動的コンパイラのための実行時分岐予測を用いた最適化手法

安江俊明† 緒方一則† 小松秀昭†

本論文では，動的コンパイラ環境における実行時分岐予測情報の収集とそれを用いた最適化の方法について述べる．従来から，より高度な最適化を実現する方法として実行時情報を用いた最適化手法が研究されてきた．特に動的コンパイラを用いた実行環境では，情報の収集とそれを利用した最適化をプログラムの同一の実行中に行うことができるために，静的コンパイラの場合に問題となる収集した入力データと実際に実行する時点での入力データの不一致による振舞いの違いの問題を生じない利点がある．反面，収集にかかる処理コストがそのまま実行時間に影響するとともに，収集される情報がプログラムの実行の一部分に限定されるために，収集される情報の量や収集時期に大きな制約がかかり，結果として収集情報の精度低下の問題が起きる．動的コンパイラ環境においては，近年関数間情報を用いた最適化についての研究がいくつかなされているが，本論文では Java プログラムを対象として関数内分岐履歴情報の活用方法について議論する．収集量と収集時期が限定された実行時情報に基づく分岐予測の精度を，プログラム全体を通じた場合の結果と比較しながら有効利用可能な情報の収集方法について考察するとともに，履歴情報を用いた多分岐命令の最適化手法として期待値モデルを用いた方法を提案し，その効果をいくつかのベンチマークプログラムを用いて評価した結果を示す．

An Optimization Technique Using Profile Based Branch Prediction for Dynamic Compilers

Toshiaki Yasue,† Kazunori Ogata† and Hideaki Komatsu†

In this paper, we describe an optimization technique using a profile-based branch prediction for dynamic compilers. Previously many works have been proposed to obtain more performance by using of profile-based optimizations in static compilers. Although static compilers can use profile information collected through the full program execution to get fully precise information for the optimizations, they are troubled with the difference of characteristics between the execution to get profile information and the truth execution. On the other hand, on a dynamic compiler environment, because the compiler can optimize a program using the very profile information of the running program, it applies the effective optimization for each individual execution. The quality of the profile information, however, tends to become worse because of the restrictions of collecting cost, quantity, and period. In this paper, we propose a profile-based multiway branch optimization technique as well as to discuss the efficiency of the small size profiling of branches for JAVA program on a dynamic compiler environment.

1. はじめに

プログラムのさらなる高速化を目指して，プログラムの挙動を仮定した最適化手法の研究がなされてきている．既存の最適化技術は，基本的にプログラムのすべての経路に対して平等に最適化を実施する．しかしながら，実際のプログラムでは，例外処理など多くの場合に実行されない処理が存在し，そのために実行頻度の高い経路の最適化が阻害される要因となっていることが知られている．この問題を解決する方法として，古くから，プログラムの履歴情報を用いた最適化技術の研究が行われてきた．これらの研究では情報収集は主に予備実行を用いて行い，その後収集した情報を用いて再度コンパイルを実施して最適化を行っている．

近年，Java [1]の普及により動的コンパイラが脚光を浴びるとともに，動的コンパイラ環境における実行履歴の活用が研究されるようになってきた[18)~22)]．しかしながら，文献 4) でも述べられているように，従来の履歴情報収集がプログラム全体の要約を求めているのに対して，動的コンパイル環境での履歴情報収集はあくまで実行予測の域を脱していない．また情報収集コストが実行オーバヘッドとしてそのまま反映されてしまうことから，主にメソッド呼び出しや実行頻度の高いメソッドやループの検出など，比較的収集する情報の単位が大きいものに限定して用いられているのが

† 日本アイ・ビー・エム（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

現状である．

本研究では，動的コンパイラ環境での，分岐履歴情報などメソッド内の履歴情報を用いた最適化技術の実現を目的とする．メソッド内の履歴情報としては，従来より様々な実行経路検出の手法[2)~6)]が提案されている．一方収集された分岐情報や経路情報を用いた最適化手法[7)~14),16)]も広く研究されており，頻度の高い経路を優先した不要命令の除去やコードの移動，基本ブロック（Basic Block）の並べ替え，多分岐命令の最適化など様々な手法が提案されている．

メソッド内の挙動情報をもとにした最適化を動的コンパイル環境へ適用する際の問題は，予測の質にある．動的コンパイル環境における情報収集は，対象とするメソッドが最適化コンパイルされるまでの期間であり，プログラムの全体を通じての挙動を掌握することは不可能である．また情報収集コストはすべてプログラムの実行時間に反映される点や，収集される情報量によるメモリ資源の消費の問題も存在することからも，多量の情報を収集することが困難な状況にある．このため，本論文では収集コストと収集情報量を小さくする観点から，少量の履歴情報収集手法を用いる．この集手法により得られる適中率の調査結果を示すとともに，プログラムをある程度実行してから情報を収集することで適中率を向上できることを示す．

一方，実行履歴を用いた多分岐命令の最適化手法としては，文献 16) で，多分岐命令など検索順序が可換な線形探索に対して実行履歴を用いて並べ替える手法が提案されている．しかしこの手法は分岐確率によっては実行履歴を用いない 2 分探索法など一般的な多分岐命令のコード生成手法[17)]（以下静的探索手法と呼ぶ）よりも遅いコードを生成してしまう場合がある．本論文では期待値モデルを用いて履歴情報を用いた探索と 2 分探索手法などの効率の良い探索手法を組み合わせることにより，速度低下を引き起こさずに高速な処理を実現できる手法を提案する．

以下では，まず 2 章で，少量の分岐履歴情報を収集した場合の分岐予測とその適中率についてベンチマークプログラムを用いて調査した結果を示す．続く 3 章では，実行履歴を用いた多分岐命令の最適化手法について評価基準となる判定式の提示とともに実装方法を示し，4 章で実行履歴を用いた最適化手法の評価を行う．

2. 分岐命令の実行履歴収集

既存の予備実行を用いた実行履歴収集で問題となるのは，予備実行と本実行での入力データの違いに起因するプログラムの挙動の変化であった．動的コンパイラにおいては，同一実行中に情報収集，最適化から実行までを実施するので，基本的に入力データの違いという問題は存在しない．動的コンパイラ環境における実行履歴の収集で問題となるのは，収集量，収集時期，収集対象，および収集に要する処理コストである．

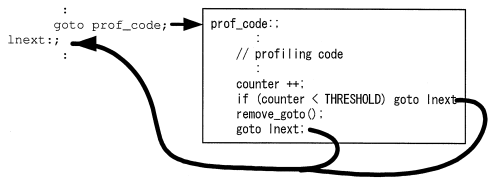

図 1 履歴情報収集処理
Fig. 1 Profile collection code.

既存の動的コンパイラでも，実行履歴を用いた最適化の研究が行われているが，そこで用いられる履歴情報は，主に関数呼び出しや，メモリ管理，例外処理など比較的処理コストの高いものである．これに対して，分岐命令はプログラム中の総数が多い半面，多分岐命令を除くと実行コストは 1 命令分であるため，情報収集コストが全体に及ぼす影響は大きい．

本章では，まず定数回の情報収集を行うための処理の実装方法について示す．続いて 2 分岐命令と多分岐命令について，SPECjvm98 ベンチマークプログラムを用いて調査した実行予測の結果をもとに履歴情報の収集方法と収集された情報の質について考察する．

2.1 履歴情報収集処理の実装

図 1 に定数回の情報収集後に自動的に収集処理を除去する方法の概略を示す．情報収集処理が本来の実行を遅くする要因をなくすために収集処理コードは正規の処理とは別に作成する．情報収集を開始する場合は，情報収集地点に分岐命令を挿入して収集処理を実行させる．処理の最後にカウンタを設定しておき，カウンタが既定回数に達したら情報収集地点に挿入した分岐命令を除去することで情報収集を完了する．収集完了後は本来の実行を妨げる命令がまったくなくなるのでその後の実行遅延を引き起こさないことを保証できる．この方法の利点は収集回数を指定することで，収集量と収集コストを制御できる点である．他方，欠点としては，短期間での情報収集となるため局所的な挙動の偏りに影響されやすい点があげられる．この問題を解決する方法として収集期間をより長くとる方法もあるが，全体として収集コストの影響が問題となるとともに，収集処理自体が複雑化するため，一概に有効とはいいがたい．

2.2 2分岐命令の実行履歴収集

2分岐命令の実行頻度の高さと情報収集オーバヘッドを考慮して，情報収集は収集開始後の連続するn回の分岐履歴情報を集積する処理として実装する．本論文で用いた実装では，各2分岐命令に対してそれぞれnビット領域を割り当て，分岐した場合には1，分岐しなかった場合には0とする長さnのビット列として格納する．分岐予測は，**表1**に示す基準に従って設定されるものとする．また情報収集時期を決定する方法として，各メソッドにそれぞれ1つのカウンタを用意して起動回数をカウントし，起動回数が指定値になった時点で情報収集を開始する処理を実装する．

表2に，履歴サイズnを1〜4とした場合の2分岐命令の分岐予測適中率を，2つの情報収集時期（初期状態履歴，定常状態履歴）において調査した結果を示す．ここで，初期状態履歴（Initial State Profile）は情報を収集する2分岐命令が属するメソッドの実行回数が0，つまりプログラムの開始時より収集を行った場合の履歴情報と定義する．また，定常状態履歴（Steady State Profile）は情報を収集する2分岐命令が属するメソッドが十分多くの回数実行された時点から収集を開始した場合の履歴情報と定義する．本論文では起動回数が1,000回を超えた時点を開始時期として設定して調査を行っている．適中率は全実行を通じての予測分岐方向への分岐数の比率として定義する．

表より，履歴サイズが1〜3回までは適中率が増加するが，3回と4回ではほぼ同程度の値を示している．また，収集時期として初期状態と定常状態を比べた場合，後者の方が最大で10%の適中率の向上が見られる．これは，プログラムの初期状態と定常状態での振舞いの違いが存在することを示すものである．compressは適中率が約68%と低いが，この原因の1つとして，後方分岐の適中率が低いことがあげられる．分岐しないとした予測した後方分岐での実際の分岐率は約87%で，この予測の失敗が全体の適中率を大きく引き下げている．後方分岐ではつねに分岐すると予測させた場合は，全体の適中率は約80%まで向上する．この問題は情報収集の方法によるところも多い．本実験では，初期状態，定常状態のいずれの場合も，メソッドの呼び出し時にすべての分岐の情報収集を開始しているため，いずれの状態においてもメソッドの実行開始から最初の4回の実行履歴を保存する．一方compressの場合は，少なくとも最初の4回の実行では分岐しないために，実験で用いた方法ではつねに初期状態しか収集できず，定常状態での振舞いを予測できなかったためである．

表1 履歴情報からの分岐予測
Table 1 Decision of binary branch prediction.

Status	Prediction
All bits are 1	Taken
All bits are 0	Not taken
Otherwise	No prediction

2.3 多分岐命令の実行履歴収集

ここではJava言語の多分岐命令であるswitch文に対する履歴情報収集方法とその精度について述べる．

多分岐命令の履歴情報収集方法は，複数の分岐先に対して履歴情報を収集できるようにカウンタを用いて履歴情報を収集する．Java言語のswitch文はcase文によりいくつかの比較定数値を定義し，実行時には与えられた式の値に等しい定数値を探索し，存在すれば対応する処理を，存在しなければdefault文☆で指定された処理を実行する．さらにdefault文の実行条件は，整数値空間において各caseラベルの定数値で分割された各区間で表される範囲に値が入るという条件の和として定義できる．

定義：switch文の探索条件式集合を，あるswitch文において，その各caseラベルの定数値と等しいかどうかを調べる条件式，およびそのcaseラベルの定数値で分割される整数空間の各区間で与えられる範囲内あるかどうかを調べる条件式からなる集合と定義する．

探索条件式集合に属する各条件式が示す範囲は互いに素であり，またそれらの範囲の和は整数空間全体に等しくなる．履歴情報収集処理では，この集合中の各条件式の示す範囲ごとに1つのカウンタを割り当てる．必要カウンタ数はcase文数をN個とすると最大$(2N+1)$個となる．**図2**に，switch文に対するカウンタ割当ての例を示す．また集合の各要素を探索条件式と呼ぶこととする．

履歴情報収集は分岐が行われるたびに分岐条件に対応するカウンタを1ずつインクリメントする．実行履歴収集は，switch文の実行回数が規定値に達するか，またはいずれかのカウンタがオーバフローした時点で終了する．本論文では各探索条件に対するカウンタのサイズを4bitとした．また実行回数の規定値はcase数の2倍と80の大きい値として設定した．

表3に，多分岐命令の分岐予測の精度について調査した結果を示す．本論文では，多分岐命令の分岐予測の評価手法として，探索条件式集合に対して，カウン

☆ default文が存在しない場合はswitch文の次の文を実行する．

表 2 2 分岐命令の分岐予測の適中率
Table 2 Hit ratio of binary branch prediction.

	Initial State Profile(%)				Steady State Profile(%)			
profile count	1	2	3	4	1	2	3	4
mtrt	77.83	83.25	85.74	86.49	80.01	83.63	84.34	84.86
jess	71.77	78.37	81.73	82.81	77.94	78.90	93.56	92.80
compress	65.02	65.72	65.71	65.71	65.02	65.72	66.44	67.59
db	97.68	98.07	98.07	98.63	97.68	98.08	98.09	98.03
mpegaudio	78.44	84.36	91.96	92.34	81.83	92.59	94.13	96.28
jack	79.41	83.47	90.09	91.97	86.77	91.09	92.08	93.49
javac	81.63	90.16	90.97	92.60	85.03	91.03	92.32	92.75

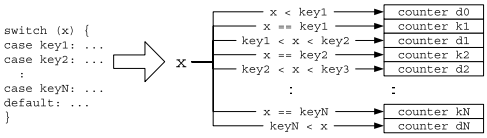

図 2 多分岐命令の履歴情報収集方法の例
Fig. 2 Example of multiway branch profile collection.

表 3 多分岐命令の分岐予測と実際の分岐結果との相関
Table 3 Correlation coefficien of multiway branch prediction.

program	static	dynamic
mtrt	0.350	0.574
jess	0.798	0.371
compress	1.000	1.000
db	0.800	0.987
mpegaudio	0.998	0.994
jack	0.934	0.918
javac	0.742	0.810

タ値から算定した予測実行確率分布と全実行を通じて求めた実際の実行確率分布との間の相関係数で評価を行った．なお予測実行確率は各条件に対するカウンタ値をカウンタ値の合計で割った値として計算し，実際の実行確率はプログラム全体の実行を通じての各条件の実行数を総実行回数で割った値として計算するものとする．相関係数−1 から 1 までの値をとり，2 つの分布の相関が高いほど 1 に近づき，逆相関の場合は負数となる．

表中の static フィールドは単純平均，dynamic フィールドは実行回数により重み付けした平均である．相関係数の値が低い mtrt と jess について調査したところ，著しく相関係数が低いのは以下の 2 つの理由のためであった．

(1) 全探索条件式の実際の実行頻度がほとんど同じために，カウンタ値の値のふらつきの影響で分布が逆相関となり，相関係数が負になってしまった場合．

(2) もともと多分岐命令が含まれていたメソッドが複数のメソッドから呼ばれており，呼び出しサイトごとに挙動が異なる場合．

前者は情報収集の精度に帰着する問題であり，予測確率を求めるうえで収集情報上の誤差の存在を考慮する必要があるといえる．後者は，メソッド・インライニングにより多分岐命令がそれぞれの呼び出し側にインラインされたときに起こる問題である．多分岐命令が他のメソッドにインライニングされた場合には，データフロー解析などを利用して解析するなどで予測の補正が必要である．

2.4 Rare Path Analysis による分岐予測の補正

プログラム中の経路についてその頻度が高いことを求めるためには長期間にわたって実行情報を収集する必要があり，収集量および収集コストともに大きくなってしまい，プログラムの実行に影響を与える．ここでは，逆に頻度の低い希少実行経路（Rare path）を求める処理について述べる．基本的なアイデアは，経路中に配置されたカウンタが実行を通じてオーバフローしなければ，そのカウンタは希少経路上に存在すると判定することである．情報収集はメソッドの開始時点から最適化処理の実施時点までの全体の期間を通じての結果のため，可能な限り最大の情報量で予測できることになる．ここでは，希少実行経路を求めるために特別なカウンタを生成するのではなく，分岐命令の履歴情報収集のための実行カウンタ，メソッド呼び出しでのレシーバメソッドの起動カウンタ，および Resolusion☆を必要とする命令において Resolution を行ったかどうか，という情報をカウンタの代用として用いて希少実行命令を検出する．さらにこれらの希少実行命令がが属する基本ブロックが後方支配する基本ブロックと支配する基本ブロックはすべて希少経路上であることを利用して，隣接する希少実行の基本ブロックを求めることができる．

表 4 に希少経路に対する調査結果を示す．ここで，bb ratio は検出された希少経路を構成する基本ブロッ

☆ Resolution は Java 仮想機械において実行時にアドレスなどの解決を行う処理である．

表 4 希少実行経路の検出
Table 4 Rare path detection.

program	bb ratio(%)	code ratio(%)	hit ratio(%)
mtrt	7.50	6.28	91.79
jess	2.97	3.35	98.96
compress	5.66	5.91	99.99
db	4.13	3.74	100.00
mpegaudio	5.45	3.13	99.98
jack	16.08	14.88	97.54
javac	16.91	14.24	99.44

クのプログラム全体に占める割合を，code ratio は検出された希少経路上のバイトコード数のプログラム全体に占める割合をそれぞれ表している．また適中率（hit ratio）は，希少経路へ分岐する分岐命令の分岐確率より算定している．希少経路の適中率は，分岐命令での適中率が低かった compress においても 99%以上であり，全体としても 9 割を超える適中率を示している．つまり，希少経路の検出は情報収集コストの小ささに比べて非常に精度の高い予測が可能であるといえる．今回の実験では，希少実行経路への分岐の予測が希少経路側になっているものが総実行回数に比べると少量のため，分岐予測の補正効果は微小となった．

3. 多分岐命令の最適化

本章では，Java 言語の多分岐命令である switch 文に対して実行時履歴情報を用いた最適化手法を提案する．

多分岐命令は，与えられた整数値に適合する条件を探索する処理であり，コンパイラにおける多分岐命令のコード生成はこの探索処理を実施する探索木の生成にほかならない．したがって，多分岐命令の最適化は効率的な探索木を作ることといえる．

実行履歴情報を用いた多分岐命令の最適化手法としては，文献 16) で，多分岐命令など検索順序が可換な線形探索に対して実行履歴を用いて並べ替える手法が提案されている．この方法では，各条件式を式 (1) に示す確率的コストで降順に並べ替えることで最適な実行順序を求めている．しかしながら，この手法は全テストを線形探索することを前提にしており，分岐確率によっては 2 分探索法など一般的な多分岐命令のコード生成手法[17]よりも遅いコードを生成してしまう場合がある．したがって，実行履歴情報を用いた多分岐命令の最適化のためには，従来の多分岐命令の静的な最適化手法も考慮した最適化手法が必須である．

$$\text{確率的コスト} = \frac{\text{実行確率}}{\text{比較コスト}} \quad (1)$$

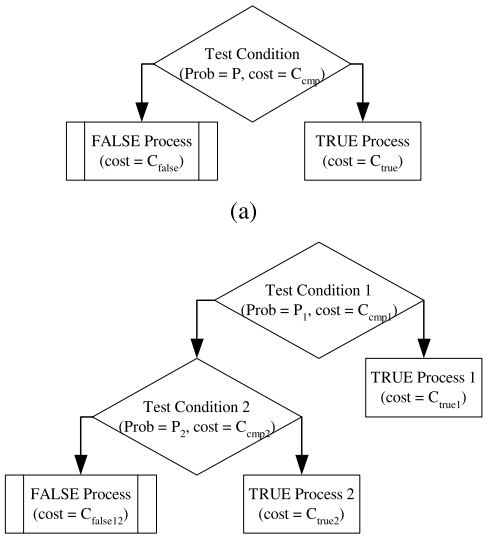

図 3 条件判定モデル
Fig. 3 Test conditional model.

本章では，まず探索条件式が与えられた場合，それを適用した場合の全体の期待コストを計算する期待値モデルを用いた評価式を導入する．この評価式により期待コストが適用前より削減できるかどうかを判断することで探索条件式による優先探索の有効性判断を可能としている．その後，これらの式を用いた探索木の生成アルゴリズムを提示する．なお以下で用いる探索条件式は 2.3 節での定義と同義である．

3.1 判 定 式

本論文で用いる条件判定のモデルを**図 3**(a) に示す．モデルは，条件判定部，成立時の処理，不成立時の処理からなり，それぞれの実行コストを，C_{cmp}，C_{true}，C_{false} とし，条件の成立確率は P で与えられるものとする．成立時の処理は，条件が成立した場合に必要となる処理を表す．たとえば単純な key との比較判定の場合は，成立時の処理はないので C_{true} は 0 となり，また，たとえば**図 4**(b) に示すような複合的な条件を含む範囲を判定条件とすれば，成立後に内包する個々の条件の判定のための処理が必要となり，C_{true} はその処理のために必要となるコストを表す．

ある探索条件式を適用した場合の期待コスト C_e は，条件の成立時と不成立時の期待値の合計値として，次式で与えられる．

$$C_e = C_{cmp} + P * C_{true} + (1 - P) * C_{false} \quad (2)$$

最適化前のコストを C_{orig} とすると，ある探索条件

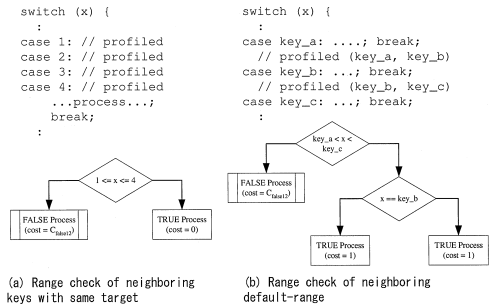

図 4 隣接条件マージの例
Fig. 4 Example of neighboring tests merger.

式を適用した場合の期待コスト C_e が C_{orig} よりも小さくなることが，この探索条件式を適用するための必要条件となる．この必要条件式より予測確率が満たすべき条件を式 (3) のように導くことができる．したがってある探索条件式の適用の有効性は，式 (3) を満たすかどうかにより判定することができる．つまりこの式を満たす限り，探索条件式の適用による期待コストの増加を起こさないことが保証される．この式は，3.2 節で示すアルゴリズムにおいて，各探索条件式に対してその条件を優先探索するかどうかを判断する評価式として用いる．

$$C_{orig} - C_e > 0$$
$$P > \frac{C_{cmp} + C_{false} - C_{orig}}{C_{false} - C_{true}} \quad (3)$$

次に複数の探索条件式の順序関係の最適化のための評価式について述べる．この場合のモデルを図 3(b) に示す．ある 2 つの探索条件式が与えられた場合に，その適用順について考える．2 つの条件 1，2 が与えられたときに，各条件が実行される無条件確率をそれぞれ，P_1，P_2，2 つの条件を 1，2 の順次適用した場合の期待コストを C_{e12}，逆順に適用した場合を C_{e21} とする．さらに両方の条件が成り立たない場合のコストを $C_{false12}$ とする．このとき期待コスト C_{e12} は，次式により計算できる．

$$\begin{aligned} C_{e12} = {} & C_{cmp1} + P_1 * C_{true1} \\ & + (1 - P_1) * C_{cmp2} \\ & + P_2 * C_{true2} \\ & + (1 - P_1 - P_2) * C_{false12} \end{aligned} \quad (4)$$

探索条件式 (1) を探索条件式 (2) に先行して評価した方が全体の期待コストを減らすことができることは，$C_{e21} - C_{e12} > 0$ が成立することと同値である．この式を変形することで，式 (5) を得る

$$\frac{P_1}{C_{cmp1}} > \frac{P_2}{C_{cmp2}} \quad (5)$$

したがって，探索条件式が複数存在する場合に全体の期待コストを最小にする探索順序は，各探索条件式 i に対して式 (6) の値を計算し，降順に並べ替えることで得ることができる．

$$\frac{P_i}{C_{cmp-i}} \quad (6)$$

この式 (6) は，文献 16) での順序条件である式 (1) と同じである．このことから最適化順序を与える評価式は条件判定モデルを一般的化したモデルにおいても従来と同じ式で求めることができる．

3.2 判定式を用いた多分岐命令の探索木生成

ここでは本論文で提案する探索木生成アルゴリズムについて説明する．実行履歴を用いて探索処理を高速化するためには，実行頻度の高い探索条件をできるだけ小さい探索コストで実行することである．この実現のために本論文では，まず実行頻度の高い探索処理を基本的に線形探索により探索する優先探索部分と，従来の 2 分探索法や間接テーブルジャンプなどを用いて優先探索して検出されない条件を探索する静的探索部分からなる探索モデルを用いる．このモデルでは，まず優先探索部分の探索処理を実施し，そこに該当する条件がない場合，続いて静的探索部分を実行する．

本アルゴリズムは，大きく 3 つの処理からなる．最初の処理は，探索条件式を式 (6) の値に対して降順に並べる処理である．これは文献 16) の処理に等しい．

次に，次に**図 5** で示すアルゴリズムにより，優先探索部分の探索木を生成する．アルゴリズムは，探索条件 $tc[i]$ に対して順に式 (3) を評価する関数 EVAL_TEST_EFFICIENCY を実行する．式 (3) の評価で必要となる値のうち，C_{cmp}，C_{true} は各探索条件式固有の値であり，あらかじめ計算しておいた値を使用する．C_{orig}，C_{false} は既存の静的探索を実施した場合の処理コストを与える．これらの処理コストは，まだ優先探索に使用されていない探索条件に関する情報を構造体 $summ$ に格納しておき，この値を用いてして関数 CALC_COST_FALSE で計算する．確率 P は条件付確率を用いる．このために使用される残存探索条件の確率の合計値も構造体 $summ$ に格納される．式 (3) の評価で無効と評価された場合は，関数 INVALIDATE_CONDITION を用いて探索条件式 tc[i] を無効化する．一方有効と評価された探索条件に対しては，さらに関数 TRY_TO_MERGE_NEIGH と関数

```
// Assume the test conditions tc[*] are sorted in descending order of (P/Ccmp).
// summ:  summary information for rest conditions
// tc[i]: test condition
// msumm: summary information for merging tests
// mtc:   merged test condition

// calculate summary info. of remaining key
summ.npairs = nkey // number of remaining keys
summ.lkid = 1 // lowest key index
summ.hkid = nkey // highest key index
FOR i = 1 to n DO
  IF tc[i] has not been invalidated THEN
    // re-calculate summary info.
    UPDATE_SUMM (summ, tc[i])
    // evaluate the efficiency of test tc[i]
    IF (! EVAL_TEST_EFFICIENCY (tc[i], CALC_COST_FALSE (summ)) THEN
      INVALIDATE_CONDITION (tc[i])
    ELSE
      // try to merge neighboring tests
      msumm = summ; // copy current summary info to msumm
      mtc = TRY_TO_MERGE_NEIGH (tc[i], msumm)
      // evaluate merged test condition
      if (mtc != NULL and EVAL_TEST_EFFICIENCY (mtc, CALC_COST_FALSE (msumm)) THEN
        // replace current test with merged test
        tc[i] = mtc
        summ = msumm // update summ with msumm
        // invalidate merged test conditions
        INVALIDATE_MERGED_CONDITIONS (mtc)
      ENDIF
    ENDIF
  ENDIF
ENDFOR
```

図 5 探索木の生成アルゴリズム
Fig. 5 Decision tree generation algorithm.

EVAL_TEST_EFFICIENCY を用いて隣接探索条件を融合した場合の有効性を評価する．融合が有効と判断した場合には，融合した隣接条件に対する探索条件式を関数 INVALIDATE_MERGED_CONDITIONS を用いて無効化する．一方融合が無効と判断した場合はもとの探索条件を優先探索条件として登録する．本論文で実装した融合処理は，同一処理を行う連続する case ラベルに対する融合処理と，隣接する範囲に対する融合処理である．これらに対する融合の例を図 4 に示す．処理が終了した時点で，無効化されなかった $tc[i]$ た優先探索を校正する探索条件となる．

最後に優先探索部分に対して式 (6) に基づく並べ替えを適用して処理を終了する．この並べ替え処理は，図 5 のアルゴリズムで，融合が実施された場合に再度最適な適用順序を得るために必要となる．

4. 評　　価

ここでは，2 分岐命令と多分岐命令の実行予測の効果を，それぞれの情報を使用した最適化による速度向上率を用いて評価する．評価は AMD Athlon 1.2 GHz 上で動作する Microsoft Windows 2000 Profesional 上で，IBM Java Virtual Machine とそこで動作する Just-in-Time (JIT) Compiler [23]で行った．IBM JIT はインタプリタと JIT コンパイラを用いた動的コンパ

イル環境である．分岐情報収集は，インタプリタに実装し，そこで収集した情報をコンパイル時に使用する．2 分岐の実行予測は，基本ブロックの並べ替え処理と Edge Profile Guided Partial Redundancy Elimination [15]で用いる．また多分岐命令の実行履歴は 3 章で示した手法を実装した．評価対象は，SPECjvm98 ベンチマークと SPECjbb ベンチマークを用いた．

2 分岐命令の予測による最適化の効果に対する評価として，**図 6** に Edge Profile Guided Partial Redundancy Elimination の効果を示す．Edge Profile Guided Partial Redundancy Elimination は，分岐予測を用いて予測分岐側を優先して冗長な演算を除去する手法である．実装方法は，Lazy Code Motion（LCM）を基本として，分岐しないと予測されている側の情報を無効化することで履歴情報を反映させる．本手法は，Gupta らの手法[13]と比べて正確さは劣る反面，実装が簡単で処理コストが通常の PRE とほとんど代わらない利点がある．グラフは，分岐予測情報の収集時期を初期状態とした場合と定常状態とした場合に対して，実行履歴情報を与えずに通常の LCM として動作した場合に対する速度向上率で評価を行った．結果より，定常状態で収集した履歴情報に基づく分岐予測の方が，初期状態のものよりも有効であることが分かる．

次に多分岐命令の最適化の効果に対する評価として，**図 7** に実行履歴を用いた多分岐命令の評価結果を示す．図中の Profile Based Switch Optimization が本論文で提案している手法，Profile Based Switch Linear Search が文献 16) の手法である．グラフは多分岐命令に対する最適化を実施しない場合に対する速度向上率を示している．グラフから分かるとおり，提案手法は速度低下を引き起こさずに最適化を実現しているのに対して，Linear Search では jess において大幅な速度低下を引き起こしてしまっている．これは従来手法を利用できないために予測確率が分散している場合には線形探索の欠点が表面化してしまうためである．また，全体を見ても提案手法が速度を向上させているのが分かる．

図 8 は SPECjvm98 ベンチマークにおいて，実行履歴を用いた場合の速度向上率を示している．また，**図 9** は SPECjbb ベンチマークにおける，Edge Profile Guided Partial Redundancy Elimination を適用した場合の速度向上率，と Profile Based Switch Optimization を適用した場合の速度向上率，およびそれらを合わせた場合の速度向上率を示す．いずれのベンチマークでも，実行履歴を用いた最適化により速度が

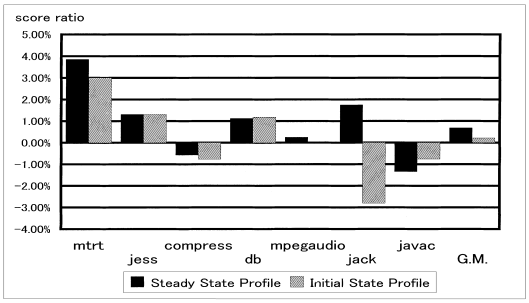

図 6 Profile Guided Partial Redundancy Elimination の効果
Fig. 6 Performance of Profile Guided Partial Redundancy Elimination.

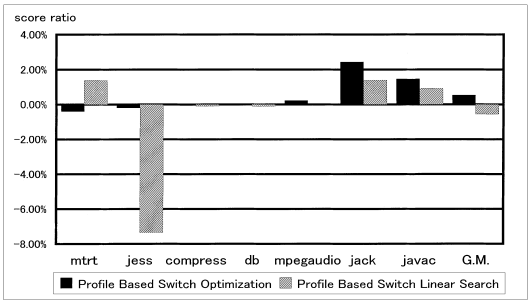

図 7 Switch Optimization の効果
Fig. 7 Switch Optimization performance.

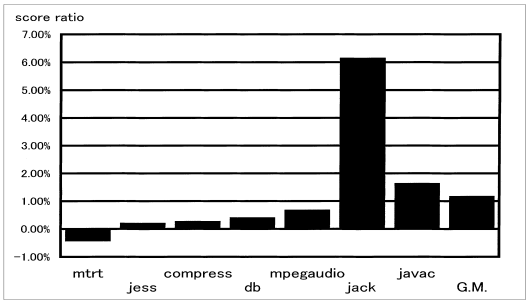

図 8 SPECjvm98
Fig. 8 SPECjvm98.

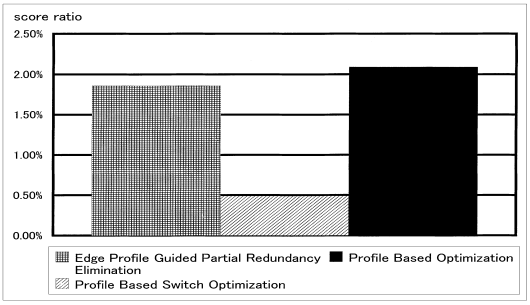

図 9 SPECjbb
Fig. 9 SPECjbb.

向上していることが分かる．

5. おわりに

本論文では，動的コンパイル環境における実行時履歴の可能性をさぐるために，条件分岐命令の予測の適中率を調べるとともに，予測を用いた最適化の効果を示した．2 分岐命令，多分岐命令とも良好な予測を実現できることを示すとともに，それらを用いた最適化により速度向上が得られることを示した．また，カウンタを用いた希少実行経路情報は特に予測の適中率が高いことを示した．

2 分岐命令では，収集情報が少ないことと，収集開始のタイミングがつねにメソッドの呼び出し時点であるために，compress のようにメソッド内で初期状態と定常状態があるプログラムでは，有効な情報を収集しきれないことが分かった．このことは，収集時期の設定をプログラムの全体的な実行だけでなく，各メソッド内での実行状態を考慮する必要があることを示唆している．収集情報量を増やすことは，最適化コンパイルを実施するまでの処理速度の低下の度合いを大きくすることから，情報量ではなく収集時期を柔軟に変更する仕組みが必要と思われる．

多分岐命令に対しては，期待値モデルを用いて履歴情報を用いた優先探索と 2 分探索方などの静的探索を組み合わせた探索手法の生成方法を提案した．従来手法では線形探索を前提とするために，たとえば実行比率がすべてのケースに分散するような場合は，実行履歴を用いない 2 分探索法などの処理よりも遅くなってしまう．本手法では期待値モデルを用いて探索コスト評価を縮小化するように探索条件式を決定することで，従来手法で問題となる速度低下を引き起こすような場合を回避しながら高速化を可能とした．

今後の課題としては，分岐予測の収集時期を柔軟化する手法に関する検討が必要である．また，多分岐命令の最適化手法については，評価式に与えるコストのパラメータの設定方法と，探索木の構築方法の改善があげられる．たとえば BTB ミスのペナルティーが大きいアーキテクチャでは，予測確率を比較コストに反映させる必要がある．さらに探索木構築の場合も，予測確率の向上を優先した手法が必要であると思われる．

謝辞 本研究を行うに際して貴重なご助言をいただいた中谷登志男氏をはじめとする日本アイ・ビー・エム（株）東京基礎研究所ネットワーク・コンピューティング・プラットフォームグループ諸氏に感謝いたします．

参考文献

1) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specifications*, Addison Wesley (1996).
2) Ball, T., Mataga, P. and Sagiv, M.: Edge Profiling versus Path Profiling: The Showdown, *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.134–148 (Jan. 1998).
3) Ball, T. and Larus, J.R.: Efficient Path Profiling, *Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.46–57 (Dec. 1996).
4) Duesterwald, E. and Bala, V.: Software Profiling for Hot Path Prediction: Less is More, *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.202–211 (Nov. 2000).
5) Zhang, X., Wang, Z., Gloy, N., Chen, J.B. and Smith, M.D.: System Support for Automatic Profiling and Optimization, *Proc. 16th ACM Symposium on Operating Systems Principles*, pp.15–26 (Oct. 1997).
6) Ball, T. and Larus, J.R.: Optimally Profiling and Tracing Programs, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1319–1360 (1994).
7) Chang, P.P., Mahlke, S.A. and Hwu, W.W.: Using Profile Information to Assist Classic Code Optimizations, *Software-Practice and Experience*, Vol.21, No.12, pp.1301–1321 (1991).
8) Pettis, K. and Hansen, R.C.: Profile Guided Code Positioning, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'90*, pp.16–27 (Jun. 1990).
9) Cohn, R. and Lowney, P.G.: Hot Cold Optimization of Large Windows/NT Applications, *Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.80–89 (Dec. 1996).
10) Ammons, G. and Larus, J.R.: Improving Data-flow Analysis with Path Profiles, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp.72–84 (Jun. 1998).
11) Gupta, R., Berson, D.A. and Fang, J.Z.: Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization, *Proc. 13th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.358–368 (Dec. 1997).
12) Gupta, R., Berson, D.A. and Fang, J.Z.: Path Profile Guided Partial Dead Code Elimination Using Predication, *Proc. International Confer-*

ence on Parallel Architectures and Compilation Techniques, pp.102–115 (Nov. 1997).
13) Gupta, R., Berson, D.A. and Fang, J.Z.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *Proc. International Conference on Computer Languages* (May 1998).
14) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Expression, *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pp.1–14 (Jun. 1998).
15) Kawahito, M., Komatsu, H. and Nakatani, T.: Eliminating Exception Checks and Partial Redundancies for Java Jus-in-Time Compilers, *IBM Research Report*, RT0350 (Apr. 2000).
16) Yang, M., Uh, G. and Whalley, D.B.: Improving Performance by Branch Reordering, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp.130–141 (Jun. 1998).
17) Spuler, D.A.: Compiler Code Generation for Multiway Branch Statements as a Static Search Problem, Technical Report 94/03, James Cook University, Townsville, Australia (Jan. 1994).
18) Sun Microsystems. The Java Hotspot Performance Engine Architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html (1999).
19) Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A Study of Devirtualization Techniques for a Java (TM) Just-In-Time Compiler, *Proc. Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.47–65 (Oct. 2000).
20) Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P.F.: Adaptive Optimization in the Jalapeno JVM, *Proc. Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.294–310 (Oct. 2000).
21) Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A Transparent Dynamic Optimization System, *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp.1–12 (Jun. 2000).
22) Cierniak, M., Lueh, G.Y. and Stichnoth, J.M.: Practicing JUDO: Java (TM) under Dynamic Optimizations, *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp.13–26 (Jun. 2000).
23) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol.39, No.1, pp.175–193 (2000).

(平成 13 年 5 月 31 日受付)
(平成 13 年 7 月 31 日採録)

安江　俊明（正会員）
1991 年早稲田大学大学院理工学研究科電気工学専攻修了．1995 年同大大学院博士後期課程退学後，日本 IBM（株）東京基礎研究所入社．最適化コンパイラ，並列処理の研究に従事．

緒方　一則（正会員）
1967 年生．1990 年，東京工業大学工学部電気電子工学科卒業．同年日本 IBM（株）入社．現在，東京基礎研究所において，Java 仮想マシンのインタープリタの研究に従事．

小松　秀昭（正会員）
1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．