プロファイル情報を用いた
大規模アプリケーションの高速化手法に関する研究

Profile Based Optimization Techniques
for Large Scale Applications

2007年 3月

安江　俊明

Toshiaki Yasue

目次

第１章　序論 9
１．１　研究の背景 9
１．２　研究の目的 11
１．２．１　実行時分岐予測を用いた多分岐処理の高速化 12
１．２．２　実行時経路情報の構造的収集 13
１．２．３　静的プロファイルを用いたプログラム起動処理の高速化 13
１．３　本論文の構成 13
第２章　実行時分岐予測を用いた分岐処理の高速化 15
２．１　はじめに 15
２．２　分岐命令の実行履歴収集 17
２．２．１　履歴情報収集処理の実装 18
２．２．２　分岐履歴格納のための領域サイズ 19
２．２．３　２分岐命令の実行履歴収集 20
２．２．４　多分岐命令の実行履歴収集 22
２．２．５　低頻度実行経路解析による分岐予測の補正 24
２．３　多分岐命令の最適化 26
２．３．１　判定式 27
２．３．２　判定式を用いた多分岐命令の探索木生成 29
２．４　実験結果 33
２．４．１　評価環境 33
２．４．２　評価方法 33
２．４．３　評価結果 34
２．５　まとめ 37
第３章　オンラインパスプロファイルの構造的収集手法 39
３．１　はじめに 39
３．２　背景 41
３．２．１　説明用プログラム 41
３．２．２　収集したプロファイルの精度の評価手法 42
３．２．３　エッジプロファイルとパスプロファイル 42
３．２．４　動的パスプロファイル収集の課題 44
３．３　構造的パスプロファイル収集手法 47
３．３．１　構造グラフの定義 48
３．３．２　構造グラフの構築 49
３．３．３　インストルメンテーションコードの生成 50
３．３．４　プロファイル収集の管理 53
３．３．５　グローバル・プロファイルの生成 55
３．３．６　最適化の例 58
３．４　パスプロファイルを用いたLoop Peeling手法 59
３．４．１　経路の分類と実行頻度の計算 61
３．４．２　適用例 61
３．５　評価 63
３．５．１　評価方法 63
３．５．２　メモリ使用量 64
３．５．３　精度の評価 66
３．５．４　プロファイル収集のオーバヘッド 70
３．５．５　パスプロファイルを用いたLoop peeling手法による評価 71

３．６　議論 72
３．７　関連研究 72
３．８　まとめ 74
第４章　静的プロファイルを用いたプログラム起動処理の高速化 75
４．１　はじめに 75
４．２　アプリケーション起動時のI/O待ち時間の問題 77
４．２．１　I/O待ち時間の定義 77
４．２．２　アプリケーション起動時におけるI/O待ち時間 77
４．２．３　解決すべき課題 79
４．３　シナリオに基づくファイル・プリフェッチ手法 80
４．３．１　サンプルプログラム 81
４．３．２　プロファイルの収集 82
４．３．３　シナリオの生成 84
４．３．４　プリフェッチの実施 92
４．４　実験結果 93
４．４．１　評価環境 93
４．４．２　評価結果 93
４．５　関連研究 94
４．５．１　プログラムの実行時のファイルアクセスパターンを用いてプリフェッチ処理を行う方法　94
４．５．２　プログラムの静的プロファイルを用いてプリフェッチ処理を行う方法 95
４．５．３　アプリケーションを書き換えてプリフェッチ処理を行う方法 95
４．５．４　コンパイラによりプログラムを解析することでプリフェッチ処理を行う方法96
４．５．５　プログラムを多重化して生成したプリフェッチ処理プロセスを用いる方法...96
４．５．６　ハードディスクのSeek時間を最小化する方法 96
４．５．７　ハードディスク上のファイルの配置を並べ替える方法 96
４．６　まとめ 96
第５章　結論 99

図目次

図２.１　履歴情報収集処理 18
図２.２　2 分岐命令の履歴情報の収集 20
図２.３　多分岐命令の履歴情報収集方法の例 23
図２.４　低頻度実行経路の検出 25
図２.５　条件判定モデル 27
図２.６　探索木生成アルゴリズム 30
図２.７　隣接条件マージの例 31
図２.８　探索木の生成アルゴリズム 32
図２.９　Profile Guided Partial Redundancy Eliminationの効果 35
図２.１０　Switch Optimizationの効果 36
図２.１１　SPECjvm98 36
図２.１２　SPECjbb 37
図３.１　２重ループを持つプログラムとそのエッジプロファイルの例 41
図３.２　オーバーラップパーセンテージ手法 42
図３.３　Restructuring by loop peeling 44
図３.４　Instrumentation code and profile counter inserted by BPP 45
図３.５　Hot paths detected by BPP. 46
図３.６　構造グラフの例 49
図３.７　構造グラフの生成アルゴリズム 50
図３.８　Algorithm to generate instrumentation code. 51
図３.９　インストルメンテーション処理の例 52
図３.１０　Algorithm for profile controller. 54
図３.１１　プロファイル制御機構 55
図３.１２　Algorithm to generate global profiles. 57
図３.１３　Hot paths detected by SPP. 59
図３.１４　エッジプロファイルを用いたLoop Peeling 62
図３.１５　Forward pathプロファイルを用いたLoop Peeling 63
図３.１６　Differences in the numbers of paths between SPP and BPP. 65
図３.１７　SPECjvm98 ベンチマークを用いたOverlap Percentageによる各閾値でのSPPとBPPの精度の比較 67
図３.１８　jBYTEmarkベンチマークを用いたOverlap Percentageによる各閾値でのSPPとBPPの精度の比較 68
図３.１９　SPECjvm98 ベンチマークとSPECjbb2000 ベンチマークによる，初回実行におけるSPPのオーバヘッドの評価 71
図３.２０　Loop peelingをもちいた速度向上の評価 71
図４.１　RedHat9 LinuxでのWAS 5.1.1 の起動時間の評価結果 78
図４.２　WAS 5.1.1 起動時のファイルからのデータ読み込みにかかる時間の時間的推移 78
図４.３　Javaプログラムで使用されるJarファイルの典型的なファイルアクセスパターン 80
図４.４　シナリオに基づくプリフェッチ手法の流れ 81
図４.５　サンプルプログラム 82
図４.６　read処理に対してプロファイル情報を収集するための処理 83
図４.７　I/Oプロファイルの例 83
図４.８　プリフェッチ処理リストの生成 84
図４.９　プリフェッチ処理の並べ替えの実施例 85
図４.１０　各Read処理の読み込み終了時刻Tiの計算 86
図４.１１　I/O処理をオーバラップさせた場合の各Read処理の開始処理を求めるアルゴリズム 87
図４.１２　各プリフェッチ処理の開始期限時刻の計算 88

図４.１３　各プリフェッチ処理の最終開始時刻を求めるアルゴリズム.....88
図４.１４　プリフェッチ処理の並び替え.....89
図４.１５　プリフェッチ処理の並び替えを行うアルゴリズム.....90
図４.１６　同期点の決定.....91
図４.１７　プリケーションとの同期点を検出するアルゴリズム.....92
図４.１８　Scenario-based prefetchingの実行モデル.....93
図４.１９　評価結果.....94

表目次

表１.１　プロファイルを用いた最適化の分類 11
表２.１　２分岐履歴情報に必要な領域のサイズに関する評価 19
表２.２　２分岐履歴情報に必要な領域のサイズに関する評価 20
表２.３　履歴情報からの分岐予測 21
表２.４　２分岐命令の分岐予測の精度 22
表２.５　多分岐命令の分岐予測と実際の分岐結果との相関を用いた精度の評価 24
表２.６　希少実行経路の検出 26
表３.１　Path profiles collected by BPP. 46
表３.２　Profile Results 58
表３.３　経路数とインストルメンテーションポイント数の比較 64
表４.１　従来手法の分類と比較 80

第１章　序論

１．１　研究の背景

近年のアプリケーションの巨大化により，プログラムの複雑さは増大の一途をたどっている．例えば，多くのビジネスアプリケーションは，企業の様々なビジネスロジックを処理するために非常に大規模なシステムとなるとともに，長年にわたる仕様の追加や変更により複雑化を増している．プログラムの構築方法においても，ビジネスロジックやビジネスプロセスを直接プログラムで記述するのではなく，さまざまなモデルを用いて記述されたプログラムを，テンプレートなどを利用して実際の実装に落とし込む作成方法などにより，ユーザーの記述と実際の実行コードの間の距離が長くなってきている．このため様々なインターフェースやＡＰＩによるプログラムのスタック構造が構築され，結果としてプログラムのパス長(path length)の増大を引き起こし，プログラムの実行速度の低下を招いている．

一般に，実行速度向上を考慮したプログラム作法は，プログラムの構造を複雑化し，プログラムの生産性やメンテナンス性を低下させる要因となる．生産性やメンテナンス性は，プログラムの規模が大きくなればなるほど重要となるため，プログラムの規模が大きくなるほどこのような複雑化を増長する記述方法は避けられる傾向にある．また現実問題として，人手によるプログラムのチューニングは膨大な時間と専門的な知識を要するため，特に性能向上が必要とされる部分に対してしか，人手によるプログラムの書き換えを行うことができていない．このように巨大化・複雑化するアプリケーションは，特にプログラムの開発効率やメンテナンスを重視するために判読性や保守性が優先されたプログラミング作法がとられる．結果として，多くの場合，実行時のオーバヘッドの増大や非効率的なプログラムとなってしまう．また，保守・管理コストを低くする観点からコードのバージョンは少ない方がよく，高速化のために特定の環境に特化したバージョンのコードを生成するようなアプローチは一般に行われ難い．

このように巨大化・複雑化したプログラムに対しては，コンパイラの最適化処理やランタイムシステムによるサポートによる高速化の実現が重要となる．例えば，コンパイラによる最適化によって，断片化して記述されたプログラムはメソッドインライニングなどを用いて大きなコンパイル単位にまとめられ，プログラム変換や冗長な処理の除去などによりパス長を短縮し，高速なコードを生成できる．またランタイムシステムは，実行時のロック処理やデータ入出力処理(I/O 処理)などを高速化する．

しかしながら，従来のプログラムの静的解析による最適化の限界がでてきている．例えば，オブジェクト指向プログラミングにより，プログラムは多数の小さなメソッドにより構成される傾向にある．このようなプログラムでは，個々のメソッドをいくら最適化しても，全体の速度向上への寄与は小さい．また，１つのメソッドの最適化についてみても，静的解析に基づく最適化では，プログラムの全ての実行可能性に対して基本的に平等に性能向上を実現するように処理が実施される．実際，多くのプログラムでは，実行される部分は一部分であり，残りの部分は例外処理など特殊な状態に対処する

ためのプログラムで，ほとんどの場合に実行されないものも多い．このようなほとんど実行されない実行経路（rare path）を最適化によって高速化しても，プログラムの速度向上にはほとんど寄与しない．また，実行のフェーズや入力データの違いでもプログラムの振る舞いが変化する場合も存在する．静的解析に基づく最適化では，このような状況に十分に対応することが困難である．結果として，最適化の機会を逸したり，効率の悪い最適化を実施したりしてしまう．

この問題を解決する１つの手段として，プロファイルを用いた最適化手法がある．プロファイルはプログラムの実行時に収集される，プログラムのさまざまな挙動を記録した情報である．実際，プログラム中の条件分岐の分岐回数，各関数の実行回数や変数のアクセス回数，変数の値の分布など，さまざまな情報がプロファイルとして収集されている．例えば，Fisher が提案した Trace Scheduling[39]では，プログラムの実行情報から実行頻度の高い経路を Trace として抽出し，この Trace の実行がもっとも効率的になるようにプログラムを変形する．場合によってはこの Trace 以外の部分の実行が遅くなるような変形も起こるが，Trace 部分の実行の速度向上によって，プログラム全体としても速度向上を得ることができる．このようにプログラムの実際の実行状態を把握し，その実行頻度の差に基づいて実行頻度の高い部分を優先した高速化を行うことで，従来の静的最適化よりも優れた性能を引き出すようにプログラムを変換することができる．

プロファイルを用いたプログラムの最適化のアプローチは，プロファイルの収集方法と最適化の実施方法によって，表１．１のように６つに分類できる．プロファイルは大きく分けて，オフラインプロファイル（Offline profile）とオンラインプロファイル（Online profile）に分類できる．オフラインプロファイルは，対象とするプログラムをプロファイル収集のために事前に１回予備実行を行い，その実行中に収集されたプロファイルである．最適化はこのプロファイルを用いて実施され，その結果生成されたコードを用いて実際の対象プログラムの本番実行を行う．オフラインプロファイルを用いた最適化では，プログラムの全実行を通じた振る舞いを収集できるために，大域的な最適化などを実施できるが，プロファイルで記述されるプログラムの特性と実際のプログラムの挙動が異なった場合に速度低下を引き起こしてしまう問題をもつ．一方オンラインプロファイルは，対象とするプログラムの本番実行中にプロファイルを収集し，そのプロファイルを用いて同じ実行中に JIT コンパイラやランタイムシステムでプロファイルを用いた最適化を実施する．オンラインプロファイルを用いた最適化では，プログラムの実行中に収集したプロファイルを利用して最適化を行うため，プロファイルで示される特性は必ず実際のプログラムの挙動となるが，プロファイルの収集はプログラムの既実行部分に対してのものなので，プログラムの実行挙動の変化などが起こった場合対処する必要がある．さらにプログラムの本実行中にプロファイル収集を行うために，その収集コストや使用するメモリなどの計算機資源に制約を受ける欠点もある．

一方，プロファイルを用いた最適化を大別すると，人手によるアプリケーションの書き換え（Hand-rewriting），コンパイラによる最適化（Compiler optimization），ランタイムシステム（Runtime system）に分類できる．人手によるアプリケーションの書き換えでは，プログラムで使用するアルゴリズムやプログラムモデルなどより大きなレベルでの変換が行えるために，非常に大きな効果を出す

ことが可能である反面，人手による作業のためにコストと時間がかかるために，適用範囲が限定されてしまう．コンパイラによる最適化は，与えられたアルゴリズムの意味を変えない範囲でしか最適化を行うことができないが，広範囲にわたって最適化を適用できる．ランタイムシステムは，ロック処理やI/O処理などのランタイムシステムを利用する処理に対して，プロファイルを用いて高速化を実現する．適用される処理は限定されるが，一般にランタイムシステムで行われる処理はその実行コストが高いので，最適化による効果は大きい．

表１.１　プロファイルを用いた最適化の分類

	Hand-rewriting	Compiler optimization	Runtime system
Offline profile	Programmer	Static compiler	Runtime system
Online profile	NA	JIT compiler	Runtime system

このように，プロファイルを用いた最適化は，それぞれに欠点も持つが適切に使用することで従来の静的最適化ではできなかった最適化を実現することが可能である．特に近年のアプリケーションのように複雑化するプログラムでは，プロファイルを活用してその実行挙動を把握し，その情報を活用してコンパイラやランタイムシステムによる高速化を実現することが非常に重要となってきている．

１．２　研究の目的

本研究は，プロファイル情報を用いて大規模アプリケーションの高速化を実現することを目的とする．前述の通り，大規模アプリケーションの高速化は，コンパイラやツールを用いてアプリケーションの事前書き換えなしに実現できることが重要である．このことから，本論文では，オンラインプロファイルを用いた最適化のアプローチ（Online profile + JIT compiler, Online profile + Runtime System）と，オフラインプロファイルを用いたランタイムシステムの高速化のアプローチ(Offline profile + Runtime System)を対象として研究を行った．

オンラインプロファイルを用いた最適化のアプローチに対して，本論文では，プロファイルの収集コストを抑えながら最適化処理に必要な高い精度のプロファイルを収集する方法の実現という課題を中心に研究を行った．オンラインプロファイルを用いた最適化では，少ない実行時コストで高い性能を引き出すことが重要となる．これは，オンラインプロファイルのアプローチでは，プロファイルを収集する処理から最適化コードを生成するコンパイル処理までのすべての処理の実行コストがアプリケーションの実行に影響を与えてしまうからである．特にプロファイルの収集コストの軽減は非常に重要な問題である．というのも，コンパイル処理は，一般にアプリケーション中で非常に実行時間を消費している部分に限定して行われるので，多少の実行時間を消費してもその結果得られる速度向上が十分大きければ結果としてコンパイルのコストは相殺される．一方，プロファイル収集処理は，収

集されるプロファイルの精度が生成されるコードの性能に大きく影響することから，高い精度のプロファイルの収集が重要である．しかしながら，プロファイルの収集もコンパイルと同様にプログラムの実行に対してオーバヘッドとなるため，高い精度のみを追求してしまうとかえってアプリケーションの性能低下を引き起こしてしまう．もし収集コストを下げるために不十分な量のプロファイルしか収集しないと，プロファイルの精度が低下してしまい，結果として最適化による十分な速度向上ができなくなってしまう．従って，プロファイルの収集コストを抑えながら最適化処理に必要十分な高い精度のプロファイルを収集することは非常に重要な課題となる．本論文では，特に軽量なプロファイル収集手法とそれを用いた最適化手法の提案を行う．

一方，オフラインプロファイルを用いた最適化のアプローチに対して，本論文では，オフラインプロファイルを用いて時間軸方向での大域的な情報を用いたランタイムシステムの高速化という課題を中心に研究を行った．一般に，オフラインプロファイルは，そこで記録されたアプリケーションの動作の再現性が大きな課題となっている．しかし，オンラインプロファイルと異なり，プログラム全体の実行挙動の情報を利用できるオフラインプロファイルは，より大きな高速化を実現できる非常に重要な手段である．本研究では，アプリケーションの実行における特定のフェーズに非常に高い再現性のあることを利用して，オフラインプロファイルを用いたランタイムシステムの高速化を実現する方法を検討した．ランタイムシステムは，時系列で発生するイベントに対して順次処理を実施する．従って，オフラインプロファイルを用いてこの時系列イベントを求め，プロファイルの大域的な解析を用いて実行時のランタイムシステムの効率的な動作を決定することで，非常に高い効果を得ることが可能となる．本論文では，特にアプリケーションを変更せずにプログラムのデータ入力処理を高速化する手法の提案を行う．

以下に，本研究で提案する内容とその結果を示す．

１．２．１　実行時分岐予測を用いた多分岐処理の高速化

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る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．

１．２．２　実行時経路情報の構造的収集

実行頻度の高い経路情報(Hot path profile)を収集することは，目的のプログラムを効果的に変形したり最適化したりするために重要である．しかしながら，従来のパスプロファイル収集手法では，精度の高いプロファイルを収集するためのコストが高いために Just-in-Time (JIT)コンパイラで利用することが困難であった．我々はこの問題を解決する手法として，JIT コンパイラに適した実行時経路情報の構造的収集手法(Structural path profiling)を提案する．キーとなるアイデアは，目的のメソッドをループの階層構造に従って分割したグラフで表し，それぞれのグラフに対して独立にプロファイルを収集することである．これよって，精度の高いパスプロファイルを低いオーバヘッドで効率的に収集することが可能となる．本手法を IBM Java Just-in-Time コンパイラに実装して評価したところ，プロファイルが頻繁に実施される状況においても，オフラインプロファイルと比較して約 90%の精度のパスプロファイル情報を，平均で 2-3%のオーバヘッドで収集できることを示した．

１．２．３　静的プロファイルを用いたプログラム起動処理の高速化

近年の大規模アプリケーションの起動時間の増加は，アプリケーションの開発や運用において大きな問題となっている．特に起動中のファイル読み込みにより発生する I/O 待ち時間は起動時間を増加させる要因の１つである．本論文では，この I/O 待ち時間を短縮する方法として，対象アプリケーションにおける I/O 処理の静的プロファイルを用いてプリフェッチ処理を実行する手法(Scenario-Based Prefetching) を提案する．従来プリフェッチ処理を用いたファイル読み込み処理の最適化手法が多数研究されてきている．しかしその多くは汎用的な手法である一方で，大規模アプリケーションの起動時のように，ファイル読み込み処理が多数のコンポーネントに分散し，多数のファイルをランダムアクセス的に読み込むような状況下ではあまり効果を得ることができなかった．これに対して，我々は大規模アプリケーションの起動時の I/O 処理がほとんど同じ順序で行われている点に着目し，(1)事前に収集された静的プロファイルを用いてアプリケーションの挙動に合わせてプリフェッチ処理をスケジューリングしたシナリオ生成しておき，(2)このシナリオを用いて実行時にプリフェッチ処理を実施することで，アプリケーションを変更することなしに起動時間を大きく削減することを可能とした．本手法を幾つかのアプリケーションで評価した結果，起動時間を約 30%削減できることを示した．

１．３　本論文の構成

本論文の構成は，以下の通りである．まず第１章において，実行時分岐予測を用いた分岐処理の高速化手法について述べる．特に，プロファイル収集のオーバヘッドを軽減するために，少量プロファイルの収集によるプロファイルの精度と最適化の効果の関連，および多分岐命令の最適化手法につい

て述べる．続いて第２章において，オンラインパスプロファイルの構造的収集手法について説明する．パスプロファイルはエッジプロファイルよりも収集コストが高くなってしまう問題がある．本章では収集コストを抑えながら高い精度のパスプロファイルを収集する手法を提案する．第３章 下のでは，静的プロファイルを用いたプログラム起動処理の高速化について述べる．オフラインプロファイルを用いてファイルの入力処理を大域的に最適化することで，プログラムの起動処理を高速化する手法を低ｓんする．最後に第４章 下のにおいて，本研究のまとめを述べる．

第２章　実行時分岐予測を用いた分岐処理の高速化

２．１　はじめに

プログラムの更なる高速化を目指して，プログラムの実行時の挙動情報（プロファイル情報）を利用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プログラム中の全ての経路が同等以上の性能になることをよう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，非常に実行頻度の高い経路が存在する一方で，例外処理などのように多くの場合には実行されない処理が存在する．このような実行頻度の偏りによって，実行頻度の高い経路の実行速度が向上することによって，実行頻度の低い経路の処理が多少遅くなっても全体として実行速度を向上することができる．

プログラム中の経路の実行頻度を求める方法として，これまで，基本ブロックの実行頻度を求める方法[5][6]，実行経路の頻度を求める方法[3][37]，実行頻度の高い経路を求める方法[4]などが提案されている．さらに，このようなプロファイル情報を用いて最適化を行う手法も提案されている．例えば，頻度の高い経路を優先したプログラムの変形[7][9][10][11][12][13][14][16]，基本ブロックの並べ替え[8]，多分岐命令の最適化[16]など様々な最適化手法が提案されている．

このようなプロファイルを用いた最適化手法で問題となるのは，どのようにプロファイル情報を収集するかという点である．通常，プロファイル情報の収集は，事前にプロファイル収集用の実行を行い，収集したプロファイル情報を用いてプログラムを最適化した後，その最適化プログラムを用いてプログラムの実行を行う．もしプロファイルの収集のためのプログラムの実行を実際のプログラムの実行と全く同じ条件で実施でき，かつプログラムの挙動が実行毎に変化しないのであれば，最適化処理ではプロファイルの情報を完全に信頼して用いることができる．収集したプロファイルの示す挙動に対して最高性能を出せるようにプログラムを最適化することで，結果として非常に高性能なコードの生成が可能となる．しかしながら，実際の多くの場合には，プロファイルの収集のための実行が実際に目的とする実行とは異なる条件でしか実行できなかったり，あるいはプログラムの挙動自体が何らかの外的要因などにより実行毎に異なってしまったりすると，収集されたプロファイル情報が示すプログラムの実行時の挙動が目的とする実行時の挙動を必ずしも正しく表さない状況が起こしまう．このような状況では，収集したプロファイルに完全に従ってプログラムを最適化してしまうと，プロファイルと異なる挙動が生じたときに，プログラムの実行速度が著しく低下してしまう状況が起こってしまうことがある．この，収集したプロファイルと実際の挙動のずれの発生が，プロファイルを用いた最適化手法における最大の課題となる．

このような課題に対する１つの解決策として，JIT コンパイラなどの動的コンパイラを用いたアプローチがあげられる．近年の Java 言語[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][19][20][21][22]．動的コンパイラ環境では，プロファイルの収集とそれによる最適化コードの使用をプログラムの同一の実行中に行うことができるため，先に述べたようなプロファイルと実際のプログラムの挙動のずれの問

題が生じない.

しかしながら，動的コンパイラ環境では別の問題が存在する．文献[4]でも述べられているように，従来の履歴情報収集（スタティックプロファイル）がプログラム全体の完全な挙動の情報を収集するのに対して，動的コンパイル環境での履歴情報収集（オンラインプロファイル）はプログラムの途中までの実行状態を表しているに過ぎない．つまり，オンラインプロファイルを使った最適化においては，プログラムの途中までの実行状態を用いてそれ以降の実行を予測しないとならず，プロファイル情報の収集時期によって精度が変動してしまう．またプロファイル情報の収集に必要な様々なコストは全てプログラムの実行時間に影響する点も問題となる．例えばプロファイルを収集するためのコードを実行するコストは対象とするプログラムの実行に対するオーバヘッドとなって速度低下を引き起こすし，プロファイル情報を格納するために必要な領域を確保するためのメモリ資源の消費はキャッシュミスや不要なスワッピングを引き起こす要因になる．このため，プログラムの実行速度の観点では，プロファイル情報の収集にかかるオーバヘッドよりも収集された情報を用いて実施される最適化による速度向上が上回ることが必要となる．同様にメモリ使用量の観点では，プロファイルに使用可能なメモリ使用量はプログラムの実行に影響を与えない範囲に制限される．従って，既存の多くの動的コンパイラでは，メソッド呼び出しや実行頻度の高いメソッドの検出など，比較的収集する情報の単位が大きく収集オーバヘッドがあまりプログラムの実行に影響しないものに限って実施されているのが現状である.

本章の目的は，プログラム中の分岐命令に対して連続する固定回数の分岐履歴情報を収集する手法の有効性を示すとともに，特に多分岐命令に対してプロファイル情報を用いた最適化手法を提案することである．分岐履歴情報の収集では，プロファイルの収集を連続する数回分に限定する方法[35]を用いて収集されるプロファイルの量を制御することができる．収集量を少なくすることで，収集のためのオーバヘッドが実行に影響しないようにすることが可能となる．しかし，特にプログラムの開始時点の挙動はそれ以降と異なる場合がしばしば存在するため，単純にプログラム開始時より固定数のプロファイルを収集するのではプロファイルの精度が低下してしまう場合がある．本章では，プロファイルの収集開始時期を，各メソッドがあらかじめ規定された回数実行されてから行うことにより，この初期状態の挙動を避けてプログラムの定常状態のプロファイルを収集することでより高い精度のプロファイルを収集する方法を示す.

収集されるプロファイルの量は，収集オーバヘッドの問題以外に，そのプロファイルを格納する領域のサイズとしても制約を受ける．一般に，プログラム中の条件分岐は平均すると数命令毎に出現すると言われており，１つの条件分岐辺りに割り当てる領域が小さくてもその数の多さによって全体として非常に多くの格納領域を必要としてしまう．例えば，４回分の分岐履歴を収集する場合に必要な領域は１つの条件分岐辺り４bit であるが，２５５回分の分岐履歴を収集する場合だと，単純に分岐回数のみを保存したとしても 16bit の領域が必要となり，結果として４倍の領域が必要となる．従来手法ではこのようなプロファイルを保存する領域のサイズは考慮されていなかったが，現実の製品レベルでの実装ではメモリの使用に大きな制限をうける．このため，許容されるメモリ使用量の制約の中

で，十分な最適化効果を導き出せるプロファイルを収集できるかどうかが重要となる．実際にベンチマークプログラムを解析した結果，現実的には２分岐命令は１名例あたり４ビット，多分岐命令は１つのカウンタあたり４ビットの領域が上限値となることがわかった．

我々は，IBM Java virtual machine 上で分岐命令のプロファイルを収集する手法を実装し，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，２分岐命令に対しては連続する４回の履歴情報の収集で約 87%の精度が得られることがわかった．さらにプログラムをある程度実行してからプロファイル情報の収集を開始することでこの精度を約 89%に向上できることを示した．他方，多分岐命令ではカウンタ辺り４ビットの領域を使用することで約 80%の精度が得られることがわかった．

さらに多分岐命令に対するプロファイル情報を用いた最適化手法として，本論文では期待値モデルによる線形探索と２分探索を組み合わせた高速な探索手法を提案する．実行履歴を用いた多分岐命令の最適化手法として，従来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探索順序を並べ替える手法[16]が提案されている．しかし，この手法を用いると，多分岐命令が常に２分岐命令を用いた線形探索として実装されるため，分岐確率に偏りがほとんどないようなプロファイルでは，２分探索や間接テーブル分岐など従来の静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）よりも実行速度が遅くなってしまう問題があった．本章では，従来の静的探索手法とプロファイルを用いた線形探索を組み合わせることで，あらゆる場合においても高速な多分岐命令の処理を可能とするコード生成手法を提案する．本手法を IBM Java Just-in-Time コンパイラに実装して評価した結果，従来の静的探索手法に対して最大で約 2.4%，平均で約 0.5%の速度向上が得られることを示した．

以下では，まず２．２節で，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２．３節では，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，２．４節で実行履歴を用いた最適化手法の評価を行う．

２．２　分岐命令の実行履歴収集

予備実行を用いた従来のプロファイルの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や外的要因によるプログラムの実行挙動の変化によって，プロファイル情報が本実行の挙動を正確に表さない場合であった．動的コンパイラ環境においては，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，基本的にこのような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など，どのようにプロファイルを収集するかという方法である．

既存の動的コンパイラの多くは実行履歴を用いた最適化が行われているが，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，１回当たりの処理コストの高いものをターゲットとしたものが多い．他方，分岐命令や基本ブロックの実行回数などプログラムの実行履歴に

対応するプロファイル情報は，収集するポイントが多い反面，分岐命令など個々の対象の実行コストが小さいために，プロファイルに要するコストがプログラム全体の実行速度の低下に及ぼす影響は大きくなってしまう．

本節では，プロファイルの収集コストを低く抑え，プログラムの実行速度に影響を与えないことを目的に，ごく少数の実行履歴情報の収集を行う処理の実装方法について述べる．最初に定数回のプロファイルを収集する基本的な仕組みを説明した後，製品レベルの実装において可能となるプロファイル保持用の領域の大きさについて議論する．その後，２分岐命令と多分岐命令について，ベンチマークでの評価を行うことで，少量の実行履歴情報の収集により得られるプロファイルの精度について議論する．

２．２．１　履歴情報収集処理の実装

まず定数回のプロファイル情報の収集を行う実装方法を図２．１に示す．この実装により，あらかじめ与えられた回数のプロファイルを収集した後，プロファイル収集コードへの分岐命令を書き換えることによってプロファイルを自立的に終了することができる．プロファイル収集コードは，目的とするプログラムの内部に埋め込まれていると不要にコードサイズを増加させたり命令キャッシュミス率を増加させたりする要因となるため，このプロファイル収集用コードは正規の処理とは別の領域に作成する．プロファイル収集の開始は，プロファイルを収集する地点を無条件分岐命令に書き換えることで行う．図の例では，初期状態では NOP 命令としていた領域を”goto prof_code”という命令に書き換えることで実施している．プロファイル収集が開始すると，プロファイル収集コードの中で収集したプロファイルの数（*counter*）をカウントアップしながらプロファイルの収集を実施する．プロファイルの収集数が予め定めた回数（*THRESHOLD*）に達した時点でプロファイル収集を停止するための処理（remove_goto()）を実行する．例ではこの処理ではプロファイルを収集する地点に挿入した無条件分岐命令（goto prof_code）を NOP 命令に書き換えることでプロファイル収集コードへの分岐を無くすことで実現している．この書き換えによってプロファイルの収集を終了した後は本来の実行を妨げる命令が全くなくなるので，プロファイル収集コードを実施したことによる影響を全く与えないことを保証できる．

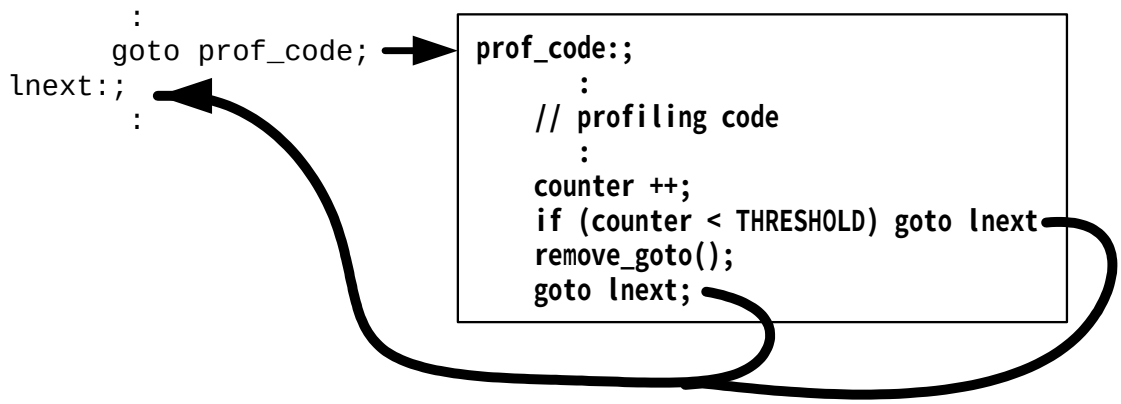


図２．１　履歴情報収集処理

２．２．２　分岐履歴格納のための領域サイズ

プログラム中の条件分岐命令の数は多く，それら１つ辺りに割り当てるプロファイル格納領域の大きさが少量であってもプログラム全体で使用されるメモリ量は無視できない量になってしまう．ここでは，ベンチマークでの評価を通じて実際にどの程度の領域の割り当てが可能であるかを求める．

２分岐命令に対して割り当てられるプロファイル格納領域がプログラムに与える影響について，SPECjvm98 ベンチマークを用いて評価した結果を表２．１に示す．表２．１は，SPECjvm98 ベンチマークの各プログラムに対して，２分岐命令の総数，全メソッドのバイトコードの合計サイズ，１つの分岐辺りのプロファイルサイズが４ビットの場合と４バイトの場合におけるプロファイル格納領域のサイズとバイトコードサイズに対する割合を示している．この表からわかる通り，１つの分岐あたり４bit のプロファイルを収集した場合は，バイトコードサイズに対するプロファイル格納領域のサイズが最大で約５％であるのに対して，１つの分岐辺り４byte のプロファイルを収集する場合は最大で４０％の領域が必要となってしまうことがわかる．

次に，多分岐命令の分岐情報の収集に必要な領域について考える．２．２．４ で後述する通り，多分岐命令は１命令辺りその分岐数に応じた数のカウンタが必要となる．このため，多分岐命令については命令数ではなく全体で必要となるカウンタ数に対しての評価を行った．表２．２は，SPECjvm98 ベンチマークの各プログラムに対して多分岐命令に必要なカウンタの総数，全メソッドのバイトコードの合計サイズ，１つのカウンタ辺りのプロファイルサイズが４ビットの場合と４バイトの場合におけるプロファイル格納領域のサイズとバイトコードサイズに対する割合を示している．この表からわかる通り，１つのカウンタあたり４bit のプロファイルを収集する場合は，バイトコードサイズに対するプロファイル格納領域のサイズが最大で約 1.7%であるのに対して，１つの分岐辺り４byte のプロファイルを収集する場合は最大で約 14%の領域が必要となってしまう．

製品レベルでの実装における許容範囲という観点を考慮すると，現実的にプロファイルの格納領域に割り当てられる領域のサイズはバイトコードサイズの 10%が上限値となると考える．このことから，プロファイルに使用可能な領域のサイズは２分岐命令では１つの分岐辺り 4bit，多分岐命令では１つのカウンタあたり 4bit の領域割り当てが上限値となる．

表２．１　２分岐履歴情報に必要な領域のサイズに関する評価

	num. of branches	total bytecode size (byte)	profile size			
			4 bit / branch		4 byte / branch	
			byte	%	byte	%
mtrt	628	16910	314	1.86%	2512	14.86%
jess	2043	21221	1022	4.81%	8172	38.51%
compress	343	6796	172	2.52%	1372	20.19%
db	641	9684	321	3.31%	2564	26.48%
mpegaudio	766	18945	383	2.02%	3064	16.17%
jack	3993	45839	1997	4.36%	15972	34.84%
javac	3967	71903	1984	2.76%	15868	22.07%

表２.２　２分岐履歴情報に必要な領域のサイズに関する評価

	num. of counters for switches	total bytecode size (byte)	profile size			
			4 bit / counters		4 byte / counters	
			byte	%	byte	%
mtrt	135	16910	68	0.40%	540	3.19%
jess	384	21221	192	0.90%	1536	7.24%
compress	64	6796	32	0.47%	256	3.77%
db	141	9684	71	0.73%	564	5.82%
mpegaudio	75	18945	38	0.20%	300	1.58%
jack	544	45839	272	0.59%	2176	4.75%
javac	2445	71903	1223	1.70%	9780	13.60%

２．２．３　２分岐命令の実行履歴収集

ここでは，まず２分岐命令に対する分岐履歴の収集方法について議論した後，SPECjvm98 ベンチマークを用いて各分岐に対して１～４個の分岐履歴を収集した場合の分岐予測の精度について評価した結果を示す.

本論文で用いる２分岐命令に対する分岐履歴情報の実装を図２．２に示す．この実装では，連続するＮ回の分岐履歴情報を，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のビット列として表現する．実際の実装では，最初にＮビットを，最下位ビットを０，それ以外を１という状態（１１…１１０）で初期化する．分岐履歴の記録は全体を上位ビット方向にシフトした後最下位ビットに記録する．またシフトする前の状態で最上位ビットが０であれば，情報収集地点に挿入した分岐命令を除去することでプロファイルの収集を終了させる．この方法によって，Ｎ個の分岐履歴の収集において収集回数と収集履歴をＮビットの領域で実装することができる.

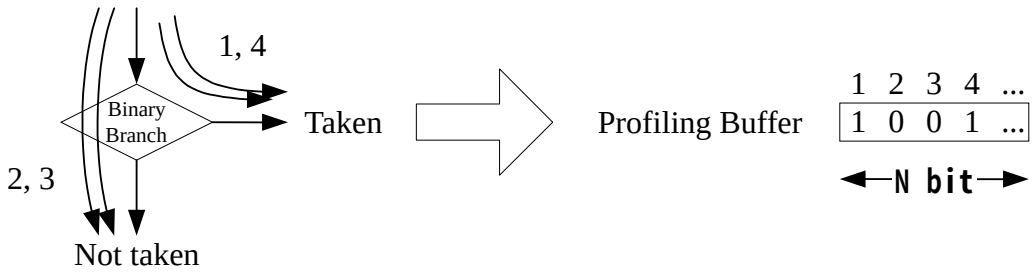


図２.２　２分岐命令の履歴情報の収集

この実装を用いて，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評価した結果を表２．４に示す．評価は，プログラムの開始と同時にプロファイルの収集を開始した場合としての初期状態履歴（Initial State Profile）と，メソッドが１０００回実行された時点でプロファイルの収集を開始した場合としての定常状態履歴（Steady State Profile）の２つの開始時期に対して，履

歴サイズに対する２分岐命令の分岐予測の精度を示している．分岐予測は，表２．３に示すように，分岐履歴中の分岐方向が全て同じ場合はその方向に分岐すると予測し，それ以外の場合は予測なしと判定する．分岐予測の精度は，分岐予測がおこなわれた分岐命令に対して，その分岐のプログラムの全実行を通じての分岐頻度を求め，この全実行における分岐頻度の高い方向と予測の分岐方向とを比較し，この方向が一致する分岐が全体に占める割合として計算した．

表２．４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る舞いの違いが存在することを示すものである．

一方，compress は適中率が約 68%と低いことがわかる．Compress のプロファイルを調査した結果，この原因として後方分岐の適中率が低いことが主な要因であることがわかった．具体的には，分岐しないとした予測した後方分岐において，実行の約 87%が分岐しており，この予測の失敗が全体の適中率を大きく引き下げる要因となっていた．さらに詳細に調査したところ，メソッドの実行が開始を基準とした場合に，最初の４回の実行では分岐しないが，プログラムの全実行を通じては非常に多くの割合で分岐が行われる後方分岐が複数存在していることがわかった．今回の実装では，条件分岐のプロファイルの開始は，つねにメソッドの実行が開始するタイミングで行っていたために，定常状態でのプロファイル収集での状況を改善することができなかったのである．この問題は，メソッドの実行開始をトリガーとしてプロファイルを収拾する方法をとる場合に共通して生じる問題である．特に今回の実装のように連続する定数個のプロファイルを収集する方法では，局所的な挙動の偏りに影響されやすいために，このような状況をうまく扱うことができない状況が起きてしまう．この問題に対する対処法としては，例えば Arnold らが提案しているようなサンプリング手法を用いて一定間隔ごとのプロファイルを収集する方法[25]などを用いる手法が考えられる．あるいは，一般に多く採用されている「後方分岐は常に分岐する」という単純なヒューリスティックを用いて予測に強制的に変更する方法も考えられる．このヒューリスティックにより補正を実施した場合，compress の適中率が約 80%に向上することもわかった．

表２.３　履歴情報からの分岐予測

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

表２.４　２分岐命令の分岐予測の精度

	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	78.9	93.56	92.8
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	92.6	85.03	91.03	92.32	92.75

２．２．４　多分岐命令の実行履歴収集

次に，多分岐命令に対する分岐履歴情報の収集方法を説明した後，ベンチマークプログラムを用いてその収集方法で収集されたプロファイルの精度を評価した結果を示す．多分岐命令の分岐履歴情報は，２分岐命令と異なり，多分岐命令の各分岐に対して１つずつカウンタを用意して分岐回数を数えることで，各分岐に対する実行頻度の比として収集する．

Java 言語の多分岐命令である switch 文では,case 文により比較定数値と対応する処理を定義する．Switch 命令は,実際にはバイトコード命令である Tableswitch 命令か Lookupswitch 命令に変換されて実行される．Tableswitch 命令は，整数区間とその区間の各値に対する分岐先アドレス列として定義される．比較定数値が連続した整数区間として表されるため，分岐先アドレスを保持する間接テーブルを用いた実装により高速な処理が可能である．一方 Lookupswitch 命令は，分岐先を，定数値と分岐先アドレスの対として表現する．比較定数値が離散値になるため，分岐先を決定するための探索処理が必要となる．したがって，本論文で提案する手法は，特に断らない場合はこの Lookupswitch 命令を最適化の対象とするものとする．

Lookupswitch命令に対する探索処理は，実行時に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への分岐を，存在しなければdefault文[1]で指定された処理への分岐を実行する．比較定数値が離散値であるために，比較定数値の間の領域はdefault文で指定された処理への分岐となる．このことは，比較定数値の間の領域は，整数区間として暗黙に定義された分岐条件と考えることもできる． したがって，探索条件式の集合は以下のように定義することができる．

定義： switch 文の探索条件式集合は，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して定義される．

[1] default文が存在しない場合はswitch文の次の文を実行する．

Switch 命令の定義から，探索条件式の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いことが保障される．履歴情報の収集処理では，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．従って，必要カウンタ数は case 文の数を N 個とすると最大(2N+1)個となる．またこれとは別に収集した履歴情報の総数を数えるカウンタを１つ割り当てる．図２．３に．switch 文に対するカウンタ割り当ての例を示す．

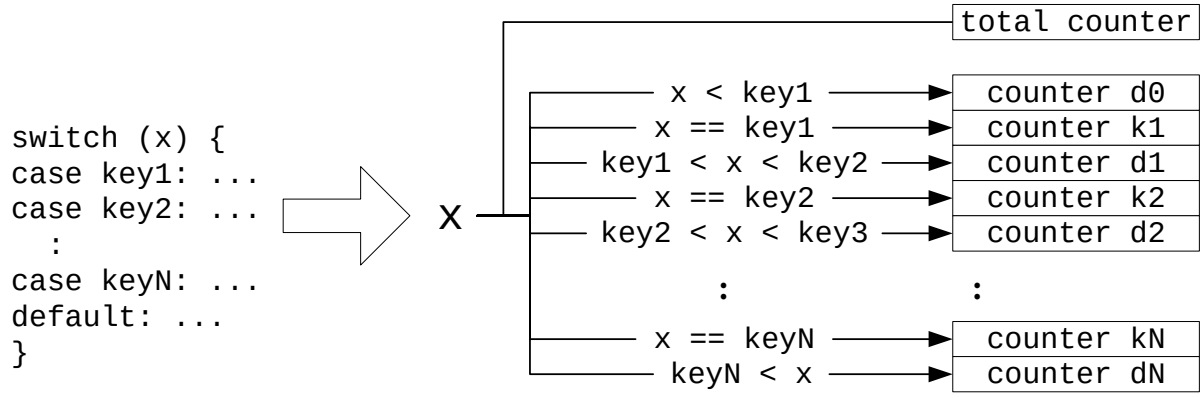


図２.３　多分岐命令の履歴情報収集方法の例

分岐履歴情報の収集では，分岐が行われる毎に，分岐総数のカウンタとターゲットとなる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．２．２．２ で述べたとおり，各カウンタのサイズの上限値は４bit であるので，ここでは最大値の 4bit で実装を行った．また，収集する履歴情報の数をカウントするカウンタは 8bit とした．履歴情報の収集は，探索条件式に割り当てられているカウンタのいずれかがオーバフローした時点か，または収集した分岐履歴情報の数が比較定数値数の２倍と 80 のうちの大きい方の値に達した時点で終了する．多分岐命令の分岐予測は，収集された履歴情報から，各探索条件式に割り当てられたカウンタの保持する値を用いて計算される分岐比率を計算することで求める．

この実装を用いて多分岐命令の分岐予測を評価した結果を表２．５に示す．分岐予測の評価は，履歴情報に基づく分岐予測の結果と，プログラムを完全に実行した場合の分岐比率との相関係数を 100 倍したものを予測の精度として使用した．表２．５では，各プログラムに対して，多分岐命令の個々の予測の精度の単純平均を計算した値（Static）と，個々の多分岐命令の精度を実行回数により重み付けして平均した値（Dynamic）を表している．この評価結果より，mtrt の予測の精度が著しく低いことがわかる．また，実行回数で重み付けをした場合においては，jess も予測の精度が低くなっている．この２つのプログラムについてその原因を調査した結果，特に以下の２点によって予測の精度が下がっていることがわかった．

(1) 全探索条件式の実際の実行頻度にほとんど差がない場合．

(2) 多分岐命令を含むメソッドが複数箇所から呼び出され，呼び出し箇所毎に挙動が異なる場合

前者のようにほとんど分岐の頻度がほとんどフラットである場合には，履歴収集期間の開始時期によって収集される履歴情報に若干の変動が生じてしまう．つまり探索条件の実行回数が１違うことが探索条件の比率の大小関係を変化させてしてしまい，結果として逆相関と判定されてしまったことが原因である．現実的には，実行頻度にほとんど差がないことから履歴情報を用いた最適化にはほとんど影響がないが，このような状況を正しく評価できる手法が必要といえる．

他方，後者は同一の分岐命令に対して複数の挙動が混じってしまう場合に起こる問題である．履歴情報の収集量が少ないと，履歴収集期間の開始時期によって収集される履歴情報の挙動が大きく異なってしまう．この問題に対する対処法としては，サンプリング手法[25]によって収集される履歴情報が特定の状況偏らないようにすることで問題を緩和することができる．

表２.５　多分岐命令の分岐予測と実際の分岐結果との相関を用いた精度の評価

Program	Static	Dynamic
mtrt	35.0%	57.4%
jess	79.8%	37.1%
compress	100.0%	100.0%
Db	80.0%	98.7%
mpegaudio	99.8%	99.4%
Jack	93.4%	91.8%
javac	74.2%	81.0%

２．２．５　低頻度実行経路解析による分岐予測の補正

実行頻度によるプロファイルの収集では，その収集期間中一度も実行されないという情報も最適化にとって有用な情報となる．全実行を通じて実行頻度の低い経路や全く実行されない低頻度実行経路（Rare path）は，収集されるプロファイルの期間や量によらず，ほとんどプロファイルが収集されない．このような領域は，その実行速度が多少遅くなってもプログラムの実行に与える影響が極めて少ない．このような領域を検出するための基本的なアイデアは，プロファイルを収集するためにメソッド中に配置されている様々なカウンタの値が，プロファイル収集期間を通じて全くカウントアップされないか，ごく少数の値であるようかカウンタを見つけることである．実際，本論文で使用している履歴情報収集のためコードは，それ自体の実行回数をカウントする仕組みを持っている．これらのカウンタが規定回数に達していなければ，そのカウンタが存在する位置は低頻度実行経路上に存在すると判断することができる（図２．４）．実際，カウンタが規定値に達してしまう場合，その規定値に達するまでの期間を知ることはできないが，逆に規定値に達していないという事実は，プログラムの開始時点以降の期間全てを通じての状態を表しており，極めて高い精度を持つ情報といえる．このような情報を収集するために，分岐命令の履歴情報収集のための実行カウンタ以外にも，メソッド呼び出しにおけるターゲットメソッドの実行回数や，バイトコードが quick 命令と鳴っているかどうかなどの情報をカウンタの代用として用いることができる．これらの命令が存在する位置に加えて，さらに

その命令が属する基本ブロックが後方支配する基本ロック，およびその基本ブロックが支配する基本ブロックは全て同様に低頻度実行経路上に存在すると定義できる．これをフロー解析により求めることで，より大きな領域に対して低頻度実行経路を検出することができる．

表２．６に，SPECjvm98 ベンチマークの各プログラムに対して，この低頻度実行経路の検出を実施した場合の結果を示す．表中の，**basic block ratio** は検出された経路を構成する基本ブロックのプログラム全体に占める割合を，**code ratio** は検出された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，低頻度実行経路へ分岐する分岐命令の分岐確率を表している．低頻度実行経路の適中率は，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，低頻度実行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．しかし，実際の最適化においては，他の経路の高速化のために，この低頻度実行経路に補償コードを挿入するなどの処理を実施できるため，多くの状況で有用な情報となる．

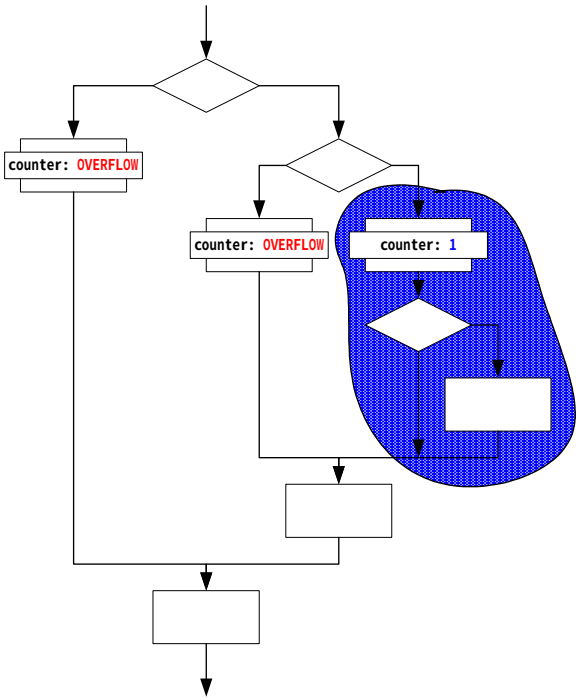


図２．４　低頻度実行経路の検出

表２.６ 希少実行経路の検出

Program	Basic block 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

２．３ 多分岐命令の最適化

本節では，Java 言語の多分岐命令である switch 文に対して本論文で提案する実行時履歴情報を用いた最適化手法について述べる．多分岐命令は，与えられた整数値に適合する処理を探索して実行する命令であり，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を生成することである．

既存の実行履歴情報を用いた多分岐命令の最適化手法として，履歴情報を用いて探索順序を決定した線形探索によって実現する手法[16]が提案されている．この方法では，多分岐命令を２分岐命令の線形探索として表現した後，探索条件の探索順序を，式(1)を用いて計算される確率的コストをもとに降順に並べ替えることで探索順序の最適化を実施する方法である．しかしながら，この手法では多分岐命令の実装を線形探索によって実装するために，分岐確率の偏りがほとんどない場合には，２分探索手法や間接テーブル分岐を用いた手法など従来のプロファイルを用いない場合の多分岐命令の処理手法[17]よりも平均的な分岐処理の時間が遅くなってしまう問題点があった．

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

この問題を解決する方法として，多分岐命令に対して線形探索と静的探索を組み合わせて探索する手法を提案する．基本となるアイデアは，全ての探索条件を線形探索で実行した場合の探索コストの期待値に対して，もっとも確率的コストの高い探索条件のみを線形探索として分離して処理する場合の探索コストの期待値を比較しながら線形探索により探索する条件を決定することである．

以下では，まず複数の探索条件が与えられた場合に，ある探索条件を線形探索として処理したほうが全体の期待コストを低くできるかどうかを判定するための期待値もモデルとその評価式を示す．その後，この評価式を使って，多分岐命令に対してプロファイルを用いた探索木を生成するアルゴリズムを示す．

２．３．１　判定式

ある探索条件の集合が与えられた場合，それらを静的探索により探索するコストをC_{orig}とする．この集合から１つの探索条件を取り出し，この探索条件の判定を独立して行う処理のモデル（図２．５(a)）を考える．このモデルは，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．図２．５(a)では，Test Conditionを実行するのに要する処理コストがC_{cmp}であり，この条件が成立した場合にさらに実施される探索処理（TRUE Process）にかかるコストがC_{true}であり，Test Conditionが不成立の場合にさらに実施される探索処理（FALSE Process）にかかるコストがC_{false}である．例えば，定数値keyとの比較で最終的な分岐先が決定する場合，成立時にはそれ以上の探索処理派必要無いのでC_{true}は０となる．もし条件成立時の探索が多段階に実施される場合，最初の条件が成立した後に次の条件の判定処理が必要となる．この最初の条件成立後の次の判定処理に必要なコストがC_{true}となる．

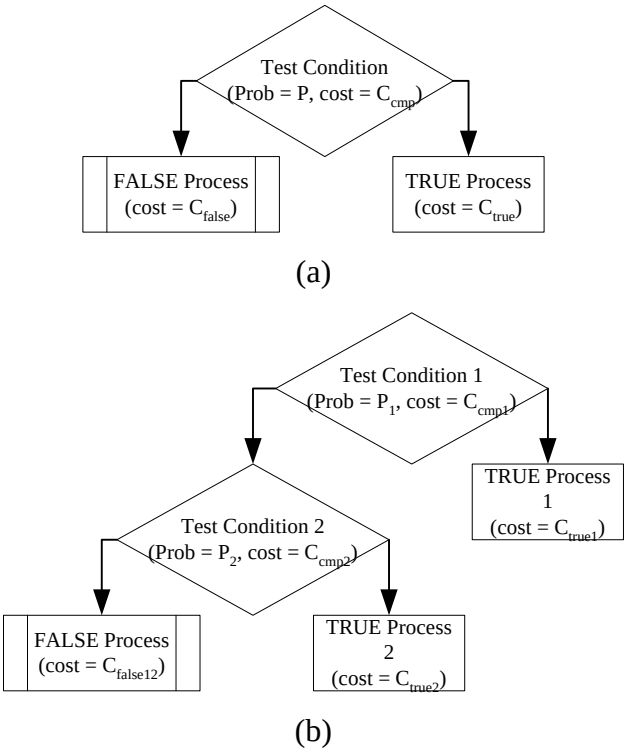


図２．５　条件判定モデル

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（２）で与えられる．この期待値は，最初の探索が確率Pで成立する場合，その探索に要するコストC_{cmp}に，その条件が成立した場合にかかるコスト$P * C_{true}$と成立しなかった場合にかかるコスト$(1-P) * C_{false}$を加えた値として計算される．

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

探索条件の集合より取り出した探索条件の判定を独立して行う処理の期待コストC_eが，全体を静的探索で行う場合の期待コストC_{orig}よりも小さくなること（式(3)）が，このような探索処理を実施するための必要条件となる．つまり，ある探索条件式がこの式を満たす限り，全体の探索処理にかかるコストの期待値は小さくなり，より高速な探索か実現できることを保証できる．

$$C_e > C_{orig} \tag{3}$$

次に，ある探索条件の集合が与えられ，そこから２つの探索条件を取り出した場合，どちらを先に判定するほうが探索のコストの期待値を小さくできるかについて考える．この場合のモデルを図２．５(b)に示す．２つの探索条件１，２が与えられたとする．このとき，各探索条件が実行される無条件確立をそれぞれ，P_1，P_2とする．また，２つの探索条件が１，２の順番で判定された場合の期待コストをC_{e12}，逆に２，１の順番で判定した場合をC_{e21}とする．さらに両方の判定条件のいずれも成立しない場合の期待コストを$C_{false12}$とする．このとき期待コストC_{e12}およびC_{e21}は，それぞれ式(4a)および(4b)により計算できる．

$$\begin{aligned} C_{e12} = {} & C_{cmp1} + P_1 * C_{true1} \\ & + (1-P_1) * C_{cmp2} \\ & + P_2 * C_{ture2} \\ & + (1-P_1-P_2) * C_{false12} \end{aligned} \tag{4a}$$

$$\begin{aligned} C_{e21} = {} & C_{cmp2} + P_2 * C_{true2} \\ & + (1-P_2) * C_{cmp1} \\ & + P_1 * C_{ture1} \\ & + (1-P_1-P_2) * C_{false12} \end{aligned} \tag{4b}$$

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．この不等式を式(4a)(4b)を用いて変形することで，式(5)を得る．

$$\frac{P_1}{C_{cmp1}} > \frac{P_2}{C_{cmp2}} \tag{5}$$

式(5)の関係はどの２つの探索条件に対しても成立する事から，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がこの関係式により与えられることとなる．すなわち，各探索条件iに対して式(6)で計算される探索順位D_iによって与えられる順序をもとに，線形探索の判定を行うことで，実行時間の期待値を最小にする探索木を生成することが可能となる．

$$D_i = \frac{P_i}{C_{cmp\,i}} \tag{6}$$

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によって求めることができることがわかった．

２．３．２　判定式を用いた多分岐命令の探索木生成

分岐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．従来手法のプロファイルを用いた最適化手法[16]の問題は，多分岐命令の処理を線形探索のみで実現してしまうために，分岐頻度に大きな偏りがない場合には，２分探索や間接ジャンプなどの静的な探索手法よりも遅くなってしまう点であった．

この問題に対して，本論文では，線形探索と静的探索を，全体のコストが低下するように融合することで高速化を実現する方法を提案する．２．３．１ で既に示したとおり，前述の線形探索の実施判定式(3)と，探索順序を決定する探索順位を計算する式(6)を用いることで，与えられた多分岐命令に対して全体のコストを評価しながらよりより高速な探索手法を生成するための評価式について述べた．

以下では，まず多分岐命令からナイーブに生成した探索条件を用いて探索木を生成するアルゴリズムを示す．その後，探索条件を融合することによる探索コストの軽減を実現する手法について検討し，そのためのアルゴリズムを発見的手法により実現する方法を示す．

(1) 探索木生成アルゴリズム

まず線形探索と静的探索を融合した探索木の生成方法について説明する．線形探索と静的探索の融合の基本は，図２．５(a)において，FALSE process を静的探索で実施したときの期待コストを計算することである．全ての探索条件を静的探索により実施した場合の期待コストよりも，１つの探索条件のみを独立した線形探索として処理するほうが全体の期待コストが下がるかどうかを判定する．次に，これを静的探索処理で実施するとした FALSE process に対して同様に評価を行う．これを最終的に期

待コストが静的探索で実施したほうが同じかよい状態になるまで再帰的に繰り返す．この処理を実行するメソッドGenerate_search_treeのアルゴリズムを図２．６に示す．

```
01:     SearchTree Generate_search_tree (Switch sw) {
02:       SearchTree ST = new SearchTree();
03:       List T = sw.Make_a_test_condition_list();
04:       T.Sort_by_weight();
05:       while (true) {
06:         c_orig = T.Calculate_static_search_cost();
07:         t = T.Get_and_pop_test_condition();
08:         c_cmp = t.Calculate_compare_cost();
09:         c_true = t.Calculate_true_side_search_cost();
10:         c_false = T.Calculate_static_search_cost();
11:         c_e = c_cmp + t.probability * c_true + (1 – t.probability) * c_false;
12:         if (c_e >= c_orig)
13:           break;
14:         // t.Try_to_merge_adjacent_test_conditions(T, c_e);
15:         ST.Append(t);
16:       }
17:       ST.Append(T);
18:       return ST;
18:     }
```

図２.６　探索木生成アルゴリズム

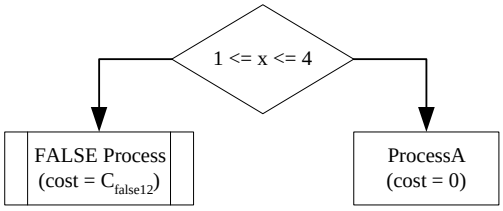
このアルゴリズムは，まず03行目で多分岐命令から生成した探索条件のリストTを04行目のメソッドSort_by_weightにおいて式(6)で計算できる探索順位によって降順に並べ直す．次に，05～15行目のwhileループにおいて，再帰的に線形探索部分を生成する．06行目のメソッドCalculate_static_search_costによってリストTの探索条件を静的探索によって探索した場合のコストc_origを計算する．静的探索コストの計算は，実際にコード生成部で用いられている探索方法とそその探索の基準を適用する．例えば，探索条件をカバーする区間の密度によって２分探索と間接テーブルジャンプが選択されている場合，それと同じ基準を用いて探索方法を決定し，それぞれについて探索コストの期待値を計算する．２分探索なら区間の大きさNに対するコストは$\log_2(N\hat{}2-1)$，間接テーブルジャンプであれば区間の大きさによらない定数（たとえば３）となる．次に07行目のメソッドGet_and_pop_test_conditionでリストTの先頭要素の探索条件をリストから取り除いてtにいれる．08行目のメソッドCalculate_compare_costで探索条件tの比較コストc_cmpを計算する．比較コストのもっとも簡単な実装は実際に必要となる比較の数として表すことができる．定数値との比較であれば１

となる．09 行目は，*t*の条件が成立した場合にさらに必要となる比較のコスト*c_true*をメソッド Calculate_true_side_search_cost で計算する．*t*が定数値との比較であれば，*c_true*は 0 となる．次に*c_false*を残っている探索条件を静的探索で探索するためのコストとして計算する．11 行目で*t*に対する期待コスト*c_e*を計算し，この値を用いて 12 行目で式(3)をつかって探索条件*t*を線形探索として処理するかどうかの判定を行う．線形探索で処理する場合は，15 行目でリストに追加する．最後に，17 行目で静的探索での処理を探索木に追加して終了する．なお，14 行目にコメントアウトしてあるメソッド Try_to_merge_adjacent_test_conditions では次の探索条件の融合処理を実行するための処理を実行する．

(2) 複数の探索条件を融合するアルゴリズム

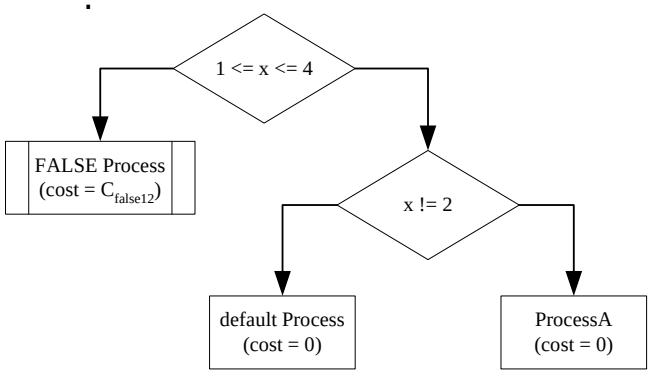
次に多分岐命令処理の高速化のために，分岐条件の融合について考える．多分岐命令では，図２．７(a)のように探索条件の一部に連続する整数値に対する比較に対して全て同じ分岐先が割り当てられている場合や，図２．７(b)のようにほとんどの場合に同じ分岐先が割り当てられている場合がしばしば存在する．このような連続区間や準連続区間に対して区間全体を評価した後にその中の例外値を除外するための比較処理を行ったり，間接テーブルジャンプを行ったりすることで，より高速な探索が可能となる場合がある．ここでは，この融合処理を実現するための方法を示す．

```
switch (x) {
  :
case 1: // profiled
case 2: // profiled
case 3: // profiled
case 4: // profiled
    ProcessA;
    break;
  :
```



(a) Range check of neighboring keys with same target

```
switch (x) {
  :
case 1: // profiled
case 3: // profiled
case 4: // profiled
    ProcessA;
    break;
  :
```



(b) Range check of neighboring default-range

図２.７　隣接条件マージの例

まず，図２．７(a)のように連続区間が全く同じ分岐先をもつ場合について考える．連続区間を判定する処理は実質的に１つの比較命令で実現でき，通常の定数値と比較処理と同じコストで処理できる．また隣接していて分岐先が等しい探索条件を融合した場合探索コストは変わらないか減少することから，このような探索条件は無条件に融合する．融合処理は，メソッド Make_a_test_condition_list で探索条件リストを生成する際に探索条件の融合処理を実施する．この処理は容易に実装できるのでここでは詳細は省略する．

一方，図２．７(b)の場合について考える．この場合，連続区間を検出した後さらに探索処理が必要となるため，無条件に融合してしまうと逆に探索コストを増加してしまう場合がありえる．動的コンパイラ環境を考慮した場合はコンパイル時間に制約があるため，ここでは発見的手法による実装方法を示す（図２．８）．まず，現在の探索条件 *t* の複製 *t_merge* を作成する（02 行目）．次にを 04 行目のメソッド Get_and_pop_near_condition により線形探索を実施することが確定した探索条件に対して近隣の探索条件を取得し，これをメソッド Merge_condition により *t_merge* に融合する（05 行目）．*T* と *t_merge* から融合した場合の期待コストを計算し，もとの期待コスト *c_e* よりもコストが少なくなる場合，その条件を融合する．これを期待コストの減少が無くなるまで繰り返すことで探索条件の融合を行う．

```
01:     void Try_to_merge_adjacent_test_conditions (List T, int c_e) {
02:       t_merge = this.clone();
03:       while (true) {
04:         t_near = T.Get_and_pop_near_condition(t_merge);
05:         t_merge.Merge_condition(t_near);
06:         c_cmp_merge = t_merge.Calculate_compare_cost();
07:         c_true_merge = t_merge.Calculate_true_side_search_cost();
08:         c_false_merge = T.Calculate_static_search_cost();
09:         c_e_merge = c_cmp_merge + t_merge.probability * c_true_merge
10:               + (1 - t_merge.probability) * c_false_merge;
11:         if (c_e_merge > c_e) {
12:           T.put(t_near);
13:           break;
14:         }
15:         this.Merge_condition (t_near);
16:       }
17:       return;
18:     }
```

図２.８　探索木の生成アルゴリズム

２．４　実験結果

本節では，ベンチマークを用いて２分岐命令と多分岐命令のプロファイルを用いた最適化の効果を評価する．既に，２．２で２分岐命令と多分岐命令に対する少量のプロファイルの収集方法を示し，ベンチマークを用いて収集されたプロファイルの精度を評価した結果を示した．ここでは，そのように収集されたプロファイルを用いて最適化を行った場合の効果を評価する．

２．４．１　評価環境

評価は AMD Athlon 1.2GHz, memory 256MB 上で動作する Microsoft Windows 2000 Professional SP1 上で行った．評価に使用した Java 仮想マシンは，IBM Java Virtual Machine J2RE version 1.3.0 と，その上で動作する Just-in-Time (JIT) Compiler[23]で行った．JVM が使用するヒープサイズは 128MB に設定した．

２．４．２　評価方法

分岐履歴情報の収集は，２分岐命令と多分岐命令のそれぞれに対して２．２節で示した方法でインタプリタに実装して実施した．２分岐命令の分岐履歴は，各分岐に対して 4bit の領域を割り当てて連続する４回の分岐方向を記録した．多分岐命令の分岐履歴は，探索条件毎に 4bit のカウンタを割り当てて分岐先の分岐比率として収集した．１つの多分岐命令あたりの履歴情報の収集は，探索条件式に割り当てられているカウンタのいずれかがオーバフローした時点である 16 回(=2^4)に達するか，または収集した分岐履歴情報の数が比較定数値数の２倍と 80 のうちの大きい方の値に達した時点で終了することとした．この終了条件は，分岐数が多い多分岐命令で分岐確率に偏りがない場合に収集量が大きくなりすぎて実行速度を低下させないためである．

収集した分岐履歴を用いた最適化は，JIT コンパイラに実装して行った．２分岐命令の分岐履歴を用いた最適化として，Edge Profile Guided Partial Redundancy Elimination (EPGPRE)[15]で用いた．この EPGPRE 手法は，従来の Partial Redundancy Elimination によって冗長な演算を除去する際に，分岐命令の分岐予測を用いて分岐頻度の高い分岐先を優先して除去を行う手法である．Lazy Code Motion を基本として，分岐頻度の低い側の情報を無効化することで分岐頻度の高い側に対する処理を優先させている．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ため，コンパイル時間が短い点で JIT コンパイラに適した最適化手法である．評価は，EPGPRE を実施しない場合に対して EPGPRE を適用した場合の速度向上率として行った．特に，分岐履歴の収集をプログラム開始時点から行った場合(Initial state profile)と，分岐を含むメソッドの実行回数が 1000 回に達した時点から収集を開始した場合(Steady state profile)に対して評価を実施し，プロファイルの精度の違いに対する影響も測定した．

また多分岐命令の実行履歴は，多分岐命令に対する最適化を全く実施しなかった場合に対する速度向上率として評価を行った．評価結果を比較するために，２．３節で示した最適化手法を実施した場

合と，文献[16]で述べられているProfile Based Switch Linear Search手法を用いた場合に対してそれぞれ評価を行った．分岐履歴情報は，分岐を含むメソッドの実行回数が1000回に達した時点から収集を開始した時点から収集したSteady state profileを用いた．

評価は， SPECjvm98ベンチマークとSPECjbbベンチマークを用いて行った．SPECjvm98ベンチマークは，各テストを10回連続して実行し，１０回のうちの最速値を評価結果として使用した．SPECjbbは，rump up timeとして３０秒，計測時間は120秒として，1 warehouse に対するスコアにより評価した．

２．４．３　評価結果

２分岐命令の予測による最適化の効果に対する評価を図２．９に示す．グラフは，プログラムの買い指示からプロファイルを収集した場合（Initial state profile）と，各メソッドが1000回実行された時点からプロファイルの収集を開始した場合（Steady state profile）の場合についての速度向上率を示している．Initial state profileとSteady state profileを比較して分かるとおり，Steady state profileを用いることで，速度向上率が上がっていることがわかる．このことから，プロファイルの収集時期を適切に設定することが効果的であるということができる．

Steady state profileの結果について見ると,mtrtでは約4%の速度向上がみられる一方で,compressとjavacは速度低下が起こっていることが分かった．この結果を表２．４の分岐予測の精度を比較すると，compressはもともと収集される分岐予測の精度が低いことが影響していると思われるが，javacは約92%の予測精度が得られており結果と矛盾する．Javacについてさらに調査したところ次の点がわかった．実験で使用した実装では，２分岐命令に対して収集した連続４回の分岐履歴から分岐予測を「分岐する」「分岐しない」「予測なし」の３値状態で設定していた．表２．４の結果は，予測した方向への分岐が他方への分岐よりも多い以下どうかで評価したため，実際に「分岐する」「分岐しない」と予測されたとしても，それは予測方向への分岐が50%より多いということを表している． EPGPREでは,分岐予測に対して実際の分岐頻度を90%と設定していたが,javacは分岐方向の偏りが大きくなく，一方に分岐すると予測した場合も，他方への分岐が少なからず実行されてしまったために，予測と異なる側への分岐先に出される補償コードの実行コストによって速度低下を引き起こしていた．逆に，mtrtでは分岐予測に対して，実際の分岐の実行の頻度が非常に高く，結果として高い速度向上を得られることができたのである．しかしながら， EPGPREでの分岐頻度の設定を低く取ってしまうと，最適化を適用できる範囲が低下してしまう．予測の精度を上げるためにはプロファイル量を多くする必要がり，少量のプロファイルから導き出せる予測の精度はどうしても限界が出てしまう．従って，例えば実行頻度の高いメソッドに対しては，別途より詳細なプロファイルを収集することで高い精度のプロファイルを収集して最適化をすることで対応する方法は考えられる．SPECjvm98ベンチマークは，全体としては約0.8%の速度向上を得ていることから，結果としては少量のプロファイルを使用した最適化は速度向上に効果があるといえる．

次に多分岐命令の最適化の効果に対する評価を図２．１０に示す．グラフは従来手法であるProfiled

based switch linear search 手法[16]と，２．３節で提案した Profile based switch optimization 手法の評価結果を示している．結果からわかる通り，mtrt 以外において，提案手法は従来手法よりもよい結果を示していることがわかる．mtrt で従来手法のほうが結果がよい理由は次の通りであった．本手法が分岐の頻度の差が少ない多分岐命令に対して２分探索コードを生成しているが，実際にはそれらの間に著しい偏りがあり，結果として単純な線形探索の探索コストが少なくなっていたためであった．プロファイルの精度が低い場合，提案手法と従来手法のどちらがよいかということは予測できないことから，この逆転現象は最適化手法の問題ではなく，むしろ収集されたプロファイルの精度の低さの問題といえる．一方，従来手法では特に 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．これは多分岐命令を常に逐次探索で処理してしまうために，分岐頻度の偏りが少ない場合に探索コストが高くなり，結果として性能低下を引き起こしてしまったためである．

図２．１１に，SPECjvm98 ベンチマークに対して分岐履歴を用いて最適化を行った場合の全体的な速度向上率を示す．mtrt は前述の理由より速度が低下してしまっているが，全体としては平均 1.2%の速度向上が得られており，少量のプロファイルを収集することで，プログラムの速度向上が得られるといえる．またより大きなプログラムに対する評価として，図２．１２に SPECjbb ベンチマークを用いた場合の結果を示す．図２．１２のグラフでは，EPGPRE による速度向上率，Profile Based Switch Optimization による速度向上率，および全プログラムに対して，履歴情報使用した場合の速度向上率を示している．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向上していることがわかる．

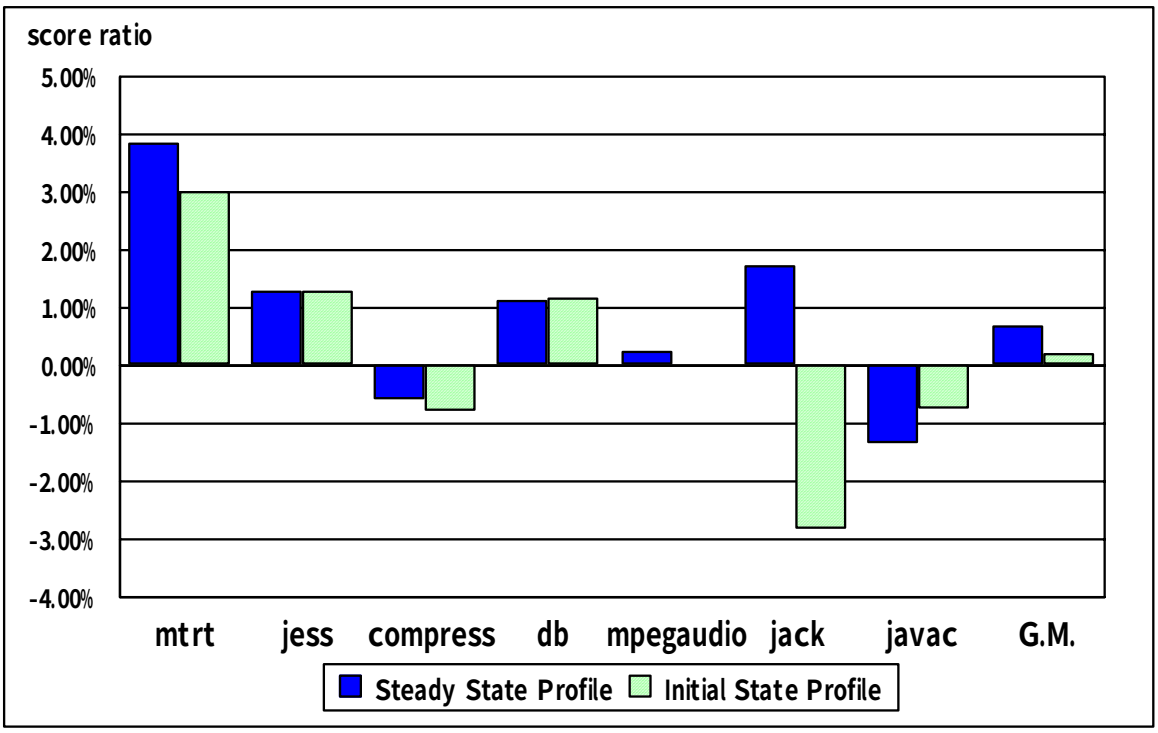


図２．９　Profile Guided Partial Redundancy Elimination の効果

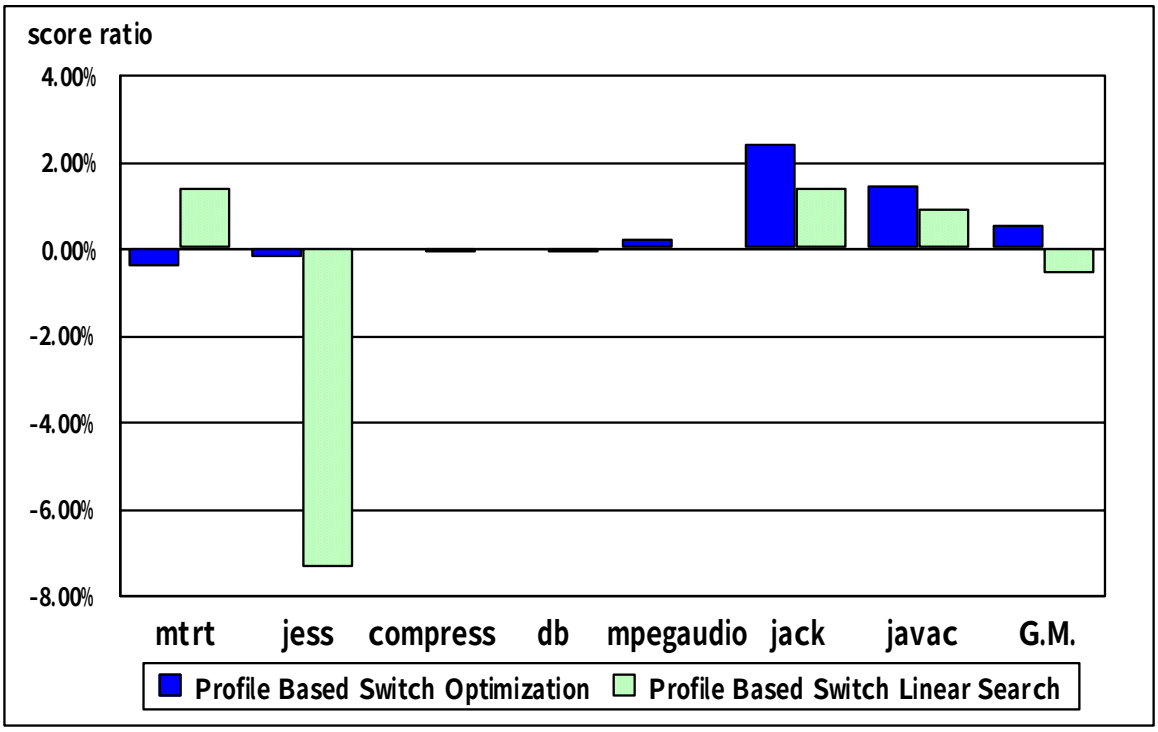


図２．１０　Switch Optimization の効果

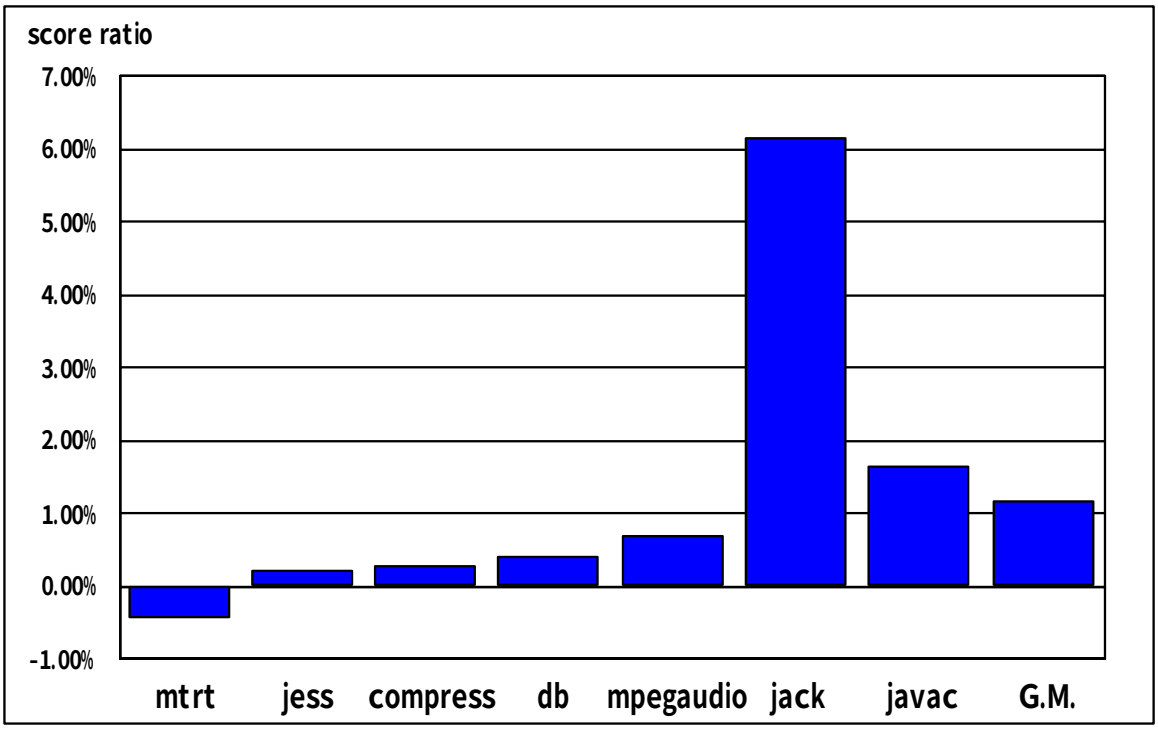


図２．１１　SPECjvm98

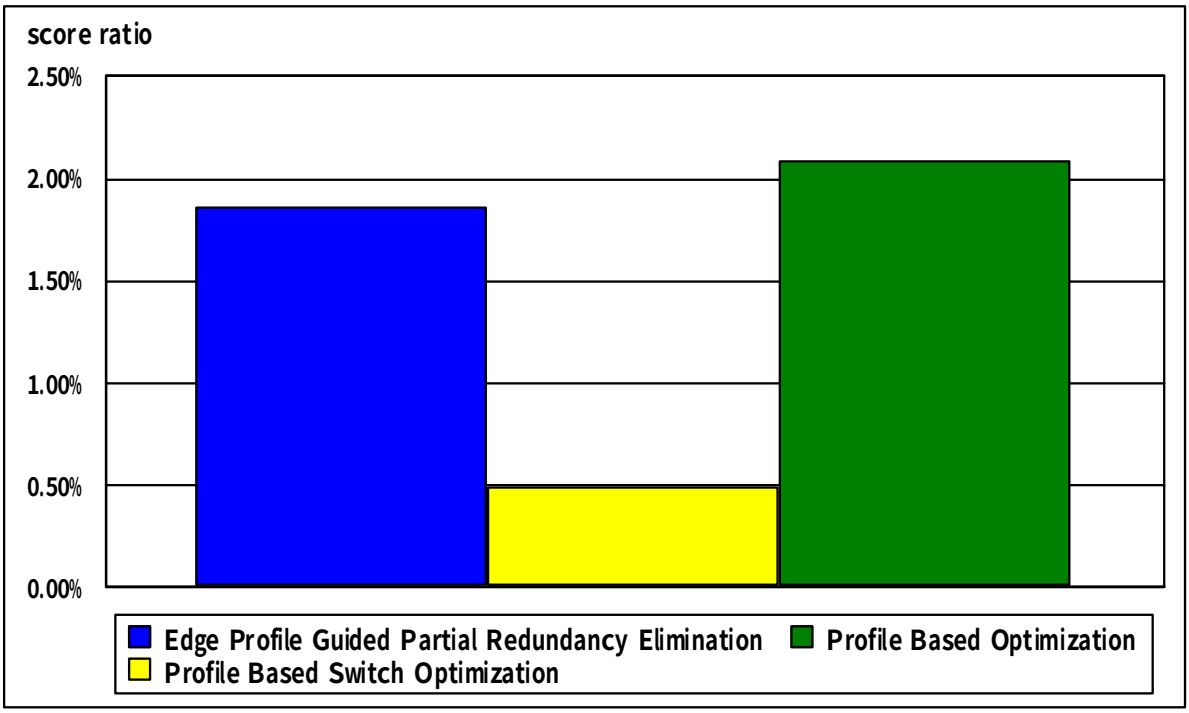


図２.１２　SPECjbb

２．５　まとめ

本章では，動的コンパイル環境において，実行時に少量の分岐履歴情報を収集して最適化を行う方法の有効性について議論した．少量の分岐履歴を収集する方法を示し，それにより収集された分岐履歴情報の精度を調べ，さらにその分岐履歴情報を用いて最適化を実施した場合のプログラムの処理速度への影響を評価した．結果として，２分岐命令，多分岐命令とも少量の分岐履歴の収集とそれによる最適化によって，プログラムの実行速度を向上することができることを示した．また，分岐履歴情報の収集に対して，収集時期を遅らせてプログラムの起動時を回避してプロファイルを収集することで，より高い精度のプロファイル情報を収集できることが分かった．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．

少量の分岐履歴では予測の精度に幅が出てしまうため，予測に対して最適化によっては想定している実行確率とずれてしまうことで，かえって性能を低下させてしまうことが分かった．これは，少量の履歴情報の収集においては解決の難しい問題であるが，例えば，実行頻度の高いメソッドに対しては，さらに詳細なプロファイルの収集を別に行うことで対処することが可能であると思われる．

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常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わかった．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，例えば実行比率が全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上げられる．例えば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．

第３章　オンラインパスプロファイルの構造的収集手法

３．１　はじめに

近年 Java 言語の普及にともなって，Java Just-in-Time (JIT)コンパイラが脚光を浴びている． JIT コンパイラ[20][21][22][32][34]は，プログラムの実行中にコンパイルを行ってネイティブコードを生成する．このため，実行中のプログラムから収集できるオンラインプロファイル情報をコンパイルに適切に使用することで，プログラムの実行に非常に適したコードを生成することができる利点がある．反面，コンパイル処理に要する時間や様々なリソースの使用がプログラムの実行に影響するため，コンパイルに対して時間や資源の使用に制約を受ける欠点を持つ．既存の JIT コンパイラでは，このような制約の中で，実行頻度の高いメソッド（ホットメソッド）の検出[20][21][22][32][34]や仮想呼び出しの最適化[20][32][34]，基本ブロックの並べ替え[20]などの様々なプロファイル情報を利用した最適化を実装している．これらにより静的コンパイラよりも高速に実行されるプログラムも少なくない．

プログラムの実行挙動を求めるプロファイル手法として，ノードプロファイル（vertex profiling）手法とエッジプロファイル（edge profiling）手法[6]がある．ノードプロファイル手法は各基本ブロックの実行頻度を，エッジプロファイル手法は制御フローエッジの実行頻度をそれぞれ収集し，分岐予測や基本ブロックの並べ替え，実行頻度の高い経路（ホットパス）の予測などのために用いられてきた．しかし，近年提案されている，データフロー解析を用いた最適化の性能向上を目的としたプログラム変形[10][14]やコード移動[12][13]，実行頻度の高い経路の分離処理[36][37]などの最適化では，複数の基本ブロック間にまたがる経路の実行状況を把握する必要があるために，ノードプロファイルやエッジプロファイルでは不十分で，より正確な実行挙動を得られるパスプロファイル(path profile)情報が必要となる．

パスプロファイル情報はプログラムの実行挙動を詳細に表現できるプロファイルの１つであり，プログラム中の有限長の経路に対する実行頻度として表現される．既存のパスプロファイル収集手法[3][37]では，対象プログラム中に存在する有限長の実在経路を用いてパスプロファイル情報を収集する．本論文ではこのように実経路を用いて定義される経路を自然な経路と定義する．

自然な経路を用いたパスプロファイル収集手法では，各経路で収集される情報はその経路の実際の実行回数となる．パスプロファイル収集処理により増加する実行時間（以下，収集コスト）は収集される実行経路数に比例すると考えると，自然な経路を用いたパスプロファイル収集手法の収集コストはメソッド中で実行される経路の数に比例して増加する．つまり非常に実行回数の多いループがあるメソッドでは収集コストがループの実行回数に比例して爆発的に増大してしまう．もし収集コストを抑えるために，メソッドの実行の途中でパスプロファイルを終了すると，それ以降の部分のプロファイルが実施されず，結果として収集情報が不完全となり，プロファイル情報の精度が低下してしまう．この性質は動的コンパイラにおけるプロファイル手法として適さない．

一般に，動的コンパイラにおいて用いられるプロファイル手法は，収集される情報を用いて実施される最適化による速度短縮よりも少ない時間しか費やすことができない．収集情報の精度は最適化結果に影響することから，プロファイルに要する時間と収集情報の精度とをバランスさせながら，必要十分な精度の情報を少ない時間で収集することが必要である．

本章では，この問題を解決する手法として構造的パスプロファイル収集手法(SPP: Structural Path Profiling)を提案する．SPP の基本アイデアは，プログラムを各ループネストレベルに分割し，それぞれのループネストに対して独立にパスプロファイル収集処理を実施することである．各ループネスト中に１つの１重ループしか存在しないように，内側ループを縮退させながらグラフを階層的に分割することで，各ループネストの収集コストが内側ループの実行状態と独立に決定できるようになる．つまり各ループネストに収集する実行経路の総数を適切に設定することで，少ないオーバヘッドで十分な精度のプロファイルを収集することが可能となる．

本章で提案するフレームワークは，既存の２つのパスプロファイル収集手法のいずれも扱うことができるが，本章では収集コストが軽い点から Ball と Larus によるパスプロファイル収集手法を用いて SPP を IBM Java Just-In-Time コンパイラに実装し，評価を行った．各メソッドで収集される実行経路数を固定した場合のプロファイル情報の精度を測定した結果，SPP は，収集経路数が 1000 の場合に 90%の精度を得られるとともに，Ball らの手法[3]と動的インストルメンテーション手法[35]を単純に組み合わせた手法に比べて少ない収集量で高い精度を出すことを示した．さらに実際の実行に対するオーバヘッドでも，非常に多くのメソッドに対してプロファイルが実施されている状況でも 2-3%のオーバヘッドしか要しないことを示した．

本章の貢献点をまとめると以下の２点となる．

- 動的コンパイラのための新しいパスプロファイルフレームワークとして構造的パスプロファイル収集手法の提案
- 提案手法の評価とその有効性の提示

以下では，まず，３．２で本研究の背景を解説してパスプロファイル情報の有効性とともに，パスプロファイル収集手法を動的コンパイラにおいて利用するための課題について示す．続いて３．３においてでこの課題を満たす新手法である構造的パスプロファイル収集手法を説明する．その後３．５において提案手法を IBM Java Just-In-Time コンパイラに実装して行った評価結果を，３．４では本結果を利用して loop peeling を実施した場合の評価結果を示し，３．６で提案手法に対して考察する．３．７では関連研究について説明し，最後に３．８で本章の結論を示す．

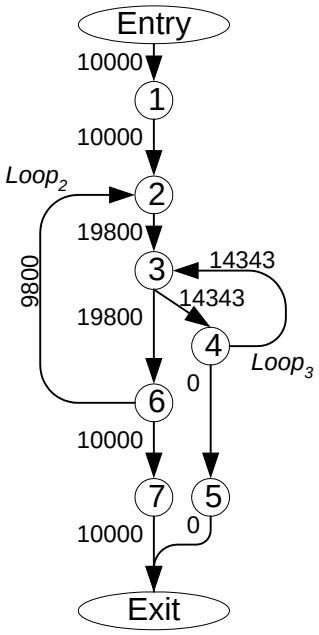
３．２　背景

本節では，JIT コンパイラにおけるパスプロファイルの使用に関する背景について説明する．本章を通じて説明の例に使用するプログラムを示すとともに，収集されるプロファイルの精度を評価する手法を示す．その後，エッジプロファイルとパスプロファイルのそれぞれを用いたプログラム変換の例を示しながら，両者の評価を通じてパスプロファイルの優位性について示す．最後に，従来のパスプロファイル収集手法を JIT コンパイラでそのまま使用する場合の問題について，プロファイルオーバヘッドとプロファイルの精度の関係から議論する．

３．２．１　説明用プログラム

まず，本章の様々な説明に用いるプログラムを図３．１に示す．図３．１(a)は，ある２重ループを含むメソッドからその制御部分のみを記述する形で示されたメソッドの断片であり，図３．１(b)は，図３．１(a)のメソッドに対応した制御フローグラフである．図３．１(b)の各エッジに付加されている値は，このメソッドを含むプログラムの実行においてこのメソッドが実際に 10,000 回呼び出された時に収集されたエッジプロファイルを表している．また，このメソッドにおいて，６→２をバックエッジとする外側のループを$loop_2$，４→３をバックエッジとする内側ループを$loop_3$と呼ぶこととする．

```
public void m()
{
  ...              // BB1
  do {
    ...            // BB2
    while (...) {  // BB3
      ...          // BB4
      if (...)
        return;    // BB5
    }
    ...
  } while (...)    // BB6
  ...              // BB7
}
```



(a) Program example　　**(b) Control flow graph with edge profile counts**

図３.１　２重ループを持つプログラムとそのエッジプロファイルの例

このメソッドの実際の実行では，次の(1)-(3)の３つの実行頻度の高い経路が存在する：(1)繰り返

しを全くしない経路：*1-2-3-6-7*，(2)外側ループ(*loop*$_2$)を繰り返す経路：*2-3-6-2*，(3)内側ループ(*loop*$_3$)を繰り返す経路：*3-4-3*．つまり，このメソッドはそのほとんどの実行ではループが実行されないが，一度ループの繰り返しが起こると，そのループの実行が多数回行われる，という特性を持っている．

３．２．２　収集したプロファイルの精度の評価手法

本章で使用するオンラインプロファイルの精度を，オフラインプロファイルとの類似度として定義する．この類似度の評価方法として，オーバーラップパーセンテージ手法(overlap percentage)[27]による評価手法を使用する．オーバーラップパーセンテージ手法は，あるプロファイルが別のプロファイルとどの程度類似しているかを評価する手法である．図３．２にオーバーラップパーセンテージ手法による評価の例を示す．あるメソッド中に存在する全経路に対して，図３．２(a)はオンラインプロファイルにより収集された各経路の実行回数の比率(%)を，図３．２(b)はオフラインプロファイルにより収集された各経路の実行回数の比率(%)を，それぞれ棒グラフで表している．このとき，オーバーラップパーセンテージによる２つのプロファイルの類似度は，２つのグラフの共通部分の面積として計算される．この例の場合，図３．２(c)に示すとおり，２つのプロファイルの類似度は８８％となる．

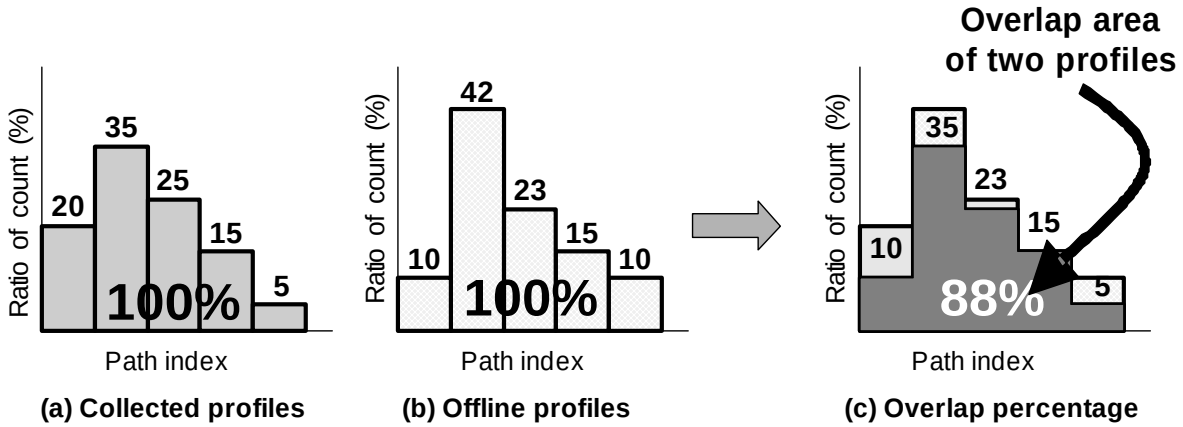


図３．２　オーバーラップパーセンテージ手法

３．２．３　エッジプロファイルとパスプロファイル

エッジプロファイルは，数値計算などのようにプログラムの実行を通じて条件分岐の分岐方向が特定の方向に分岐するプログラムでは，非常に高い精度でホットパスの検出を検出することができる[2][38][39][40]．しかしながら，ビジネス処理などの非数値計算プログラムなどでは条件分岐の分岐方向の偏りがない場合が多く，この様に明確な分岐方向を持たない分岐を含むプログラムに対してはエッジプロファイルでホットパスを正確に検出することが困難となる．もちろん，このような非数値計算プログラムに対しても大まかにホットパスを予測することは可能[2]である．しかし，プログラムが最高性能を発揮するように変換するように，loop peeling や loop unrolling，tail duplication，partial redundancy elimination などのプログラムの最適化手法を適切に利用するためには，より高

精度のパスプロファイルを用いてプログラムの実行状態を正確に把握することが必要である．

例えば，図３．１のプログラムの最適化方法について考える．このプログラムは既に述べたとおり，３つの Hot Path が存在しているが，それらの経路が互いに重なり合っている．このために図３．１(b)にあるエッジプロファイルから分かるとおりノード３とノード６の条件分岐の分岐頻度はトータルではほぼ均等に分岐する結果となっている．このプログラムに対して，例えば loop peeling を用いて図３．３(a)に示されるよう３つのホットパスを分離することで，各ホットパスに対してより速い実行コードを生成することが可能となる．このコードでは，全ての条件分岐の分岐頻度の偏りが大きくなるために，コードのローカリティだけでなくハードウェアの分岐予測の的中率の向上によって高速実行が可能となる．

しかしながら，エッジプロファイルでは，ノード３とノード６の分岐頻度の偏りがほとんどないことから，３つのホットパスである，*1-2-3-6-7*, *2-3-6-2*, *3-4-3* を検出することはできない．結果として，単に *1-2-3-6-7* のパスのみをホットパスとして予測するか，場合によっては，それぞれのループは常に１回のみ繰り返しが起こると予測し，*1-2-3-4-3-6-2-3-4-3-6-7* という誤ったホットパスを導き出す可能性すらある．仮にホットパスとして *1-2-3-6-7* のみが検出された場合，ホットパスに対するコードのローカリティを優先した最適化を実施すると，図３．３(b)のように loop peeling は実施せずに基本ブロックの並べ替えのみが実施されたコードが生成される．このコードでは，検出されなかった２つのホットパスが実行される場合や，実行されるホットパスが変わった場合に分岐予測ミスが発生し，実行速度低下を引き起こしてしまう．他方，詳細は後述するが，このプログラムに対して Ball と Larus のパスプロファイル収集手法[3]をそのままオンラインプロファイルとして使用すると，２つのホットパス *1-2-3-6-7* と *3-4-3* のみが検出さる．結果として図３．３(c)のコードが生成されるが，エッジプロファイルの場合と同様に，検出されなかったホットパス *2-3-6-2* を高速に実行することができない．

一方，パスプロファイルはエッジプロファイルのスーパーセットであり，パスプロファイルから容易にエッジプロファイルを生成することが可能である．つまり，パスプロファイルを収集することで，パスプロファイルを利用した最適化とエッジプロファイルを利用した最適化の両方を使用することが可能となる．パスプロファイルの収集コストは，文献[3]によれば経路数が増大した場合に用いられるハッシュ手法を使用しない場合にはエッジプロファイルと同等のコストで収集できることが報告されている．すなわち，経路の数が予め設定した値以内の場合に対してパスプロファイルを収集するようにすることで，エッジプロファイルの収集コストとほぼ同等のコストで，パスプロファイルを収集することが可能となる．

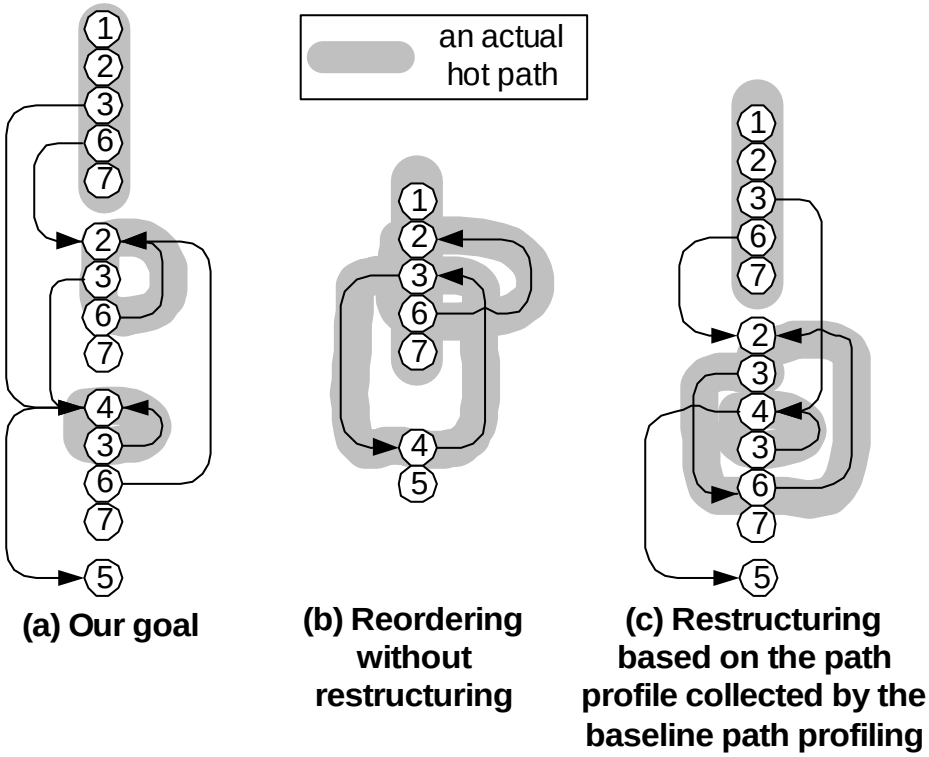


図３.３ Restructuring by loop peeling

３．２．４ 動的パスプロファイル収集の課題

動的コンパイラにおけるパスプロファイル収集手法で満たすべき要件は，収集コストと収集されるプロファイルの精度をどのようにバランスさせるかという点である．すなわち，オフラインパスプロファイル[3][37]と異なり，オンラインパスプロファイルでは，たとえプロファイルを収集する対象がほっとメソッドに限定されたとしても収集に要する時間や資源が制限されるので，どのような量のプロファイルをどのように収集するかが重要となる．

従来のパスプロファイル収集手法を用いて高精度のプロファイルを収集するためには，少なくとも各メソッド呼び出しで実行される全ての経路に対するパスプロファイルを収集する必要がある．これは，従来のパスプロファイルはプログラムの実行された各経路の実行回数に従って記録されるので，各経路の比率を正確に収集するためには少なくともメソッドの実行を単位としてプロファイルを収集する必要がある．すなわち，もしメソッド中に非常に多数回実行されるループが存在する場合，プロファイル収集のオーバヘッドの爆発を引き起こしてしまう．一方，このオーバヘッドの爆発を防ぐために，例えば ephemeral instrumentation 手法[35]を用いて収集する総経路数を予め定めた値に限定したとすると，収集されたプロファイルがメソッドの振る舞いを正しく表していることを補償できなくなってしまう．例えば，プロファイルを収集しているメソッドにいくつかの非常に実行回数の多いループが存在したとする．メソッドの開始時にプロファイルを始めたとして，そのいくつかのループ

の実行途中で収集プロファイル量が設定値に達してしまうと，その時点でプロファイルの収集が終了してしまう．つまり，それ以降のループに対して全くプロファイルの収集が行われなくなってしまう．すなわち，そのメソッドの一部分に対してしか収集が行われていないプロファイルをメソッド全体のプロファイルと解釈してしまうことによって，本来の実行状態とは異なるプロファイルとなってしまい，結果として精度の高いパスプロファイルを収集することができなくなってしまう．

ここで，図３．１のプログラムに対して，Ball と Larus の手法[3]をそのままオンラインパスプロファイルの収集に使用した場合の例を示す．彼らの手法は本来オフラインプロファイル収集手法として設計されているので，オンラインパスプロファイル収集手法として使用するためにはプロファイルの収集を制御する仕組みが必要となる．ここでは，プロファイルカウンタを用いて，収集対象とする各メソッドに対して収集されたパスプロファイルの総数をカウントし，その値が予め設定された数のプロファイルを収集した時点でそのメソッドに対するプロファイルを終了する方法を用いる．本論文では，このオンラインパスプロファイル収集手法を Baseline Path Profiling (BPP)と呼ぶものとする．

図３．４，表３．１，図３．５は，BPP を図３．１のプログラムに適用して 300 のプロファイルを収集した場合の結果である．図３．４は，図３．１のプログラムに対して，BPP を用いてオンラインパスプロファイルを収集するために挿入されるインストルメンテーションコードとプロファイルの総数をカウントするプロファイルカウンタ，およびその挿入点となるエッジを，制御フローグラフ上で示している．

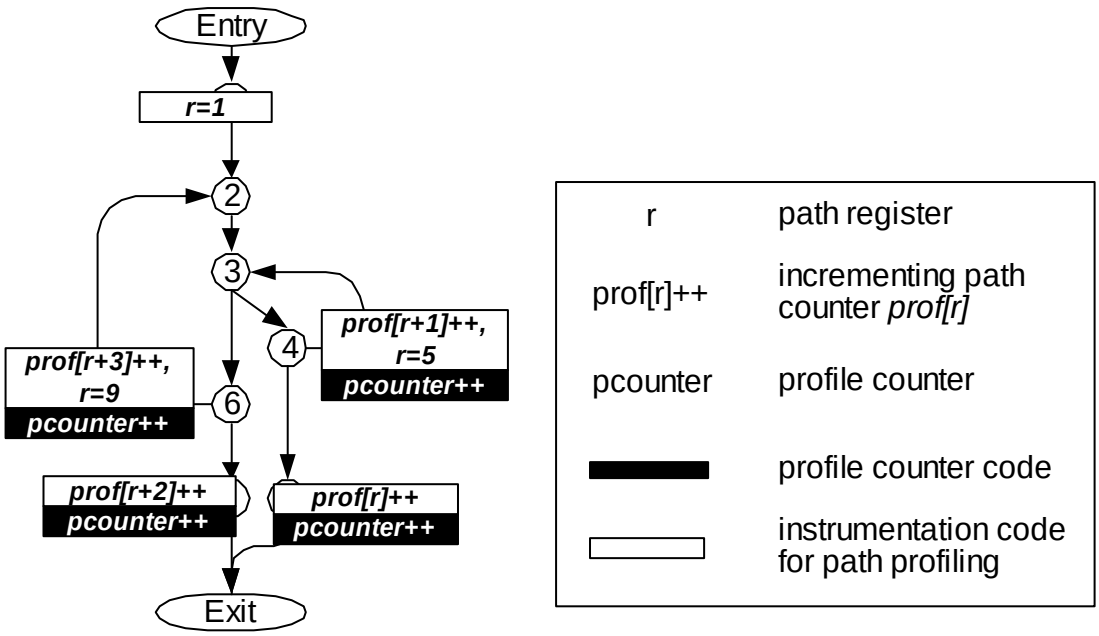


図３．４　Instrumentation code and profile counter inserted by BPP

表３．１ Path profiles collected by BPP.

path index	Paths	Onlne profiles	Offline profiles
1	Entry-1-2-3-4-5-Exit	0	0
2	Entry-1-2-3-4-(3)	5	345
3	Entry-1-2-3-6-7-Exit	124	9,462
4	Entry-1-2-3-6-(2)	0	193
5	(4)-3-4-5-Exit	0	0
6	(4)-3-4-(3)	167	13,660
7	(4)-3-6-7-Exit	4	345
8	(4)-3-6-(2)	0	338
9	(6)-2-3-4-5-Exit	0	0
10	(6)-2-3-4-(3)	0	338
11	(6)-2-3-6-7-Exit	0	193
12	(6)-2-3-6-(2)	0	9,269
Total		300	34,143

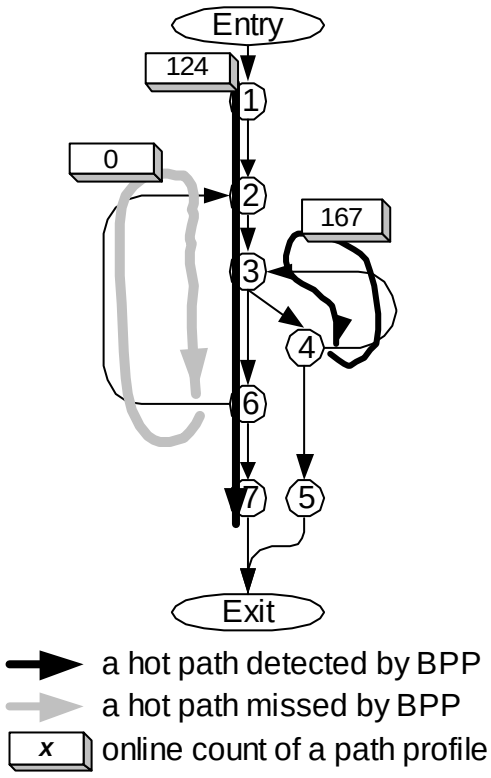


図３．５ Hot paths detected by BPP.

表３．１は，図３．１のプログラムに対してBPPによって収集したプロファイル（Online Profiles）と，オフラインプロファイルで全実行を通じて収集したプロファイル（Offline Profiles）の結果で

ある．オフラインプロファイルでは，３つのホットパス，1-2-3-6-7，3-4-(3)，2-3-6-(2)，を全て検出できている一方で，オンラインプロファイルでは， 1-2-3-6-7，3-4-(3)，の２つのホットパスのみしか検出できていない．すなわち，もう一つのホットパスである，2-3-6-(2)，を検出することができていない．これは，このホットパスは外側のループである$loop_2$ を形成する経路であるが，このループの実行が行われるまえにパスプロファイルの収集が終了してしまったことが原因である．図３．５の太線はBPPで検出された２つのホットパスを，グレーの線は検出できなかったホットパスを，それぞれのプロファイルカウントとともに示している．この結果をもとに最適化を実施した結果が前述の図３．３(c)である．結果として，BPPではこのメソッドを十分に最適化することはできないことがわかる．

オンラインパスプロファイルを実現するための別の手法として Arnord らの Instrumentation sampling 手法[25]を利用して BPP を実施する方法が考えられる．彼らのフレームワークは，プロファイルを収集するメソッドのコードを複製し，コンパイラで生成されたカウンターベースのサンプリング手法をもちいてインストルメンテーションコードを挿入されたコードと挿入されていないコードの実行の切り替えを行う．サンプリング・レートによってプロファイル収集のオーバヘッドと収集されるプロファイルの精度が制御される．これによって，オフラインプロファイルに比べて実行時のオーバヘッドを大きく軽減することを実現している．この手法を用いることで，原理的には収集されるプロファイルが分散し,BPP で問題となっているようなメソッドの途中でプロファイルの収集が終了してしまうような状況を回避することは可能である．しかしながら，そのために適切なサンプリング・レートを決定することは非常に困難である．サンプリング・レートが低すぎるとプロファイル収集が完了するまでに長い時間を要してしまい，結果としてプロファイルを用いてより実行速度の速いコードへと最適化を実施する再コンパイルの時期が遅れてしまう．逆にサンプリング・レートが高すぎればプロファイル収集が完了するまでの時間は短くなるが,BPP のようにホットループのプロファイル収集を失敗してしまう可能性が高くなる．適するサンプリング・レートはメソッドによって異なり，また現実的にプログラムの静的解析で検出することは困難であることから，この方法でも問題を完全に解決することはできない．

３．３　構造的パスプロファイル収集手法

前節で述べたとおり,BPP の主な問題点はプロファイルの収集数をループネスト毎に独立して設定できないことである．高精度のプロファイルを収集するためには，目的のメソッドに対してプロファイル収集を保証する必要があり，少なくとも１回のメソッドの実行を通じてプロファイルを収集し続けねばならない．このためメソッド中に実行頻度の高いループが存在すると高いプロファイル収集コストが必要となってしまう．

この問題を解決する手法として，本節では，動的コンパイラにおいて前節の問題を解決する新しいパスプロファイル収集手法である，構造的パスプロファイル収集手法(Structural Path Profiling,

SPP)を提案する．SPP の主なアイデアは，ターゲットメソッドをそのループ構造に基づいてループのネスト毎に階層グラフ（Structure Graph）と呼ぶグラフに分割し，その各グラフ毎に独立にプロファイルを実施することである．構造グラフの構築においては，各ネストが内包する内側ループをループノードという一つのノードに縮退させることで，各ストラクチャーグラフはその内側ループのプロファイルオーバヘッドの影響を受けずにパスプロファイルを収集することが可能となる．

このアプローチがオンラインパスプロファイル収集における問題を解決する理由は以下の２点である．１つはプロファイルの収集をループの１つの階層に限定することでプロファイルのオーバヘッドを小さく抑えることができる点であり，もう１つは各構造グラフに対して適切なプロファイル収集量を設定することで，個々の構造グラフのプロファイル結果からそのメソッド全体のパスプロファイルを，十分な精度で生成することができる点である．

構造グラフを用いる利点は，各グラフでは内側ループが全て縮退されており，最大で１つの１重ループしか存在しないことである．つまり，内側ループの実行と独立にパスプロファイル情報を収集できるので，そのグラフ中のループの実行回数に無関係に収集する実行経路数を決めてもグラフ内の全域に対して実際の実行頻度に比例したパスプロファイル情報を収集できることを保証できる．このため，各構造グラフに対して適切にプロファイル量を設定することで，少ないオーバヘッドで十分な精度のプロファイルを収集することが可能となる．

以下では，まず構造グラフを定義した後，SPP の各処理について説明する．続いて収集されたパスプロファイルを用いた最適化の例を示した後，構造グラフ上で定義される構造的経路について考察する．

３．３．１　構造グラフの定義

構造グラフは，始点ノード(Entry Node)，終点ノート(Exit Node)，基本ブロックを表すノード，および強連結領域を表すループノード(Loop Node)と，それらの間の制御フロー関係を表すエッジにより構成される．図３．６は，図３．１で示されるプログラムを構造グラフで表したものである．図中の太線のノードはループノードであり，点線のエッジは仮想エッジ(Virtual Edge)である．仮想エッジはもともとの制御フローグラフには存在しないエッジであるが，ループへ出入りするエッジに対応しており，そのラベルで対応する元のエッジが示されている．

構造グラフは２種類の構造グラフ，外郭構造グラフ(Outline Structure Graph)とループ構造グラフ(Loop Structure Graph)に分かれる．外郭構造グラフはもとの制御フローグラフ中の全ての強連結領域をループノードに縮退したグラフとして定義される．また，メソッドの開始基本ブロックにあたるノードを外郭構造グラフに対するカウントノードと定義する．もしこのノードが強連結領域内に位置する場合は，その強連結領域を縮退したループノードをカウントノードとする．図３．６(b)が外郭構造グラフの例である．

ループ構造グラフは，その内側ループを全てループノードに縮退させた強連結領域として定義され

る．また，その強連結領域を構成するループのヘッダノードをループ構造グラフのカウントノードと定義する．図３．６(c)がノード２をヘッダノードとするループネストに対応するループ構造グラフを表し，図３．６(d)がノード３をヘッダノードとするループネストに対応するループ構造グラフを表している．構造グラフの階層構造関係は，その内包関係により定義される．

SPP はターゲットメソッドを各ループネストで分割するために，ループネストに跨る経路を検出することはできない．しかし，ループの最適化は基本的にループの各ネストに対しておこなわれるので，現実的には影響は少ないと考える．

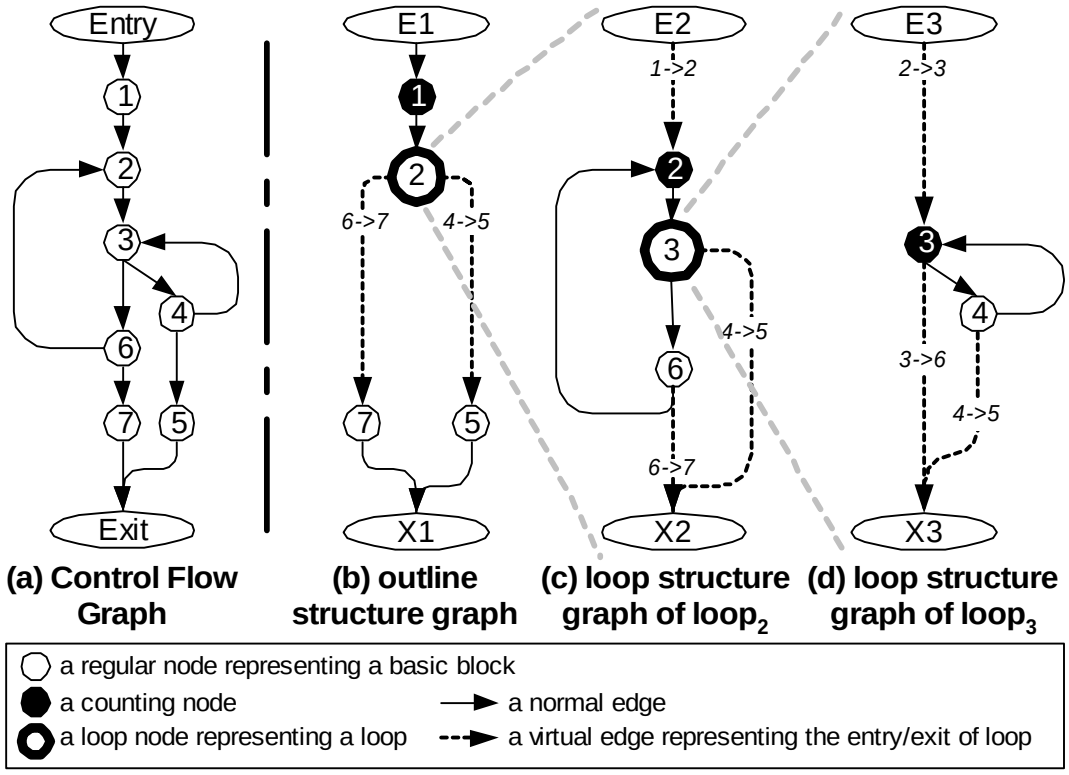


図３.６　構造グラフの例

３．３．２　構造グラフの構築

SPP では，まず対象メソッドの制御フローグラフから構造グラフを生成する．図３．７に構造グラフの生成アルゴリズムを示す．まずターゲットメソッドに対して制御フローグラフを生成した後，Havlak のアルゴリズム[28]を用いたループ検出処理 *analyzeLoop* 関数を呼ぶことでメソッド内の全てのループを検出する．Havlak のアルゴリズムは，Tarjan のループ検出のアルゴリズムを拡張したもので，irreducible loop に対してもほぼ線形時間で処理することができる．ループ領域を検出した後，グラフ内のループ領域をループノードに縮退することで構造グラフを作成する．もしループが無限ループで制御上の出口がまったく存在しない場合は，そのループを縮退して生成したループノードから終点

ノードへのダミーエッジを作成する．各仮想エッジは，次のインストルメンテーションコード生成時に使用する情報として，もとの制御フローグラフにおける対応するエッジの情報を設定する．

```
Procedure makeStrGraph (Method m)
  Generate control flow graph og from m
  // loop region detection by Havlak's algorithm
  analyzeLoop (og, Entry)
  make a hierarchy of structure graphs of m from loop nesting tree
       by calling the procedure analyzeLoop
  // structure graph construction
  foreach sg in structure graphs of m do
    make loop nodes from all inner loops in sg
    foreach ln in loop nodes of sg do
      lg := corresponding loop structure graphs to ln
      change destination of all incoming edges to ln and make virtual entry edge of lg
       corresponding to them
      change source of all outgoing edges to ln and make virtual exit edges of lg
       corresponding to them
    enddo
  enddo
```

図３.７　構造グラフの生成アルゴリズム

プログラムの中には複数のネストレベルで１つのヘッダノードを共有する多重ループが存在するが，このアルゴリズムでは，このような場合に対して，これらを１つのループ構造グラフで表してしまう．SPP はこの様に生成されたループ構造グラフに対しても問題なく動作するが，この様なループに対して，本アルゴリズムを適用する前にヘッダノードを分離させる処理を行うことで，それらに対して異なるループ構造グラフを生成する方が，より正確なパスプロファイルを収集することが可能となる．

３．３．３　インストルメンテーションコードの生成

図３．８に構造グラフに対してインストルメンテーションコードを生成するアルゴリズムを示す．このアルゴリズムでは，まず従来のパスプロファイル手法のアルゴリズムを用いてパスプロファイルのためのインストルメンテーションコードを生成する．本論文ではそのオーバヘッドの小ささから Ball と Larus の手法を用いているが，Young と Smith の方法[37]を代わりに用いることもできる．

続いて各構造グラフのカウンタノードに対してプロファイルカウンタ用のコードを生成し，カウンタノードが実行されるたびに１ずつカウンタがインクリメントされるようにする．このカウンタノードは，各構造グラフにおいて収集されたパスプロファイルの総数をカウントするためのカウンタで，

実行時にプロファイル収集の完了を検出するために用いられる．もし構造グラフに既約ループ（irreducible loop）が含まれる場合，そのループの外からループ内のカウントノード以外のノードへ入るエントリーエッジに対しても，同様にプロファイルカウンタを生成する．既約ループの場合でも，ループの繰返時は必ずカウントノードを通過するので，ループ外からの経路の実行に対するカウントを行うコードをエッジに対して生成することで収集されるパスプロファイルの総数を正確にカウントすることができる．

```
Procedure GenInstCode(Method m)
  // generate instrumentation code for each structure graph
  // and counters used for adjusting profiles
  foreach sg in structure graphs of m do
    // generating instrumentation code of path profiling
    // by applying Ball-Larus path profiling to sg
    pplyBallLarusPathProfiling (sg)
    generate a profile counter to the counting node of sg
    foreach e in edges of sg do
      if (instrumentation code is assigned to e) then
        find insertion point p for the code of e
        register instrumentation code of e to p
      endif
    enddo
  enddo
  // insert instrumentation code into $m$ ¥¥
  foreach n in all regular nodes of m do
    if (hasInstInfo(n)) then
      insert instrumentation code to n
    endif
  enddo
  foreach e in all regular edges of m do
    if (hasInstInfo(e)) then
      insert a basic block b into e
      insert instrumentation code into b
    endif
  enddo
```

図３.８　Algorithm to generate instrumentation code.

最後に各構造グラフのインストルメンテーション情報を，メソッドの元の制御フローグラフ上の対応位置にインストルメンテーション命令として挿入する．仮想エッジ上のインストルメンテーション情報はその仮想エッジに対応する実際のエッジに対してコードの挿入が行われる．エッジ上に割り当てられているインストルメンテーション情報は，そのエッジの始点ノードの出力エッジが１本のみなら始点ノードの基本ブロックに，そうではない場合でそのエッジの終点ノードの入力エッジが一本のみなら終点ノードに対応する基本ブロックに，いずれも該当しなければ新しい基本ブロックを作成してコードを挿入する．インストルメンテーションコードの生成は，動的インストルメンテーション手法[30][35]を用いて実行時にインストルメンテーションコードの ON/OFF ができるように生成する．

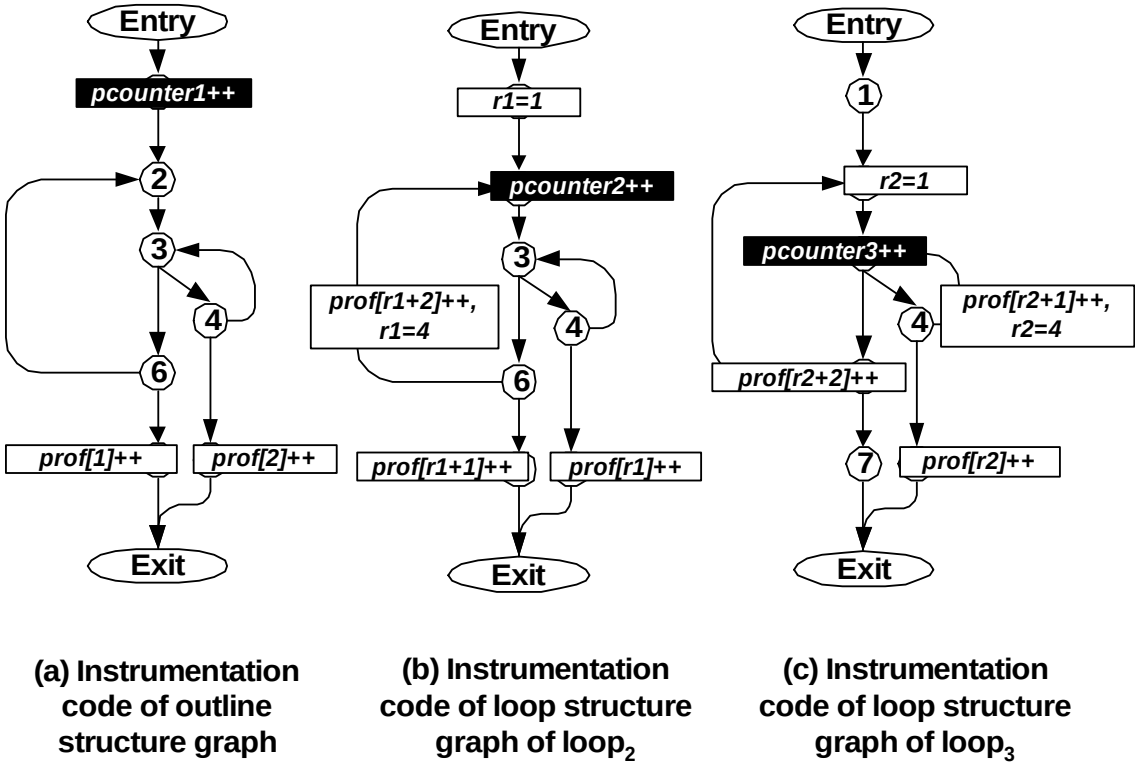


図３.９　インストルメンテーション処理の例

Ball-Larus の手法では，図３．９に示すように３種類のインストルメンテーション（経路番号の初期化，経路番号の更新，経路カウンタの更新）を経路上に配置し，これらが順に実行されながらパスレジスタ *r* を更新し，最終的にパスカウンタ *prof[r]* をインクリメントすることでプロファイルの収集を行う．このため，プログラムの実行状態を考慮せずにインストルメンテーションの収集を開始すると，パスレジスタの初期化処理を行わずにインストルメンテーション順序の途中から実行が開始されてしまう状況が発生する可能性がある．つまり経路番号の初期化が行われないために不正な経路番号がパスカウンタに到達してしまい，誤ったパスプロファイルを収集してしまう．この状況を完全に防ぐためには高い実行時コストが必要となってしまう．不正な経路番号が偶然経路数内の値をとることもあるが，これは最大でも同時に実行されるスレッド数以下であり，ほとんどの場合に無視できることから，現実装ではパスカウンタの更新で経路番号の範囲チェックを行い不正な値でのカウンタの更新を回避するにとどめいている．

３．３．４　プロファイル収集の管理

SPP では，各構造グラフに対するパスプロファイル収集処理をプロファイル制御機構で制御しながらパスプロファイルの収集を実施する．図３．１０にこのプロファイル制御機構のアルゴリズムを示す．ターゲットとするホットメソッドに対してパスプロファイルの収集は，そのメソッドの外郭構造グラフ *og* を引数として関数 *startProfiling(og)* メソッドを呼ぶことで処理が開始する．プロファイル収集中，プロファイルを実施している構造グラフのプロファイルカウンタが閾値に達すると，その構造グラフ　*sg* を引数として，関数 *profileController()* が呼ばれる．

関数 *profileController()* は，*isFinished[sg]* の値をチェックし，構造グラフ *sg* とそれ以下の階層の全ての構造グラフに対して既にプロファイルが終了したかどうかを確認する．まだ終了処理が実施されていない場合，まず構造グラフ *sg* に対するパスプロファイル処理を停止し，*sg* の直下の階層の各ループ階層グラフに対して順にプロファイルを開始する．もし *sg* が最下層の構造グラフの場合は，関数 *setCompletion(sg)* を呼んで，このメソッドのパスプロファイルが完了したかどうかチェックする処理を実施する．

プロファイル管理機構は，定期的にパスプロファイルが実施されている構造グラフに対して関数 *checkCompletion(sg)* を実行して，プロファイルの進行状況をチェックする．もしある構造グラフのパスプロファイル収集が長時間にわたって行われている場合，その構造グラフは実行頻度が低い領域と判断してプロファイル処理を停止し，次のレベルの構造グラフへとプロファイルを進める．最終的に，外郭構造グラフ *og* に対して *isCompleted[og]* が TRUE となった時点でそのメソッドに対するプロファイル処理が完了となる．

図３．１１に，図３．６の例に対するプロファイル制御の例を示す．図中の数字は処理順序を表している．まずタイマー・サンプリング・プロファイラにより検出されたホットメソッドの外郭構造グラフに対して開始される．そのエントリーカウンタが閾値に達した時点で外郭構造グラフに対するパスプロファイル処理を停止し，構造グラフの階層構造に従って，下階層へ向かって順にパスプロファイ

ル処理を実施する．全ての構造グラフに対してパスプロファイルの収集が終了した時点で，メソッドに対するパスプロファイル処理が完了する．図３．９(a)-(c)は，各構造グラフに対してインストルメンテーションを有効にした状態を示しており，図３．１１の(2)，(6)，(10)において有効化されるインストルメンテーション処理にそれぞれ対応している．

```
// All the elements of isStarted[*],isFinished[*], and isCompleted[*] are initialized
// with FALSE.

procedure startProfiling(Graph sg)
    isStarted[sg] := TRUE
    enable all instrumentation code blocks of sg

procedure profileController(Graph sg)
    if (not isFinished[sg]) then
        disable all instrumentation code blocks of sg
        isFinished[sg] := TRUE
        if (sg has inner loop structure graphs) then
            foreach lg in inner loop structure graphs of sg do
                if (not isStarted(lg)) startProfiling(lg)
            enddo
        else
            setCompletion(sg)
        endif
    endif

procedure setCompletion(Graph sg)
    isCompleted[sg] := TRUE
    if (sg is a loop structure graph) then
        og := outer structure graph of sg
        foreach lg in inner loop structure graphs of sg do
            if (not isCompleted[lg]) return
        enddo
        setCompletion(og)
    endif
```

図３．１０ Algorithm for profile controller.

```
procedure checkCompletion(Graph sg)
    if (isCompleted[sg]) return
    if (not isFinished[sg] and elapsedTime(sg) > threshold) then
        profileController(sg) // time out for sg
    else
        foreach lg in inner loop structure graphs of sg do
            if (not isCompleted[lg]) then
                checkCompletion(lg)
                if (not isCompleted[lg]) return
            endif
        enddo
        isCompleted[sg] := TRUE
    endif
```

図３.１０ Algorithm for profile controller. (continued)

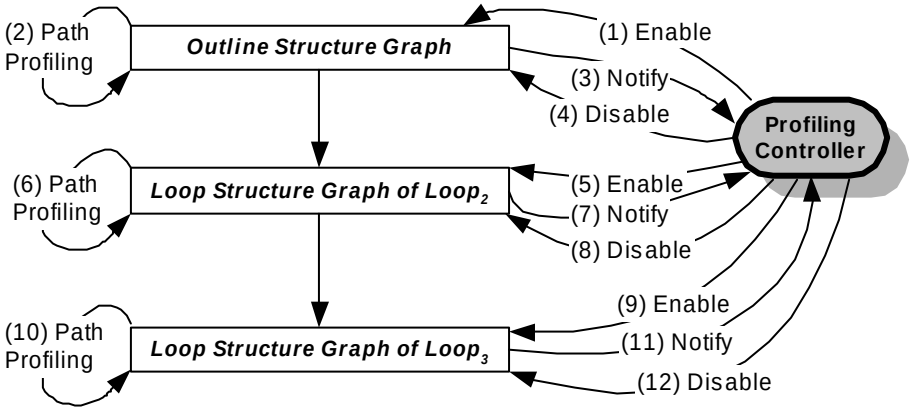


図３.１１　プロファイル制御機構

３．３．５　グローバル・プロファイルの生成

最後に収集した各構造グラフのローカル・パスプロファイルを補正して，実際の実行頻度に相当するグローバル・パスプロファイルを生成する．この補正処理は，各構造グラフのローカル・パスプロファイルに実際の頻度に応じた補正係数 A_x を乗じることで行う. ある構造グラフにおけるパスプロファイル情報は，そのループの外からループへ入る経路の実行頻度に対する相対的な頻度と考えることができる．従って，ループ外からループへ入る経路の実行頻度と，上位層の構造グラフ中にあるループノードの実行回数との比（補正係数）を計算し，この値をその構造グラフの各パスプロファイル情報の値に掛け合わせることで，この２つの構造グラフ間のパスプロファイル情報の値が対等な関係と

なる．以下に，ループ構造グラフXの補正係数A_Xの計算式を示す．ここで，C_pは，経路pに対して収集された実行回数のプロファイル値，$P_X(a)$は構造グラフ X 中のノードaを通る経路の集合，$P_X\ (entry)$は構造グラフXにおいてループ外からループへ入る経路の集合，YはXの上位層の構造グラフ，N_Xは構造グラフXに対応する上位階層の構造グラフ中のループノード，をそれぞれ表すものとする．

$$A_X = \begin{cases} 1 & if\ X \text{ is outline} \\ \dfrac{A_Y \cdot \sum_{q \in P_Y(N_X)} C_q}{\sum_{p \in P_X(entry)} C_p} & \text{otherwise} \end{cases}$$

この式を用いて補正係数を計算するアルゴリズムを図３．１２に示す．プロファイルを収集したメソッド m がループを含む場合，このメソッドに対して関数 $adjustSPP(m)$を実行すると，外郭構造グラフから順に，関数 $adjustProfiles(sg)$によって各構造グラフに対する補正係数 $A[sg]$ が計算される．

補正処理の例として，表３．２に示してあるローカル・プロファイルからグローバル・プロファイルを生成する手順を示す．表の各項は，構造グラフ毎に，パスインデックス(Path index)，経路リスト(Paths),収集したローカル･プロファイル(Local Profile),補正したグローバル･プロファイル(Global Profile)，プログラムの全実行を通じて収集した場合のオフラインプロファイル(Offline Profile)，および対応する BPP のパスインデックス (Path index of BPP)を示している．BPP のパスインデックスでカッコに入っているインデックスは，そのパスの一部分が対応していることを示している．

$Loop_2$の補正係数A_Xは，Yが外郭構造グラフであるのでA_Y=1 となり，また外郭構造グラフの経路は全てループノード２を通ることから，補正係数A_Xの計算式の分子は(1 * 100)となる．一方$Loop_2$を表す構造グラフの経路のうちで，始点ノードからの経路は，L2-1，L2-2，L2-3 の３つの経路であるので，分母は 0+50+0=50 となる．従って，$Loop_2$の補正係数A_Xは，(1*100)/51=1.96 となる．$Loop_2$のグローバル・プロファイルは，ローカル・プロファイルの各経路の値に 1.96 をかけた値となる．同様に$Loop_1$の補正係数は，Yは$Loop_2$の構造グラフとなるので，A_Y=1.96 となる．$Loop_2$の構造グラフにおいて全ての経路がループノード３を通るので，補正係数A_Xの計算式の分子は(1.96 * 100)となる．一方$Loop_3$を表す構造グラフの経路のうちで，始点ノードからの経路は，L3-1，L3-2，L3-3 の３つの経路であるので分母は 0+2+56=58 となる．従って，$Loop_3$の補正係数A_Xは，(1.96*100)/58=3.38 となる．この値をローカル・プロファイルの各経路の値に乗じることでグローバル・プロファイルを得ることができる．表よりグローバル・プロファイルの各経路の比率とオフラインプロファイルの各経路の値の比率が非常に類似していることがわかる．

```
procedure adjustSPP(Method m)
    if (loop structure graphs exist in m) then
        og := get outline structure graph of m
        A[og] := 1
        adjustProfiles(og)
    endif

procedure adjustProfiles(Graph sg)
    foreach ln in loop nodes in sg do
        // calculate the frequency of ln
        loopCount := 0
        foreach p in paths passing though ln do
            loopCount += profile count of p
        enddo
        lg := loop structure graph of ln
        entryCount := 0
        foreach p in paths starting from the entry node of lg do
            entryCount += profile count of p
        enddo
        A[lg] := A[sg] * (loopCount / entryCount)
        if (lg has inner loop structure graphs) then
            adjustProfiles(lg)
        endif
    enddo
```

図３．１２　Algorithm to generate global profiles.

表３.２ Profile Results

Graph	Path index	Paths	Online profiles		Offline profiles	index of BPP
			Local profiles	Global profiles		
Outline Structure	O-1	E1-1-[2]-7-1	100	100.0	10000	3,(2,4,7,11)
Graph	O-2	E1-1-[2]-5-1	0	0.0	0	1,(2,4,5,9)
Loop Structure	L2-1	E2-2-[3]-X2	0	0.0	0	1,(5)
Graph of Loop2	L2-2	E2-2-[3]-6-X2	50	98.0	9800	3,(2,7)
	L2-3	E2-2-[3]-6-(2)	1	2.0	200	4,(8)
	L2-4	(6)-2-[3]-X2	0	0.0	0	9,(5)
coefficient	L2-5	(6)-2-[3]-6-X2	1	2.0	200	11,(7)
1.96	L2-6	(6)-2-[3]-6-(2)	48	94.1	9600	12,(8)
Loop Structure	L3-1	E3-3-4-X3	0	0.0	0	1,9
Graph of Loop3	L3-2	E3-3-4-(3)	2	6.8	683	2,10
	L3-3	E3-3-X3	56	189.3	19117	3,4,11,12
	L3-4	(4)-3-4-X3	0	0.0	0	5
coefficient	L3-5	(4)-3-4-(3)	40	135.2	13660	6
3.38	L3-6	(4)-3-X3	2	6.8	683	7,8
Total			300	634.1	63943	

３．３．６ 最適化の例

表３.２に示されるグローバル・プロファイルから，このメソッドの３つのホットパスを求めることができる．まず，グローバル・プロファイルから，**O-1**，**L2-2**，**L2-6**，**L3-3**，**L3-5** の５つの経路がホットパスであることが分かる．このうち，**O-1**，**L2-2**，**L2-6** はループノードを含む経路である．経路 **L3-5** はループノードを含まず，また外側ループとの接続も持たないことから，この **L3-5** の経路，***3-4-(3)*** がこのメソッドのホットパスであることが分かる．また，**L2-2**，**L3-3** の経路がホットパスとなっていることから，２つのループはほとんどの場合繰り返しが行われずに実行が終了してしまうことが分かる．従って，**O-1**，**L2-2**，**L3-3** のホットパスから，経路 ***1-2-3-6-7*** がホットパスであることが分かる．同様に，**L2-6**，**L3-3** の組み合わせから，経路 ***2-3-6-(2)*** もホットパスであることが分かる．図３．１３(a)-(c)は，表３.２のグローバル・プロファイルから検出された各構造グラフにおいてホットパスを太線で示しており，またそれらの組み合わせによって得られた結果が図３．１３(d)に示されている．

収集されたプロファイルから推定したホットパスの情報を用いて，２つの最適化手法，loop peeling と Pettis と Hansen のコードレイアウト手法[8]を適用することができる．ここでは Loop peeling はホットパスを分離する方法として利用し，コードレイアウトはホットパスのコードが連続領域に配置されるために利用する．まずパスプロファイル情報からループを繰り返さない経路とループを繰り返す経路がそれぞれ独立したホットパスであることがわかり，Loop Peeling を適用してこれらのホットパスを分離する．次に，変換後のパスプロファイル情報から生成したエッジプロファイルを用いて Pettis と Hansen の基本ブロックの並べ替えアルゴリズム[8]を用いてホットパスが連続になるように配置する．これらを適用した結果が図３．３(a)である．

この変形は３つの利点がある．第１点は，分岐命令の分岐頻度が特定の方向に大きく偏る（ノード３とノード６）ことで，ハードウェアの分岐予測の適中率が向上し，命令パイプラインの効率が改善される点である．これは，本変形によってノード３とノード６の分岐方向がほぼ一方向に定まったためである．第２点，は独立した３つの Hot Path がそれぞれ連続になるように配置することで，コードのローカリティが向上し，I-キャッシュ効率が上がる点である．３つめとしては，第３点は無条件分岐や前方分岐の数が激減している点である．

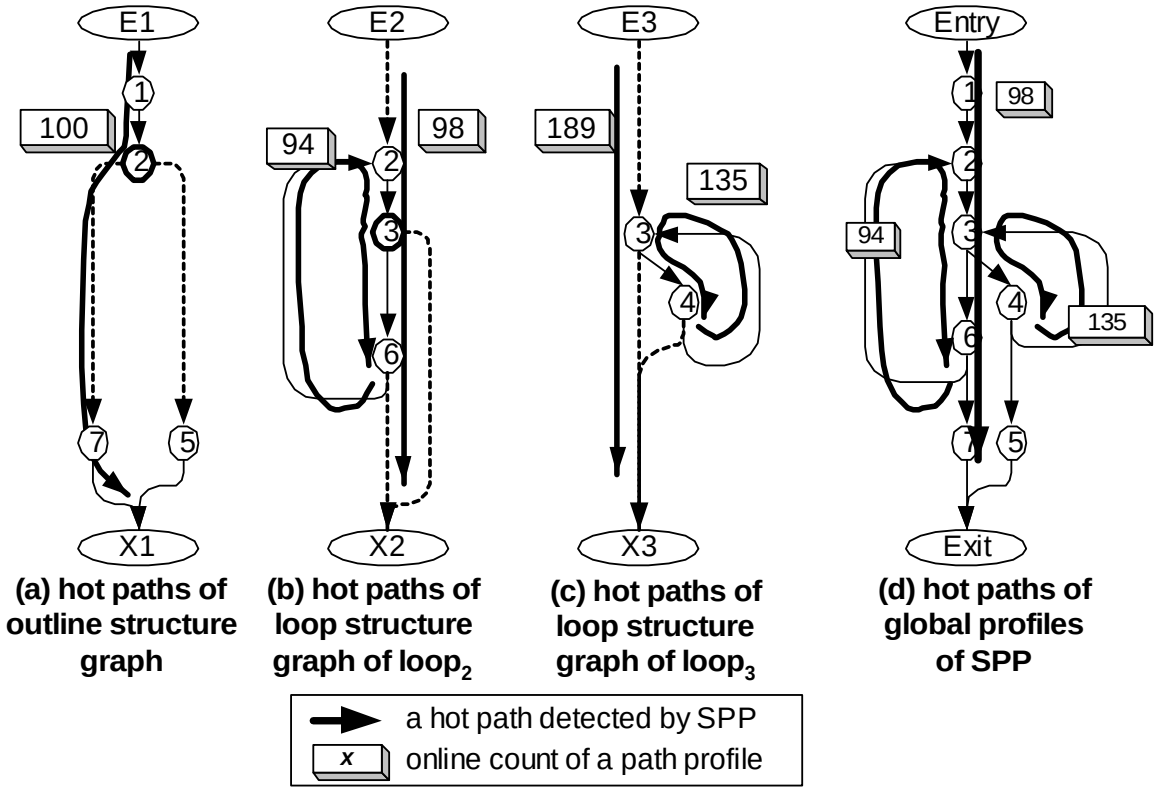


図３.１３　Hot paths detected by SPP.

３．４　パスプロファイルを用いた Loop Peeling 手法

前節において，SPP によるプロファイル収集手法の説明を行った．SPP の目的は，そこで収集したプロファイルを用いて最適化を行い，プログラムをさらに高速化することにある．本節では，SPP を用いて収集したプロファイルを用いた最適化の実装として，パスプロファイルを用いた Loop Peeling 手法について述べる．

Loop Peeling[24][60]は，ループの最初の何回かの繰返しの実行分をループの直前に展開（複製）する手法である．Loop Peeling は，もともとループの最適化を阻害する要因をループ外へ移動する手法として用いられてきたが，近年のプログラムの実行履歴を用いた最適化手法の発展により，動的分

岐予測の向上，I-cache 効率の向上，およびほとんど実行されないループのループ制御処理の除去などを実現する手法として利用されるようになった．実際，現実のプログラムにおいては，ループの実行挙動が最初の数回の実行とそれ以降の実行の動作が異なるループが存在する．例えば，ほとんどの場合にループが１回しか実行されない場合や，１回目の実行と２回目以降の実行で実行される経路が異なる場合などである．このようなループに対して Loop peeling を適用して，実行挙動の異なる最初の数回の実行をループ外へ取り出してそれ以降の実行と分離することで，それぞれの部分でのプログラムの実行挙動を安定させて高速化を実現することができる．ループ外へ移動するイテレーションの数 N は，ループ内の不変変数や依存関係を解析して求める方法などが提案されている．しかしながら，Loop peeling はコードの複製を伴う最適化のため，常に適用してしまうと，コード増大の問題を生じるだけでなく， I-cache ミスを増加させてしまう原因にもなる．不要なコード増加を抑えながら効果を引き出す適用方法が必要である．

プロファイルを用いた Loop Peeling の適用方法に関する従来手法としては，edge profile を用いる方法[38]や，K-bounded general path profile を用いる方法[36]などが提案されている．前者は edge profile を使用してループの繰返し回数を判断し，繰返しが少ないループに対して，ループ境界をまたいで Superblock formation を適用することで loop peeling を実施する方法である．しかしながら，edge profile の性質上，ループの挙動が edge profile で示される実行頻度で均一に実行されることを仮定しており，図３．１のプログラムのように，多くの場合は１回目のイテレーションでループを抜けるが，一度ループが繰返された場合には引き続き多数回の繰返しを実行するようなループに対しては，効率的に loop peeling を適用することができない．他方，Young と Smith の手法は，K-bounded general path という経路情報を用いた superblock formation により loop peeling を実施する手法である．K-bounded general path 情報は，プログラム中の任意の点における，その点に到る長さ K の経路に対する実行頻度情報を提供する．したがって，経路長 K をループ内の経路長を考慮して適切に設定することで，初回のイテレーションの振る舞いに基づいて loop peeling を実施することが可能となる．しかしながら K-bounded general path の収集コストは最適化による短縮時間に比べて非常に高いために，プロファイル時間が実行時間に加味される動的コンパイラにおいては現実的に利用することが難しい．

ここでは，Ball と Larus のパスプロファイル収集手法[3]で収集されるパスプロファイルにより表される経路情報（以下，Forward Path と呼ぶ）を用いて loop peeling を実施する方法を示す．Forward Path は，ループを形成する backedge を経路の分割点とすることで，コントロールフローグラフ（以下ＣＦＧ）上の経路を，有限個の有限長経路の集合として表現する．従って，ループ内の全ての経路は，ループ外よりループに入る経路と，backedge を始点としてループの繰り返し時の経路に大別できる．本手法では，ループの最初の実行パスの実行傾向とループ繰り返し時のパスの実行傾向をそれぞれ求め，ループの最初の実行パスがループを繰り返さない経路であり，かつループ繰り返し時の実行パスがループを繰り返す経路である，という状態を検出した場合にループを１回分剥くことで loop peeling を実施する．

３．４．１　経路の分類と実行頻度の計算

対象ループの挙動を調べるために，Forward Path Profile 中の情報のうちで，対象ループの外からループに入る各 edge(entry edge)を含む経路と,対象ループの backedge を始辺とする経路を利用する．これらの経路を，以下の４つの経路の集合に分類して収集された経路情報より各分類の実行頻度を計算することでループの挙動を解析する．

P1：対象ループの外からループに入る edge を含み，ループを脱出する edge を含む経路の集合

P2：対象ループの外からループに入る edge を含み，ループの backedge を終端辺とする経路の集合

P3：対象ループの backedge を始辺とし，ループを脱出する edge を含む経路の集合

P4：対象ループの backedge を始辺とし，ループの backedge を終端辺とする経路の集合

次に,以下の３つのいずれかの条件が成立する場合,そのループに対して Loop Peeling を実施する．

(1) ループのbackedgeを始辺とする経路において，ループのbackedgeを終端辺とする経路のプロファイル数 N_{P4} とループを脱出するedgeへの経路のプロファイル数 N_{P3} の比から計算される平均繰返し回数 N_{P4} / N_{P3} が，規定値 R_{min} よりも小さい場合

$$\frac{N_{P3}}{N_{P4}} < R_{\min}$$

(2) ループのentry edgeを含む経路において，ループのbackedgeを終端辺とする経路のプロファイル数 N_{P2} とループを脱出するedgeを含む経路のプロファイル数 N_{P1} の比 N_{P2} / N_{P1} が，規定値 R_{ENTRY} よりも小さい場合

$$\frac{N_{P2}}{N_{P1}} < R_{ENTRY}$$

(3) ループのentry edgeを含む経路における各経路の実行頻度の分布 P_{ENTRY} と，backedgeを始点とする経路における各経路の実行頻度の分布 $P_{BACKEDGE}$ のOverlap percentageの値 $OP(P_{ENTRY}, P_{BACKEDGE})$ が，規定値 S よりも小さい場合

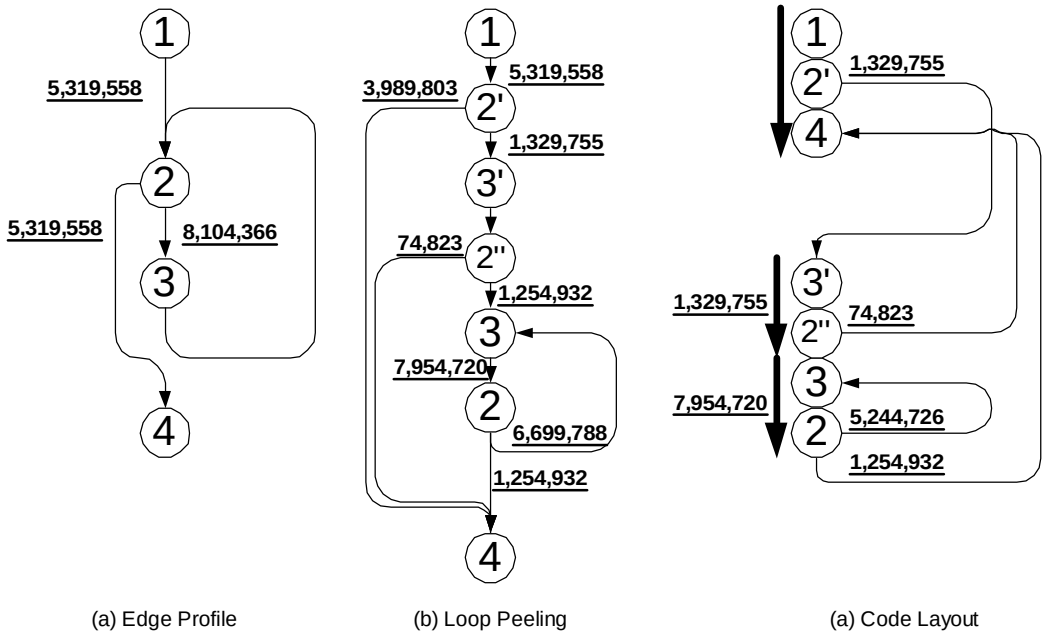
$$OP(P_{ENTRY}, P_{BACKEDGE}) < S$$

３．４．２　適用例

本例では,SPECjvm98 の mtrt で実行される Scene.FindLightBlock というメソッドの最内ループの実行の振る舞いを簡略化したＣＦＧで表した例を用いて説明する．図３．１４(a)は，このループの実

行履歴の edge profile を表している．edge profile から，このループの平均繰返し回数は 1.5 回（≒ 8,104,366/5,319,558）となる．コードサイズ増加を考慮して loop peeling の実施回数を１回とした場合はこのループは loop peeling されない．またコードサイズ増加の制限を緩めて２回までの loop peeling を許した場合，loop peeling の結果は図３．１４(b)のようになり，edge profile を用いて基本ブロックを並べなおした結果は，図３．１４(c)のようになる．

一方，同じループに対する forward path profile は図３．１５(a)のようになる．このループでは，entry edge からの経路（図中の実線）では，約 75%の実行が最初の繰返しでループを終了することを表している．一方，ループを繰り返した場合の経路は，backedge からの経路（図中の点線）で表され，平均５回（≒6,774,611/1,329,755）繰り返されることがわかる．本手法を用いた場合，まずこのループは平均５回繰り返されるので，条件(1)による loop peeling は実施しない．次に，entry edge からの経路の実行履歴により，このループは１回目の実行でループを終了する場合が多い（75%）ことから条件(2)に適合し，loop peeling の対象として検出される．loop peeling をした結果は，図３．１５(b)となり，さらに code layout した結果は図３．１５(c)となる．図３．１４(c)と図３．１５(c)はともに，loop peeling をしない場合に比べて，条件分岐による taken jump は 2,613,598 回少なくなっていることがわかる．しかしながら，Edge profile を用いた従来手法の場合，１回余分に loop peeling しているためにコード増加を招いているとともに，繰返し回数の多いループを peeling してしまったことにより，I-cache 効率を低下させてしまっている．これらの点から，本発明が edge profile を用いた従来手法に比べて効率よく loop peeling を実施できることがわかる．



(a) Edge Profile (b) Loop Peeling (a) Code Layout

図３.１４ エッジプロファイルを用いた Loop Peeling

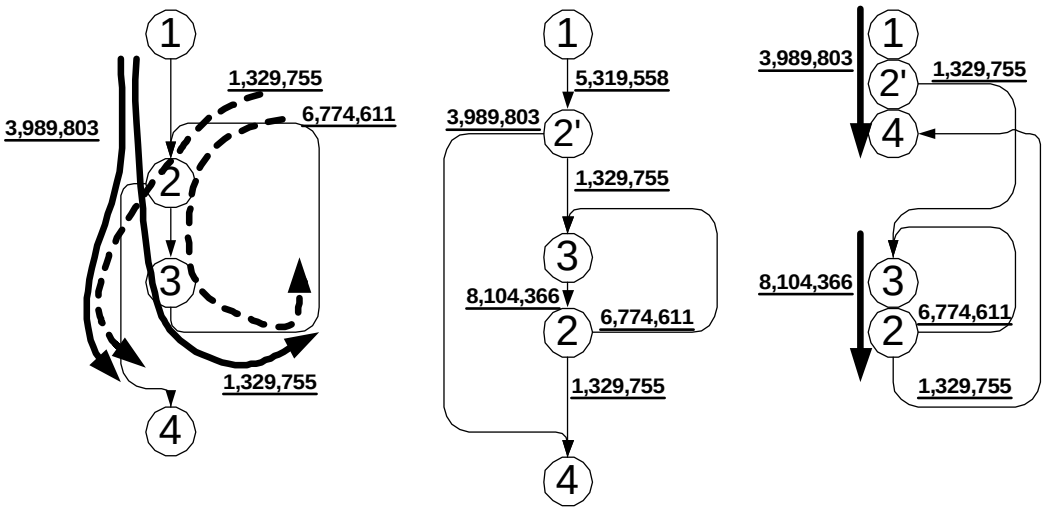


図３.１５　Forward path プロファイルを用いた Loop Peeling

３．５　評価

本節では BPP に対する SPP の有効性を検証するために，いくつかの評価結果を示す．本評価のために，SPP を IBM Java Just-In-Time コンパイラ上に実装した．各構造グラフでのパスプロファイル収集手法として Ball-Larus のパスプロファイル収集用のインストルメンテーションコード生成アルゴリズムを利用した．実験で使用した IBM Java VM はインタプリタ実行を含む多段階コンパイルシステムである[34]．システムはホットメソッドを検出すると，そのメソッドの最適化レベルを１段上げて最適化を実施する．評価では，(1)インタプリタ実行，(2)高速コンパイル実行，(3)最適化コンパイル実行の３段階を設定し，最適化コンパイル実行への遷移を決定したメソッドに対してのみ SPP を利用したパスプロファイルの収集を実施する．

３．５．１　評価方法

全ての評価は，IBM Aptiva A-series 6832 (Pentium 4 2 GHz Uniprocessor, 512 MB memory)で動作する Windows 2000 SP2 上で測定した．使用した Java JVM は，IBM Developer Kit for Windows, Java Technology Edition version 1.3.1 prototype build を使用した．

評価に用いたベンチマークは SPECjvm98 1.04，SPECjbb2000[33]，jBYTEmark の３つである．SPECjvm98 ベンチマークは，インタラクティブモード，デフォルトの入力サイズ，初期ヒープサイズ 256MB，最大ヒープサイズ 256MB で実行した．SPECjbb2000 ベンチマークは，パスプロファイルのオーバヘッドの程度を測るために，SPEC の標準的な測定方法とは異なり，１ウェアハウスに対して ramp-up を行わない実行をもちいて評価を行った．jBYTEmark は，個々のテストを別々の JVM で実行した．

SPP の比較対象である BPP は，３．２．４ で定義したとおり， Ball-Larus の手法を動的に適用す

るように変更した方法である．対象メソッドに対して Ball-Larus の手法に対して各インストルメンテーションコードを生成してプロファイルを収集する．またメソッドに対する総収集プロファイル数をカウントして，その数が閾値に達した時点でプロファイルを終了する．

SPP, BPP ともに，各メソッドに対して同じ数のパスプロファイルを収集するように閾値を設定した．SPP では，メソッド中の各構造グラフに対して，指定された閾値をグラフ数で割った値分だけ経路情報を収集した．例えば構造グラフが４つあるメソッドで，閾値 1000 が指定されている場合，各構造グラフではそれぞれ 250 の経路情報を収集し，メソッド全体で 1000 の経路を収集する．

パスプロファイルを格納するバッファのサイズを制限するために，SPP，BPP ともに経路数が 1000 以内のメソッドのみにプロファイルを実施した．もし対象メソッドの経路数が 1000 を超えた場合は，パスプロファイル収集の代わりに，エッジプロファイルを収集した．今回使用したベンチマークでは，経路数が 1000 を超えるメソッドは高々数個であり，評価にはほとんど影響を与えない．

３．５．２　メモリ使用量

ここでは，SPP と BPP の実行に必要なメモリ使用量について評価した結果を示す．表３．３において，最初の２つの欄は，プロファイルを格納するバッファのサイズの指標として，SPP と BPP の各ベンチマークにおける静的な経路の総数を示している．次の２つの欄は，インストルメンテーションコードによるコードサイズの増加の指標として，SPP と BPP によって挿入されるインストルメンテーションコードの挿入点の総数である．各インストルメンテーションコードは，通常２～１９命令で構成される．

表３．３　経路数とインストルメンテーションポイント数の比較

		Total number of paths		Total number of instrumentation code blocks	
		SPP	*BPP*	*SPP*	*BPP*
SPECjvm98	mtrt	2,617	337	909	807
	jess	4,332	5,534	1,642	1,420
	compress	2,142	2,829	627	560
	db	2,324	2,943	755	671
	mpegaudio	2,673	3,255	1,021	879
	jack	4,379	5,197	1,365	1,224
	javac	12,234	13,563	4,376	3,780
SPECjbb2000		7,912	9,321	2,744	2,224
jBYTEmark	Assignment	1,037	1,781	500	361
	Bit field Ops	851	1,334	366	275
	FFT	816	1,297	355	263
	FP Emulation	1,235	1,774	520	424
	Huffmann	933	1,600	419	305
	IDEA	851	1,333	377	277
	LU	956	1,553	433	314
	NeuralNet	936	1,413	434	313
	Num Sort	864	1,325	1,325	270
	String Sort	892	1,393	397	290
Total		47,966	60,815	18,565	14,657
Ratio		0.789	1.000	1.267	1.000

この表より，SPP の経路数は BPP に対して平均で約 21%減少することがわかる．これは，SPP がループの各ネストを独立のグラフに分割し，ネスト間に跨る経路を除去したためである．例えば，図３．１６に示すメソッドでは，SPP が１０の経路となるのに対して，ＢＰＰは１２の経路を生成する．

一方，インストルメンテーションコードの挿入ポイントの数は，ＳＰＰの必要数は，ＢＰＰに対して約 26%増加している．Ball-Larus のパスプロファイル手法では，インストルメンテーションコードは，パス変数の初期化および更新と，最終的な経路数のカウントを行う点に挿入される．ＳＰＰはメソッドを複数のグラフに分割して，それぞれのグラフに対してインストルメンテーションコードを挿入するので，パス変数の初期化や経路数のカウントのためのインストルメンテーションがＢＰＰよりも増加してしまう．例えば，図３．１のプログラムに対するインストルメンテーションコードの挿入点の数は，ＳＰＰでは図３．９から分かるとおり１０箇所にインストルメンテーションコードが挿入されるが，ＢＰＰでは図３．４から分かるとおり５箇所にインストルメンテーションコードを挿入するだけとなる．

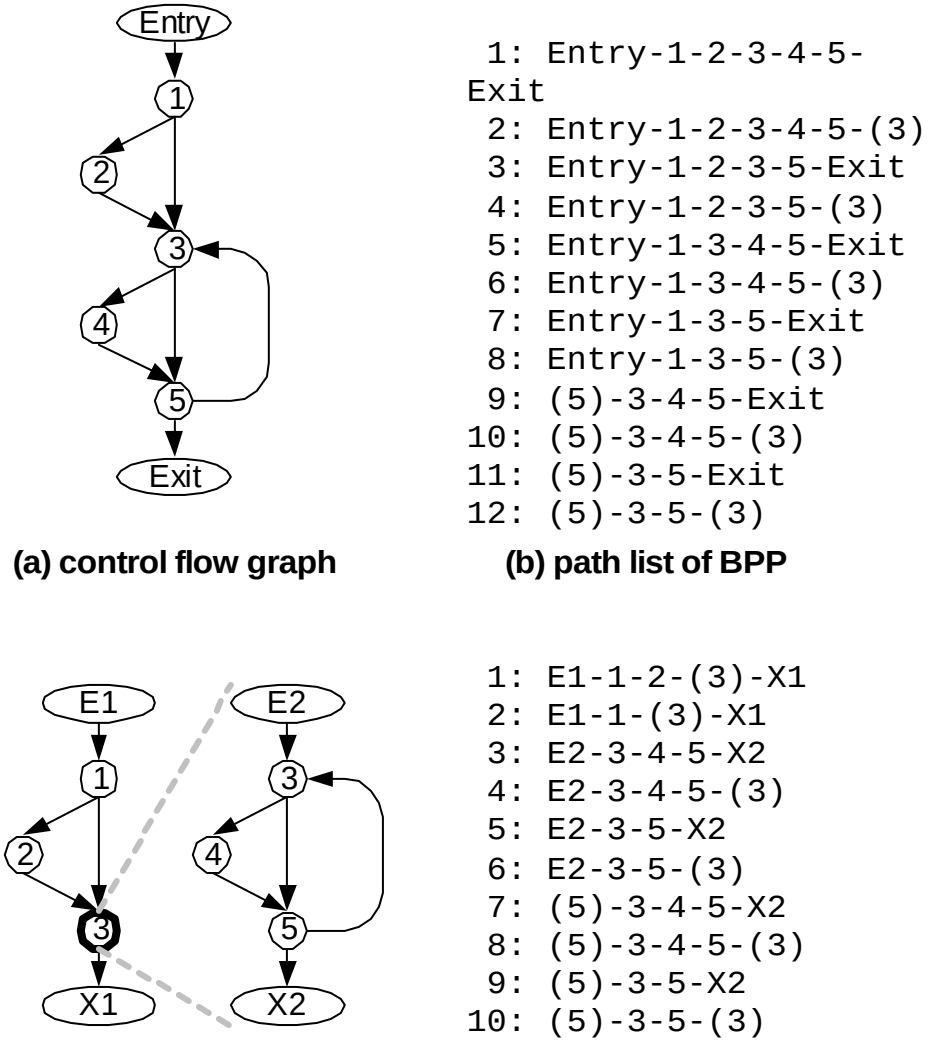


図３.１６　Differences in the numbers of paths between SPP and BPP.

メソッドによっては構造グラフに分解することで経路数が増加する場合が存在するが，評価結果より SPP の経路数は BPP に対して平均で約 20%減少することがわかる．経路数は分岐の数に指数関数的に影響するために，グラフを分割することで経路を減少させていることがわかる．一方 SPP のインストルメンテーションポイント数は BPP に対して約 14%増加している．これは，グラフが分割されることで主に経路番号の初期化処理が増加したことが主な原因である．

３．５．３　精度の評価

パスプロファイル収集における我々の目的は，実行頻度の高い経路および実行頻度の少ない経路のともに信頼できる情報を収集することであり，その信頼度は結果として，最適化におけるコストと利得の評価の正確さにつながる．したがって，この目的においては単に実行頻度の高い経路を検出できたかどうかの評価[41]ではなく，収集された全経路の情報に対する正確さを評価する必要がある．

収集されたパスプロファイル情報の精度を評価する方法として，類似度の評価方法として，Arnold と Ryder が Instrumentation sampling 手法[25]の評価に使用した手法である overlap percentage という評価手法[27]を用いた．overlap percentage では，収集したプロファイルにおける各経路の実行頻度の比率が，プログラムの全実行を通じて収集した場合の比率とどのていど類似しているかという点で比較する．つまり，各項の比率で表される２つの多項分布に対して重なる領域の面積を計算することでそれらの類似度を求める．各パスプロファイルの大小関係や頻度の低い部分の類似性も算定に加えられるため，最適化を目的としたパスプロファイルの精度を評価するうえで適した評価尺度の一つといえる．例えば表３．２で示される SPP の overlap percentage は 99.7%であり，表３．１で示される BPP の overlap percentage は 69.7%となる．

図３．１７は，収集する実行経路数の閾値を 100 から 10000 まで変化させた場合の，overlap percentage による SPP と BPP の精度を表している．評価結果より，compress と db は全域にかけて SPP の精度が BPP の精度を上回っていることがわかる．これは，３．２．４ で述べたとおり BPP がいくつかのホットパスをプロファイルし損ねているためである．また，mpegaudio では SPP は少ない閾値でも高い精度を出していることが分かる．

一方，jess，jack，javac では，いくつかの閾値において，SPP が BPP より若干低い精度を示している．これは，SPP では，各メソッドにおいて与えられるプロファイル数を各ループネストを表す構造グラフに均等に分配しているために，ループの数が非常に多いメソッドでは個々の構造グラフに対して少量のプロファイルしか収集できないために収集されるプロファイルの精度が下がってしまったためである．この問題に対しては，メソッド内の各構造グラフに対して均等にプロファイル数を割り当てるのではなくグラフの形状などから適切なプロファイル数を割り当てる方法や，構造グラフ１つ当たりに割り当てるプロファイル数の下限値を設定する方法などにより改善することが可能である．

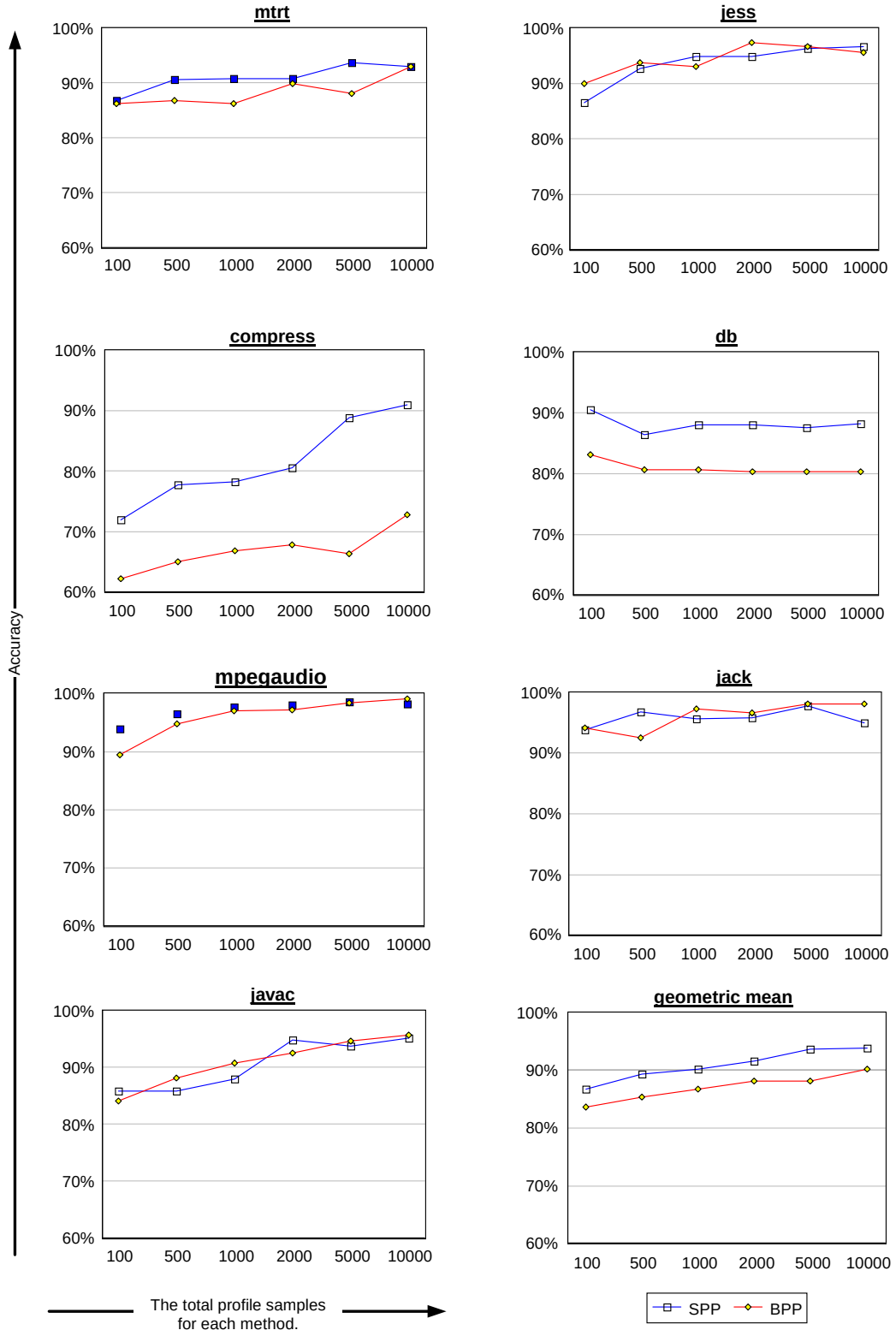


図３.１７　SPECjvm98 ベンチマークを用いた Overlap Percentage による各閾値での SPP と BPP の精度の比較

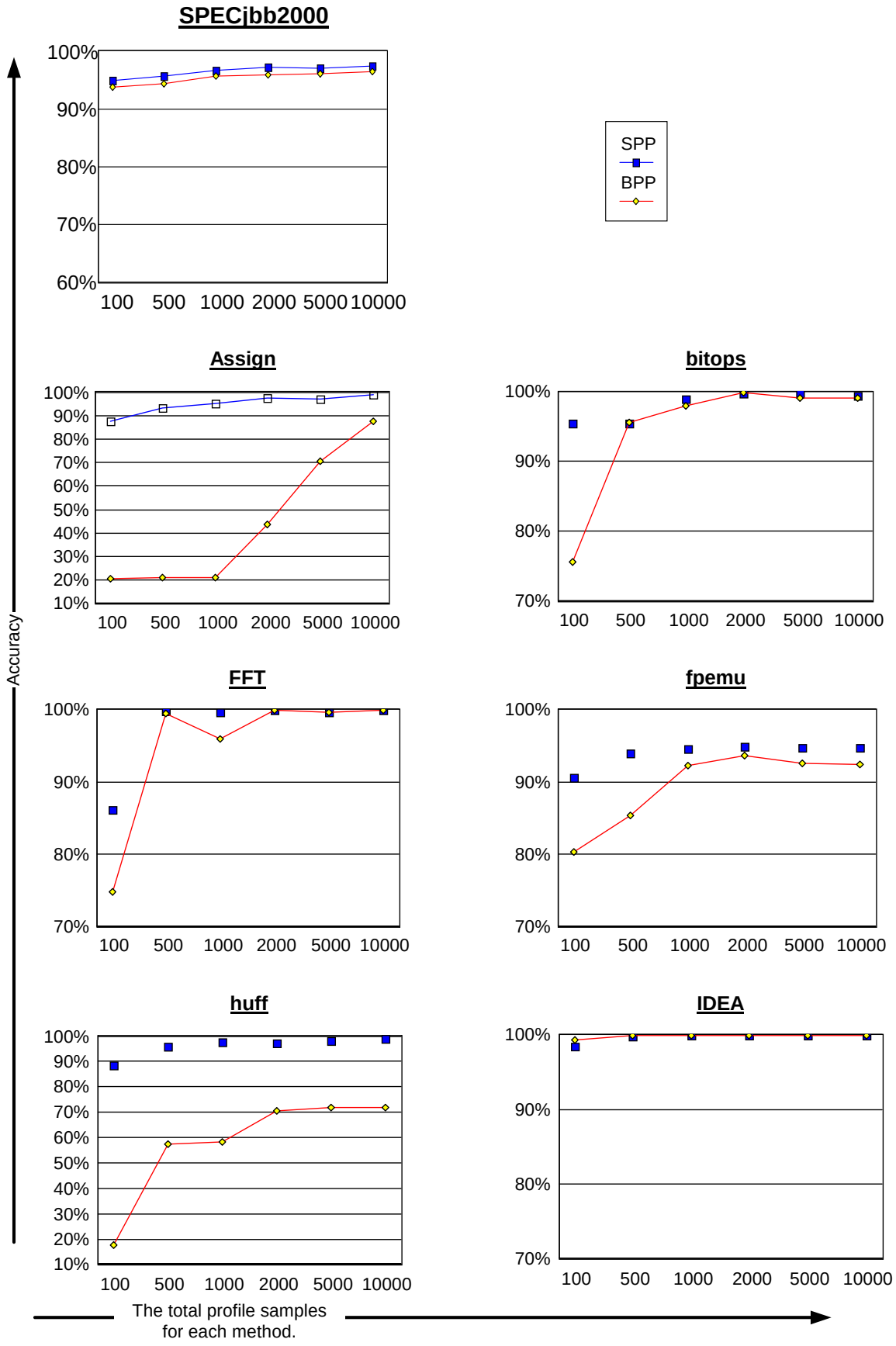


図３.１８　jBYTEmark ベンチマークを用いた Overlap Percentage による各閾値での SPP と BPP の精度の比較

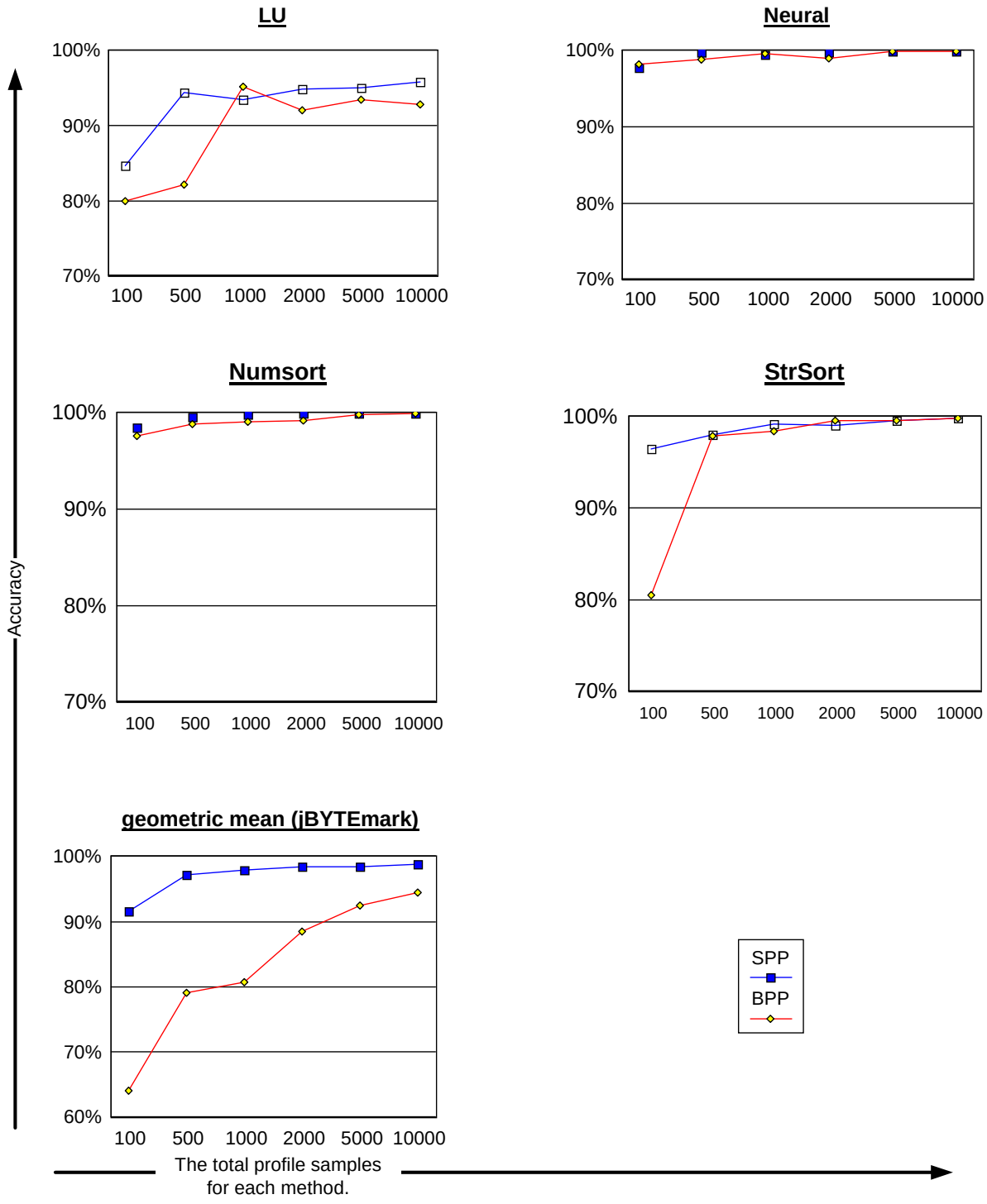


図３.１８　jBYTEmark ベンチマークを用いた Overlap Percentage による各閾値での SPP と BPP の精度の比較（続き）

図３．１８は，同じ比較を SPECjbb2000 ベンチマークと jBYTEmark ベンチマークに対して行った結果である．特に jBYTEmark ベンチマークのほとんどのテストが数値計算プログラムを元にしておりほとんどの条件分岐の予測が一定方向に分岐することから，SPP では非常に高い精度のプロファイルが収集できている．一方 BPP では幾つかのテストにおいて閾値が低い範囲で非常に低い精度のプロファイルとなっている．これも３．２．４ で述べた理由によるものである．

全体としてみると，SPECjvm98 ベンチマークの幾何平均(geometric mean)において，SPP は閾値 1000 で 90%の精度を出しているが，同じ精度を出すために BPP では閾値を 10000 にしなければならないことが分かる．また，jBYTEmark の幾何平均においては，SPP の精度は全ての閾値に対して 90%を超えている一方で，BPP は 90%の精度を達成するのに 5000 のプロファイルを収集するひつようがあることがわかる．以上より，パスプロファイル情報の収集コストは同じ閾値の場合は SPP も BPP もほとんど変わらないので，SPP は BPP よりも少ないオーバヘッドで高い精度のプロファイルを収集できるといえる．

３．５．４　プロファイル収集のオーバヘッド

SPP が実際の実行に与えるオーバヘッドとして，実行速度の低下を閾値を 100 から 10000 まで変化させた場合について測定した結果を図３．１９に示す．ここで SPP のみを評価しているのは，SPP と BPP は各メソッドに対して同数の実行経路数を収集するので，同じ閾値に対する実行時オーバヘッドはほとんど変わらないためである．またコンパイル時間の差についても，SPP は BPP に比べてループの検出と構造グラフの生成処理を余分に要するが，基本的には線形時間の処理でありコンパイル時間全体に対する影響の差は無視できる．

図３．１９の各グラフは，各ベンチマークを１回だけ実行した時に，プロファイルを実施しない場合に対するプロファイルを実施した場合の速度低下率を表している．つまり，この速度低下率は，インストルメンテーションコード生成にかかる時間，実行中のパスプロファイル収集処理にかかる時間，およびパスプロファイルの実施により生じる最適化のための再コンパイルの遅延による損失時間など，SPP の実施に伴う全てのオーバヘッドが含まれた値を示している．また１回目の実行を測定するのは，１回目の実行では多数のメソッドに対してプロファイルが実施されるため，プロファイルに対する速度低下の影響が著しく現れるためである．実行の回を重ねて定常状態に達した時点では，ほとんどのホットメソッドは際コンパイルされた状態となり，新たなプロファイル処理が発生しないため SPP によるオーバヘッドは発生しない．

compress と db はホットメソッドが 10 個程度しか存在しないために，ほとんどオーバヘッドが現れていない．一方，javac は約 60 個のホットメソッドに対してプロファイルが実施されるために，速度遅延が大きく現れている．平均で見ると，プロファイルが頻繁に起こる状況下で 2-3¥%のオーバヘッドと非常に少ないオーバヘッドであることを示している．

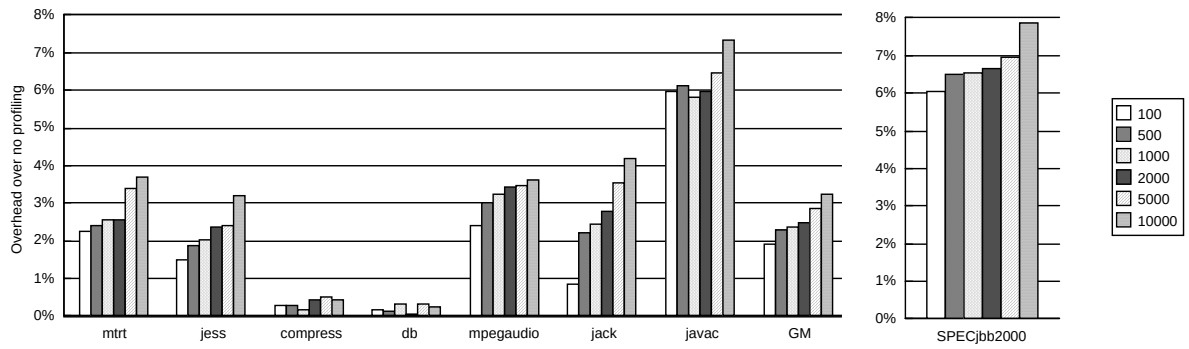


図３.１９　SPECjvm98 ベンチマークと SPECjbb2000 ベンチマークによる，初回実行における SPP のオーバヘッドの評価

３．５．５　パスプロファイルを用いた Loop peeling 手法による評価

収集したパスプロファイルを用いてプログラム最適化した場合の効果を評価するために，ここでは３．４で述べたパスプロファイルを用いた Loop peeling 手法を IBM Java Just-In-Time コンパイラ上に実装してプログラムの速度向上率の評価を行った．

本評価では，最内ループに対して，全て Loop Peeling を適用した場合（all loop peeling）と，ＳＰＰにより取得したパスプロファイルを用いて Loop Peeling が有効と判断した場合に限定して loop peeling を適用した場合(selective loop peeling)の速度向上率の比較により評価を行った．図３．２０に示す評価結果より，パスプロファイルを用いた loop peeling では，常に速度向上を得ているのに対して，無条件に loop peeling を適用した場合は，db において多少の速度向上を得ているものの，compress と javac で大きく性能低下を引き起こしている．このことから，パスプロファイルを用いた loop peeling の有効なループを適切に検出できていることがわかる．

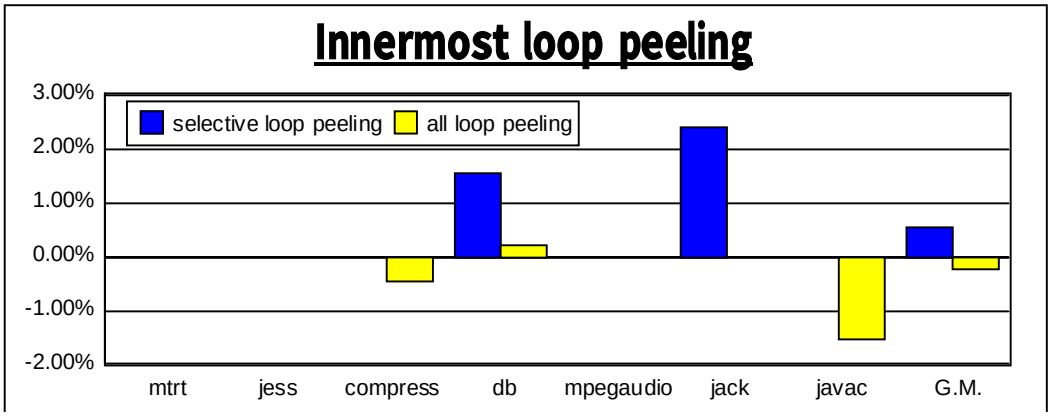


図３.２０　Loop peeling をもちいた速度向上の評価

３．６ 議論

今回実験を行った実装では,SPP によるプロファイルの期間が BPP の実装期間よりも長くなる場合がある．これは，SPP では，アウトライン構造グラフから内側構造グラフへとプロファイルを多段階にいっており，メソッド中のネストの深さに依存してプロファイル期間が変化するためである．収集されるプロファイルの量は同じであることから，プログラムの実行に与える影響は SPP と BPP では変わらない．実際には，SPP では各構造グラフのプロファイル収集は独立の仕組みとなるので，全ての構造グラフに対するプロファイルを同時に開始することで期間を短縮することは可能である．なお内包関係にある構造グラフでは，プロファイル収集に必要な状態を保持する変数を共用できなくなるため，その分の実行時の作業領域が必要となる．

SPP では，３．５．３ の評価でも述べたとおり，構造グラフあたりに割り当てられるプロファイルの収集数が著しく少ない場合，プロファイルの精度が低くなってしまう問題が存在する．これは，例えば，ほとんど実行されないループを含むようなメソッドやメソッド内に多数のループが存在する場合に特に問題となる．これは現在の実装が，メソッドに割り当てられているプロファイル収集数を各構造グラフに対して均等に分配しているために生じている．この問題に対しては，例えば構造グラフの形状から収集するプロファイルの数を変えるような手法が考えられる．例えば，ループ内に分岐がない場合には収集数を減らし，ループ内に分岐が多く含まれる場合には収集数を増やすということが考えられる．

さらに，一歩進んだ方法としては，SPP が外側構造グラフから順にプロファイルを実施することを利用して，ある構造グラフのプロファイルの収集が終了した時点でプロファイルを解析し，その中に含まれるループノードの実行頻度を求める．そして，ループノードの実行頻度に応じてそれ以下の部分に対する収集数を動的に割り振っていく．極端な場合としては，全く実行されなかったループノードに対してはプロファイルを実施しない方法も考えられる．この場合，実行はほとんどされないが，一度実行されると非常に実行頻度の高いホットなループになるようなものを検出することが困難になるが，隣接する実行頻度の高い領域を低いオーバヘッドで収集することが可能となる．

３．７ 関連研究

これまでに，静的コンパイラのためのパスプロファイル収集手法として，２つの手法が提案されている．１つは Ball と Larus による手法[3]であり，もう１つは Young と Smith による手法[37]である．Ball-Larus の手法は，ループを含まない経路に対するパスプロファイル処理を高速に実行する手法である．各分岐エッジに割り当てた経路番号を計算するための定数値を，グラフ中の木に対する弦となるエッジに集約させることでインストルメンテーションポイント数を最小化しているとともに，経路途中の処理は経路番号を計算するために変数に定数値を加算する処理なので，処理コストが平均で約 31%と非常に少ない．

Young-Smith の手法は，プロファイルする経路として，あるノードに到達するまでの過去K個の分岐ノードの履歴の列（K-bounded general path）を用いている．実行に伴い path CFG と呼ばれるグラフを構築しながらプロファイルを実施する．path CFG は，各ノードが元の CFG における経路を表し，各エッジがその始端ノードが指す経路の最後のノードの分岐エッジを表す．彼らの手法は Ball-Larus の手法に比べてそのオーバヘッドが非常に大きいかわりに，個々のノードに対してより詳細な情報が得られる．彼らはさらに K-bounded general path を用いていくつかの最適化手法[36][37]も提案し，その有効性を示している．

インストルメンテーションによるプロファイルのオーバヘッドを軽減する試みも提案されている．Arnold と Ryder は Instrumentation sampling 手法を提案している[25]．この手法では，もとのコードを複製した後，複製コードにのみインストルメンテーションコードを挿入する．もとのコードには，その開始点と各ループの後方分岐エッジにカウンタを用いたチェックコードを挿入し，定期的に実行を遷移させてインストルメンテーションコードを実行することで，インストルメンテーションコードの実行頻度を下げている．彼らの手法では，SPP に比べて利点と欠点を併せ持っている．我々は彼らのフレームワークを実装していないので正確な比較はできないが，BPP の経路数をカウントするインストルメンテーションコードに対してカウンタを用いて指定した間隔でプロファイルを収集するように実装することで彼らの手法の精度について評価を行った．結果として，彼らの手法はＳＰＰとほぼ同程度の精度のパスプロファイルを収集することができた．またプロファイル収集のオーバヘッドも，基本的に収集した経路数に比例すると考えればＳＰＰと同程度であるといえる．両者の相違は，ＳＰＰが連続してプロファイルを収集するのに対して，彼らの手法はサンプリングを用いて一定間隔に間引いてプロファイルを収集する．つまり，彼らの手法は，プロファイルの収集を分散させることで，局所的な挙動の偏りに陥ることなくプロファイルを収集することが可能となる利点がある．反面，彼らの手法は２つの欠点も持つ．１つは，彼らの手法はコードを複製するために余分のメモリ領域を必要とする点である．もう１つは，サンプリング間隔をある程度大きくしないとＢＰＰと同様の問題を生じてしまうことから，プロファイルの収集期間がＳＰＰよりも長くなってしまう点である．これはプロファイルを用いた最適化コンパイルを実施するタイミングの遅延につながり，結果的にプログラムの性能を低下させる要因となる．

プログラム中の実行頻度の高い経路を検出する手法もいくつか提案されている．Chillimbi と Hirzel は実行頻度の高いメモリアクセスの順序列(Hot Data Stream)を検出し，動的にプリフェッチ命令を挿入する手法を提案している．彼らは Instrumentation sampling 手法を改良した Bursty Tracing 手法[26]を用いて，メモリアクセスの履歴を収集し，これを Sequitur アルゴリズム[31]を用いて圧縮しすることで頻度の高い部分列を抽出する．彼らの手法は確かにプログラムのホットパスを検出できるが，それらを用いたプログラム変換や最適化の実施はさらなる処理が必要となる．

Dynamo[21]は，バイナリコードを入力としてプログラムの実行時に最適化を実施する動的最適化システムである．このシステムでは，最適化の対象となるホットトレースを検出するために，NET[4]と呼ばれる手法を用いる．NET では，あらかじめプログラム中に存在する後方分岐の到達点

(start-of-trace)にカウンタを挿入しておき，カウンタ値が閾値を越えた時点における，その位置からの実行経路(next execution trace)をホットパスとして検出する．これらの処理は実行頻度の高い領域を検出する方法であるが，パスプロファイル収集手法のように具体的な実行頻度が得られるわけではない．

３．８ まとめ

本論文では，動的コンパイラのための実用的なパスプロファイル収集手法として，構造的パスプロファイル収集手法を提案した．本手法では，プログラムをループネスト毎に構造グラフに分割したのち，各構造グラフに対して動的インストルメンテーション手法を用いてパスプロファイル処理を実施することで，プロファイルの収集コストをループの実行回数とは独立に制御することを実現とした．これにより，十分な精度のプロファイルを少ないオーバヘッドで収集することを可能とした．

本手法を IBM Java Just-In-Time コンパイラに実装して評価したところ，各メソッドに対して実行経路数を 1000 収集した場合に，そのプロファイル精度が約 90%となるとともに，自然な経路を用いたパスプロファイル収集手法に比べて少ないプロファイル量で高い精度を上げられることを示した．また実際のオーバヘッドは,プロファイルが頻繁に行われる状況でもオーバヘッドが約 2-3¥%と非常に少ないことを示した．

第４章　静的プロファイルを用いたプログラム起動処理の高速化

４．１　はじめに

近年の Web アプリケーションやアプリケーションサーバなどに代表される大規模アプリケーションは，Java 言語の普及等もあいまって広く構築されるようになってきた．しかしアプリケーションの規模が増大し複雑化するに従って，その起動時間の増加が問題となってきている．起動時間の増加は，システムの運用時間の低下だけでなくプロビジョニングなどのオン・デマンドな運用に対しても障害となる．さらに，アプリケーションの開発時にはより高頻度に起動と終了が繰り返されるために，開発効率の低下を引き起こしている．特にアプリケーションサーバは，目的とするアプリケーションを実行するためのフレームワークであり，その起動時間が多くのアプリケーションに対して影響を与えてしまう点で問題が大きい．従ってこのようなアプリケーションの起動時間を高速化することは非常に重要な課題である．

アプリケーションの起動時間を増加させる要因は様々あるが，その１つの要因としてファイル読み込みに伴う I/O 待ち時間が上げられる．特にアプリケーションの起動時は実行ファイルや設定ファイルなど多くのファイル読み込みが発生する一方で，処理が逐次的に進行するために，I/O 処理が他の計算処理とオーバラップされない場合が多く，I/O 処理待ち時間がそのまま起動時間に現れてしまう場合が多い．

この I/O 待ち時間は，HDD 上のファイルからデータを読み込む速度と CPU の処理速度の差により生じる．HDD からのデータ読み込み処理は，大きく分けると HDD のヘッドを目的のデータがある領域に移動させる処理(シーク処理)と，HDD から目的のデータを読み取るための処理(データ転送処理)からなる．シーク処理時間はヘッドが移動する回数や距離に比例して増加する．他方，データ転送処理時間は HDD から読み出されるデータの量に比例して増加する．従って，同じ量のデータを読み出す限り必ず同じ時間を必要とする．

I/O 待ち時間を減らすために解決すべき課題は大きく２つある．第１点としては，アプリケーションの起動時は多数のデータが読み込まれるので，読み込み処理を起動処理とオーバラップさせながら I/O 資源の利用効率を上げることである．第２点としては，特に Java 言語で構築されたシステムでは，JAR ファイル中のクラスファイル読み込みに伴い多数のランダムアクセス読み込みが発生する．ランダムアクセス読み込みは，読み込みの度にシーク処理を実施するため，このオーバヘッドが I/O 待ち時間の増加の要因の１つとなっている．このため，何らかの方法でシーク処理時間を削減する方法が必要である．

I/O 待ち時間を削減する従来手法は大きく３つに分類できる．１つめは非同期 I/O 処理を用いてアプリケーション自体を書き直す手法，２つめは何らかの方法で読み込み処理に先行してプリフェッチ処

理を発行してデータをファイルキャッシュに読み込ませる方法，３つめは HDD 上の読み出し時にヘッドの移動する距離が最小になるようにしてシーク処理時間を削減する方法である．このうち非同期 I/O を用いる方法は，プログラマがプログラムの知識を用いて記述するために原理的には最も効果の高い方法であるが，現実的には大規模なシステムでのプログラムの書き換えは困難である．

プリフェッチ処理を用いる従来手法は，さらに，コンパイラなどで解析的にプログラムにプリフェッチ処理を挿入する方法 [52][53]，元のプログラムからプリフェッチ処理用のプログラムを生成して投機的に実行させる方法[42][43]，動的プロファイル情報を用いて実行時にファイルのアクセスパターンを検出してプリフェッチ処理を実行する方法[47][48]，アプリケーションにディレクティブなどを挿入してプリフェッチ処理を行う手法[49][50][51]，静的プロファイルを用いてプリフェッチ処理を行う方法[44][45][46]に分類できる．このうち静的プロファイルを用いる方法以外の方法は，汎用的な手法である反面，プリフェッチ処理と実際の read 処理の時間的間隔をあまり大きく取れないことが多く，連続して多くのファイル読み込みが発生するような場合には効果が少ない．他方，静的プロファイルを用いる手法は，ファイル読み込みに再現性のあるプログラムにしか効果がないが，アプリケーションが読み込むファイルの情報を使って十分に早い時期からプリフェッチ処理を実施できるため，再現性のあるプログラムに対しては大きな効果を得ることができる．しかし，従来手法では，プロファイルされた状態と実際の実行が一致しているかどうかの検出に重点が置かれており，プロファイルを用いていかに高速化するかという観点では議論されていなかった．

シーク処理時間を削減する方法としては，OS の I/O 要求キュー中にある要求の実行順序をヘッドの移動距離が最小になるように並べ替える手法[54][55]や，静的プロファイルを用いて HDD 上のファイルの配置を並べ替える手法[56][57][58]などが提案されている．前者の手法は汎用的ではあるが，同時に I/O 要求キューに存在する要求の間でのみシーク処理時間の最適化が行われるため，逐次的に読み込みを行うプログラムにおけるシーク処理時間を短縮することはできない．また後者の方法は OS などシステムレベルでの実装が必要となるため，既存のシステムで容易に利用できる方法ではない．

本論文では，静的プロファイルから作成したシナリオを用いてプリフェッチ処理を実行する手法を提案する．提案手法は，まず事前に収集した I/O 処理の静的プロファイル情報を解析してプリフェッチ処理を実施するためのシナリオを作成する．シナリオ作成では，シーク処理オーバヘッドの軽減を実現するために,HDD 上の連続領域に対するプリフェッチ処理を連続して行うようにプリフェッチ処理の順序の並べ替えを行う．実行時には，このシナリオを用いてアプリケーションとは別の専用スレッド上でシナリオに従ってプリフェッチ処理を実施する．従来の静的プロファイルを用いる手法ではプロファイルされた順にプリフェッチ処理が実施されていたのに対して，本手法ではこの並べ替えにより起動時に発生する I/O 待ち時間自体を短縮することができる．また本手法は，OS のファイルキャッシュを用いてプリフェッチされたデータを保持させるので，実装に特殊な仕組みを必要とせず，既存のほとんどのシステム上で実装可能である利点もある．

RedHat9 Linux 上で IBM WebSphere Application Server 5.1.1 の起動時間による提案手法の評価を

行ったところ,本手法を用いてプリフェッチ処理を実施することで起動時間を 20-33¥%短縮できることがわかった．またプリフェッチ処理を実施する順序を並べ替えることで，約 10¥%の速度向上が得られることがわかった．

以下では，まず４．２において，アプリケーション起動時の I/O 待ち時間の問題について述べる．続いて４．３において我々が提案する Scenario-based prefetching 手法について説明し，４．４において提案手法を用いた評価結果を示す．最後に４．５で関連研究について説明する．

４．２　アプリケーション起動時の I/O 待ち時間の問題

本節では，まず I/O 待ち時間について説明した後，アプリケーション起動時におけるファイル読み込みに伴う I/O 待ち時間の影響について示す．その後，従来手法を示しながら，アプリケーション起動時間の短縮のために解決すべき課題について述べる．

４．２．１　I/O 待ち時間の定義

現在の計算機では,CPU が HDD からデータを読み出す速度はメモリなどからデータを読み出す速度に比べて遙かに遅いので，この速度差から I/O 待ち時間が生じる．本論文では，I/O 待ち時間を CPU が HDD 上のデータを読みに行くときに発生する際に生じる CPU のアイドル時間と定義する．HDD からデータを読み出す時間は，シーク処理時間とデータ転送処理時間に分けられる．シーク処理時間は HDD のヘッドを読み出すデータが置かれている位置に移動する処理にかかる時間であり，データ転送処理時間は HDD から目的のデータを実際に読み出す際にかかる時間である．シーク処理時間はヘッドが移動するたびに発生するオーバヘッドであるため，プログラムのデータのアクセスの仕方によって変化する．例えば逐次アクセスで HDD 上の連続領域を読み出す場合はシーク処理コストが最小となり，ランダムアクセスで毎回異なる位置からデータを読むと増大する．一方，データ転送処理時間は読み出すデータ量にのみ比例してかかる時間でり，同じデータを読み出す場合は読み出す順番によらず同じ時間を必要とする．

４．２．２　アプリケーション起動時における I/O 待ち時間

実際の I/O 待ち時間がアプリケーションの起動にあたえる影響として，Redhat9 Linux において IBM WebSphere Application Server(WAS) 5.1.1 の起動処理時間を測定した結果を図４．１に示す．図４．１では，WAS の起動時間を，計算機の起動直後に測定した場合(cold startup)と，WAS を１回起動し終了することで起動時に読み込まれる全てのファイルがファイルキャッシュ上に保持されている状態で測定した場合(warm startup)の結果を示している．この２つの実行の差は，HDD からのデータ読み込み処理の有無であることから，WAS の起動においては，HDD からのファイル読み込みによる I/O 待ち時間に起因するオーバヘッドが起動時間の 44%を占めていることがわかる．実際，WAS 5.1.1 では，起動中にファイルから合計で 21.54MB のデータを読み込み，その読み込み処理には合計で 7.6 秒の時間を要している．

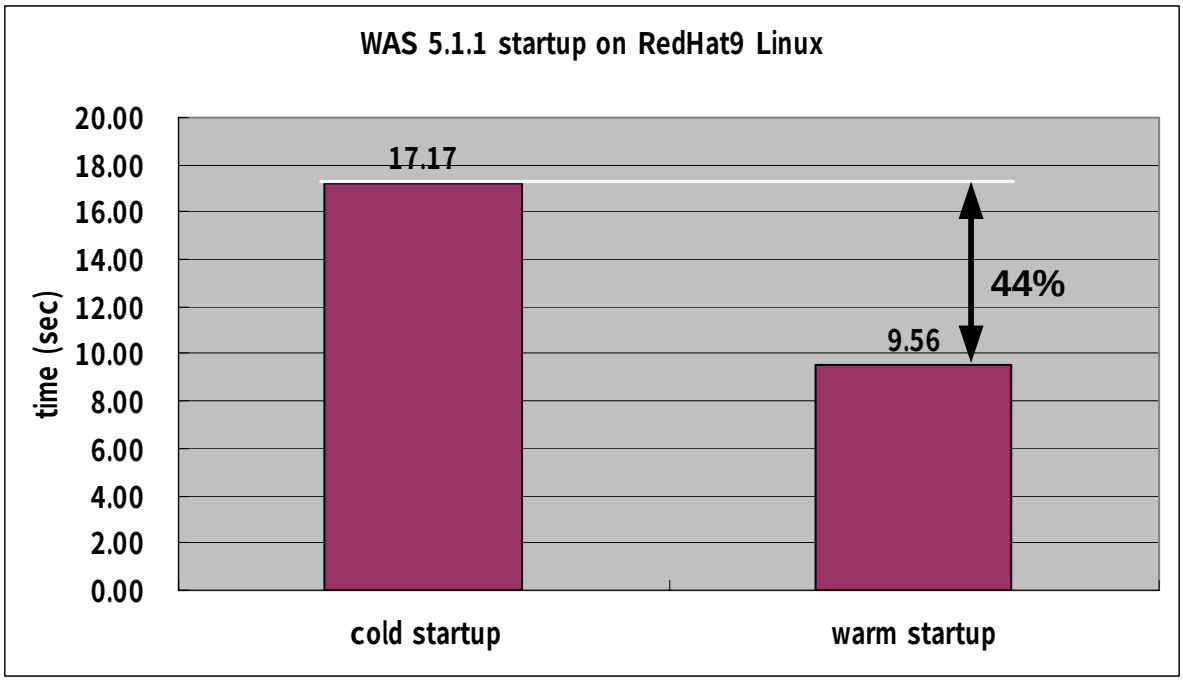


図４.１　RedHat9 Linux での WAS 5.1.1 の起動時間の評価結果

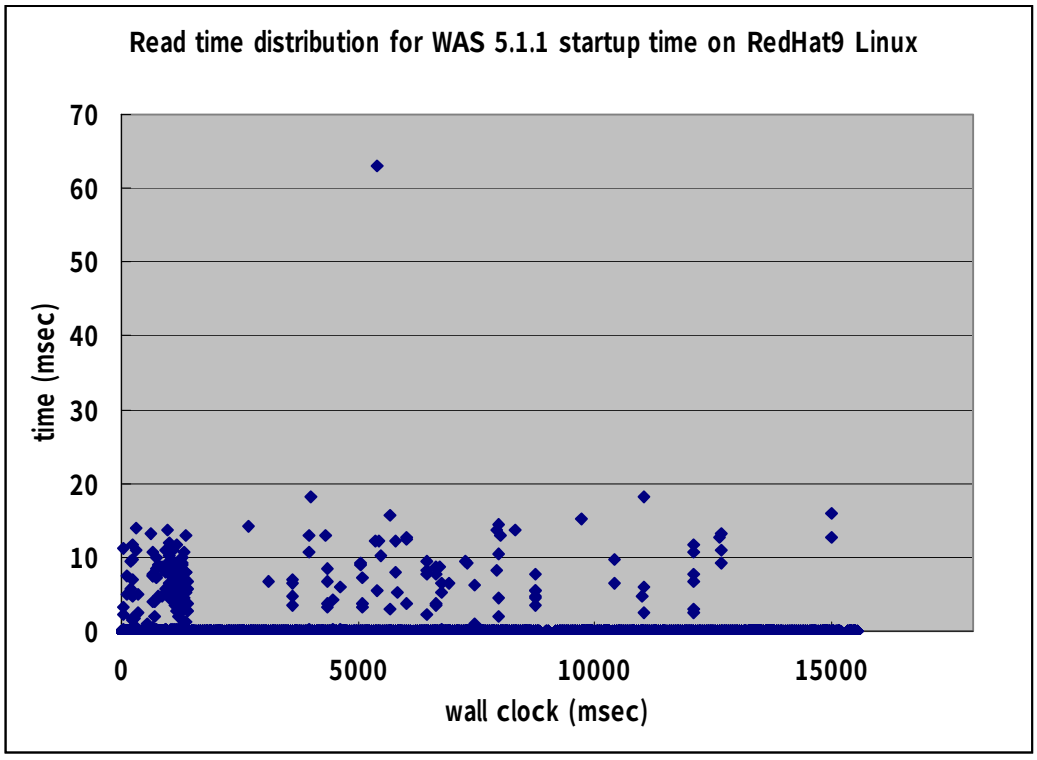


図４.２　WAS 5.1.1 起動時のファイルからのデータ読み込みにかかる時間の時間的推移

また，図４．２は，同じく WAS 5.1.1 に対して，その起動開始直後から起動完了までの間に発生するファイルからの読み込み処理に要する時間をあらわしたグラフである．データを読み込む時点でそのデータが既にファイルキャッシュに保持されている場合はほとんど I/O 待ち時間を生じずに処理することができる．このグラフにおける多数のほぼ０時間での読み込みは，このようにファイルキャッシ

ュからデータを読み出したことを意味している．他方，読み込むデータがファイルキャッシュ上に存在しない場合，HDD からデータを読み出すことになるために多くの時間を必要とする．HDD からの直接読み込みは，読み込み処理全体に対する頻度は決して多くなくても，その読み込み時間がプログラムの実行に与える影響は無視できない．これはグラフより HDD からの読み込みには平均 10ms，最大で 65ms を要していることからもあきらかである．

起動直後の数秒間に多くの I/O 待ちを起こす読み込みが発生し，その後も頻度は多少下がるが I/O 待ちを起こすファイル読み込みが発生していることがわかる．つまりアプリケーションの起動時には多くのファイル読み込みが発生し，その I/O 待ち時間が起動時間に大きく影響を与えているといえる．また，システム自体を再起動しない場合でも，アプリケーション自体の実行の結果としてファイルキャッシュから起動時に読み込まれるファイルが追い出されてしまう場合は，その起動処理に対して常に cold startup の場合と同様の I/O 待ち時間を被ってしまう．

４．２．３　解決すべき課題

大規模アプリケーションの起動時はファイルの読み込み量が多い．前述の結果では，WAS の起動時間の約 44%が I/O 待ち時間であり，この値はアプリケーションによっては 70%近くに達する場合もある．このことから，I/O 待ち時間を最小にするためには，ファイルからのデータ読み込みと計算処理とをオーバラップさせることで，起動処理中の I/O 資源の使用率を上げることが重要な課題となる．

表４．１に従来手法の分類と比較を示すように，従来手法においては静的プロファイル手法を除くとプリフェッチを起動直後から連続的に実施することは原理上不可能であった．静的プロファイルを用いる手法は，汎用性は低いがアプリケーションの起動処理などの再現性の高いアプリケーションに対しては有効な方法である．しかし，従来の静的プロファイルを用いる手法では，その論点は主にプリフェッチ処理に使用するプロファイルが実際のアプリケーションの実行と一致しているかどうかの検出に重点が置かれており，収集されたプロファイルを用いていかにアプリケーションを高速化するかという観点では論じられていなかった．例えば，Lei の手法[44]や Griffoen らの手法[45]では，プリフェッチ処理はファイル単位であるため，ファイルの一部しかアクセスしないような場合を考慮していない．また Grimsrud らの方法[46]は，HDD 上の各セクター毎に次にアクセスされやすいセクターの情報を保持しておくことで効率的なプリフェッチ処理を実現しようとしている．しかし HDD の読み出し順序はアプリケーションのアクセス順序と同じであるため，ランダムアクセス読み込みにより発生するシーク処理のオーバヘッドの影響をそのまま被ってしまう．特に Java 言語で構築されているシステムでは，JAR ファイルからのクラスの読み出しが，そのファイルフォーマットの構成上ランダムアクセス読み込みとなり，多数のクラスファイルの読み込みによって生じるシーク処理のオーバヘッドの増大が実行速度を低下させる原因の一つとなっている．例えば図４．３では，１つの Jar ファイルから 2 つのクラスを読み出す例を示している．まず，(1)Jar ファイルの Central directory 領域から最初のクラスの位置情報などを取得する．(2)この情報を使って最初のクラスのデータを Jar ファイルから読み込む．続いて，(3)2 つ目のクラスの位置情報を Central directory から読み出した後，(4) ２つめ

のクラスのデータを読み込む．このように，Jar ファイルからのクラスの読み込みでは，Jar ファイルの尾部にある Central directory と実際にクラスのデータが位置する部分とのアクセスが交互に行われる．このシーク処理のオーバヘッドを削減することもアプリケーションの起動時間短縮のためには重要な課題となる．

表４.１　従来手法の分類と比較

	Application controlled approach	Compiler directed approach	Common access pattern heuristics	Speculative execution approach	Seek optimization	Profile based approach
Basis	**Static analysis**	**Static analysis**	**Dynamic profile**	**Speculative execution**	**dynamic profile**	**Static profile**
Advantage	**アプリケーション情報を用いることで効果的なプリフェッチが可能**	**解析的にプリフェッチを生成することが可能**	**実装が容易**	**ほとんどのアプリケーションに対して汎用的に適用可能**	**システム全体としてシーク時間の削減を実現可能**	**再現性のあるプログラムに対して効果的なプリフェッチが可能**
Disadvantage	**アプリケーションの書き換えが必要**	**解析能力の問題で適用範囲がループなどに限定されてしまっている**	**不規則なアクセスに対してはほとんど対応できていない**	**プリフェッチ処理用スレッドの適切な制御が困難**	**逐次的なアクセスに対して効果がない**	**再現性のないプログラムでは使えない**
Fully overlap the read	△	△	△	△	×	◎
Reduce the seek time overhead	×	×	×	×	○	×
Control the active cache size	○	×	×	×	×	×

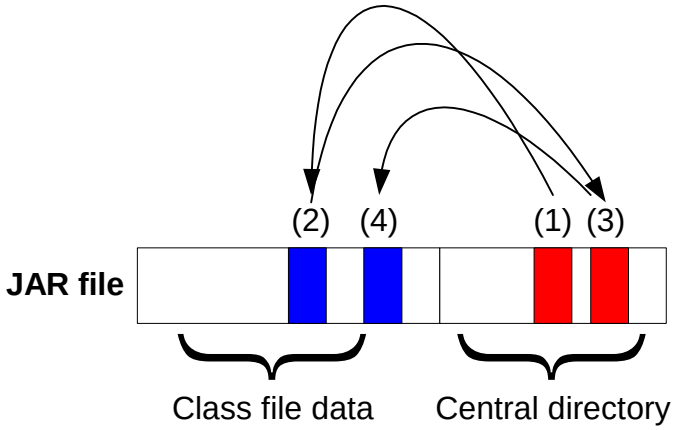


図４.３　Java プログラムで使用される Jar ファイルの典型的なファイルアクセスパターン

４．３　シナリオに基づくファイル・プリフェッチ手法

前節で述べた課題を解決する方法として，我々は Scenario-based prefetching 手法を提案する．本手法では，静的プロファイルを解析して生成したシナリオを用いてプリフェッチ処理を実施する．提案手法では，静的プロファイルを用いてアプリケーションのファイル読み込みの実行を事前に解析し，別プロセスを用いてアプリケーションの実行とファイルからのデータ読み込みを最大限にオーバラッ

プさせるようにプリフェッチすることで，アプリケーションの I/O 待ち時間を減少させる．

図４．４に，提案手法の流れを示す．本手法の処理では，まずアプリケーションの事前実行により I/O プロファイルの収集を行う．次に，収集したプロファイルを解析してアプリケーションの実行にあわせたプリフェッチを実行するためのシナリオを生成する．最後に，アプリケーションの実行とともに，シナリオに従ったプリフェッチを実施するプロセスを実行することで，アプリケーションが読み込むデータのプリフェッチを実現する．

以下では，まず説明に使用するサンプルプログラムについて説明した後，本手法の３つのフェーズについて順に説明する．

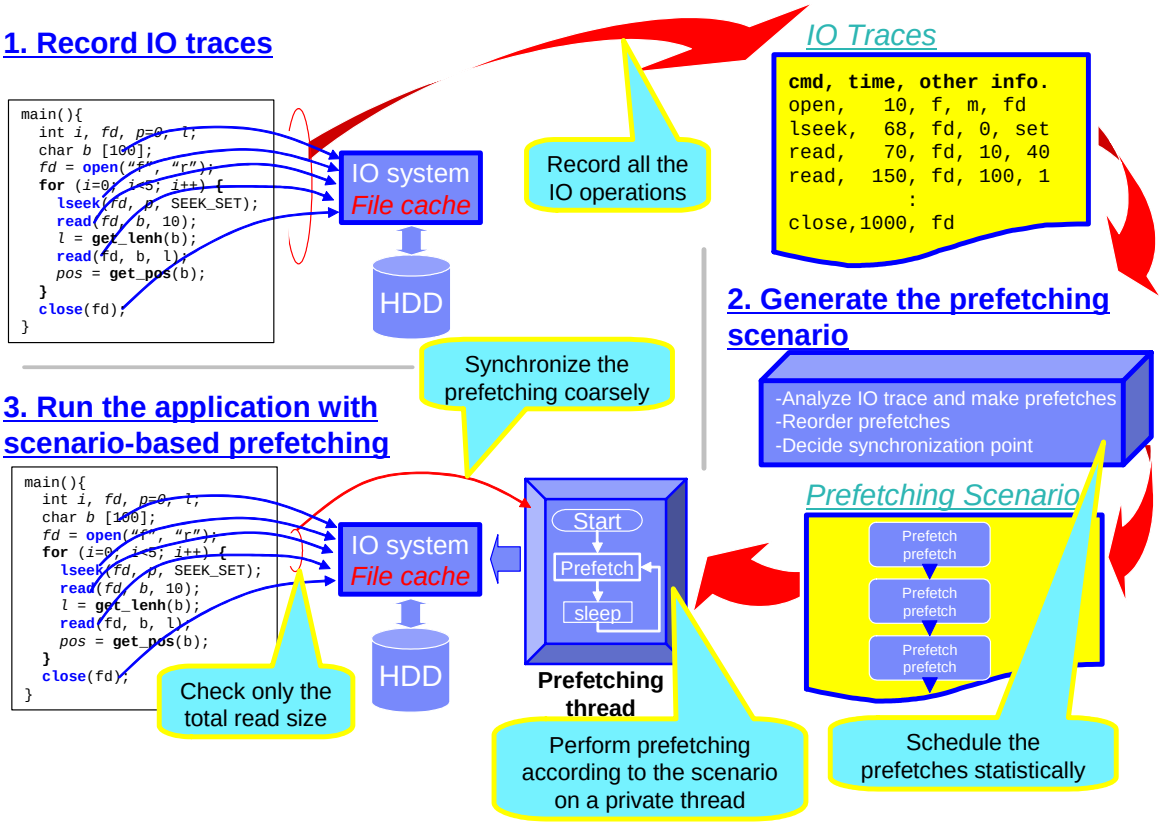


図４.４　シナリオに基づくプリフェッチ手法の流れ

４．３．１　サンプルプログラム

図４．５に，本節で説明に使用するサンプルプログラムを示す．図の左側に示されるプログラムは，プロファイルを収集する対象のプログラムである．このプログラムは，１つのファイル *fileA* をオープンした後，lseek 命令を１回実施して読み取り位置を設定した後　２回の read でデータを読み取るという一連の処理を５回繰り返し，合計で１０回の read 命令が実行されるた後，ファイルをクローズする．右側の図はこのプログラムを実行した場合の read 命令の実行時間と処理時間を示したタイム・チ

ャートと，各 Read 命令で読み込まれるファイル上の領域である．この例で用いるシステムでは，4KB 単位でファイルをキャッシュする物とする．図４．５の右図に示す *fileA* の青線で示す長方形領域がファイルキャッシュで扱うブロックとなる．このプログラムでは，lseek 命令により読み取り位置を設定した後の最初の read 命令で読み取るデータはファイルキャッシュ上に存在しないために，タイム・チャート上の長方形で示されるように read 命令の実行後に I/O 待ち時間が発生している．一方２番目の read 命令で読み取るデータは１番目の読み取りによりファイルキャッシュ上に保持されているために，その読み取り処理はほぼ０時間で終了している．

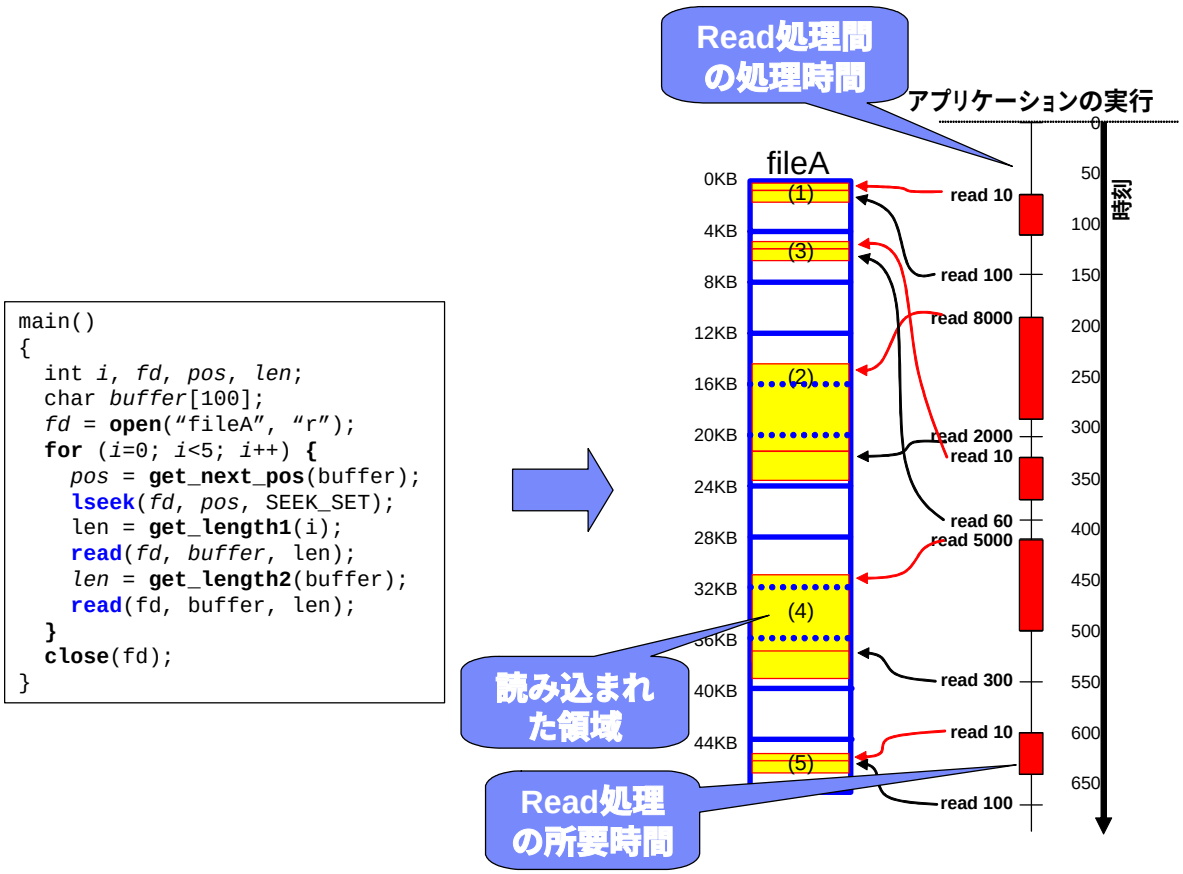


図４.５　サンプルプログラム

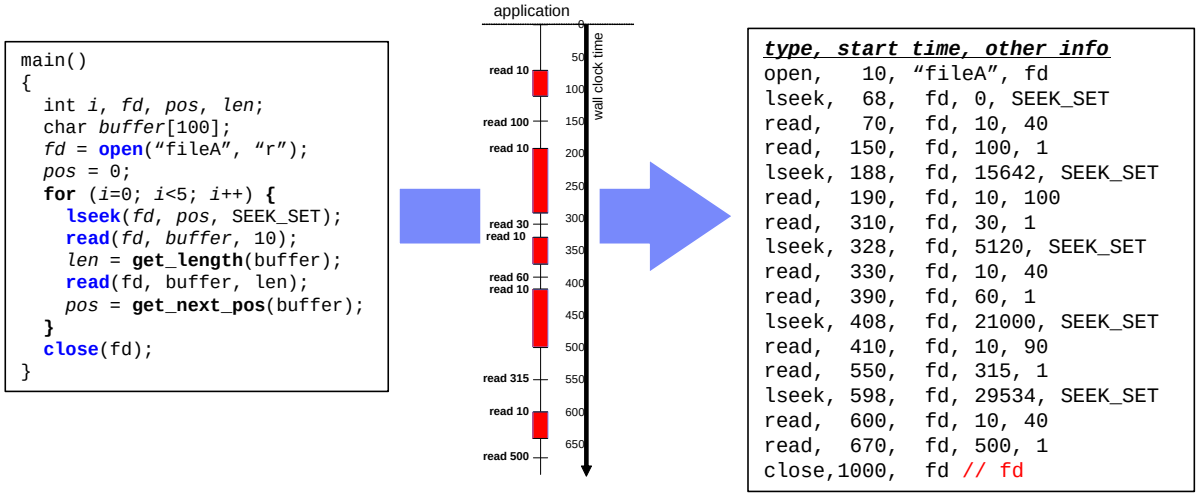
４．３．２　プロファイルの収集

本手法では，まず対象アプリケーションの I/O 処理に関するプロファイルを収集する．収集されるプロファイルの情報は，対象プログラムにより実行される I/O 処理(open，close，read，write，lseek など)の引数と返り値，thread ID，時刻，実行に要する時間等を時系列で記録したトレース情報である．今回の実験では，Linux で用いられている weak alias による関数解決の仕組みを利用して，*LD_PRELOAD* 環境変数でフック関数のライブラリを指定し，I/O 処理用の各システムコールをフックすることでプロファイルを収集した．例えば，read 関数の場合は，図４．６に示すように，システム関数である read と同じ名前の関数を定義した関数を用意する．フック関数中では，必要なプロファイルを

収集するとともに，read 関数の実体の名前である__libc_read 関数を呼ぶ．フック関数中で使用されている関数 SBP_GetPerformanceCounter()は，時刻に相当する値を返す関数，関数 SBP_dumpProfile()は，渡された引数をプロファイルとしてログファイルに書き出す関数である．このようにして図４．５のプログラムに対して収集した I/O プロファイルの例を図４．７に示す． 図４．７(c)では，まず時刻 10 においてファイル *fileA* に対して open 命令が実行されてファイル識別子 *fd* が返されたことが記録される．次に，時刻 68 において lseek 命令が実行され，ファイル識別子 *fd* の読み出し位置を０にセットされたことが記録される．その後，時刻 70 において read 命令が実行され，ファイル識別子 *fd* から 10 byte のデータが読み出され，その処理に時間 40 を要していることが記録される．このように，ファイルへの全てのアクセスが順次記録されていく．

```
01:     ssize_t read(int fd, void *buff, size_t n) {
02:       ssize_t result;
03:       long long t_start, t_end;
04:       int tid;
05:       totalReadSize += n;
06:       t_start = SBP_GetPerformanceCounter();
07:       result = __libc_read (fd, buff, n);
08:       t_end = SBP_GetPerformanceCounter();
09:       SBP_dumpProfile (fd, buff, n, result, t_start, (t_end - t_start));
10:       return result;
11:     }
```

図４.６　read 処理に対してプロファイル情報を収集するための処理



(a) An example　　(b) execution sequence of IO operations　　(c) IO trace

図４.７　I/O プロファイルの例

４．３．３　シナリオの生成

次に，収集したプロファイルを解析し，プリフェッチ処理用のシナリオを生成する．この処理は，大きく３つのステップからなる．まず収集したI/Oトレースからプリフェッチ処理リストを生成する．次に，プリフェッチ処理の並べ替えを実施する．その後アプリケーションとの同期点を決定し，最終的なシナリオを生成する．以下に，この３つのステップを順に示す．

(1) プリフェッチ処理リストの作成

まず，I/Oプロファイルを順に走査し，ファイルデスクリプタの値を用いてファイル毎に時系列順のI/O処理リストを生成する．同じファイルが複数回オープンされている場合は，１つのI/O処理リストにまとめる．

続いて，ファイル毎にread処理のうちで実際にI/O要求を発行するread処理を検出する．既存のOSでは，read処理で読まれるデータがファイルキャッシュ上に存在しない場合，OSにより規定されたI/O要求のサイズに正規化されたI/O要求を生成してHDDからデータの読み込みが実行される．例えば実験に使用したLinuxでは4KB単位でHDDからデータが読み出される．一度HDDから読み込まれたデータはファイルキャッシュ上に保持されるので，同じキャッシュブロック上に存在するデータを読みだすread処理に対してI/O要求は生成されない．ファイル毎に時系列順のread処理リストを走査し，各キャッシュブロックのデータを最初に読み出すread処理を検出し，そのread処理に対応させてプリフェッチ処理のリストを生成する．図４．８に，図４．５で示されるサンプルプログラムに対するプリフェッチ処理リストの生成の例を示す．中央の図の赤線が付されたread処理が，各キャッシュブロックのデータを最初に読み出すread処理である．この例では全部で５つ存在するので，この５つを実際のアクセス順に収集することで，右図で示すプリフェッチ処理リストを得ることができる．

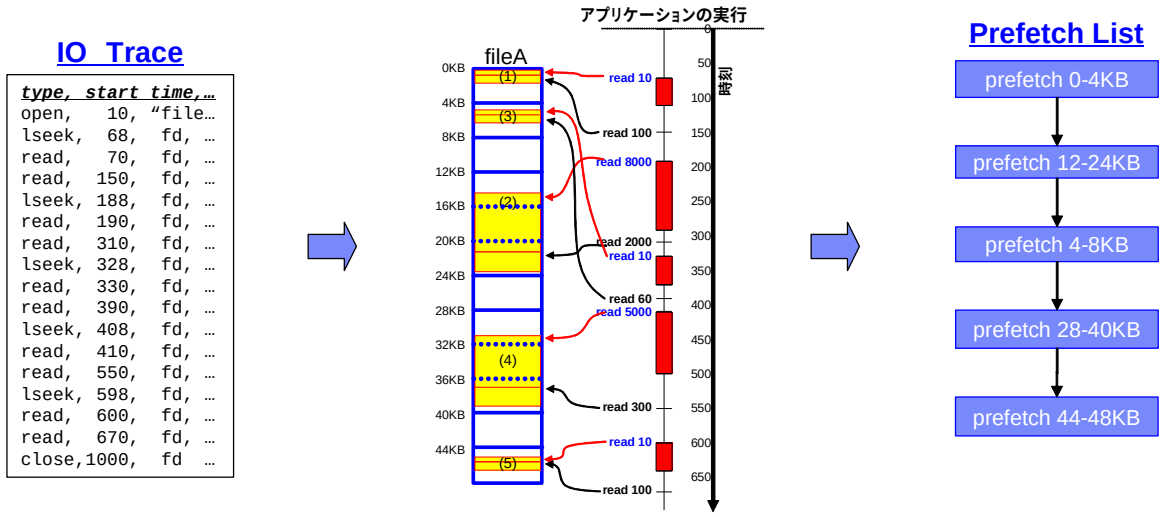


図４．８　プリフェッチ処理リストの生成

(2) プリフェッチ処理の並べ替えの実施

次に，隣接したプリフェッチ領域に対して非連続的にプリフェッチされている処理を連続化するようにプリフェッチ処理を並べ替える．この並べ替えの実施条件は，並べ替えによりアプリケーションの実行時間が長くならないことである．また本処理では，各ファイルがＨＤＤ上に連続に配置されていることを仮定している．並べ替え処理の流れを図４．９に示す．

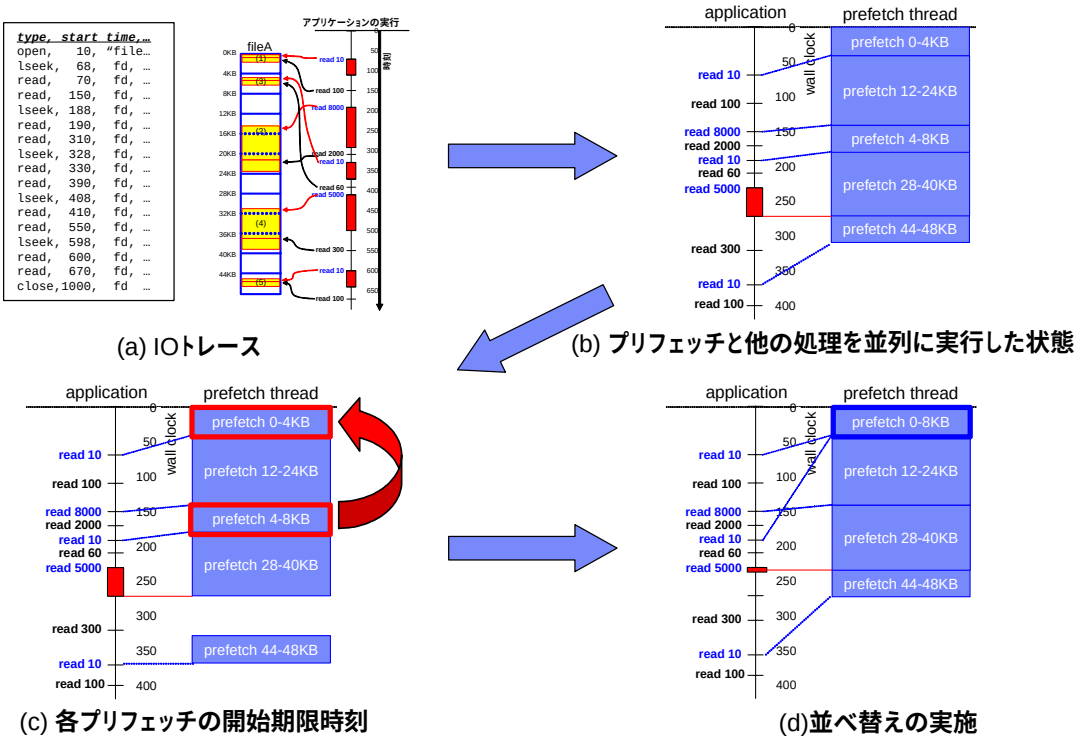


図４．９　プリフェッチ処理の並べ替えの実施例

1. 各 read 処理の読み込み完了時刻の計算

I/0プロファイルを解析し，ファイルからの読み込み処理とアプリケーションの実行が完全に並列に実施できたと仮定した場合における，各read命令の読み込み完了時刻 T_i を計算する．この完了時刻T_iは，以後の並べ替え処理によって各Read処理の完了時刻が，並べ替えを行わない場合に比べて遅くならないことを保障するために使用する．プログラムの実行が逐次的に行え割れる場合，プログラム中の計算処理とI/0待ち時間は模式的に図４．１０の左側の図のように表すことができる．プログラム開始後 T_{C1} 時間の計算処理を実施した後，I/0待ち時間 T_{r1} を伴うファイル読み込み処理が実施される．この読み込み処理の完了時刻を T_1 とする．このようにして，I/0待ち時間を生じる i 番目の読み込み処理 R_i に対しては，その読み込み処理に先行して，T_{Ci} の計算処理が行われ，続いてI/0待ち時間 T_{ri} のファイル読み込みが行われ，Rn の読み込み完了時刻は T_i となる．

このように逐次的に行われる処理を，図４．１０の右図のように，計算処理と読み込み処理が並列に

実施されたと仮定した場合の T_i の値を求める．読み込み処理間のオーバラップはないとすると，i 番目の計算処理 C_i が開始されるためには，$i-1$ 番目の読み込み処理 R_{i-1} が終了している必要がある．もし，R_1 から R_{i-1} までの処理時間の合計時間が C_1 から C_{i-1} までの処理時間の合計よりも長い場合，C_i の実行は，R_{i-1} の実行完了を待ってから開始することになる．このことから，i番目の読み込み処理 R_i の読み込み完了時刻 T_i は，次の式で与えられる．

$$T_0 = 0$$

$$T_i = \max(T_{i-1} + T_{Ci}, \sum_{k=1}^{i} T_{ri})$$

図４．１１は，この式に基づいてプリフェッチ処理 p に対する読み込み処理の完了時刻 $T_i(p)$ を計算するアルゴリズムである．

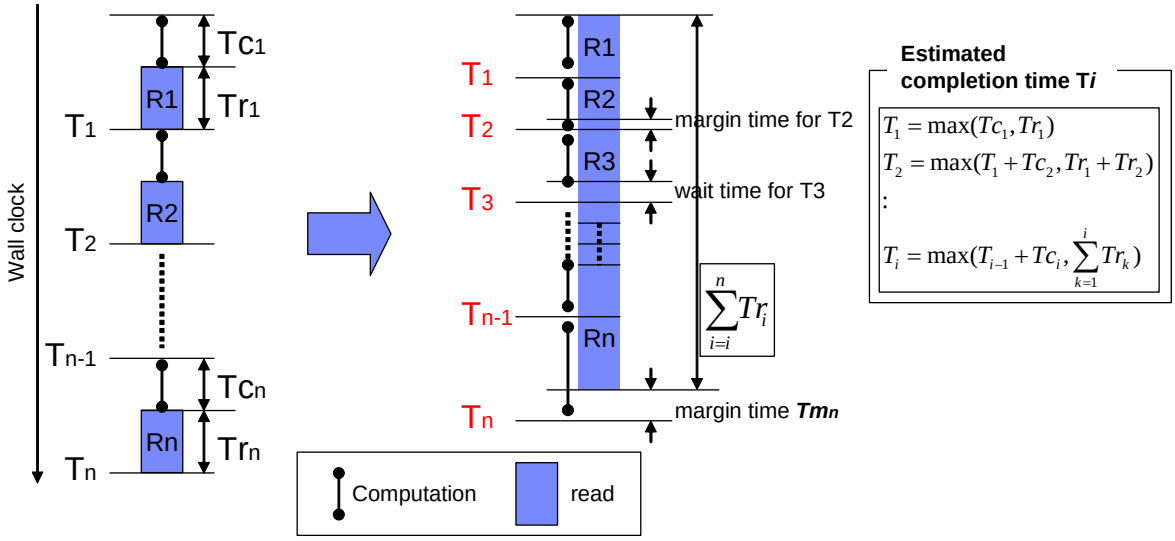


図４．１０　各 Read 処理の読み込み終了時刻 Ti の計算

```
01:     void calculateTi () {
02:       Tr_total = 0;
03:       T_total = 0;
04:       for (P = プリフェッチ処理リスト中の先頭要素から順に繰り返す) {
05:         Tc = (Pに対応するread処理のプロファイル上の開始時刻 )- Tclr;
06:         Tclr = (Pに対応するread処理のプロファイル上の完了時刻);
07:         Tr_total += Tr(P);
08:         if (T_total + Tc ＞Tr_total) {
09:           T_total += Tc;
10:         } else {
11:           T_total = Tr_total;
12:         }
13:         Ti(P)= T_total;
14:       }
15:     }
```

図４.１１　I/O 処理をオーバラップさせた場合の各 Read 処理の開始処理を求めるアルゴリズム

2. 各プリフェッチ処理の開始期限時刻の計算

次に，全てのread処理の完了時刻Tiを遅らせない範囲で，各プリフェッチ処理のとることができる最も遅い開始時刻（開始期限時刻）を計算する．図４．１２に示すように，読み込み処理で読み出すデータがファイルキャッシュ上にある場合は読み込み時間０で読み込みを完了できると仮定すると，i番目の読み込み処理 R_i に対する開始期限時刻を D_i とすると，その読み込み完了時刻 T_i を遅らせないためには，２つの条件を満たす必要がある．１つは（$T_i - T_{ri}$）の時刻までにプリフェッチ処理を開始することである．もう１つは$i+1$ 番目の読み込み処理 R_{i+1} の開始期限時刻 D_{i+1} を補償するために，（$D_{i+1} - T_{ri}$）の時刻までにプリフェッチを開始する必要がある．以上より，i番目の読み込み処理 R_i に対するプリフェッチ処理の開始期限時刻 Di は，次の式で与えられる．

$$D_n = T_n$$
$$D_i = \min(T_i, D_{i+1}) - T_{ri}$$

図４．１３は，この式を用いてプリフェッチ処理pに対する開始期限時刻 $D_i(p)$ を計算するアルゴリズムである．

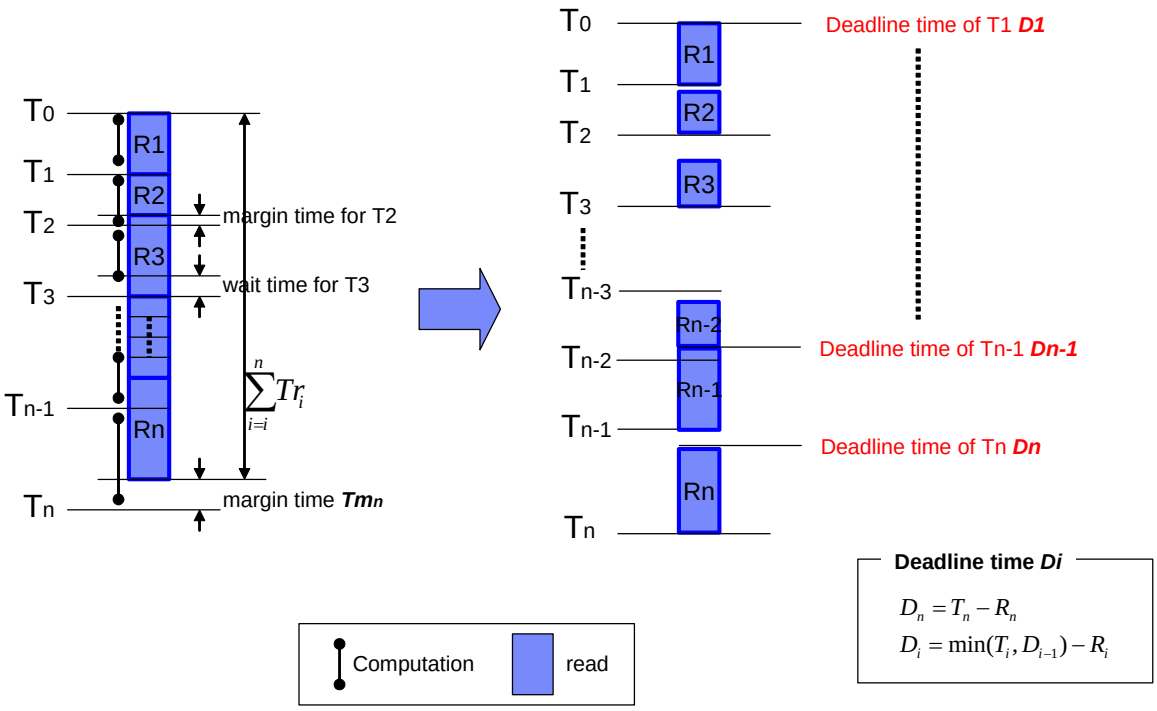


図４.１２　各プリフェッチ処理の開始期限時刻の計算

```
01:    void calculateDi () {
02:      P0 = (プリフェッチ処理リストの最後のプリフェッチ処理);
03:      Di(P0) = Ti(P0) - Tr(P0);
04:      T_deadline = Ti(P0);
05:      for (P = P0の直前の要素から順に，リストを逆順にたどりながら繰り返す) {
06:        if (T_deadline < Ti(P)) {
07:          Di(P) = T_deadline - Tr(P);
08:        } else {
09:          Di(P) = Ti(P)-Tr(P);
10:        }
11:        T_deadline = Di(P);
12:      }
13:    }
```

図４.１３　各プリフェッチ処理の最終開始時刻を求めるアルゴリズム

3. プリフェッチ処理の並べ替え

最後に，各プリフェッチ処理の最終開始時刻を遅らせないことを条件として，隣接する領域へのプリフェッチ処理を連続に行えるかを調べながらプリフェッチ処理の並べ替えを実施する．本手法では，リストスケジューリングをベースとしたヒューリスティック手法をもちいて，プリフェッチ処理リス

トを先頭から順に走査しながら並べ替えが可能かどうかを調べていくアルゴリズムを用いる．図４．１４に示すように，あるプリフェッチ処理に対して，それに隣接する領域をプリフェッチする処理を探索し，隣接領域へのプリフェッチ処理が存在する場合，それらを結合して１つのプリフェッチ処理とした場合に要する処理時間が増加してもそれ以降のプリフェッチ処理の開始期限時刻が維持できる場合，その隣接領域へのプリフェッチ処理を並べ替えて現在のプリフェッチ処理と融合して１つのプリフェッチ処理とする．

図４．１５に，この並べ替えのためのアルゴリズムを示す．アルゴリズム中の関数 isLowerSelected()，isUpperSelected()は，現在処理中のプリフェッチ処理であるＰがアクセスする領域に隣接する領域のプリフェッチ処理が存在する場合に，直前の領域と直後の領域のどちらを融合するほうが効果的かを計算する評価関数である．また，関数 concatinateLowerPrefetch()，concatinateUpperPrefetch()は，隣接領域のプリフェッチ処理を融合する関数である．この処理中にプリフェッチ処理される領域とプリフェッチ処理にかかる時間が更新される．

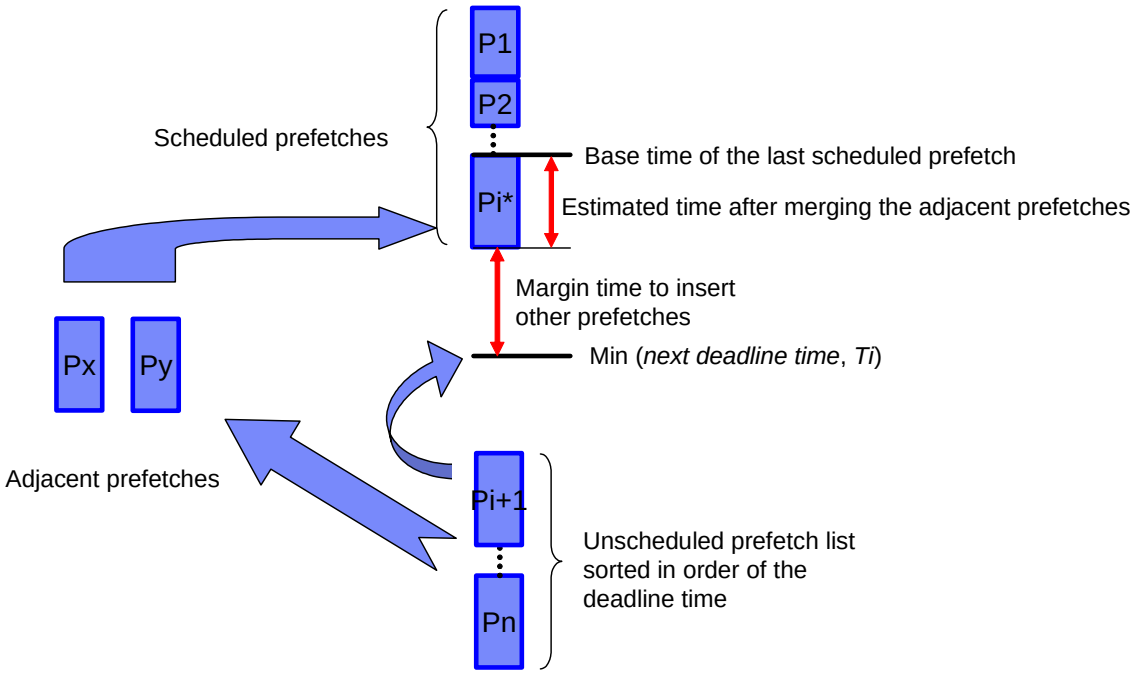


図４.１４　プリフェッチ処理の並び替え

```
01:     void reorderPrefetchList () {
02:       P1 = (プリフェッチ処理リストの先頭要素);
03:       (P1 の開始時刻) = 0;
04:       P2 = (プリフェッチ処理リストでP1 の次の要素);
05:       while (P2 != NULL) {
06:         F1 = Di(P2) - Ti(P1);
07:         F2 = Ti(P) - (P1 の開始時刻);
08:         free_time = min (F1, F2);
09:         if (free_time ＞ (P の処理にかかる時間)) {
10:           for (;;) {
11:             lower = (P の領域の直前の領域をプリフェッチする処理);
12:             upper = (P の領域の直後の領域をプリフェッチする処理);
13:             if (isLowerSelected (P, lower, upper)) {
14:               concatinateLowerPrefetch (P);
15:             } else if (isUpperSelected (P, lower, upper)) {
16:               concatinateUpperPrefetch (P);
17:             } else {
18:               break; // end of for
19:             }
20:           }
21:         }
22:         (P2 の開始時刻) = (P1 の開始時刻) + (P1 の処理にかかる時間);
23:         P1 = P2;
24:         P2 = (プリフェッチ処理リストでP1 の次の要素);
25:       }
26:     }
```

図４.１５　プリフェッチ処理の並び替えを行うアルゴリズム

(3) 同期点の決定

本手法では，最終的にシナリオを用いてアプリケーションと独立のスレッド上でプリフェッチ処理を実施する．プリフェッチ処理がアプリケーションの実行と独立に実行される場合，両者の間に何らかの同期処理が必要となる．例えば，プリフェッチ処理が先行しすぎてしまうと，必要以上に多くのファイルをプリフェッチしてしまい，まだ読み込みが終わっていないデータをファイルキャッシュから追い出してしまう状況が起こる．これはプリフェッチ処理の効果を無くすだけでなく同じデータを複数回読み込むことになり起動時間の増加を引き起こしてしまう．この問題を解決するために，本手法では，プリフェッチ処理リストを走査しながら，ファイルキャッシュが保持しているアクティブなデータのサイズを計算し，この量が閾値を超えた場合に，各プリフェッチ処理の最終開始時刻を遅ら

せない範囲でプリフェッチスレッドを停止させる期間を検出する．

この処理では，まずトレースを順方向にスキャンしながら，ファイルキャッシュに保持されるアクティブなデータの量があらかじめ決めた上限値を越える時刻T_uを検出する．次に，アクティブなキャッシュの値があらかじめ規定した下限値を下回る時刻か，次のプリフェッチの採集開始時刻のどちらか早い方の時刻T_Lを求める．この時刻T_uから時刻T_Lまでがプリフェッチ処理を停止する時刻となる（図４．１６）．実際には，提案手法ではアプリケーションのRead命令が読み出すデータ量を用いてプリフェッチ処理スレッドとの同期をとるため，この同期に使用可能なRead命令が存在するという条件が付加される．

図４．１７はこれらの処理を実施するアルゴリズムである．アルゴリズム中の makeNode()関数は，指定された範囲のプリフェッチ処理を実行した後，targetRead で指される read 処理で同期するための情報を生成する関数である．

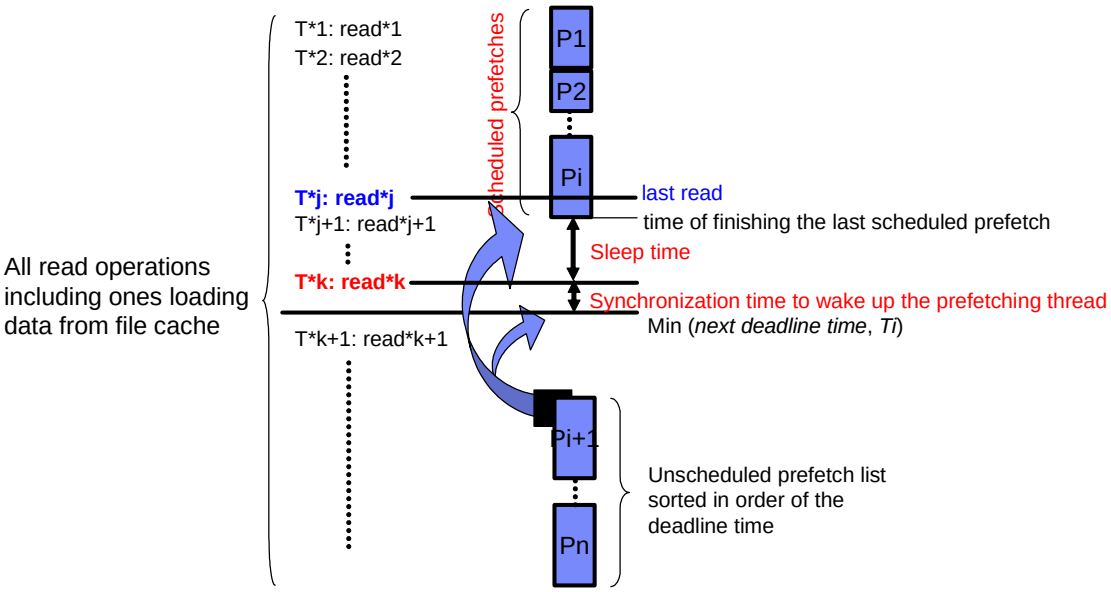


図４．１６　同期点の決定

```
01:    void decideSyncPoint () {
02:      Tr_total=Tp_current=Tc_offset=Tc_base=profidx=n_states=totalCacheSize=0;
03:      for (i = プリフェッチ処理リストの各要素を順にたどりながら繰り返す) {
04:        currentCacheSize = 0;
05:        startPref = P;
06:        while (P != NULL) {
07:          Tp_current += (Pの処理にかかる時間);
08:          totalCacheSize += (Pによりキャッシュに追加されるデータサイズ);
09:          totalCacheSize -= (Pの処理が完了する時刻までのアプリケーションの 実行で
                                                使用済みとなるキャッシュサイズ);
10:          T_target = (次のプリフェッチ処理要素の最終開始時刻) - (同期に要する時間);
11:          T_nextRead = (Pの処理が完了した時刻以降で最初に実行されるread処理の時刻);
12:          if (totalCacheSize ＞ (保持するキャッシュサイズの閾値) &&
13:              T_target ＞ Tp_current && T_target ＞ T_nextRead) {
14:            break;
15:          }
16:          P = (プリフェッチ処理リストにおけるPの次の要素);
17:        }
18:        targetRead = (T_targetの直前に実行されるRead処理);
19:        makeNode(startPref, P, targetRead);
20:    }
```

図４.１７　プリケーションとの同期点を検出するアルゴリズム

４．３．４　プリフェッチの実施

生成されたシナリオを用いて，対象プログラムの実行に同期してプリフェッチ処理を実行する．プリフェッチ処理の実行は，アプリケーションと独立のプリフェッチ処理スレッド上で実施される．アプリケーションの実行進度を得るために，read 処理をフックする関数を用意し，この関数中でファイルから読み出されるデータの合計値を元にプリフェッチ処理スレッドに対して同期のためのシグナルを送る．

プリフェッチ処理の実施例を図４．１８に示す．図４．１８ (a)はアプリケーションの実行に従って，read 処理でデータサイズａ， ｂ， ｃ…， ｇのデータを順に読み込む様子を示している．図４．１８(b)はプリフェッチ処理のシナリオで，幾つかのノードを直線状に並べたグラフとして表現されている．各ノードは一連のプリフェッチ処理を保持し，各エッジはその入力先のノードの開始条件としての読み込みデータ総量の値を保持する．あるノードのプリフェッチが完了すると，プリフェッチ処理スレッドはスリープ状態となる．アプリケーションが進行し読み込み総量が条件の値以上となった時点で

フック関数からシグナルが送られ，プリフェッチ処理スレッドが次のノードの処理を開始する．以降は同期処理を繰り返しながら，終端ノードに到達した時点でプリフェッチ処理が終了する．

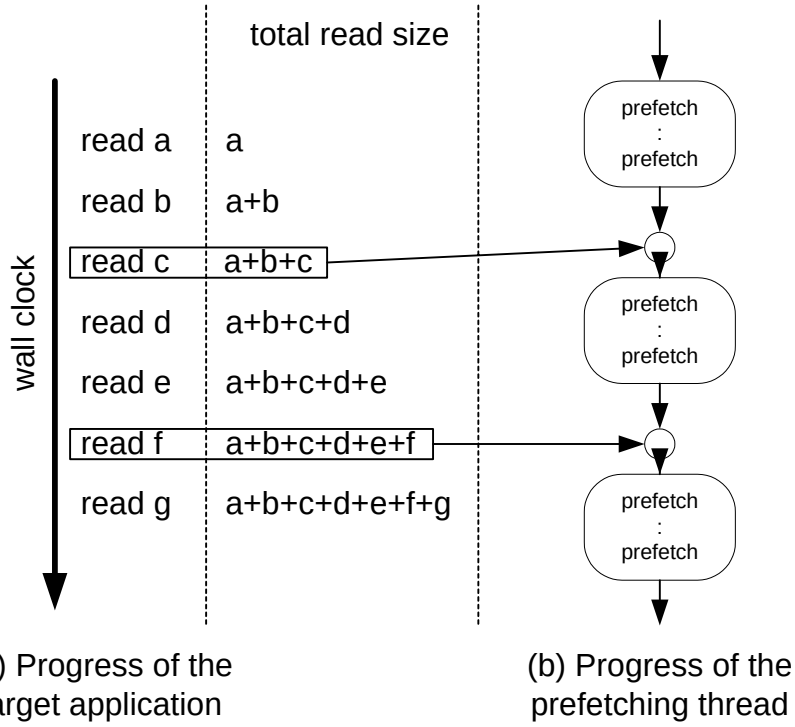


図４.１８　Scenario-based prefetching の実行モデル

４．４　実験結果

４．４．１　評価環境

評価は Intel Xeon 2.8GHz x 2, 2GB memory の IntelliStation MPro 上で実施した．使用した HDD は，HITACHI Deskstar DK250 120GB ATA100 モデルである．使用した Java VM は，IBM Java SDK 1.4.2 J9 prototype build である．評価は，WAS 単体の場合の起動時間，WAS に SPECjAppServer2002 ベンチマークをインストールした状態での WAS の起動時間，WAS に SPECjAppServer2004 ベンチマークをインストールした状態での起動時間を測定した．測定項目は，cold startup でプリフェッチ処理を実施しない場合の起動時間(cold startup)，cold startup でプロファイルの順にプリフェッチ処理を実施した場合の起動時間(cold+prefetch)，cold startup で Scenario-based prefetching 手法を用いてプリフェッチ処理を並べ替えた場合の起動時間(SBP)，warm startup でプリフェッチ処理を行わない場合の起動時間(warm startup)の４点である．

４．４．２　評価結果

図４．１９に，本手法の評価として，WebSphere Application Server 5.1.1 の起動時間を評価した結

果を示す．評価結果より，Scenario-based prefetching 手法を用いた場合，WAS 単体で 28¥%，SPECjAppServer2002 で 33%，SPECjAppServer2004 で 21%の速度向上があったことがわかる．また，いずれの場合でも，単純にプロファイルの順にプリフェッチ処理を実施した場合に比較して，プリフェッチ処理の並べ換えにより約 10%の速度向上がえられていることがわかる．一般に，本手法による効果は，対象とするアプリケーションにおいてファイル読み込み処理と他の計算処理がどの程度並列に実行できるかに依存する．今回の実験から，アプリケーションや Java VM に全く変更を加えない状態で平均 30%の速度向上を得ることが出来る点は，その実装のポータビリティとともに，本手法の有効性を示しているといえる．

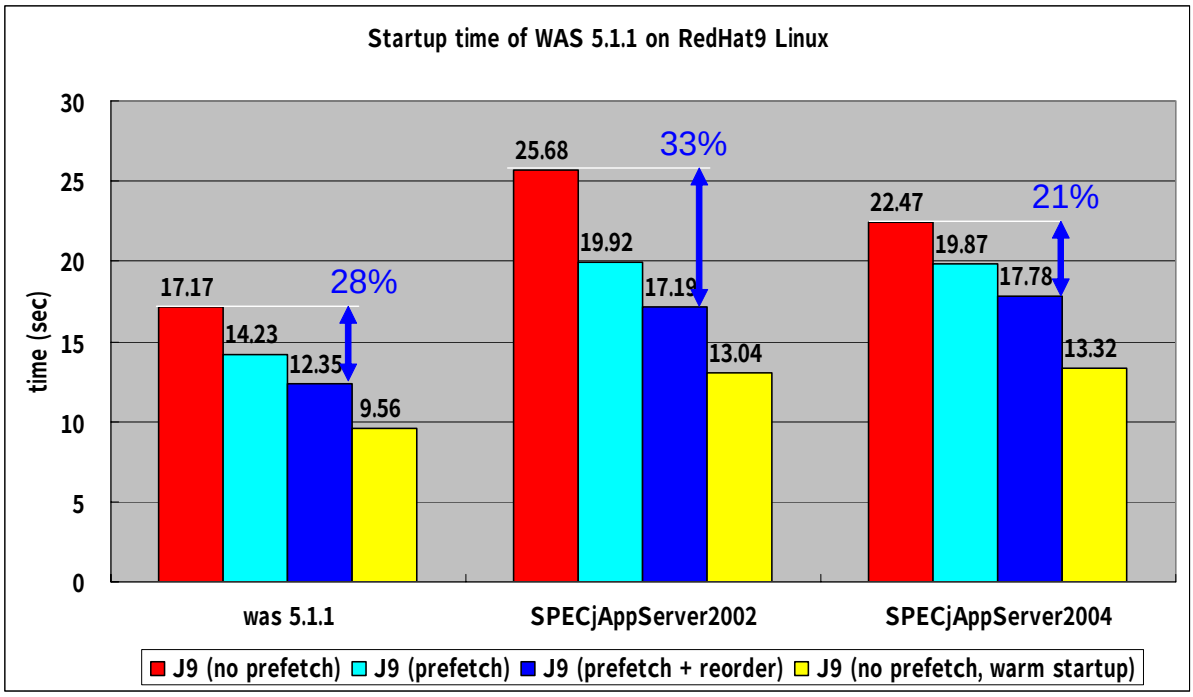


図４.１９　評価結果

４．５　関連研究

従来手法は，大きく７つの方法が提案されている．以下にそれらについて説明する．

４．５．１　プログラムの実行時のファイルアクセスパターンを用いてプリフェッチ処理を行う方法

この方法は，実行時にファイルのアクセスパターンを検出し，そのアクセスパターンに従って次にアクセスされる領域を予測してプリフェッチ処理を行う方法である．従来手法[47][48]では，特に UNIX におけるファイルのアクセスが，ファイル全体に対して逐次読み出しする場合がほとんどであるという特性を利用して，逐次読み出しに対してプリフェッチ処理を実施する方法である．しかしこの方法はファイルアクセスが一定のパターンで行われる場合のみ有効であるので，JAR ファイルからのクラス

読み込みのように，ファイルを不規則にアクセスする場合には効果がない．

４．５．２　プログラムの静的プロファイルを用いてプリフェッチ処理を行う方法

この手法は，静的プロファイルを用いる点で本手法と類似する．Lei らの手法[44]では，各実行ファイルに対して，その実行ファイルがアクセスするファイルとその順序を予め access tree として記録しておき，それらと実行時のファイルのオープン順序を比較して類似する access tree を検出し，それに基づいて次に open されるであろうファイルをプリフェッチ処理する方法である．また，Griffion らの方法[45]は，あるファイルとそのファイルが open されたときに次に open される可能性のあるファイルとの関連を，各 edge に頻度をもたせたグラフ（probability graph）であらわしておき，プリフェッチ処理するファイルを予想する方法である．しかし，いずれの方法もプリフェッチ処理の対象がファイル単位であるので，JAR ファイルのようにファイルの一部しか読み込まないファイルに対しては，不要データ読み込みによる遅延が生じてしまう問題がある．他方，Grimsrud らの方法[46]は，HDD の各クラスタに対して，そのクラスタがアクセスされた時に，次にアクセスされやすいクラスタの情報をその重みとともに保持してプリフェッチ処理を実施する方法である．この方法は，ランダムアクセスファイルに対しても有効である点が利点である．ただし，情報が HDD 上のセクターに関連付けられているために同じファイルを異なるアプリケーションから異なるパターンでアクセスする場合には，かえって性能低下を引き起こす問題もある．

４．５．３　アプリケーションを書き換えてプリフェッチ処理を行う方法

この方法は，アプリケーションを書き換えることで I/O wait 時間を削減する方法である．この方法として最も単純な方法は，アプリケーション作者がそのアルゴリズムを非同期 I/O 処理を用いて書き換える方法である．しかし非同期 I/O 処理を用いたアプリケーションの書き換えは，アプリケーションの規模が増大するに従って時間的，および作業量的に困難となる．アプリケーションのアルゴリズム自体は書き換えずに，I/O システムに対してアプリケーションの I/O アクセスに関する情報を伝える方法として，Win32 API におけるファイルアクセスのヒント[49]や，Patterson らによる I/O システムに disclosure と呼ぶ I/O アクセスのヒント情報を伝える方法[50]などが存在する．どちらの場合も，提供された情報を用いて I/O システムがプリフェッチ処理を生成して実施する．Win32 API の場合，ファイルに対して逐次アクセス属性とランダムアクセス属性を設定できるが，ランダムアクセス属性に対しては過去２回分の履歴しか保持しておらず，複雑なパターンには対応できない．また Patterson らの方法では，アプリケーション中に ioctrl 命令を使って I/O システムにファイルのアクセスパターンなどを伝えるコードを挿入し，I/O システムはその情報を元にプリフェッチ処理を実施する．この際，I/O システムは評価関数を用いてプリフェッチ処理によりメモリ上に保持されるデータ量とプリフェッチ処理による効果をを考慮して適切なプリフェッチ処理を生成する．しかしながら，この方法も結局は特定のパターンをもったファイルアクセスにしか対応できていない．

４．５．４ コンパイラによりプログラムを解析することでプリフェッチ処理を行う方法

この方法[52][53]は，コンパイラによりプログラムを解析することでファイルのアクセスパターンを検出し，そのアクセスパターンに従ってプリフェッチ処理コードをプログラム中に挿入する．しかしながら，現状では手続き間解析の能力の問題で，ループ内などの限られた範囲でしかプリフェッチ処理コードを生成することができていない．

４．５．５ プログラムを多重化して生成したプリフェッチ処理プロセスを用いる方法

この方法[42][43]では，対象とするプログラムから不要処理や副作用命令などを除外することでプリフェッチ処理プロセスを生成し，対象プログラムに対して投機的に実行することで，プリフェッチ処理を実施する方法である．プリフェッチ処理プロセスは基本的にもとのプログラムの実行を縮退させて生成するので，どのようなアクセスパターンに対しても対応可能であるが，反面プログラムを縮退させてしまうためにプリフェッチ処理プロセスの実行状態が元のプログラムの実行と必ずしも一致せず，誤ったデータをプリフェッチ処理してしまう状況を防げない問題が存在する．また，プリフェッチ処理プロセス自体のオーバヘッドにより，本来の実行を遅延させてしまう点も問題となる．Fraserらの方法[42]では，OS を修正し I/O 待ち時間が発生した場合にのみこのプロセスを実行させるように制御することで，元のプログラムを遅くしないことを保証する方法を提案しているが，プリフェッチ処理プロセスの実行時間を限定するためにプリフェッチ処理の効果が減少してしまう欠点もある．

４．５．６ ハードディスクの Seek 時間を最小化する方法

この方法[54][55]は，シークタイム最適化，あるいはエレベータシーキングとして知られている方法である．I/O システムの I/O 要求キューにある I/O 要求を，ハードディスクのシーク処理コストを最小限にするように並べ替える方法である．この方法は同時に I/O 要求キューに入っている I/O 要求に対してのみ適用されるので，処理と処理の間に依存関係があるような読み出し処理など同時に要求キューに入らない場合には効果がない．

４．５．７ ハードディスク上のファイルの配置を並べ替える方法

この方法[56][57][58]は，静的プロファイル情報を元に HDD 上のファイルの配置を実行時に連続アクセスとなるように並べ替えることで，シーク処理時間を最小化する手法である．アプリケーションの変更を必要としない点は我々の手法と同じであるが，OS 内に実装しないとならない点から，全てのシステムで利用できる方法ではない．

４．６ まとめ

本稿では，Scenario-based prefetching 手法という，静的プロファイルを用いてアプリケーション

に対してプリフェッチ処理を実施する方法を提案した．この方法では，静的プロファイルを解析して，シーク処理に伴うオーバヘッドを削減するようにプリフェッチする順序を並べ替えて生成したシナリオを用いてプリフェッチ処理を実施する．本手法を Redhat9 Linux 上で IBM WebSphere Application Server 5.1.1 を用いて評価した結果，平均約 30%の速度向上を得ることができた．

本手法はアプリケーションや実行環境の変更を必要としない点で，多くの計算機上で利用できるポータブルな手法である．大規模な既存のアプリケーションに対してもプログラムの書き換えをおこなうことなしにプリフェッチ処理による高速化を実施できる点は非常に有用である．特に，本論文で評価に用いたアプリケーションサーバは，その上で多くのＷｅｂアプリケーションが動作するフレームワークであり，この起動時間は多くのＷｅｂアプリケーションの運用や開発に影響を与える．本手法によるアプリケーションサーバの起動処理時間の短縮は，プロビジョニングなどのオンデマンドな運用における反応時間の向上や，Ｗｅｂアプリケーション開発時における頻繁な起動と終了の繰り返しにおいての開発効率の向上を実現する上で非常に有効な手法といえる．

第５章　結論

本研究の目的は，プログラムの実行前の書き換えを行うことなく，プロファイルを用いてプログラムを高速化することであった．プログラムの実行前に書き換えることは，プログラムを特定の状況に特化してしまい，結果としてプログラムの汎用性が減少する．特に近年の Java プログラムのように，１つのプログラムを異なるアーキテクチャで実行されるので，特定の環境や入力データに特化した最適化を実施したプログラムを作成してしまうと，プログラムのポータビリティが大きく損なわれてしまう．本論文では，この課題に対して２つのアプローチによりプログラムの高速化を実現する方法を提案した．１つは Just-in-Time コンパイラを用いて実行時にプログラムのプロファイルを収集し最適化する方法であり，もう１つは実行時サポートプログラムを用いてファイル読み込みを高速化する方法である．本論文では，このように事前の書き換えを伴わない最適化手法について議論を行ってきた．

本論文における研究成果を以下にまとめる．

- 関数内分岐履歴情報の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し，少量のプロファイル収集でも最大で 4%，平均 0.8%の速度向上が可能であることを確認し，その有効性を示した．
- 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て最大で 2.5%，平均 0.6%の速度向上を確認し，その有効性を示した．
- 高精度の実行時経路情報を低コストで収集するためのプロファイル手法として，構造的収集手法(Structural path profiling)を提案し，ベンチマークによる評価によってオフラインプロファイルと比較して約 90%の精度のパスプロファイル情報を，平均で 2-3%のオーバヘッドで収集できることを確認し，その有効性を示した．
- 対象アプリケーションにおける I/O 処理の静的プロファイルを用いてプリフェッチ処理を実行する手法（Scenario-Based Prefetching）として，事前に収集された静的プロファイルを用いてアプリケーションの挙動に合わせてプリフェッチ処理をスケジューリングしたシナリオ生成しておき，このシナリオを用いて実行時にプリフェッチ処理を実施する手法を提案し，実アプリケーションの起動時間を約 30%削減できることを確認し，その有効性を示した．

本論文を通じて提案してきた高速化手法は，いずれもプログラムの事前書き換えを必要としない方法である．この様な手法はプログラムの可搬性の向上とともに，細かなバージョンの違いによるプログラムの保守管理のコストの増加の問題も解決する．つまり，今後のプログラムの最適化手法は，単に実行時の高速化だけではなく，プログラムの大規模化に伴う開発コストや管理コストを軽減への影響を含めて，開発から管理までのライフサイクルを考慮しすることの重要性がますます増加すると思われる．

謝辞

本論文の執筆に当たり，学部・修士時代よりご指導頂きました，早稲田大学理工学部コンピュータ・ネットワーク工学科　村岡洋一教授に，深く感謝いたします．研究という道を歩むための多くの知識とともに，研究者としてのあり方など数多くのことを学ばせていただきました．また，本論文の審査をお引き受け頂き，審査を通して貴重なご助言をいただきました，早稲田大学理工学部コンピュータ・ネットワーク工学科　筧捷彦教授，後藤滋樹教授，中島達夫教授，に深く感謝いたします．公聴会・予備公聴会でいただいたコメントや議論によって，本論文をさまざまな角度から再考することができ，より広い観点よりまとめることができました．

本論文は，日本アイ・ビー・エム（株）東京基礎研究所で１９９５年に始まった Java Just-in-Time コンパイラプロジェクト，およびミドルウェアの高速化プロジェクトで得られた研究成果をまとめることで完成することができました．本プロジェクトを運営し，また本研究に終始ご協力を賜り，多くのご指導を賜りました，日本アイ・ビー・エム（株）東京基礎研究所　中谷登志男氏に深く感謝いたします．そのご指導を通じて，研究の価値を追及する視点や，研究の進め方など，研究者として備えるべき能力の多くを学ぶことができました．また，技術的な観点より数多くの助言とさまざまな議論を頂いた，日本アイ・ビー・エム（株）東京基礎研究所　小松秀昭氏に深く感謝いたします．幅広い技術領域を通じて関連する研究や有用なアイデアなどを教えていただき，研究の位置付けや価値を明確にし，深みのある研究を行うことができました．また，本論文執筆にあたり，技術的なご指導とともに論文執筆のための様々なご配慮を頂いた　日本アイ・ビー・エム（株）東京基礎研究所　小野寺民也氏に深く感謝いたします．また本研究を進める上で，プロジェクトを通じて終始熱心にご指導，ご討論を頂いた，日本アイ・ビー・エム（株）東京基礎研究所　郷田修氏（当時），菅沼俊夫氏，河内谷清久仁氏，石崎一明氏，小笠原武史氏，川人基弘氏，竹内幹雄氏，緒方一則氏，稲垣達氏氏，古関聰氏，今野和浩氏（当時），百瀬浩之氏（当時）田端邦夫氏（当時），に深く感謝いたします．また，Java Just-in-Time コンパイラプロジェクトに関わり支えて下さいました，日本アイ・ビー・エム（株）東京基礎研究所，日本アイ・ビー・エム（株）大和研究所，IBM T. J. Watoson Research Center, IBM United Kingdom Hursley Laboratory, IBM Austine Laboratory, IBM Canada Toronto Laboratory, の多くの方々に感謝いたします．

本研究の礎となる知識の多くは，早稲田大学村岡研究室に在籍中に身につけることができました．早稲田大学村岡研究室でともに学んだ先輩，同輩，後輩一同に深く感謝いたします．特に，研究室の先輩であり，共同研究者である，早稲田大学理工学部コンピュータ・ネットワーク工学科　山名早人教授に深く感謝いたします．並列処理システムー晴ープロジェクトを通じて，研究の仕方や論文の書き方など，研究者になるための多くの基礎的な事柄を一から教えていただき，自分の中で研究者としての道を確立するための大きな助けとなりました．

最後に，研究以外の側面で，精神的な支えとなってくれた，妻　倫子，研究の大いなる励みとなってくれた二人の子ども達　颯達，憧翔，そして，特に長期にわたる大学時代の生活を支えていただい

た両親に心からの感謝を送りたいと思います.

参考文献

[1] J. Gosling, B. Joy, and G. Steele, "The Java Language Specifications," Addison Wesley, 1996.

[2] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling versus Path Profiling: The Showdown," In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 134-148, Jan. 1998.

[3] T. Ball, J. R. Larus, "Efficient Path Profiling," In Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, pp. 46-57, Dec. 1996.

[4] E. Duesterwald, and V. Bala, "Software Profiling for Hot Path Prediction: Less is More," In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 202-211, Nov. 2000.

[5] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automatic Profiling and Optimization," In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 15-26, Oct. 1997.

[6] T. Ball, and J. R. Larus, "Optimally Profiling and Tracing Programs," ACM Transactions on Programming Languages and Systems, Vol.16, No.4, pp. 1319-1360, Jul. 1994.

[7] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," Software-Practice and Experience, Vol. 21, No.12, pp.1301-1321, Dec. 1991.

[8] K. Pettis, and R. C. Hansen, "Profile Guided Code Positioning," In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation'90, pp. 16-27, Jun. 1990.

[9] R. Cohn, and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications," In Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, pp. 80-89, Dec. 1996.

[10] G. Ammons, J. R. Larus, "Improving Data-flow Analysis with Path Profiles," In Proceedings of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation, PLDI'98, pp. 72-84, Jun. 1998.

[11] R. Gupta, D. A. Berson, and J. Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," In Proceedings of the 13th annual IEEE/ACM international symposium on Microarchitecture, pp. 358-368, De. 1997.

[12] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp.102-115, Nov. 1997.

[13] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," In Proceedings of International Conference on Computer Languages, May 1998.

[14] R. Bodik, R. Gupta, and M. L. Soffa, "Complete Removal of Redundant Expression," In Proceedings of the ACM SIGPLAN'98 conference on Programming language design and implementation, pp.1-14, Jun. 1998.

[15] M. Kawahito, H. Komatsu, and T. Nakatani, "Eliminating Exception Checks and Partial Redundancies for Java Jus-in-Time Compilers," IBM Research Report, RT0350, Apr. 2000.

[16] M. Yang, G. Uh, and D. B. Whalley, "Improving Performance by Branch Reordering," Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pp. 130-141, Jun. 1998.

[17] D. A. Spuler, "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] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A Study of Devirtualization Techniques for a Java(TM) Just-In-Time Compiler," In Proceedings of Conference on Object-oriented programming, systems, languages, and applications, pp. 47-65, Oct. 2000.

[20] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive Optimization in the Jalapeno JVM," In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, OOPSLA '00, pp. 294-310, Oct. 2000.

[21] V. Bala, E. Duesterwald, S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," In Proceedings of the ACM SIGPLAN'00 conference on Programming Language Design and Implementation, PLDI'00, pp. 1-12, Jun. 2000.

[22] M. Cierniak, G. Y. Lueh, and J. M. Stichnoth, "Practicing JUDO: Java(TM) Under Dynamic Optimizations," In Proceedings of the ACM SIGPLAN'00 conference on Programming language design and implementation, pp. 13-26, Jun. 2000.

[23] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," IBM Systems Journal, Vol.39, No.1, pp. 175-193, 2000.

[24] S. S. Muchnick, "Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, San Francisco, California

[25] Arnold, M., and Ryder, B. G. A Framework for Reducing the Cost of Instrumented Code. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01, pp. 168-179, Jun. 2001.

[26] Chilimbi, T. M., and Hirzel, M. Dynamic Hot Data Stream Prefetching for General-Purpose Program, In Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation, pp.199-209, Jun. 2002.

[27] Feller, P. T. Value profiling for instructions and memory locations. Master Thesis CS98-581, University of California, San Diego, Apr. 1998.

[28] Havlak, P. Nesting of Reducible and Irreducible Loops. ACM Transactions on Programming Languages and Systems, 19(4), pp.557-567, 1997.

[29] Hirzel, M., and Chilimbi, T. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling, In Workshop on Feedback-Directed and Dynamic Optimizations (FDDO), 2001.

[30] Hollingsworth, J. K., Miller, B. P., Goncalves, M. R., Naim, O., Xu, Z., and Zheng, L. MDL: A Language and Compiler for Dynamic Program Instrumentation. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT '97, pp. 201-212, Nov. 1997.

[31] Nevill-Manning, C. G., and Witten, I. H. Linear-time, incremental hierarchy inference for compression, In Proceedings of the Data Compression Conference (DCC'97), pp.3-11, 1997

[32] Paleczny, M., Vick, C., and Click, C. The Java HotSpot Server Compiler. In Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01), pp. 1-12, Apr. 2001.

[33] Standard Performance Evaluation Corporation. SPECjvm98 available http://www.spec.org/osg/jvm98.

[34] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages ¥& Applications, OOPSLA '01, pp. 180-194, Oct. 2001.

[35] Traub, O., Schechter, S., and Smith, M. D. Ephemeral Instrumentation for Lightweight Program Profiling. Technical Report, Harvard University, 1999.

[36] Young, C., and Smith, M. D., Better Global Scheduling Using Path Profiles. In Proceedings of 31th International Conference on Microarchitecture, MICRO-31, pp.115-123, 1998

[37] Young, C., and Smith, M. D., Static Correlated Branch Prediction. ACM Transactions on Programming Languages and Systems, 21(5), pp.1028-1075, 1999.

[38] Cohn, R. and Lowney, P. G., Feedback directed optimization in Compaq's compilation tools for Alpha, in Proceedings of 2nd ACM Workshop on Feedback-Directed Optimization, 1999.

[39] Fisher, J., Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Transactions on Computers, vol. C-30, no. 7, pp. 478-490, 1981.

[40] Hwu, W. and Mahlke, S. A. and Chen, W. Y. and Chang, P. P. and Warter, N. J. and Bringmann, R. A. and Ouellette, R. G. and Hank, R. E. and Kiyohara, T. and Haab, G. E. and Holm, J.

G. and Lavery, D. M., The Superblock: An Effective Technique for VLIW and Superscalar Compilation, Journal of Supercomputing, vol. 7, no. (1,2), pp.229-248, 1993.

[41] Wall, D. W., Predicting program behavior using real or estimated profiles, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.59-70, 1991.

[42] Fraser, K. and Chang, F., Operating System I/O Speculation: How two invocations are faster than one, in Proceedings of the USENIX 2003 Annual Technical Conference, pp.325-338, 2003

[43] Yang, C. K., Mitra, T. and Chiueh, T. C., A Decoupled Architecture for Application-Specific File Prefetching, in Proceedings of the USENIX Annual Technical Conference, pp.157-170, 2002

[44] Lei, H. and Duchamp, D., An Analytical Approach to File Prefetching, Proceedings of the USENIX Annual Technical Conference, pp.275-288, 1997

[45] Griffioen, J. and Appleton, R., Reducing File System Latency using a Predictive Approach, in Proceedings of the USENIX Summer 1994 Technical Conference, pp.197-208, 1994

[46] Grimsrud, K. S., Archibald, J. K., and Nelson, B. E., Multiple Prefetch Adaptive Disk Caching, IEEE Transaction on Knowledge and Data Engineering, vol.5, No.1, pp.88-103, 1993

[47] Feiertang, R. J. and Organisk, E. I., The Multics Input/Output System, in Proceedings of Third Symposium on Operating System Principles, pp.35-41, 1971

[48] McKusick, M. K., Joy, W. J., Leffler, S. J. and Fabry, R. S., A Fast File System for Unix, ACM Transactions on Computer Systems, Vol.2, No.3, pp.181-197, 1984

[49] Richter, J., Advanced Windows - The Developer's Guide to the Win32(R) API for Windows NT(TM) 3.5 and Windows 95, Microsoft Press, 1995

[50] Patterson, R. H. et. al., Informed Prefetching and Caching, in Proceeding of Fifteenth Symposium on Operating System Principles, pp.79-95, 1995

[51] Cao, P., Felten, E. W., Karlin, A. and Kai Li, Implementation and Performance of Application-Controlled File Caching, in Proceedings of First USENIX Symposium on Operating Systems Design and Implementation, pp.165-178, 1994

[52] Brown, A. D. and Mowry, T. C., Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently, in Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), pp.31-44, 2000

[53] Mowry, T. C., Demke, A. K. and Krieger, O., Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Application, in Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.297-306, 1993

[54] Gotlieb, C. C. and MacEwen, G. H., Performance of Movable-Head Disk Strage Devices, Journal of ACM, Vol.20, No.4, pp.604-623, 1973

[55] Geist, R. and Daniel, S., A Continuum of Disk Scheduling Algorithms, ACM Transactions on

Computer Systems, Vol.5, No.1, pp.77-92, 1987

[56] Akyurek, S. and Salem, K., Adaptive block rearrangement, ACM Transactions on Computer Systems (TOCS), Vol.13, No.2, pp.89-121, 1995

[57] Vongsathorn, P. and Carson, S. D., A system for adaptive disk rearrangement, Software Practice and Experience, Vol.20, No.3, pp.225-242, 1990

[58] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H., Semantically-Smart Disk Systems, in Proceedings of Second USENIX Conference on File and Storage Technologies, pp.73-89, 2003

[59] Song, L., and Kavi, K. M., A Technique for Variable Dependence Driven Loop Peeling, In proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), 2002.

[60] Song, L., and Kavi, K. M., A Technique for Variable Dependence Driven Loop Peeling, In proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), 2002.

研究業績

●は筆頭著者であるもの.

○は筆頭茶者ではないもの.

論文誌（査読有）：　１８件（うち主著５件）

○ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, ″A region-based compilation technique for dynamic compilers,″ ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 28, Issue 1, pp. 134-174, January 2006.

○ Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani, ″Design and evaluation of dynamic optimizations for a Java just-in-time compiler,″ ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 27, No. 4, pp. 732-785, July 2005.

● 安江　俊明, 小松　秀昭, 中谷　登志男, ″静的プロファイルを用いたファイル・プリフェッチ手法の提案,″ 情報処理学会論文誌プログラミング, Vol. 46, No. SIG 14 (PRO 27), pp. 55-65, Oct. 2005.

● Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani, ″Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers,″ The Journal of Instruction Level Parallelism, Vol. 6, pp. 1-28, April 2004.

○ Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani, ″Evolution of a Java just-in-time compiler for IA-32 platforms,″ IBM Journal of Research and Development, IBM Research in Asia Issue, Vol. 48, No. 5/6, pp. 767-795, 2004.

● 安江　俊明, 菅沼　俊夫, 小松　秀昭, 中谷　登志男, ″動的コンパイラにおける実行時経路情報の構造的収集手法の提案,″ 情報処理学会論文誌プログラミング, Vol. 44, No. SIG 15(PRO 19), pp. 24-35, Nov. 2003.

● 安江　俊明, 緒方　一則, 小松　秀昭, ″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,″情報処理学会論文誌プログラミング, Vol.43, No. SIG 1(PRO13), pp.75-84, 2002.

○ 石崎　一明, 安江　俊明, 川人　基弘, 小松　秀昭, ″コード書き換えによる動的メソッド呼び出しの直接 devirtualization,″ pp.124-136, 情報処理学会論文誌, Vol. 43, No.1, 2002

○ Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu and Toshio Nakatani, ″Design, implementation, and evaluation of optimizations in a Java(TM) Just-In-Time compiler,″ Concurrency: Pract. Exper. pp.457-475, 2000

○ Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu and Toshio Nakatani, "Overview of the IBM Java Just-in-Time Compiler," IBM System Journal, Vol.39, No.1, pp.175-193, 2000

● 安江　俊明, 村岡　洋一, "データフロー計算機のためのコンパイル技法 - 制御フローグラフからデータフローグラフへの最適変換アルゴリズム," 電子情報通信学会論文誌, Vol.J78-D-I, No.2, pp.189-199, 1995.

○ 山名　早人, 安江　俊明, 村岡　洋一, 山口　善教, "分散共有メモリ型並列計算機における１重Doacross 型ループの実行時間算出法,"電子情報通信学会論文誌, Vol. J78-D-I,No.2, pp.170-178, 1995.

○ 山名　早人, 安江　俊明, 石井　吉彦, 村岡　洋一, "並列処理システムにおけるマクロタスク間先行評価方式," 電子情報通信学会論文誌, Vol.J77-D-I, No.5, pp.343-353, 1994.

○ 石井　吉彦, 安江　俊明, 山名　早人, 村岡　洋一, "データ駆動計算機におけるDOACROSS型ループに対するフロー処理方式の提案,"電子情報通信学会論文誌, Vol. J-75-D-I, No.7, pp.440-449, 1992.

○ 山名　早人, 安江　俊明, 神舘　淳, 村岡　洋一, "並列処理システム-晴-におけるフローグラフ展開を用いた条件分岐の並列実行,"早稲田大学情報科学教育センター 紀要,Vol.12, pp.8-18, 1991.

○ H. Yamana, J. Kohdate, T. Yasue, and Y. Muraoka, "An Environment for Dataflow Program Development of Parallel Processing System -Harray-," Systems and Computers in Javan, Vol.22, No.8, pp.26-38, 1991.

○ 山名　早人, 神舘　淳, 安江　俊明, 村岡　洋一, "並列処理システム-晴-におけるデータフロープログラム開発環境,"電子情報通信学会論文誌 ,Vol.J73-D-I, No.6, pp.569-579, 1990.

○ H. Yamana, Y. Kusano, T. Yasue, J. Kohdate, T. Hagiwara, and Y. Muraoka, "parallel Processing System -Harray-," Computing Systems in Engineering, Vol.1, No.1, pp.111-130, 1990.

書籍：　１件（うち主著１件）

● Yasue, T. and Muraoka, Y., "A FORTRAN Compiler and a Visual Environment of Program Development for Dataflow Machines," in Parallel Language and Compiler Research in Japan, Chapter 7, KLUWER ACADEMIC PUBLISHERS, 1995.

国際会議（査読有）：　８件（うち主著２件）

○ K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler," ACM

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), pp. 187-204, October 26-30, 2003.

● T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani, An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers, In Proceedings of The Twelfth International Conference on Parallel Architectures and Compilation Techniques (PACT-2003), pp. 148-158, 2003.

○ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, "A Region-Based Compilation Technique for a Java Just-In-Time Compiler," ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), pp. 312-323, June 9-11, 2003.

○ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, "An Empirical Study of Method Inlining for a Java Just-In-Time Compiler," 2nd Java Virtual Machine Research and Technology Symposium (JVM '02), pp. 91-104, August 1-2, 2002.

○ Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, "A Dynamic Optimization Framework for a Java Just-In-Time Compiler," ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), pp. 180-194, October 14-18, 2001.

○ K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A Study of Devirtualization Techniques for a Java(TM) Just-In-Time Compiler," In Proceedings of Conference on Object-oriented programming, systems, languages, and applications, pp.47-65, Oct. 2000.

○ K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu and T. Nakatani, "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler," ACM 1999 Java Grande Conference, 1999.

● T. Yasue, H. Yamana, and Y. Muraoka, "A FORTRAN Compiling Method for Dataflow Machines and its prototype compiler for parallel processing system -Harray-," Fifth International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, LNCS 757, pp.482-496, 1992.

国内会議（査読有）： ４件（うち主著３件）

● 安江 俊明, 金子 正教, 萩原 純一, 新開 正史, 山名 早人, 村岡 洋一, "超並列のためのマルチアーキテクチャコンパイラ開発環境-はれだす-," JSPP'92, pp.139-146, 1992.

○ 山名 早人, 安江 俊明, 石井 吉彦, 村岡 洋一, "先行評価を用いたマクロタスクの多段先行実行方式の提案," JSPP'92, pp.117-122, 1992.

● 安江 俊明, 神舘 淳, 山名 早人, 村岡 洋一, "並列処理システム-晴-におけるプロトタイプFORTRANコンパイラ," JSPP'91, pp.285-292, 1991.

● 安江 俊明, 神舘 淳, 山名 早人, 村岡 洋一, "並列処理システム-晴-におけるデータフロー

プログラム開発環境,” JSPP’90, pp.233-239, 1990.

国内会議（査読無）： ３０件（うち主著８件）

- ● 安江 俊明, 小松 秀昭, 中谷 登志男, ″静的プロファイルを用いたファイル・プリフェッチ手法の提案,” 情報処理学会プログラミング研究会, 2005.
- ● 安江 俊明, 菅沼 俊夫, 小松 秀昭, 中谷 登志男, ″動的コンパイラにおける実行時経路情報の構造的収集手法の提案,” 情報処理学会プログラミング研究会, 2003.
- ● 安江 俊明, 緒方 一則, 小松 秀昭, ″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,” 情報処理学会プログラミング研究会, 2002.
- ● 安江 俊明, 仲 顕照, 小松 秀昭, 深澤 良彰, ″バイトコードパターンマッチによる Java インタプリタの高速化手法,″情報処理学会プログラミング研究会, PRO2000-1, 2000.
- ○ 仲 顕照, 安江 俊明, 小松 秀昭, 深澤 良彰, ″バイトコードパターンマッチングを用いた Java インタプリタの高速化,” SS99-60, 信学技法, Vol.99, No.548, pp.1-8, 2000.
- ● 安江 俊明, 萩原 純一, 金子 正教, 村岡 洋一, ″分散メモリ型並列計算機用 FORTRAN コンパイラの構築,” 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第３回シンポジウム, pp.2-11, 1993.
- ● 安江 俊明, 金子 正教, 萩原 純一, 田原 歩, 山名 早人, 村岡 洋一, ″超並列のためのマルチアーキテクチャコンパイラはれだすの内部表現,” 処全体 1992 年 3U-8, 1992.
- ○ 萩原 純一, 安江 俊明, 金子 正教, 山名 早人, 村岡 洋一, ″分散メモリ型並列計算機における DO ループ処理方式の提案,” 信学技報, CPSY92-15, pp.47-54, 1992.
- ○ 山名 早人, 石崎 一明, 安江 俊明, 村岡 洋一, ″並列処理システム-晴-における条件分岐の先行評価制御方式,” 情処研報, 91-ARC-89-19, pp.135-142, 1991.
- ○ 金子 正教, 中里, 安江 俊明, 山名 早人, 村岡 洋一, ″ループ並列化手法’Dependent-flow ループ化’の提案,” 信学技報, CPSY91-32, pp.221-228, 1991.
- ○ 石崎 一明, 安江 俊明, 山名 早人, 村岡 洋一, ″先行評価に適した並列計算機のネットワーク構成,” 情処研報, 91-ARC-89-20, pp.143-150, 1991.
- ● 安江 俊明, 神舘 淳, 山名 早人, 村岡 洋一, ″並列処理システム-晴-における FORTRAN to Dataflow コンパイラ,” 信学技報, CPSY90-25, pp.77-82, 1990.
- ○ 山名 早人, 安江 俊明, 神舘 淳, 村岡 洋一, ″並列処理システム-晴-におけるマクロブロック管理方式,” 信学技報, CPSY90-24, pp.71-76, 1990.
- ○ 石井 吉彦, 安江 俊明, 山名 早人, 村岡 洋一, ″データ駆動計算機におけるカラー管理方式の実装 - DOACROSS 型ループの高速化をめざして,” 信学技報,90-37, pp.149-154, 1990.
- ○ 石井 吉彦, 安江 俊明, 山名 早人, 村岡 洋一, ″科学技術計算用並列処理システム-晴-におけるカラー管理方式の評価,” 信学技報,CPSY90-3, pp.17-24, 1990.
- ● 安江 俊明, 神舘 淳, 萩原 純一, 山名 早人, 村岡 洋一, ″並列処理システム-晴-の低レベ

ルプログラムビジュアル開発環境,"信学技報,CPSY89-2, pp.9-16, 1989.

○ 山名　早人, 草野　義博, 神舘　淳, 安江　俊明, "並列処理システムー晴ーにおけるＣＤフロー（Controlled Dataflow）方式," 信学技報, CPSY89-55, pp.63-68, 1989.

○ 神舘　淳, 安江　俊明, 山名　早人, 村岡　洋一, "並列処理システム-晴-におけるフローグラフ展開を用いたコンパイル手法," 信学技報, CPSY89-56, pp.69-74, 1989.

○ 神舘　淳, 安江　俊明, 萩原　純一, 村岡　洋一, "並列処理システム-晴-の低レベルソフトウェア開発環境," 情処研報, 88-ARC-73-3, pp.17-22, 1988.

○ 他　全国大会１０件, ユーザ会１件

特許出願：　７件

● Toshiaki Yasue, and Hideaki Komatsu, "Scenario-based prefetching," 2004.

● Toshiaki Yasue, Toshio Suganuma, and Hideaki Komatsu, "Structural Path Profiling on Dynamic Compilers," 2003.

○ Motohiro Kawahito, Hideaki Komatsu, and Toshiaki Yasue, "An elimination technique of redundant array range check by multi range checking," JA9-98-215, 1999.

○ Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Hideaki Komatsu, and Mikio Takeuchi, "A Code Generation Technique to Keep Trade-Off between Register Pressure and Parallelism of Program," 1999.

● Toshiaki Yasue, Kazunori Ogata, Kazuaki Ishizaki, and Hideaki Komatsu, "Dynamic compiling and transferring scheme with multiple transferring points for a partially executed method," 1999.

○ Motohiro Kawahito, Toshiaki Yasue, and Hideaki Komatsu, "An elimination technique of redundant array range check," 1998.

● Toshiaki Yasue, Hideaki Komatsu, and Takeshi Ogasawara, "A method inlining technique of a method including virtual method invocations for a Java Just-in-Time Compiler," 1998.