2003－ARC－154　(6)
2003／8／5

間歇的プロファイリングの提案とSPECint95による評価

加藤文彦[†]　大津金光[†]
横田隆史[†]　馬場敬信[†]

実行時にプログラムのパスの挙動が動的に変化する場合、コンパイラによる静的な最適化処理には限界が存在する。その場合、実行時にパスの挙動を調べてホットパスを検出し、その情報を基に動的に最適化を行なうことで速度向上の可能性が出てくる。一般に、ホットパスを検出する方法としてパスプロファイリングが用いられるが、その際に発生するオーバーヘッドが問題となる。本稿では、プロファイリングを非周期的に行うことで、ホットパスの検出精度をある程度保ちつつオーバーヘッドを削減を目指す“間歇的プロファイリング”を提案し、SPECint95ベンチマークを用いてホットパスの検出精度とオーバーヘッドの評価を行なう。

A Proposal of Intermittent Profiling and its Evaluation by SPECint95

Fumihiko Katou,[†] Kanemitsu Ootsu,[†] Takashi Yokota[†]
and Takanobu Baba[†]

When execution paths change dynamically, static optimization by compiler has a limitation. If we detect hot-paths by capturing path's behavior and optimize the program based on the information, we can achieve the efficient execution performance. As the path profiling is used to detect path's behavior, but it's overhead may be a serious problem. In this paper, we propose Intermittent Profiling which reduces the overhead and maintaining the hot-path detection accuracy, and evaluate the accuracy of hot-path detection and overhead by SPECint95 benchmarks.

1. はじめに

実行時にプログラムのパスの挙動が動的に変化する場合、コンパイラによる静的な最適化処理には限界が存在する。この場合、実行時にプロファイリングにより頻繁に実行されるパス(ホットパス)の変化を検出し、その情報を基に動的に最適化を行なうことで速度向上の可能性が出てくる。

その際に、正確なプロファイリングを行なうことにより、より効率の良い最適化処理が可能となるが、プロファイリングによる実行時のオーバーヘッドにより、実行性能が低下する可能性がある。

ソフトウェアのみで正確なプロファイリングを行なう手法の中で、オーバーヘッドが低い手法としてEfficient Path Profiling[1]が存在するが、プログラム全体に対して適用しプロファイリングを行った場合、実行時のオーバーヘッドが大きくなり高速化が困難になるという問題がある。この問題に対する対処法として、プログラムカウンタ(PC)を定期的に取得することで頻繁に実行される部分(ホットスポット)を検出する、サンプリングの手法[4]と組み合わせ、サンプリングにより検出されたホットスポットについてさらにパスプロファイリングを行い、パスの挙動を正確に解析する、という手法[5]が存在する。この手法は、PCサンプリングによりプロファイルリングを行なう部分を限定することでオーバーヘッドを削減できる反面、プロファイリング処理が2段階になり、ホットパス検出までの遅延が大きいという問題が存在する。

そこで、オリジナルコードとプロファイルコードを挿入したコードを用意し、実行時にそれらを状況に応じて切り替えることでオーバーヘッドを削減を目指す**間歇的プロファイリング**を提案する。本手法を用いた場合、コードの切り替え方次第でプロファイリングを行う時間が変わり、それによって精度やオーバーヘッドが変動するため両者の間のトレードオフが問題となる。

本稿では、間歇的プロファイリングをEfficient Path Profilingに対して適用し、SPECint95を対象としてプロファイリングを行なうことで、ホットパスの検出精度とオーバーヘッドの評価を行なう。

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

2. プロファイリング

2.1 間歇的プロファイリング

本手法は、プログラムの実行前に各々の関数に対し、任意のプロファイリング手法を用いてプロファイルコードを挿入した関数と、挿入しない関数を用意し、２つのコードを切り替える為のコードを関数の先頭に挿入する。そしてプログラム実行時には、プロファイルコードを挿入した関数と挿入しない関数を必要に応じて切り替えながら実行する。この場合、プロファイルコードのサイズが増大しキャッシュのヒット率が低下したり、関数の切り替えに要するオーバーヘッドが増えるといった欠点が存在する。しかし、プロファイルコードによるオーバーヘッドが大きい場合は、関数の切り替えることでプロファイルコードを実行する回数を大幅に減らし、そのオーバーヘッドを削減することで、全体のオーバーヘッドを低く抑えることが可能となる。

本手法はオーバーヘッドの大きいパスプロファイリングなどに適用することが可能であるが、実装する際には関数の切り替え方が問題となる。そこで、同じくコードの切り替えによってオーバーヘッドを抑えるエッジプロファイリングの手法である Ephemeral Instrumentation[2] を参考とし、次の方法を用いる。

プロファイルコードを挿入した関数側のパスの実行回数を計測する部分に、その関数でパスの計測を行った回数を測定するコード、及びその回数が既定値に達するとプロファイルコードを挿入していない関数の対応する部分へと分岐するコードを挿入する。そして、関数の先頭にはプロファイルを取った回数が既定値未満ならプロファイルコードを挿入した関数へ、そうでなければプロファイルコードを挿入していない関数へと分岐する条件分岐を挿入する。プログラム実行時には、一定間隔毎にタイマー割り込みを用いて各関数のパスの計測を行った回数をリセットし、再びその関数が呼び出されたときにプロファイルコードを挿入した関数へと分岐させる。ここで、プロファイル回数が既定値に達し、プロファイルを取らない関数へと分岐する処理を unhook、unhook の閾値を Unhook Threshold(UT)、タイマー割り込みを行ないパスの計測を行った回数をリセットする処理を rehook、rehook を行なう間隔を Rehook Interval(RI) と定義する。

この方法を用いると、rehook が発生してもその後関数が呼び出されない場合、プロファイリングが行なわれないという問題があるが、タイマー割り込み時の処理を最小限にし、プロファイルコードを挿入しない関数に対して一切コードを挿入しないため、オーバーヘッドを極力抑えることができる。

2.2 Efficient Path Profiling

間歇的プロファイリングのプロファイリングコードを挿入するアルゴリズムとして用いた Efficient Path Profiling(EPP) について説明する。EPP は、プログラムコードに対していくつかのプロファイルコードを挿入するという、ソフトウェアレベルの処理のみでパスプロファイリングを行う手法である。先ず、プログラムの基本ブロック間にカウンタを挿入し、それによってパスの番号付けを行う。そしてパスの終端部分となるバックエッジにおいてパスの実行回数を計測する。EPP では、基本ブロック間のカウンタの数は最小に抑えられており、正確なプロファイリングを行う手法の中ではオーバーヘッドが低いという特徴である。

EPP の適用例を図 1に示す。図の黒塗り四角の部分はカウンタを挿入した部分を示し、隣に行った処理が記されている。r の式で書かれている部分を通る度に、カウンタ r を加算しパスの番号付けを行い、各パスは図の右側の様に番号付けされる。そして、パスのバックエッジもしくは関数の呼び戻し部分において、r で示される番号のパスの実行回数を計測 (+1) する。

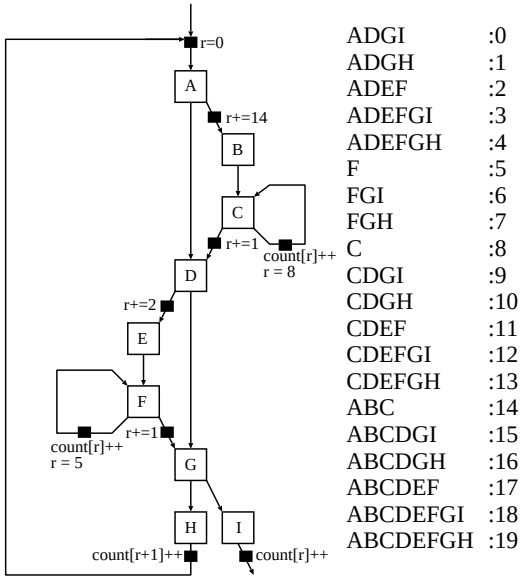

図 1 EPP の適用例

2.3 間歇的プロファイリングの実装

EPP を用いて間歇的プロファイリングを適用した例を図 2に示す。初めに、関数が呼び出されると、図の黒塗り三角の部分において、プロファイルを取った回数が UT 未満ならば EPP を適用した関数 (図の右側) へ、そうでなければプロファイルを行わない関数 (図の左側) へと分岐する。EPP を適用した関数へと分岐した場合、各バックエッジ (図の黒塗り丸の部分) において、プロファイルを取った回数を測定し、その回数が既定値に達するとプロファイルを取らない関数へ分岐 (図の破線部分) する。一度この unhook 処理が行なわれると、一定時間が経過してタイマー割り込みが発生したときに rehook 処理を行ない、その後その関数

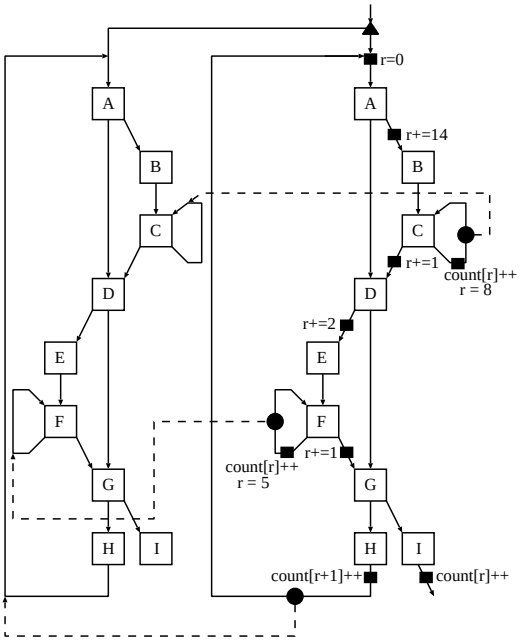

図2 EPPによる間歇的プロファイリングの適用例

が呼び出されるまで、プロファイルを行なわない関数を実行する。こうすることで、unhook時は余分に実行するコードが関数切り替えコードに限られるため、オーバーヘッドの軽減が期待できる。

3. 評　　価

3.1 評価環境

評価には、SimpleScalarをベースにしたシミュレータSIMCA[6)7)]にタイマー割り込み機能を追加したものを用い、シミュレーションパラメータは表1に示す値を用いた。

表1 シミュレーションパラメータ

スレッドユニット	4命令同時実行 out-of-order実行
1次キャッシュ(命令)	direct-map 16KB(ラインサイズ32B) レイテンシ1クロック
1次キャッシュ(データ)	4-way set-associative 16KB(ラインサイズ32B) レイテンシ1クロック
2次キャッシュ	4-way set-associative 256KB(ラインサイズ64B) レイテンシ6クロック
メモリ	レイテンシ18クロック

評価用アプリケーションにはSPECint95の129.compressを用い、それをSIMCA用gccクロスコンパイラ(version 2.7.2.3 最適化オプション -O3)を用いて生成したコードを元に評価を行なった。また、入力データセットにはtrainを用いた。

3.2 評価方法

間歇的プロファイリングを用いた場合、UTやRIの設定値によってはオーバヘッドが元よりも大きくなってしまう可能性がある[3)]。そこで、実用的なUT, RIの範囲を調べる必要がある。

そこで、間歇的プロファイリングを適用したときのオーバーヘッドを、

$$overhead(IP) = \frac{(OH(PP)+OH(UH))\times UT+OH(RH)}{RI}\times 100(\%)$$

と定義する。この内rehookのオーバーヘッドは、

$$overhead(rehook) = fracOH(RH)RI$$

である。OH(PP)はパスの計測1回当りのオーバーヘッド、OH(UH)はunhook(図2の黒丸部分)の処理一回当りのオーバーヘッド、OH(RH)はrehook処理一回当りのオーバーヘッドを表しており、次式で表される。

$$OH(PP) = \frac{EC(EPP) - EC(ORI)}{T(PP)}(clock)$$

$$OH(UH) = \frac{EC(IP1) - EC(EPP)}{T(PP)}(clock)$$

$$OH(RH) = \frac{EC(IP2) - EC(IP1)}{T(RH)}(clock)$$

EC(ORI)はオリジナルコードの実行サイクル数、EC(EPP)はEPPを適用したコードの実行サイクル数、EC(IP1)はUTとRIを無限大としたときの間歇的プロファイリングの実行サイクル数、EC(IP2)はUTのみ無限大とし、RIはrehookが一回以上行なわれるように設定した間歇的プロファイリングの実行サイクル数、T(RH)はそのときにrehookが行われた回数、T(PP)はEPPを適用したコードにおいてパスの計測を行った回数を表す。

こうして定義したオーバーヘッドを用いてUTやRIの範囲を求めるため、129.compressの関数compressのみを対象として、実験的に各種プロファイリングを行う。この際、オーバーヘッドの目標を次のように設定する。rehookによるオーバーヘッドはタイマー割り込みによるもので、プロファイリングによるオーバーヘッドとは独立なため、出来る限り抑える必要があるため、このオーバーヘッドの目標を1%以下とする。また、全体のオーバーヘッドは、オーバーヘッドの削減に重視し、EPPを適用したコードの1／10以下として、得られた結果から計算によってUTとRIの範囲を求める。

次に、EPPに対し間歇的プロファイリングを適用することで、オリジナルのEPPと比べてホットパスの検出精度やオーバーヘッドがどのように変化するのかを調べる。今度は、129.compressのすべての関数に対して、間歇的プロファイリングを用いたEPPを適用したコードとオリジナルのEPPを適用したコード用い、上記の手順によって求められた範囲内でUTやRIを変化させ、ホットパス検出精度とオーバーヘッドの評価を行なう。但し、関数の中にパスが一種類しかな

い場合は、プロファイリングを行わなくても実行されるパスは自明なため、その関数にはプロファイリングを行なわないものとする。

評価の基準として、検出率と誤検出率を用い、それらは次式で定義する。

検出数 ＝ EPPと間歇的プロファイリングの両方において検出されたホットパス数

誤検出数 ＝ 間歇的プロファイリングで検出されたがEPPでは検出されなかったホットパス数

$$\text{検出率} = \frac{\text{検出数}}{EPP\text{で検出したホットパス数}}$$

$$\text{誤検出率} = \frac{\text{誤検出数}}{EPP\text{で検出したホットパス数}}$$

検出率が高く、誤検出率が低いほど精度の高いホットパス検出ができていると言える。

プログラム全体の中のホットパスの検出や、各関数毎のホットパスの検出が可能であるか調べるため、これらの定義の基にホットパスの基準を、実行頻度が全体の1％、実行頻度上位5つ、各関数の実行頻度1位、としたときのホットパス検出精度について評価を行なう。

またオーバーヘッドは次のように定義する。

$$overhead = \frac{Profile - Original}{Original} \times 100(\%)$$

Original:オリジナルコードの実行サイクル数

Profile :プロファイルコードを挿入したコードの実行サイクル数

3.3 評価結果

3.3.1 UTとRIの範囲

129.compressの関数compressに限定して、EPPや間歇的プロファイリングを用いたEPPを適用し、実行した結果を表2に示す。

表2 関数compressにおけるプロファイル結果

EC(ORI)	5871354(clock)
EC(EPP)	6624279(clock)
EC(IP1)	7320048(clock)
EC(IP2)	7325550(clock)
T(RH)	7(回)
T(PP)	254718(回)

となった。以上の結果から、

$$a = \frac{6624279 - 5871354}{254718} = 2.96(clock/回)$$

$$b = \frac{7320048 - 6624279}{254718} = 2.73(clock/回)$$

$$c = \frac{7325550 - 7320048}{7} = 786(clock/回)$$

となり、オーバーヘッドは、

$$overhead(IP) = \frac{(2.96 + 2.73) \times UT + 786}{RI} \times 100(\%)$$

その内rehookによるオーバーヘッドは、

$$overhead(rehook) = \frac{786}{RI} \times 100(\%)$$

であるので、これを1％以下にする場合、

$$1 \geqq \frac{78600}{RI}$$

$$RI \geqq 78600 \fallingdotseq 100000(clock)$$

とすれば良い。また、EPPによるオーバーヘッドは、

$$overhead(EPP) = \frac{6624279 - 5871354}{5871354} \times 100(\%) = 12.824(\%)$$

となるので、

$$1.282 = \frac{569 \times UT + 78600}{RI}$$

$$UT \leqq \frac{RI}{444} - 138 \fallingdotseq \frac{RI}{1000}(回)$$

と定義できる。

3.3.2 ホットパス検出精度

求めたUTとRIの範囲より、UTとRIを以下の値とし、間歇的プロファイリングを行った。

RI=100000, UT=50, 100

RI=500000, UT=50, 100, 500

RI=1000000, UT=50, 100, 500, 1000

RI=5000000, UT=50, 100, 500, 1000, 5000

RI=10000000, UT=50, 100, 500, 1000, 5000, 10000

単位は、RIがclock、UTが回数とする。

(1) ホットパスを実行頻度1％以上とした場合

始めに、ホットパスの定義を実行頻度 1％以上としたときの間歇的プロファイリングによる検出率と誤検出率をそれぞれ表3, 表4に示す。

表3 ホットパスの定義を1％としたときの検出率

RI＼UT	50	100	500	1000	5000	10000
100000	56%	56%	-	-	-	-
500000	50%	56%	63%	-	-	-
1000000	56%	56%	81%	81%	-	-
5000000	56%	56%	81%	81%	88%	-
10000000	50%	56%	81%	81%	100%	94%

表4 ホットパスの定義を1％としたときの誤検出率

RI＼UT	50	100	500	1000	5000	10000
100000	44%	44%	-	-	-	-
500000	31%	25%	31%	-	-	-
1000000	31%	25%	25%	25%	-	-
5000000	38%	31%	13%	25%	25%	-
10000000	56%	44%	31%	25%	19%	19%

この表から、UTが小さいときは検出率は50％程度だが、UTを大きくするにつれて検出率が高くなる傾向があり、UTを5000以上としたとき、9割近くのホットパスが検出できていることが分かる。129.compressでは、主要な関数は一度呼び出すと長時間実行するといった傾向があり、今回用いた間歇的プロファイリ

ングのアルゴリズムの場合、rehookを行ってからプロファイルを取り始めるまでの遅延が大きい。そのため、ある程度まとめてプロファイルを取ることにより、rehookされているのにプロファイルを取らない時間を減らし、その結果パスの挙動の変化が捉えやすくなったと考えられる。

誤検出率に関しては、完全になくすことはできないが、こちらもUTやRIを大きくすることによって減少させることができる。これらの誤検出されるパスの大半は、実行頻度が0.9％程度とホットパスの基準とほぼ同じパスであり、それらを検出してもそれほど問題はないと言える。

(2) ホットパスを実行頻度の上位5個とした場合

次にホットパスの定義を実行頻度の上位5つとしたときの検出率を表5に示す。なおこの場合は、

誤検出率 = 100 − 検出率(%)

となるため、誤検出率は省略する。

表5 ホットパスの定義を実行頻度上位5つとしたときの検出率

RI＼UT	50	100	500	1000	5000	10000
100000	60%	60%	-	-	-	-
500000	60%	60%	80%	-	-	-
1000000	60%	60%	60%	60%	-	-
5000000	60%	60%	60%	60%	100%	-
10000000	60%	60%	60%	60%	100%	100%

この表からは、UTとRIが大きいときに検出率が100％、誤検出率が0％となっており、ほぼ完璧なホットパス検出ができていることが読み取れる。また、検出率に関しては表3と同じ傾向が見られる。このことから、ホットパスの条件を変更してもホットパスの検出結果が大きく変化することはないと言える。

しかし、検出率100％の場合でもパスの実行頻度の順番は変化しており、その例は表6に示す。

表6 間歇的プロファイリングとEPPの比較

パス番号	346	1219	1566	1568	1582
EPP	8.93%	8.52%	6.97%	16.40%	8.53%
IP	7.74%	7.32%	14.24%	14.27%	11.78%

この表では、IPがUT=10000, RI=10000000の間歇的プロファイリングを行なったときのパスの実行頻度を表しており、EPPにおける実行頻度と比較を行なっている。また、UT,RIがこれ以外の場合も同様の結果となっている。これは、間歇的プロファイリングでは関数別にunhookを行なうため、実行頻度が高い関数ほどプロファイリングを行なう割合が少なくなることが原因と考えられる。しかしこれらのパスは、いずれも実行頻度が高いパスなため、実行頻度の順番の変化するが、ホットパスの検出は正しく出来ていると言える。

(3) ホットパスを各関数の実行頻度1位とした場合

最後にホットパスの定義を各関数の最も実行頻度が高いパスとしたときの検出率を図7に示す。なお129.compressには実行されない関数も存在するため、ホットパスを検出できた関数は8個となった。

表7 各関数の実行頻度1位をホットパスとしたときの検出率

RI＼UT	50	100	500	1000	5000	10000
100000	75%	75%	-	-	-	-
500000	63%	75%	75%	-	-	-
1000000	75%	75%	75%	75%	-	-
5000000	75%	75%	75%	75%	100%	-
10000000	75%	75%	75%	75%	100%	100%

この場合も、誤検出率は次式の通りとなるので省略する。

誤検出率 = 100 − 検出率(%)

この場合も、UTとRIを大きくすると検出率が100％、誤検出率が0％となっており、完璧なホットパス検出が行なわれていることが分かる。従って、各関数のホットパスを検出することは可能と言える。

3.3.3 オーバーヘッド

129.compressのすべての関数に対し間歇的プロファイリングを行なったときのオーバーヘッドを表8に示す。

表8 プロファイリングによるオーバーヘッド

EPP	40.26%					
RI＼UT	50	100	500	1000	5000	10000
100000	6.22%	7.49%	-	-	-	-
500000	5.27%	5.78%	9.69%	-	-	-
1000000	4.91%	5.25%	7.72%	10.92%	-	-
5000000	4.44%	4.50%	5.10%	5.85%	11.59%	-
10000000	4.36%	4.46%	4.77%	5.20%	8.69%	12.93%

この表を見ると、オーバーヘッドをEPPのオーバーヘッドを1／10以下とするのを目標としていたのに対し、実際にはそれよりもかなり大きくなっている。また、UT=50の部分に注目すると、RIを大きくしてもオーバーヘッドが4％台から下がらないことも分かる。これは、関数compressでは関数が呼び出す回数が25回と少ないのに対し、関数putbyteでは呼び出す回数が約25万回となっている。そのため、呼び出し回数の増加に伴って、関数を呼び出した直後の関数を切り替える部分において発生するオーバーヘッドが増大したことが原因であると考えられる。

また、間歇的プロファイリングでは、EPPと比べてコードサイズが大きく、その結果キャッシュのアクセス状況が悪化した可能性がある。そこで、プロファイルリングを行う関数を実行した割合が低いにも関わらず、オーバーヘッドが低くならなかった、UT=50, RI=10000000としたときの間歇的プロファイリングと、EPPのキャッシュのアクセス状況を調べた。その結果を表9に示す。

表9は、EPPと間歇的プロファイリング(=IP)におけるアクセス数、キャッシュミスした回数、キャッシュのミス率を表している。この表から、間歇的プロファ

表9　間歇的プロファイリングによるキャッシュアクセスの変化

	L1(命令)	L1(データ)	L2
EPP(accesses)	67088913	18267305	1086688
EPP(miss)	19485	622690	142739
EPP(miss_rate)	0.03%	3.41%	13.14%
IP(accesses)	46846505	13935035	1080264
IP(miss)	21193	616772	145024
IP(miss_rate)	0.05%	4.43%	13.42%

イリングではEPPと比べてL1,L2共にキャッシュのアクセス回数が減っているのに対し、キャッシュミスした回数はL1命令やL2において逆に増え、L1データの場合もほとんど減っていない。その結果、キャッシュのミス率が増加し実行サイクル数が増加したと考えられる。

しかしながら、当初の目標よりは大きいが、間歇的プロファイリングを適用したEPPのオーバーヘッドは、RIとUTの値に関わらずオリジナルのEPPの1／3以下に抑えられている。そのため、RIとUTの範囲を示す式は、少なくとも間歇的プロファイリングを適用することによって、オーバーヘッドがEPPよりも増加するのを防ぐ目安としては有効であると言える。

3.4 考　　察

以上の結果から、間歇的プロファイリングにおいて、RI=10000000,UT=5000程度としたとき、プログラム全体の中でのホットパスの検出や関数単位でのホットパスの検出において、EPPと同程度の検出精度を保ちつつ、オーバーヘッドをEPPの1／3以下に削減する可能であることが分かる。

そこで、RIとUTの範囲を示す式に従いRI,UTをある程度大きくすることで、プロファイル精度を保ちつつオーバーヘッドを減らすことが可能になると考えられる。

4. おわりに

本稿では、オーバーヘッド削減を目的とした“間歇的プロファイリング”を提案し、それをEPPに対して適用して、SPECint95の129.compressを用いて評価を行なった。

始めに関数compressにのみ間歇的プロファイリングを適用し、プロファイリングのオーバーヘッドを明確化することによって、間歇的プロファイリングを適用することで逆にオーバーヘッドが大きくならないための、二つのパラメータの範囲を示した。

そして、求めたパラメータの範囲に従ってプロファイリングを行ない、その結果から間歇的プロファイリングによってオーバーヘッドを削減しつつ、ホットパスの検出を行なうことが可能であることを示した。

ただし、今回の結果はあくまで対象を129.compressに限定しているため、他のアプリケーションに対してもプロファイリングを行ない、今回得られた結論が正しいのかを確かめる必要がある。その結果、UT,RIをどの様に設定しても良好なホットパス検出ができないことが判明した場合、rehookされたときにすぐにプロファイリングをとるアルゴリズムへと変更する必要がある。

今後の予定として、本手法は、実行時のホットパス検出を目的としているため、パスがどの程度実行された段階でホットパスとして検出するかを決める閾値を検討が挙げられる。また、本手法を用いてプログラムにプロファイルコードを挿入する作業を、現在手作業にて行なっているため、コード挿入に非常に時間がかかる。そのため、コード挿入を自動的にやってくれるプログラムの作成が求められる。

謝辞　本研究は、一部日本学術振興会科学研究費補助金（基盤研究(B)14380135、同(C)14580362、若手研究14780186）の援助による。

参 考 文 献

1) Thomas Ball, James R. Larus. “Efficient Path Profiling,” Proc. 29th Ann. IEEE/ACM Int'l Symp. Mycroarchitecture, IEEE CS Press, Los Alamitos, Calif., pp. 46-57, 1996.
2) Omri Traub, Stuart Schechter, and Michael D. Smith, “Ephemeral Instrumentation for Lightweight Program Profiling,” http://www.eecs.harvard.edu/hube/publications/muck.pdf, June 2000.
3) 加藤 文彦、野中 雄一、青木 政人、大津 金光、横田 隆史、馬場 敬信、“一時的プロファイルによるホットパス検出法の精度とオーバーヘッドの検討,” 情報処理学会第65回全国大会論文集, 講演番号5Z-6, pp.1-129〜1-130, March 2003.
4) 青木 政人、大津 金光、横田 隆史、馬場 敬信、“サンプリング情報に基づいた実行時最適化手法とその評価,” 情報処理学会ハイパフォーマンスコンピューティング研究会(HOKKE-2003), March 2003.
5) 野中 雄一、大津 金光、横田 隆史、馬場 敬信、“パスプロファイルによるホットパス検出とオーバーヘッドの評価,” 情報処理学会研究報告, Vol.2002, No.81, pp.103-108, (2002-ARC-149), August 2002.
6) J. Hnang. The SImulator for Multi-threaded Computer Architecture(SIMCA), Release 1.2. http://www-mount.ee.umn.edu/~lilja/SIMCA/
7) J. Y. Tsai, J. Hang, and et al, “The Superthreaded Processor Architecture,” IEEE Transactions on Computers, Vol.48, No.9, pp.881-902, 1999.
8) Evelyn Duesterwald, Vasanth Bala. “Software Profiling for Hot Path Prediction: Less is More,” 9th ASPLOS , pp.202-211, 2000.