．例えば，あるプログラムがあり，このプログラムの全（静的）命令が 3 個の基本ブロックにまとまることがわかったとする．(これは非常に短いプログラ

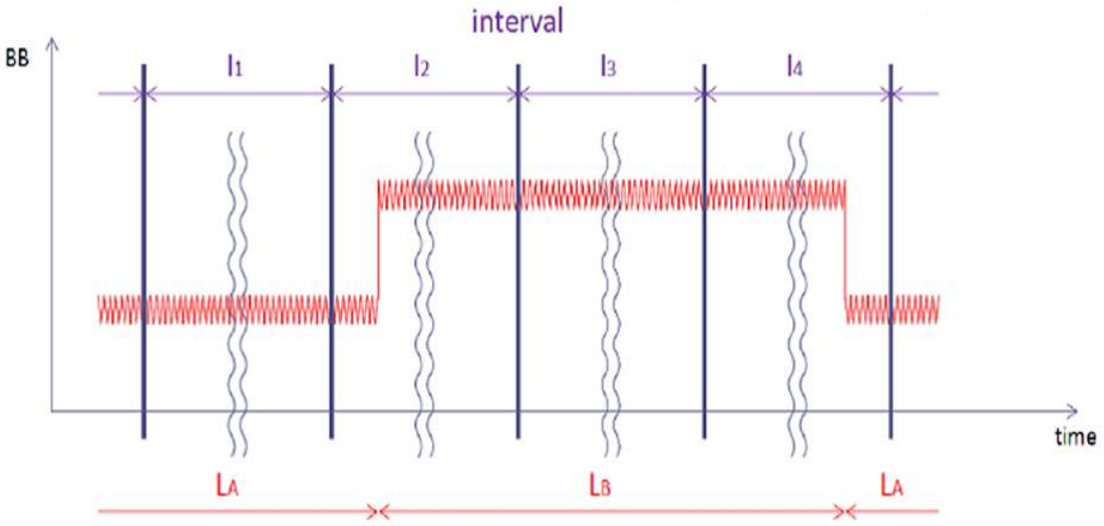

図 2.1: SimPoint の問題点

ムの例である.）この場合，インターバルに対して定義される基本ブロックベクトルは要素数が 3 となる．あるインターバルの基本ブロックベクトルが（0 3 1) ならば，このインターバルの 1 番目の基本ブロックを 0 個，2 番目の基本ブロックを 3 個，3 番目の基本ブロックを 1 個含むことになる．

フェーズの検出方法およびクラスタリング　まず，プログラムを，事前にエミュレータで実行して動的命令列を得る．動的命令列を固定長で分割してインターバルを得る．通常分割するサイズは 1M から 100M 程度である．SimPoint ではそれぞれのインターバルに対して基本ブロックベクトルを求める．こうして得られたベーシックブロックを k-means 法を用いてクラスタリングする．同一のクラスタに分類されたベーシックブロックベクトルを持つインターバル同士は同じフェーズであるといえる．得られた結果から，それぞれのクラスタで代表的なインターバルを取り出し，これをシミュレーション・ポイントとする．このシミュレーション・ポイントから先ほど述べたようにプログラム全体を実行したときの評価指標を推定することができる．

SimPoint の問題点　SimPoint ではインターバルの長さが一定である．このためフェーズの切れ目を含んだインターバルが生じてしまう [1]．図 2.1 では横軸は命令数で数えた時間，縦軸は基本ブロック ID をとっている．この図においてはフェーズ LA とフェーズ LB が存在する．ところがこれを固定長インターバルで分割した場合，図のようにフェーズの切れ目を含むインターバル，I2 と I4 ができてしまう．このようなインターバルの基本ブロックベクトルは，全く関係のないクラスタに分類され

てしまったり，またI2のみ，I4のみからなるクラスタを新しく生成される可能性がある．そしてその結果，性能指標の評価に誤差が生じてしまう恐れがある．これはSimPointのインターバルの長さが一定のために生じる問題である．

2.2 その他の先行研究

2.2.1 可変長セグメントを用いる手法

本研究室の先行研究として[1]がある．この研究は，先のSimPointの問題点に対処するために固定長インターバルではなく可変長セグメントに区間を分割する．

メディアンフィルタによるフェーズの顕在化 この手法では，まずプログラムのベーシックブロック列に対して1次元のメディアンフィルタをかける．メディアンフィルタはエッジを保ったまま平滑化ができるので，ベーシックブロック列の大きな変化だけが保存され，小刻みな動きは除去される．これによって，フェーズの顕在化を図っている．

可変長セグメント メディアンフィルタに通し，フェーズが顕在化したベーしクックブロック列で，閾値以上ベーシックブロックが離れているところをフェーズの切れ目とみなし，区切る．これをセグメントと呼んでいる．この可変長なセグメントはフェーズの切れ目を含んでいないため，精度よくクラスタリングを行うことが可能である．

2.2.2 Software Phase Marker

最後に上のレイヤーからのアプローチとしてSoftware Phase Marker[12](以下，SPM)を紹介する．SPMではフェーズの切れ目をソフトウェアのレイヤーから探すアプローチを取っている．

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を見て，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
- 二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．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，非常に細かな区間が多く発生してしまうベンチマークも見受けられ，そのようなベンチマークでは，SimPoint よりも悪い結果となっている．

2.2.3　SimFlex

ランダムサンプリング　SimFlex 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．ただし，SimPoint において [7] ら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．

ライブポイント　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去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を一定命令分前から行い，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　先行研究の問題点

上記で述べた手法はいずれれも，SimFlex を除いて，PC や call グラフ上での振る舞いが似ているものを同じフェーズとして検出する手法である．しかし，1 章でも述べたが，これらの手法の「プログラムの同じような（静的）部分を実行してい

る（動的）区間は同じフェーズである」という仮定には，反例がある．例えば同じコードで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，CPI に大きな影響を及ぼす．すなわち，プログラムの同じ部分を実行していてもプロセッサの状態によってプロセッサの性能は変化してしまう．SimPoint ではこのために，一部のベンチマークの予測精度が大きく悪化している．

第3章　提案手法

SimPoint はプログラムカウンタの振る舞いに着目したフェーズ検出の手法であった．それに対し，本提案手法ではあらかじめ，特徴的なアーキテクチャでプログラムをシミュレーションして得られた区間 IPC をもとにフェーズを検出する．

3.1　原理

n 種のアーキテクチャ $a_i\,(i = 1, 2, \ldots, n)$ で，2 つの区間 s，t をシミュレートした結果，$2n$ 個の IPC 値 $c_1(s), c_1(t), c_2(s), c_2(t), \ldots, c_n(s), c_n(t)$ を得，これらに対して $c_1(s) \simeq c_1(t)$，$c_2(s) \simeq c_2(t), \ldots, c_n(s) \simeq c_n(t)$ が成り立つとする．アーキテクチャ a_i が互いに十分異なっていれば，区間 s と t は同じフェーズで，未知のアーキテクチャ a_{n+1} に対しても $c_{n+1}(s) \simeq c_{n+1}(t)$ となることが期待できる．

図 3.1 は異なる 4 種類のアーキテクチャにおいて，400.perlbench の test 入力を全長実行した際の IPC の遷移を示している．この図において，たとえアーキテクチャが異なっても，同じプログラムを実行すればフェーズがアーキテクチャ間で保存されていることがわかる．この図からの類推で，推定対象の未知のアーキテクチャについても，これと非常に似たような IPC の遷移となることが期待される．

事前シミュレーション　提案手法では，あらかじめシミュレーションする a-kite

3.2　提案手法の工程

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. シミュレーション対象のプログラムの動的命令列を区間に分割
2. 複数の特徴的なアーキテクチャをもつプロセッサによる事前シミュレーション
3. 区間ごとの IPC ベクトルの生成およびクラスタリング

以下ではそれぞれ工程を順番に見ていく．

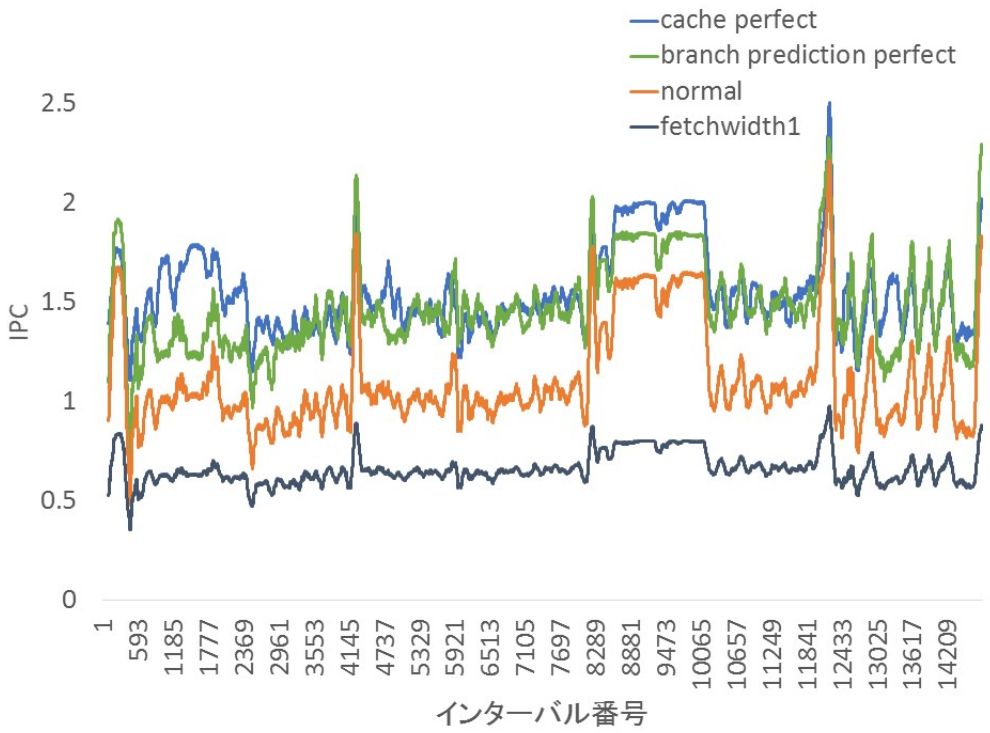

図 3.1: 400.perlbench の全長実行の様子

3.2.1 基準アーキテクチャ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アーキテクチャで事前にシミュレーションする．このようなアーキテクチャを**基準アーキテクチャ**と呼ぶこととする．1 章のサーキットのたとえ話からもわかる通り，基準アーキテクチャ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．

事前シミュレーションの際には，区間ごとの IPC を求める．区間の区切り方については，SimPoint のような固定長のものや [1] で述べた可変長のものを割り当てることも可能である．

3.3 IPC ベクトルの作成

次にそれぞれの区間でフェーズを検出する際の特徴量となる**IPC ベクトル**を作る．IPC ベクトルとは基準アーキテクチャのその区間での IPC を並べたベクトルである．この IPC ベクトルの要素数は実際にシミュレーションしたアーキテクチャの数である．また，要素はそれぞれのアーキテクチャのその区間における IPC である．例えば，アーキテクチャa_1，a_2，a_3 でプログラムを全実行したとする．ある区間に

おいて，IPC が a_1 は 1.1，a_2 が 1.2，a_3 が 0.9 だとすると，この区間における IPC ベクトルは（1.1 1.2 0.9）と表すことができる．このようにしてプログラムの全区間において IPC ベクトルを生成する．二つの区間の IPC ベクトルの距離が近いということは，その区間は事前シミュレーションしたどのアーキテクチャでも性能差が少ないということである．すなわち二つの区間は同じフェーズであると考えることができる．

3.3.1 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う．

ある IPC ベクトル，V に対して

1. V と全てのクラスタの中心との距離を比較
2. 距離が閾値以内のクラスタがあれば，V を最も距離の短いクラスタに分類し，中心を再計算
3. 閾値以内のクラスタがなければ新しいクラスタを作成
4. 次のセグメントに対して，(1) から (3) を繰り返す

このクラスタリング方法では，閾値の大小によってクラスタ数が決まる．また，クラスタの数が大きくてしたがって，k-means 法のように何回も実行して最適なクラスタ数を求める必要はない．

第4章　提案手法の実装

本章では，今回の評価のために行った提案手法の実装について述べる．4.1 節で基準アーキテクチャの検討を行い，4.3 節でクラスタリングの高速化について説明する．

4.1　基準アーキテクチャの検討

IPC 変動の 3 要因と基準アーキテクチャ　1 章の車のたとえ話からもわかる通り，基準アーキテクチャは互いに十分異なっているのが望ましい．ここで，どのような性質の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 変動の主な要因はキャッシュミス，分岐予測ミス，命令の依存関係である．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．図 4.1 は 400.perlbench の test 入力における IPC の遷移を示している．この図では青色が，一般的な 4-way のスーパスカラプロセッサ（評価で述べる基本構成のアーキテクチャに同じ) である．そして，橙色が青色のキャッシュヒット率を 100%にしたもの，灰色が橙色からさらに分岐予測のヒット率を 100%にしたもの，黄色が灰色からさらにフェッチ幅を 1 にしたものとなっている．すなわち，青→橙→灰→黄の順番に，先の節で述べて，IPC 変動の要因を一つずつ取り除いたときの IPC の様子である．これを見れば，基本構成の青が激しく IPC が変動してフェーズが生じているのに対して，IPC 変動の 3 つの要因全てを取り除いた黄色はほぼ一直線であり．ほとんどフェーズが見られない．このことからも，プロセッサの性能のフェーズを生み出したしているのは先の 3 つの要因によるところが大きいと考えられる．そこで，今回は評価にあたり，これらの 3 要因について特徴的な振る舞いを示すアーキテクチャを基準アーキテクチャとして選定する．

今回の評価に際して，基準アーキテクチャは以下の 5 つとした．

- キャッシュパーフェクト：キャッシュのヒット率を 100%にしたもの
- 分岐予測器パーフェクト：分岐予測のヒット率を 100%にしたもの
- スカラプロセッサ
- 基本構成：一般的なスーパスカラプロセッサ

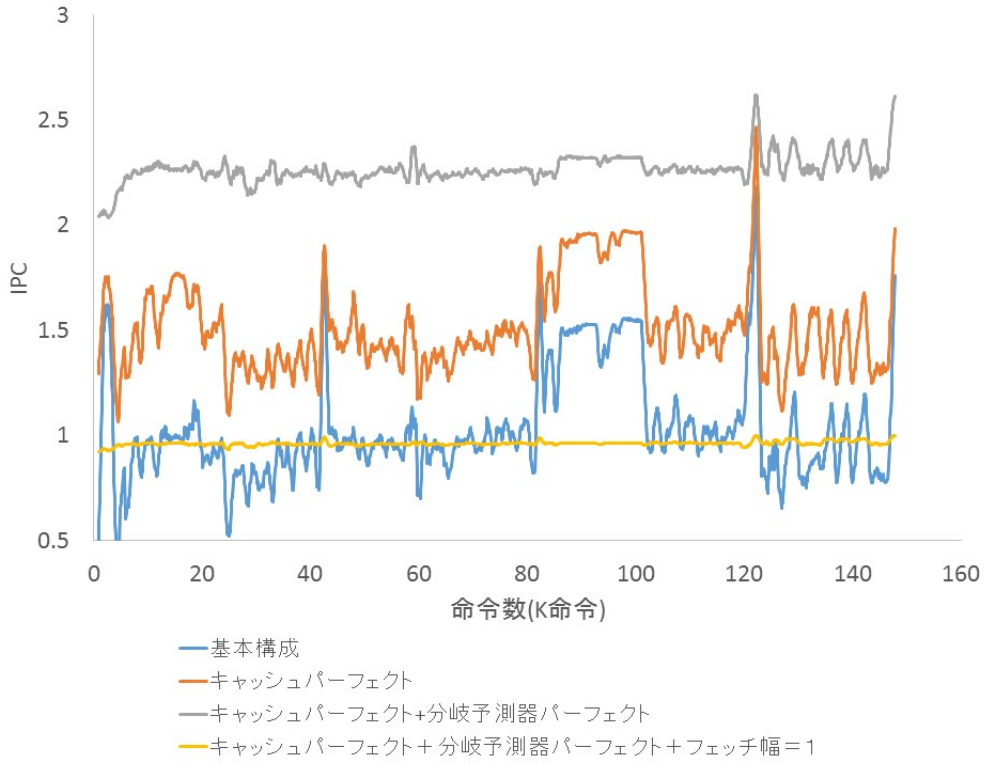

図 4.1: 400.perlbench の全長実行の様子

- 多階層キャッシュ：多階層キャッシュをもつアーキテクチャ

以降でそれぞれについて説明を行う．

4.1.1 パーフェクトなアーキテクチャと基本構成

最初の 3 つは，いずれも IPC 変動の要因のうち，どれか一つの影響を全く受けない，いわば ”理想的な ” アーキテクチャである．将来開発されるプロセッサの性能はこの理想的なアーキテクチャと基本構成の” 間” にあると考えられる．すなわち，これらの理想的なアーキテクチャは性能の上限であり，これ以上に特徴的な IPC の変動はないと考えられる．そこで，これら 3 つの理想的なアーキテクチャと，それ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を基準アーキテクチャに追加している．

4.1.2 多階層キャッシュを持つアーキテクチャ

キャッシュ容量のフェーズ　キャッシュは様々な容量を取ることが可能であり，容量固有のフェーズが生じる可能性がある．このようなフェーズは，上で述べた他の 4

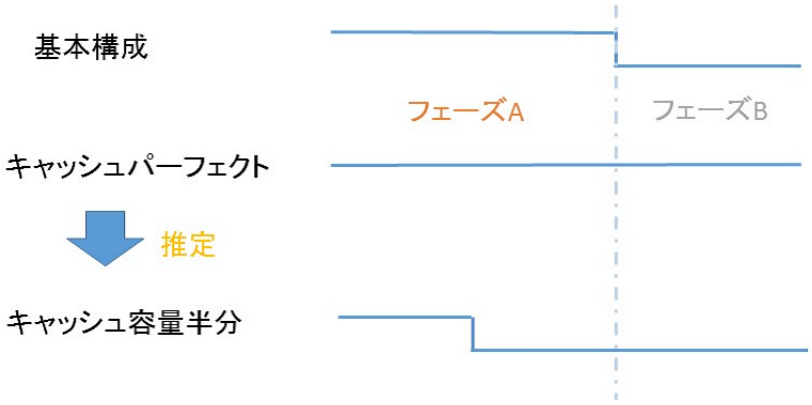

図 4.2: キャッシュのレイテンシが支配的なフェーズ

つのアーキテクチャでは補足できない可能性がある．図 4.2 は，ワーキングサイズが徐々に大きくなるような区間における IPC の様子の模式図である．基本構成のアーキテクチャでは，ワーキングサイズがキャッシュ容量からあふれると，IPC が下がる．一方キャッシュパーフェクトなアーキテクチャでは IPC の変化は見られない．この二つの IPC の振る舞いから，提案手法では，フェーズは図のＡとＢの二つに分類される．ところが，基本構成のキャッシュ容量を半分にしたアーキテクチャでは，IPC の低下が基本構成に比べ早めに生じる．このような状況では，ＡとＢの分類は適当とは言えず，これらのフェーズをもとにした性能の推定の精度も下がってしまう．同様に，キャッシュ容量を変えた様々なアーキテクチャで固有のフェーズが生じる可能性がある．

多階層キャッシュ　キャッシュ容量固有のフェーズに対応するために，基準アーキテクチャにキャッシュ容量を違えた無数のアーキテクチャを追加するのは現実的ではない．そこで解決策として，様々な容量のキャッシュからなる，多階層キャッシュを持ったアーキテクチャ（以下これを多階層キャッシュと呼ぶ）を基準アーキテクチャにを加える．図 4.3 は多階層キャッシュの先の図 4.2 の区間での IPC の振る舞いを示している．多階層キャッシュは，ワーキングサイズが徐々に大きくなる区間においては，図のように徐々に IPC が低下していく．よって，一定の性能差おきに別々のフェーズとして分類できるので，様々なキャッシュ容量に対応してフェーズがとれる．

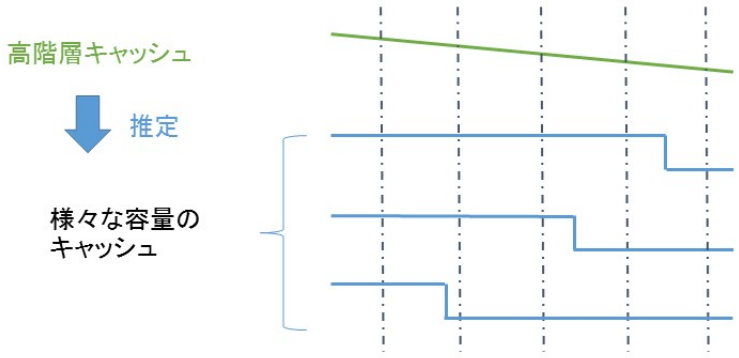

図 4.3: 多階層キャッシュ

4.2 区間の区切り方

提案手法も SimPoint 同様に，シミュレーションするプログラムを区間に切り分ける必要がある．今回は SimPoint 同様に固定長インターバルで区間でプログラムを区切ることとする．固定長で区切った場合，SimPoint の説明の際に触れた通り，フェーズの切れ目を含んだ区間生じ，クラスタリングの精度が下がってしまう可能性がある．しかし，提案手法でインターバルから生成される IPC ベクトルは，SimPoint のベーシックブロックベクトルに比べ次元数は非常に低い．このため，SimPoint に比べ，インターバルを短くして区間数が増えても，後に述べるクラスタリングの高速化を行えば，クラスタリングすることができる．すなわち，提案手法は，フェーズの切れ目の影響を無視できるほどに，細かくインターバルを取ることが可能である．

4.3 クラスタリングの高速化

3.3.1 節の 4 つの手順に忠実に実装したアルゴリズムでは，1 番目のすべてのクラスタの中心点との距離の計算がベクトル数の二乗のオーダーで増加する．この部分は，あるベクトルに最も近い中心点を持つクラスタを見つける，最近傍探索問題である．この部分の計算量を減らすため、今回は以下の高速化を行った．

IPC ベクトルの分布　図 4.4 は perlbench の test 入力において，基準アーキテクチャが 2 つの場合の IPC ベクトルの分布の様子を示している (以下の説明では簡単のため，基準アーキテクチャが 2 つ，すなわち，IPC ベクトルの要素数が 2 の場合で行う)．この図からわかるように，IPC ベクトルの分布は一直線上に乗っている．これは 3.1 節の図 3.1 で見た通り，アーキテクチャが異なっても似たような IPC の振る舞いに相関が発生するからである．また，図 4.5 の模式図のように，IPC ベクトルのクラスタの中心に関しても同じような傾向がある．

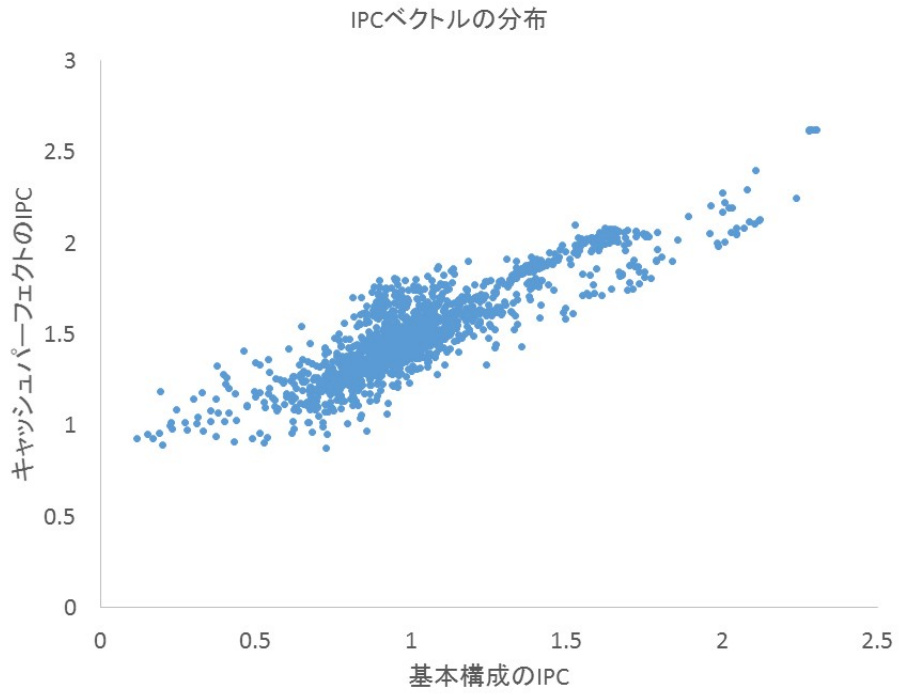

図 4.4: 400.perlbench IPC ベクトルの様子

高速化　あるベクトル (v_1,v_2) が，もっとも近いクラスタを探す場合を考える．もっとも近いクラスタの中心が (C_1,C_2) だった場合，IPC の振る舞いの相関から，二つの要素の和 v_1+v_2 は C_1+C_2 と非常に近い値となる可能性が高い．このような性質を利用し，高速に距離の近いクラスタを探す．あらかじめ，クラスタを，その中心の要素の和を計算し，その値の大小で，グループ分けしておく（図のようなｘ＋ｙ＝定数の二つの破線で囲まれた帯状の領域に含まれる点同士が同じグループ）．あるベクトル (v_1,v_2) が最も近いクラスタを探す場合，ベクトルの要素の和 v_1+v_2 を計算し，これに近いグループから順番に探していく．（図 4.5 の場合，まず d のグループを探し，次に近接する c および e のグループを探すことになる）ここで，a や g のような遠すぎるグループは，距離が閾値以内のクラスタが存在しないため，探索を省略できる．図 4.6 は，忠実に実装した場合と高速化を施した場合の 401.bzip2 の test 入力で得られた，IPC ベクトルのクラスタリングに要した時間である（閾値を変化させることにより，クラスタの数が変化している）．グラフより，クラスタリング時間の高速化が見られる．

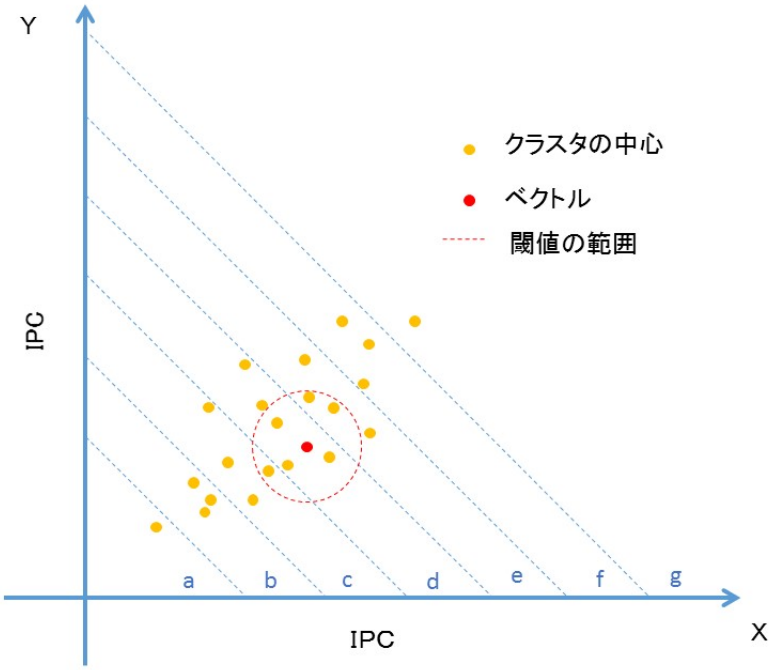

図 4.5: 最近傍探索の様子

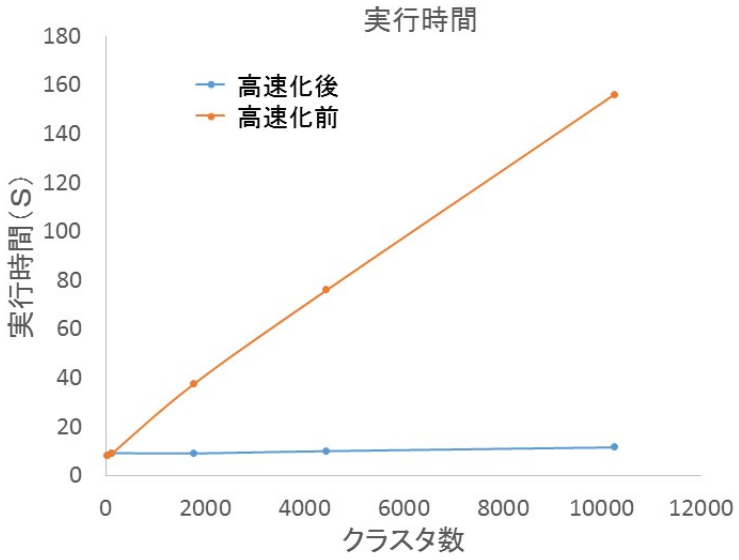

図 4.6: 実行時間の高速化

第5章　評価

本章ではまず，評価環境について説明を行う．次に，4.1 節で述べた理想的なアーキテクチャのみで基準アーキテクチャを構成した場合にについて，SPEC2006 の ref 入力で先頭から 100 G命令目までのシミュレーション・ポイントの検出およびそれを用いた推定を行った．さらに，基準アーキテクチャに，多階層キャッシュを含めて，SPEC2006 の test 入力の全長で検出および推定を行った．

5.1　評価環境

5.1.1　基準アーキテクチャ

今回の評価にはプロセッサ・シミュレータとして「鬼斬弐」を用いてシミュレーションを行った．. ベースとなるスーパスカラプロセッサの構成は以下の通りである。（以下これを「基本構成」と呼ぶ）

baseline ベースとなるスーパスカラのアーキテクチャである．表 5.1 に詳細の構成を示す.

cache perfect 基本構成のキャッシュのヒット率を 100%に変更したもの

bpred perfect 基本構成の分岐予測器の予測ヒット率を 100%に変更したもの

scalar スカラプロセッサ

poly cache 基本構成のキャッシュを高階層にしたもの．詳しいパラメータは巻末の付録に記載.

なお，鬼斬は，アウト・オブ・オーダーなスーパスカラプロセッサのシミュレーターなので，インオーダーなスカラプロセッサのシミュレーションには対応していない．スカラプロセッサに関しては，ごく初歩的なパイプラインプロセッサを想定し，ロード命令以外のすべての命令のレイテンシを 1 とし，分岐予測ミスの影響もないモデルを仮定した．この場合 IPC は主記憶及びキャッシュのアクセスレイテンシの合計から，計算される．そこで，インオーダーにキャッシュシミュレーションのみを行ってアクセス回数から IPC を算出した.

表 5.1: Configuration of Processor

ISA	Alpha21164A
pipeline stages	Fetch:3,Rename:2,Dispatch:2,Issue:4
fetch width	4 inst.
issue width	Int:2，FP:2，Mem:2
instruction window	Int:32,FP:16, Mem:16
branch predictor	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

5.1.2 推定対象アーキテクチャ

先節で述べた基準アーキテクチャの事前シミュレーションからシミュレーション・ポイントを得た．このシミュレーション・ポイントを用いて IPC 推定を行ったアーキテクチャは以下の 4 つである．

cache half 基本構成において L1 および L2 キャッシュの容量を半分にしたもの

pht singlebit 分岐予測器の PHT のカウンタビットを 1 にしたもの

fetch eight 8-way のスーパスカラプロセッサ．power8 相当の演算器構成にしたもの

norcs 本研究室で提案している非レイテンシ指向レジスタキャッシュシステム [10]. レジスタキャッシュ容量が十分である場合には性能低下がほとんどないので，今回は実験のため，敢えて 容量が極端に少ないモデルを採用している．

これらのアーキテクチャのうち，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のアーキテクチャである．最後の norcs は，これらの要因が絡み合った実際の研究で提案されたアーキテクチャに対して提案手法が適用可能か評価するためのものである．

参考までに，上記のアーキテクチャの baseline に対する性能は，SPEC2006 の ref 入力，先頭 100G 命令で，表に示すとおりである．

表 5.2: 推定対象のアーキテクチャの性能

アーキテクチャ	IPC(baseline 比)
cahce half	−7.6%
pht singlebit	−2.5%
fetch eight	11.1%
norcs	−9.2%

5.2 ref 入力先頭 100G 命令の推定

この評価では，SPEC2006 の 22 種類のベンチマーク（巻末の付録に掲載) の ref 入力先頭 100G 命令に対し，提案手法と SimPoint でシミュレーション・ポイントを検出した．得られたシミュレーション・ポイントを用いて推定した IPC と，実際の 100G 命令分のシミュレーションの結果得られた IPC を比較し，誤差を算出した．ここで，評価に用いたベンチマークプログラムは付録 A の通りである．

インターバルを変化させたときの様子　図 5.1 は，シミュレーションポイントの割合を横軸に，シミュレーション・ポイントを用いて行った IPC 推定の誤差率を縦軸に取ったグラフである．ここでは，原点に近いほど，少ないシミュレーションポイントでより正確な IPC の推定が行えていることを意味する．提案手法は実線で，SimPoint については点線で，表 5.3 と表 5.4 に従ってパラメータを変化させた時の様子を示している．このグラフは，合計 88 通り (ベンチマーク 22 種類×アーキテクチャ 4 種類）の組み合わせで推定を行った際の，幾何平均値をプロットしている．この図より，提案手法を示す実線が SimPoint を示す破線より，原点に近い位置にあり，提案手法の方が全体の平均で，優れたシミュレーション・ポイントを選択できていることがわかる．また，提案手法において，インターバルを変化させた場合，インターバル 10K を示す橙色が最も原点に近いところにあり，もっとも望ましいパラメータであることがわかる．

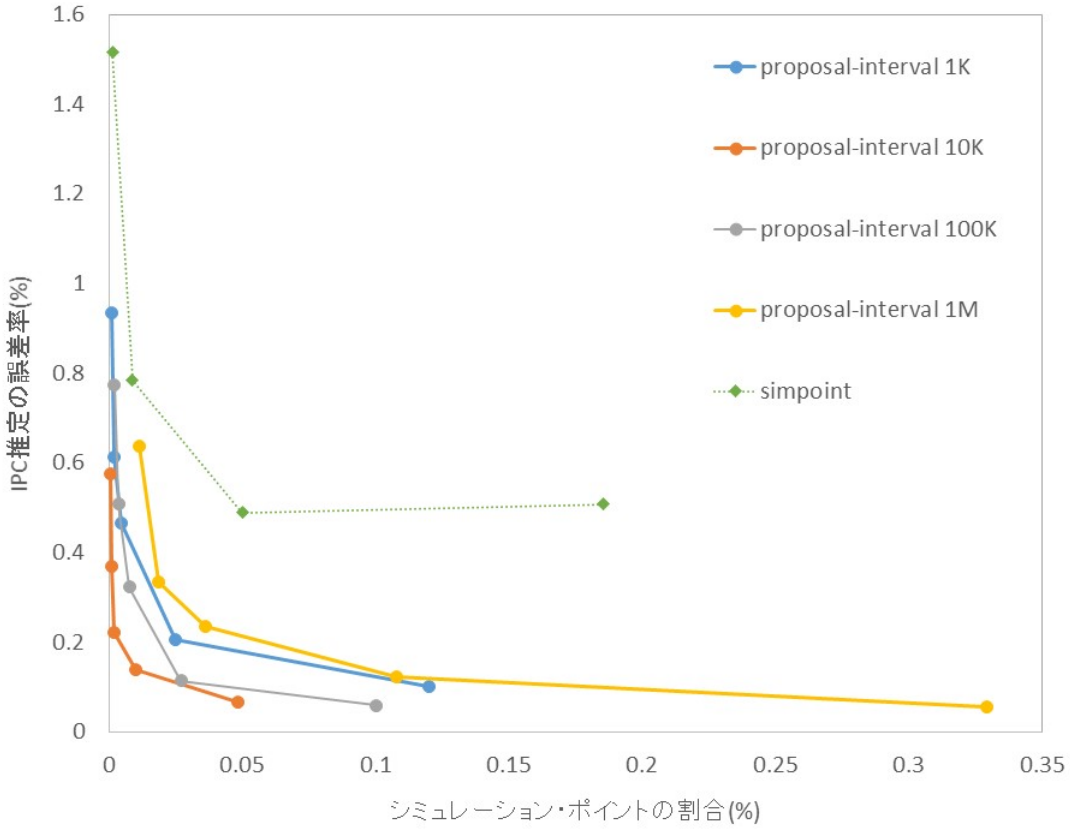

図 5.1: インターバルを変化させた際のシミュレーション・ポイントの割合と誤差率

表 5.3: 提案手法 評価パラメータ

基準アーキテクチャ	baseline cache-perfect bpred-perfect scalar
インターバル	10K 命令ごと
クラスタリングの SimPoint 閾値	0.05，0.1，0.2，0.3，0.4 の 5 通りで シミュレーション・ポイントを検出

表 5.4: SimPoint 評価パラメータ

インターバル	10K，100K，1M，10M の 4 通りで シミュレーション・ポイントを検出
K の最大値	(上のインターバルの順番に) 300，200，100，30

アーキテクチャごとの比較　次に，アーキテクチャごとの提案手法 (図の煩雑さ回避のため，10K 命令のみ) と SimPoint との比較を行う．図 5.2 にプロットさてているのは，4 つの評価対象アーキテクチャについて，22 種類のベンチマークの推定結果を幾何平均したものである．また，同じ色の線同士は同じアーキテクチャの推定結果を示している．
アーキテクチャごとの推定結果の平均には，提案手法も SimPoint も目立ったばらつきは見られない．

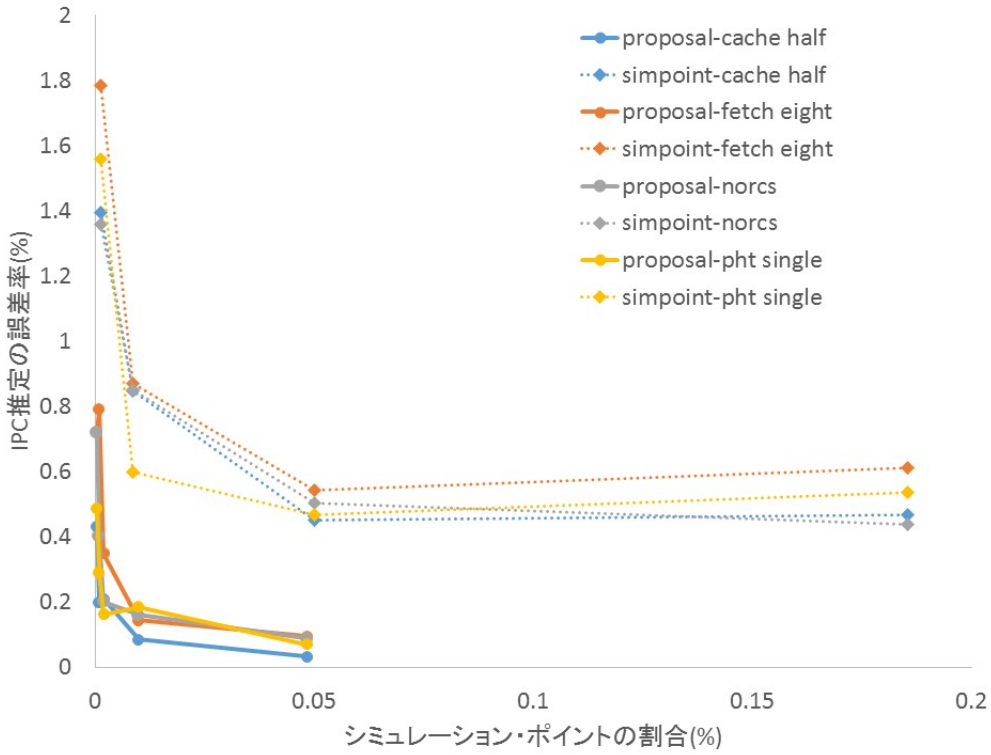

図 5.2: アーキテクチャごとのシミュレーションポイントの割合と誤差率の平均

5.2.1　パラメータを固定した場合

次に，個々のベンチマークの概観をを見ていく．ここでは，norcs について，パラメータは提案手法がインターバル 10K 命令，閾値 0.05，SimPoint がインターバル 10M 命令で固定した場合の結果を示す．図 5.3 は 22 種類のベンチマークについて，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，図 5.4 はその拡大図を示している．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．22 例中，16 例がシミュレーション・ポイントの割合と IPC 推定の誤差率の削減を同時に達成している．

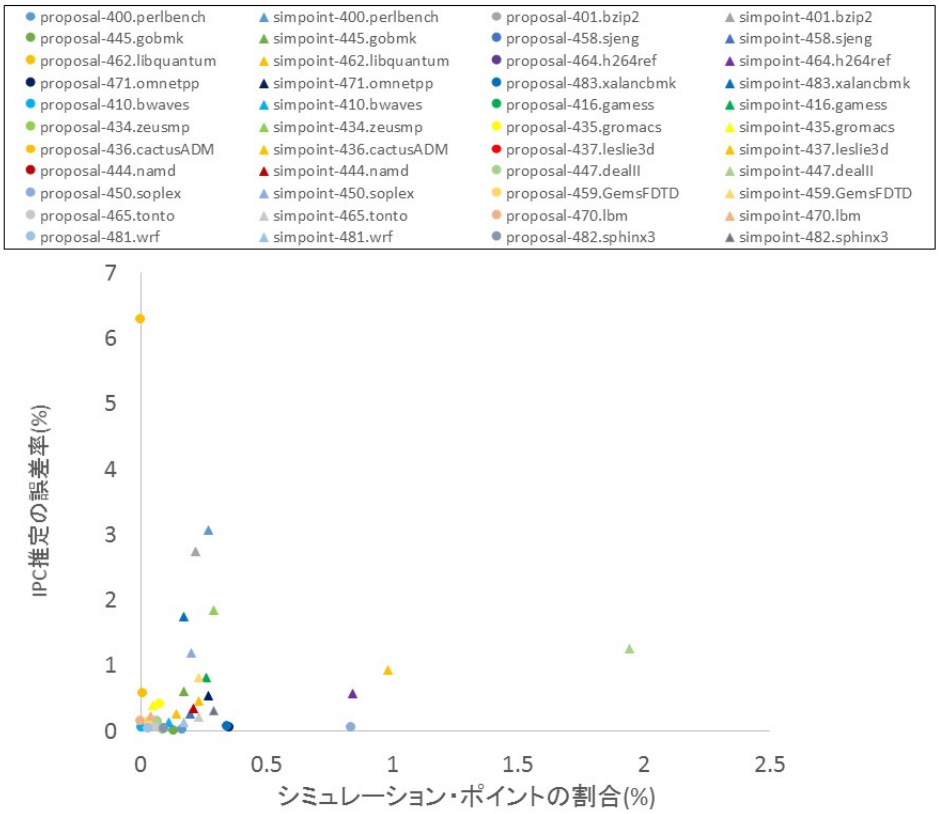

図 5.3: シミュレーションポイントの割合

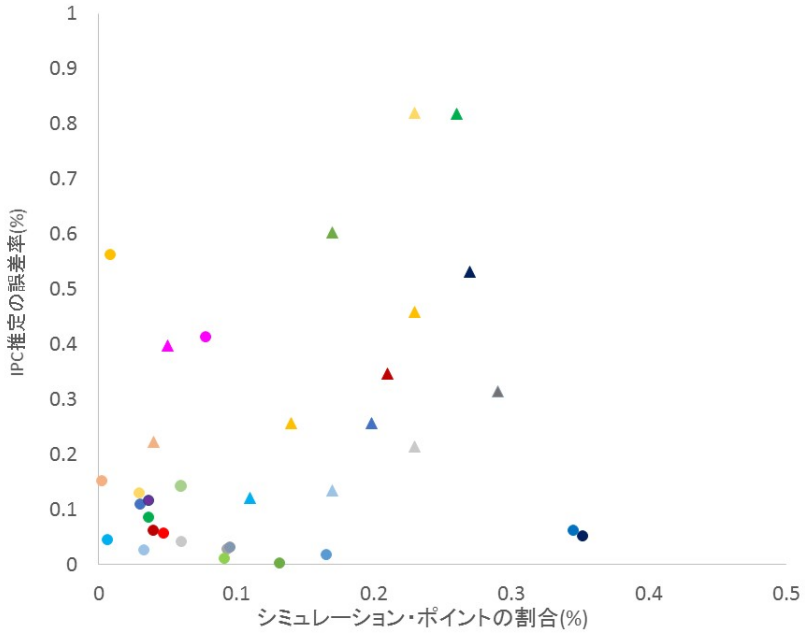

図 5.4: IPC 推定の誤差

5.2.2　ベンチマークごとの検討

結果の中から特徴的だったものについて，フェーズを可視化して掲載する．ただし，以下に掲載するフェーズの色分けグラフ描画点の関係上，提案手法，SimPointともにインターバル 10M 命令で取っている．

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良い結果が出たベンチマークの一つである．図 5.5 は 401.bzip2 のシミュレーション・ポイントの割合を横軸に，IPC 推定の誤差を縦軸にとったものである．これを見ると，どのアーキテクチャでも，SimPoint より，優れたシミュレーション・ポイントが選ばれていることがわかる．図 5.6 および図 5.7 は 401.bzip2 の同じフェーズに分類したインターバルの実際の norcs でのシミュレーションの IPC を示している．図は点一つがインターバル 1 つ分に相当し，そのインターバルの実際の IPC を縦軸に，そのインターバルが何命令目に位置しているかを横軸に取っている．図において同じ色どうしのインターバルは同じフェーズであることを示している．また，黒い点はシミュレーション・ポイントとして選ばれた点を示している． 図 5.6 は，SimPoint，図 5.7 は提案手法の，フェーズに分類されたインターバルの様子を示している．（すべてのフェーズをグラフに乗せると煩雑になるため，大きなクラスタを形成した上位 10 フェーズのみをプロットしている．また，上下のグラフで色の対応関係はないことに留意されたい.）．SimPoint のフェーズの分類では，黄色のフェーズや赤色のフェーズに分類された点の IPC に大きなばらつきがみられる．一方で提案手法は SimPoint に比べ，同じフェーズに分類された区間同士の IPC がそろっており，きれいな帯が認められる．これは，提案手法が SimPoint に比べ，IPC を推定する上で正確なフェーズの分類が行えていることを示している．

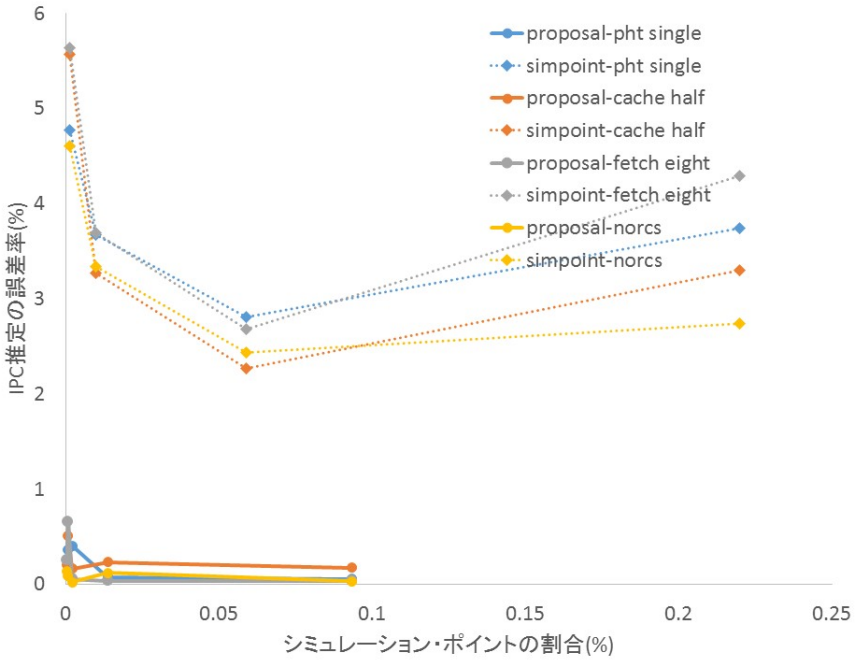

図 5.5: 401.bzip2 の推定結果

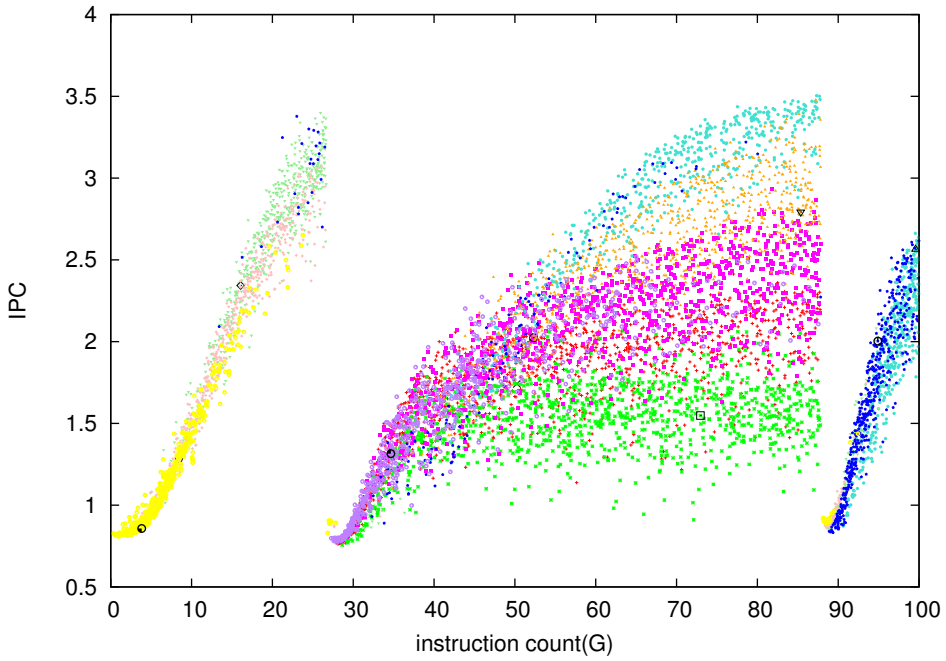

図 5.6: 401.bzip2 norcs の SimPoint によるクラスタリング

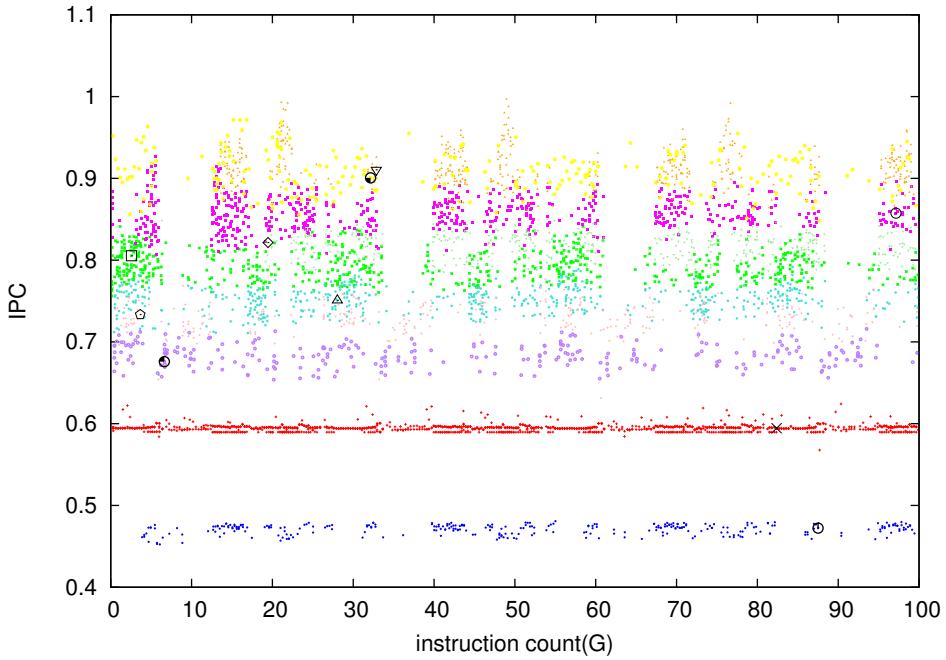

図 5.7: 401.bzip2 norcs の提案手法によるクラスタリング

483.xalancbmk　次に 483.xalancbmk について比較を行う．483.xalancbmk でも，提案手法は SimPoint に比べ優れた推定を行っている．図 5.8 と図 5.9 はそれぞれ，SimPoint と提案手法のフェーズの分類を，fetch eight に適用したものである．このグラフからもわかる通り，提案手法のフェーズの分類の方が IPC の分布がまとまっており，先に掲載した norcs の推定結果にもある通り，提案手法のほうが優れたシミュレーションポイントを選択している．ところが，同じフェーズの分類を，cache-half に適用した場合，IPC の分布が急激に広がってしまっている (図 5.10)．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．実際に，図 5.11 より，IPC 推定の際の誤差は，norcs に比べて誤差が大きくなっていることがわかる（ただし，SimPoint を示す破線は一部フレームアウトしているが，提案手法に比べると依然悪い結果になってる)．

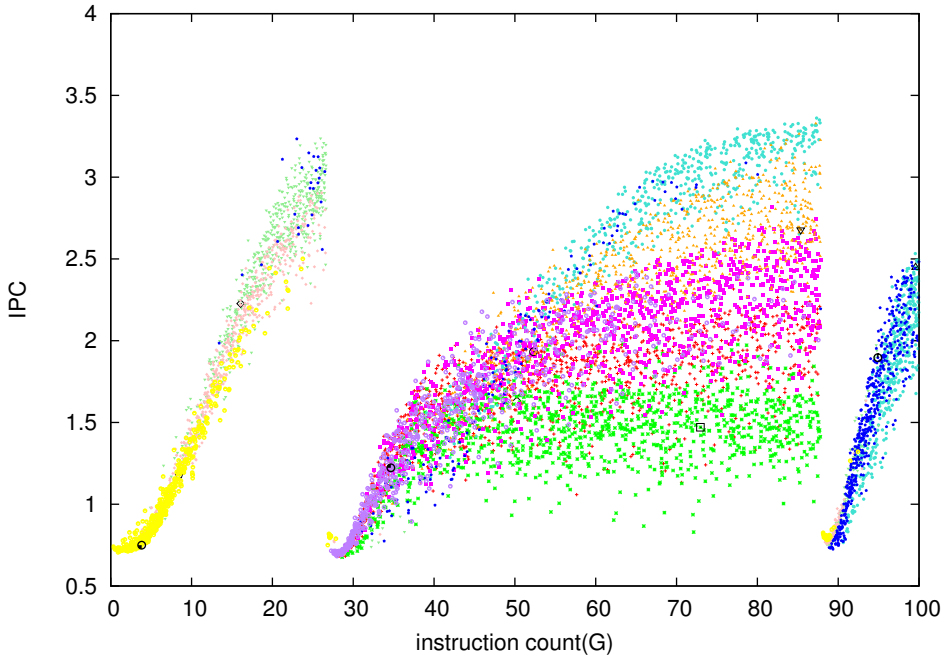

図 5.8: 483.xalancbmk fetch-eight の SimPoint によるフェーズの色分け

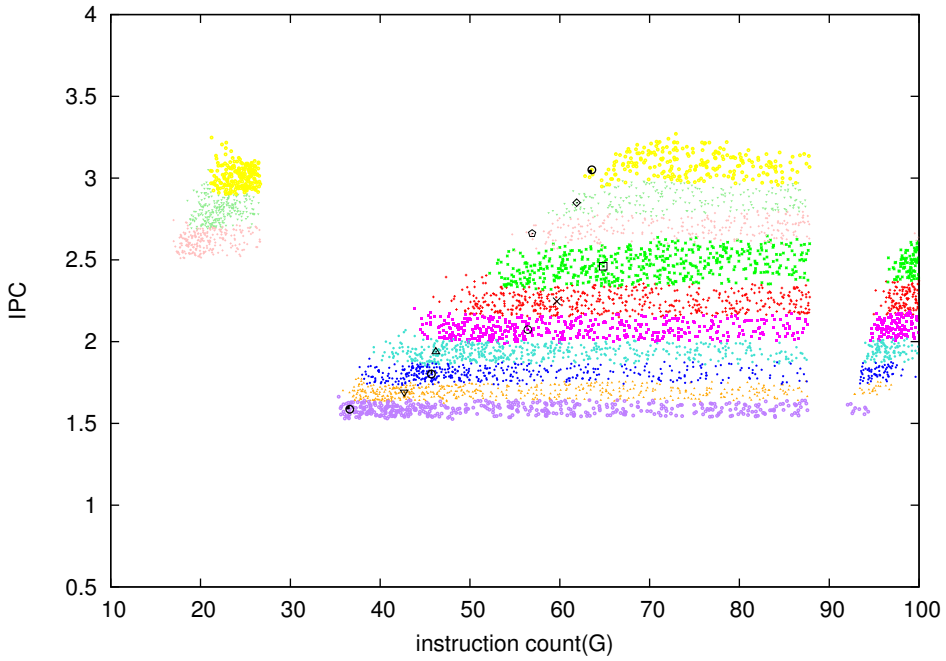

図 5.9: 483.xalancbmk fetch-eight の提案手法によるフェーズの色分け

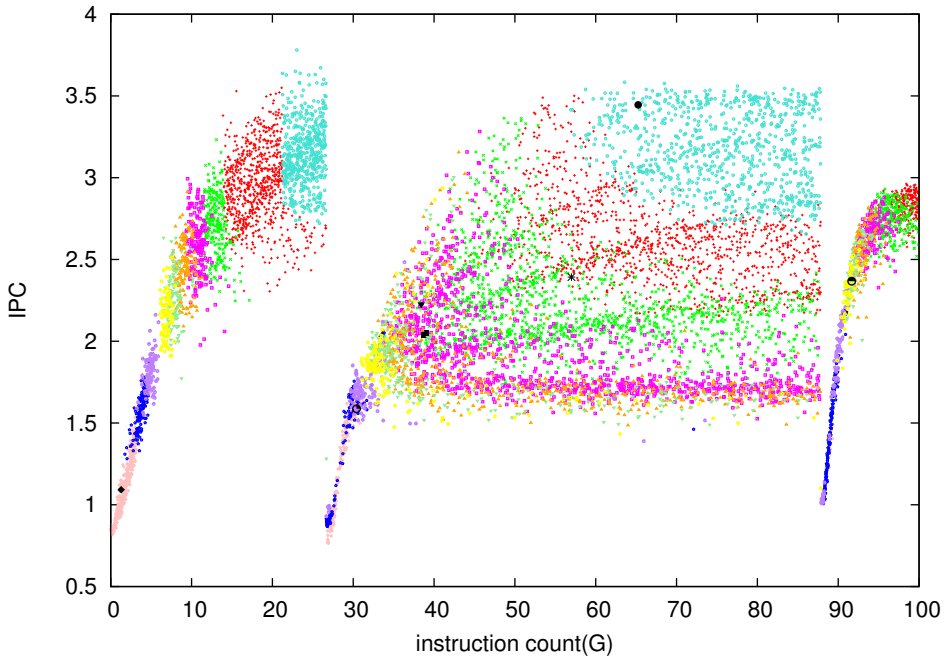

図 5.10: 483.xalancbmk cache-half の提案手法によるクラスタリング

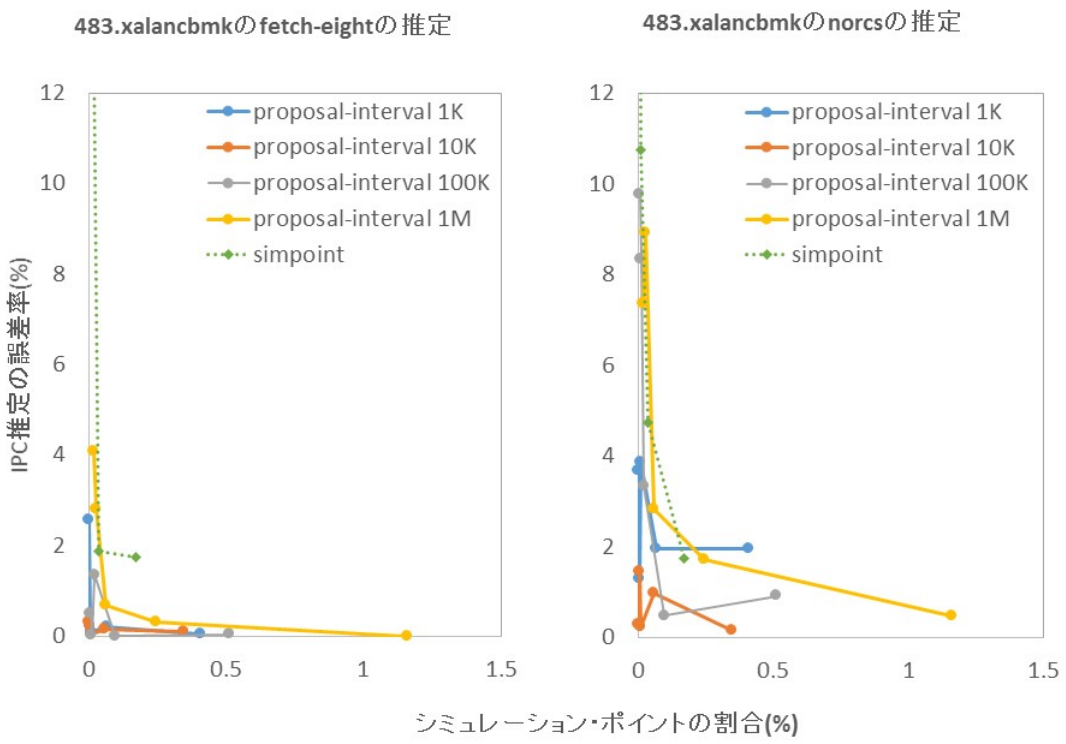

図 5.11: 左 norcs 右 cache half 483.xalancbmk の推定結果

その他のベンチマーク　本評価では 22 種類のベンチマーク × 4 種類のアーキテクチャの合計 88 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 SimPoin t 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 4 例である．その一つである，libquantum の norcs の推定では，提案手法において閾値を 0.05〜0.4 で変化させても得られるシミュレーション・ポイントの割合が 0.0004 パーセント程度と，極めて少ないままで，誤差率も 5 パーセント程度と非常に大きかった．しかし，閾値を 0.001 まで下げると，SimPoint に対してシミュレーション・ポイントと IPC 推定の誤差率を同時に削減することができた．この他に 471.omnetpp の cache half，444.namd の cache half，465.tonto の fetch eight の 3 例は，SimPoint の方が提案手法より良い結果を示したパラメータがあった．これらについては原因を現在調査中である．

5.3　多階層キャッシュの評価

この評価では 5.2.2 節で述べた，483.xalancbmk のキャッシュ容量によるフェーズの変化の結果を受けて，基準アーキテクチャに多階層キャッシュを入れて，評価を行った．なお，評価は SPEC2006 の 9 種類のベンチマーク (巻末の付録に一覧を掲載) を 4 種類のアーキテクチャで test 入力全長実行により行った．

5.3.1　多階層キャッシュの効果

多階層キャッシュを基準アーキテクチャに追加したときの効果をフェーズの色分けグラフで確認する．図は 483.xalancbmk の test 入力を cache-half で全長実行した際の，多階層キャッシュを基準アーキテクチャに含まない場合（図 5.12）と含む場合（図 5.13）の上位 10 フェーズの色分けである．図 5.12 では，桃色のフェーズや緑色のフェーズ，淡桃色のフェーズに分類された点の IPC が上下に散らばっている．一方で，多階層キャッシュを基準アーキテクチャに含めたものでは，このような上下に伸びたフェーズの分布が抑えられている．これは，今まで同じフェーズだと誤判定されていたフェーズが細かく別々のフェーズに分割された，上位 10 フェーズに入ってこなかったためである．すなわち，高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している．
図 5.14 は，基準アーキテクチャに多階層キャッシュを含めた場合と含めなかった場合の推定結果の比較である．多階層キャッシュを含めることにより，やや結果が改善されていることがわかる．

その他のベンチマーク　ところが，図 5.15 において，9 種類のベンチマークの推定結果の平均を 4 つのアーキテクチャで見てみると，すべての場合について．多階層キャッシュがないものの方がよい結果になっている．この原因現在調査中であるが，多階層キャッシュありの場合となしの場合で，同じ命令数分のシミュレーションポイントを選択した時に，前者は後者に比べてキャッシュのフェーズが” 補強” される分，シミュレーション・ポイントに漏れるフェーズがある可能性がある．

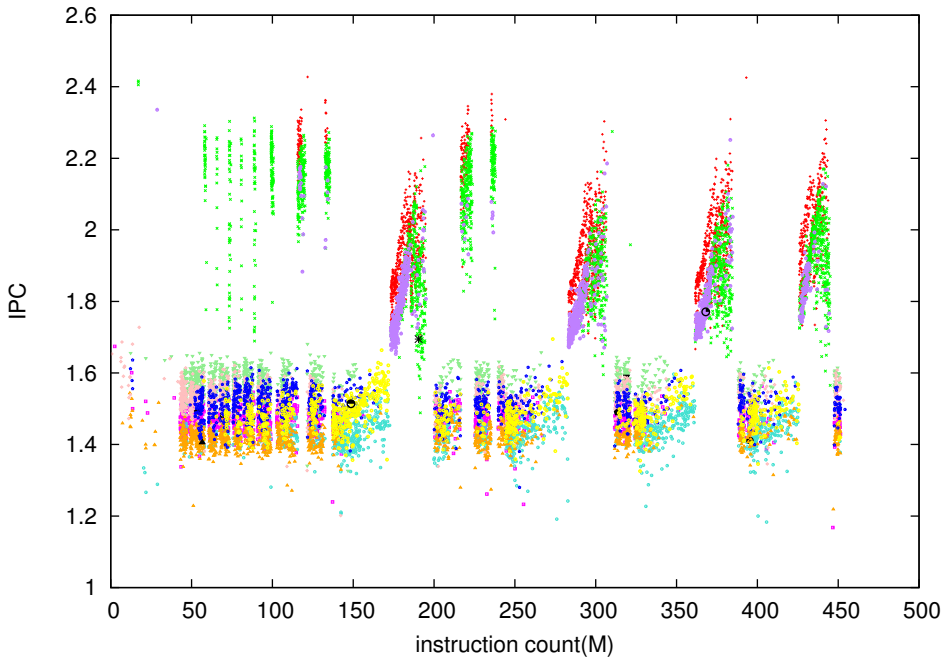

図 5.12: 基準アーキテクチャに多階層キャッシュを含まない場合のフェーズ

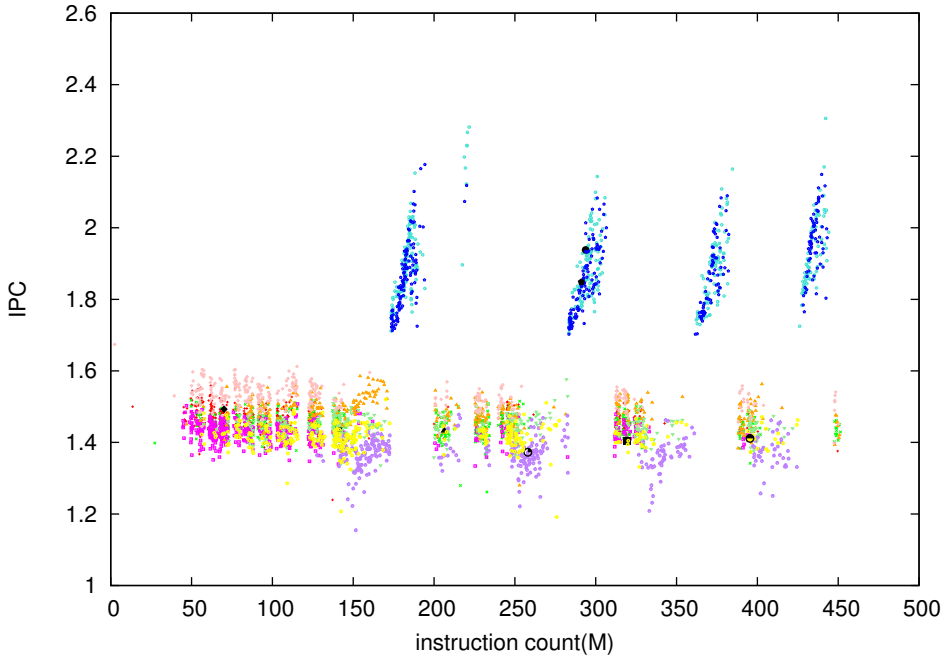

図 5.13: 基準アーキテクチャに多階層キャッシュを含んだ場合のフェーズ

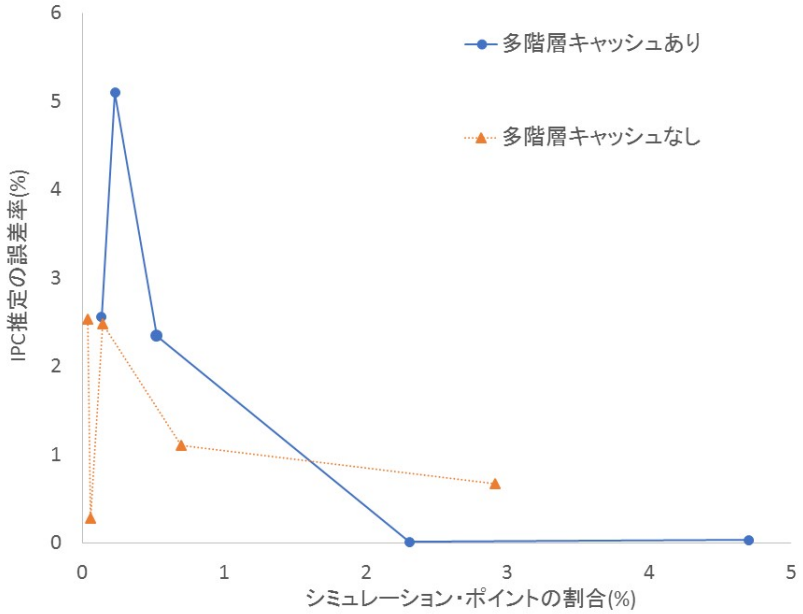

図 5.14: test 入力 483.xalancbmk の cache half の推定結果

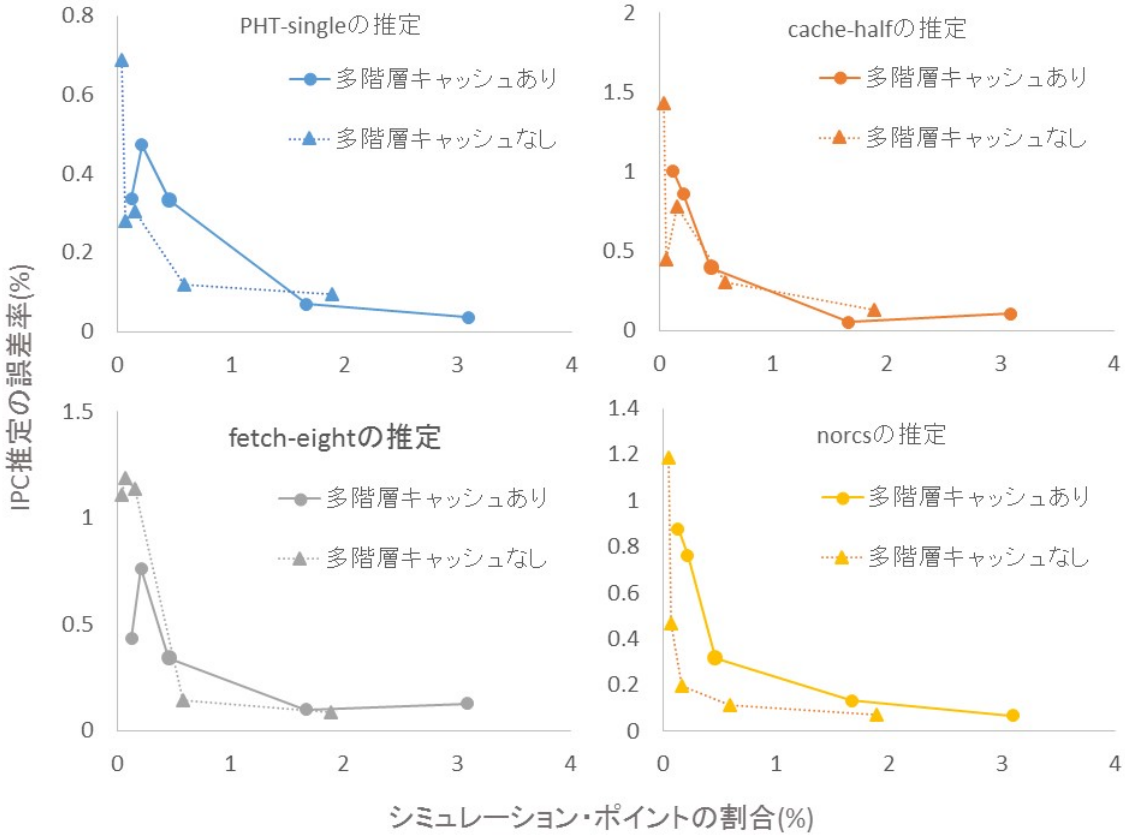

図 5.15: 10 種類のベンチマークの平均の比較

第6章　おわりに

6.1　今後の課題

多階層キャッシュの改良　今回，様々なキャッシュ容量のフェーズを反映させるために基準アーキテクチャに多階層キャッシュを追加したものについて評価を行ったが，追加前と比べ一部のベンチマークでしか効果が確認できなかった．多階層キャッシュについては，容量の分布やレイテンシの設定の自由度が高く，適切なパラメータを設定できていない可能性があり，再検討が望まれる．

命令の依存関係のフェーズ　図は 445.gobmk の test 入力を fetch-eight でシミュレーションした際のフェーズの色分けである．ここで，緑色の点は上下に幅のある分布になっている．これは，命令の依存関係のフェーズに関して，本来異なるフェーズに分類されるべき点同士が同じ一つのフェーズにまとめられていることを示している．このようなフェーズが生じる原因として，スーパスカラの way 数を増やすことにより，さらに並列性が引き出せる部分と，もうこれ以上並列性が引き出せない部分が同じフェーズとして緑色にまとめられている可能性がある．このようなフェーズを正確に分類するにはスーパスカラの way 数を大きくしたアーキテクチャの追加について今後検討していく必要がある．ただ，シミュレーション・ポイントとして選択されている黒い点を見ると，緑色のフェーズのシミュレーション・ポイントは，緑色の点の IPC の分布の中では平均的な点が選択されている．したがって，今回の推定では目立った誤差にはなっていない．

事前シミュレーション　今回の事前シミュレーションでは，ref 入力 100G 命令を鬼斬でサイクルアーキュレートにシミュレーションするのに 1 か月の期間を要した．このように，提案手法では，事前シミュレーションにはそれなりの計算資源と時間が必要である．今回の基準アーキテクチャの事前シミュレーションはスカラプロセッサを除いて，すべてのアーキテクチャでスーパスカラプロセッサのパイプラインの挙動をサイクルアーキュレートに行っている．しかし，これは，少々やりすぎであるといえる．

例えば，先の章で見たキャッシュのフェーズを補強する為の，多階層キャッシュアーキテクチャの事前シミュレーションについて考える．この場合，キャッシュシミュ

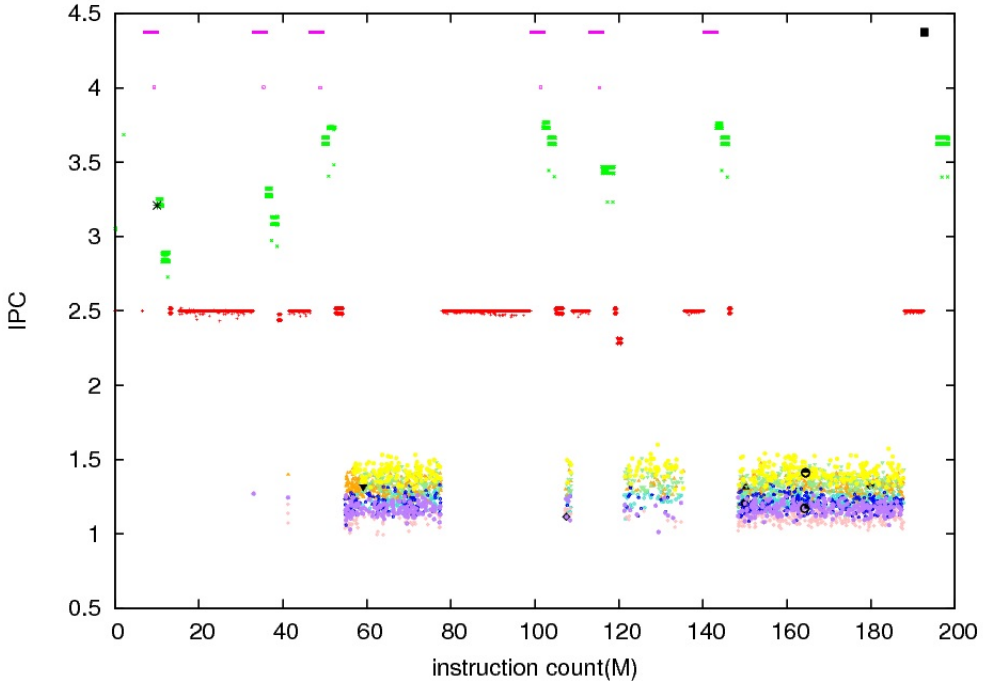

図 6.1: gobmk の fetch-eight のフェーズ

レーションを行いヒット率を得れば取りたいフェーズは原理的には得られたと考えられる．これは，提案手法が性能差が生じているところを別々のフェーズとして区別する手法であるためである．その” 性能差” は IPC の差でなければならないわけではない．

キャッシュのシミュレーションはパイプラインのシミュレーションに比べて非常に軽く，100G 命令のシミュレーションでも，数日程度で終わる．このように，プロセッサの特定のユニットだけを，シミュレーションし，その性能を測定することで，事前シミュレーショの計算コストを抑えることについても今後検討すべきである．これは，今後 ref 入力の全長実行に対してい評価を行う際にも有効な手法となりうる．

6.2 まとめ

本稿では，複数の特徴的な基準アーキテクチャのシミュレーション結果から帰納的にシミュレーション・ポイントを推定する手法を提案した．また，SPEC2006 のベンチマークのシミュレーション・ポイントを選出し，IPC 推定を行って，実際のシミュレーションにより得られた IPC との誤差を評価した．評価の結果，提案手法が SimPoint よりも，少ないシミュレーションポイントで正確な IPC の推定が行えることを確認した．

付録 A　　評価に用いたベンチマーク一覧

ref 入力先頭 100G 命令	test 入力全長
400.perlbench	400.perlbench
401.bzip2	403.gcc
445.gobmk	435.gromacs
458.sjeng	445.gobmk
462.libquantum	450.soplex
464.h264ref	459.GemsFDTD
471.omnetpp	462.libquantum
483.xalancbmk	471.omnetpp
410.bwaves	483.xalancbmk
416.gamess	
434.zeusmp	
435.gromacs	
436.cactusADM	
437.leslie3d	
444.namd	
447.dealII	
450.soplex	
459.GemsFDTD	
465.tonto	
470.lbm	
481.wrf	
482.sphinx3	

付録B　　多階層キャッシュの容量

階層番号	index bit 数	way 数	ofset bit 数	容量	レイテンシ
1	1	8	6	1(KB)	1
2	2	8	6	2(KB)	4
3	3	8	6	4(KB)	7
4	4	8	6	8(KB)	10
5	5	8	6	16(KB)	13
6	6	8	6	32(KB)	16
7	7	8	6	64(KB)	19
8	8	8	6	128(KB)	22
9	9	8	6	256(KB)	25
10	10	8	6	512(KB)	28
11	11	8	6	1(MB)	31
12	12	8	6	2(MB)	34
13	13	8	6	4(MB)	37
14	14	8	6	8(MB)	40
15	15	8	6	16(MB)	43
16	16	8	6	32(MB)	46
17	17	8	6	64(MB)	49
18	18	8	6	128(MB)	52
19	19	8	6	256(MB)	55
20	20	8	6	512(MB)	58

関連図書

[1] 早川薫, 倉田成己, 坂井修一, 坂井修一. 可変長セグメントを用いたフェーズ検出手法. SWoPP 2013

[2] 赤松雄一, 五島正裕, 坂井修一. 固定長インターバルを用いないフェーズ検出手法. SACSIS, 2011.

[3] G. Hamerly and C. Elkan
Learning the k in k-means CS2002-0716 University of California 2002

[4] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood.
Using machine learning to guide architecture simulation. Machine Learning Research, 2006.

[5] Greg Hamerly, rez Perelman, Jeremy Lau, and Brad Calde.
SimPoint 3.0: Faster and More Flexible Program Analysis, 2005.

[6] Erez Perelman Greg Hamerly Brad Calder.
Picking statistically valid and early simulation points. Parallel Architectures and Compilation Tech-niques (PACT), 2003

[7] Greg Hamerly Erez Perelman Brad Calder.
How to Use SimPoint to Pick Simulation Points

[8] Timothy Sherwood and Erez Perelman and Greg Hamerly and Suleyman Sair and Brad Calder
Discovering and Exploiting Program Phases,ISCA,2002

[9] M. Hind, V. Rajan, and P. F. Sweeney.
Phase detection: A problem classi

cation. Technical Report 22887, IBM Research, 2003.

[10] 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一:
回路面積指向レジスタ・キャッシュ, SACSIS,2008

[11] Jiaxin Li, Weihua Zhang, Haibo Chen, and Binyu Zang.
Multi-level phase analysis for sampling simulation. Technical report, 2013.

[12] Jeremy Lau, Erez Perelman, and Brad Calder.
Selecting software phase markers with code structure analysis. In Code Generation and Optimization(CGO), 2006.

[13] Arun A. Nair, Lizy K. John
Simulation Points for SPEC CPU 2006, 2008

[14] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsaf, James C. Hoe
SimFlex: Statistical Sampling of Computer System Simulation

著者発表論文

口頭発表

1. 帰納的なシミュレーション・ポイント選出手法の改良
   福田隆, 倉田成己, 五島正裕, 坂井修一
   情報処理学会研究報告 2014-ARC-212, No. 20, pp. 1-8 (2014).

2. 既存アーキテクチャのシミュレーション結果を用いる汎用シミュレーションポイント検出手法
   福田隆, 倉田成己, 五島正裕, 坂井修一
   情報処理学会研究報告 2014-ARC-211, No. 12, pp. 1-7 (2014).

受賞

福田隆: 第 203 回計算機アーキテクチャ研究会若手奨励賞，2014 年

謝辞

本研究を進めるにあたり，本当にたくさんの方々にお世話になりました．

指導教員である坂井先生には相談会などにおいて多くのご指導をいただきました．また五島先生には論文の添削や研究の進め方だけに限らず，論理的な考え方など，さまざまな面でご指導頂き，時には長時間の説教ももらいながら，一生ものの助言を数多く頂きました．

倉田成己氏はいつも研究室で席が隣ということもあり，気軽に思い付いたことを相談させてもらい，たくさんの指導や助言を頂きました．八木原晴水さん，長谷部環さんには，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました．また，同期の方々のおかげで大変楽しい研究生活が送れました．その他多くの研究室メンバーにも研究活動において様々な面でご協力いただきました．本当にみなさまありがとうございました．