Vol.2015-OS-135 No.12
Vol.2015-EMB-39 No.12
2015/11/24

実行パスの解析による性能障害の原因関数の推定

深井 賢[1]　中村 啓太郎[1]　吉村 剛[1]　河野 健二[1]

概要：近年，あらゆるコンピュータシステムに対して，高い信頼性が求められるようになっている．システムの信頼性が損なわれる要因のひとつに性能障害がある．性能障害とは，システムの応答時間が遅くなり，処理効率が悪くなることを指す．Linux などの大規模ソフトウェアにおける性能障害は，その原因を特定するのが容易ではない．本稿では，多数のスレッドが並行動作する状況において，特定のスレッドに起きた性能障害の原因関数を推定する方法を提案する．スレッドの実行パスを比較することにより遅延の発生している関数を推定する．日立から報告された実際の障害事例に本提案を適用したところ，性能障害時に実行パス内で特に処理が遅くなっている関数を特定し，障害の原因関数として推定することができた．

1. はじめに

一般のユーザに広く普及しているような情報サービスが増加している中，いわゆるミッション・クリティカルなシステムではないコンピュータシステムに対しても，高い信頼性が求められるようになっている．高い信頼性を脅かす要因の一つとして性能障害が知られている．

性能障害とは，システムの応答時間が遅くなることで，処理効率が悪くなる障害を指す．一般ユーザが広く使用しているシステム上で発生する性能障害は使用ユーザに対して深刻な影響を与える．2012 年 10 月には，Amazon EC2 で 7 時間程性能障害が発生し，その間 Reddit などの有名サイトがダウンし，多くのサービスがサービス停止に追い込まれているという事例も存在する [1].

こうした性能障害は再現性が低く，原因特定が難しいことが多い．これは性能低下の原因となる要素が多岐にわたる，クラッシュなどと異なり障害発生の瞬間が把握しづらいためである．性能低下の原因は，ワークロード，実行環境の組み合わせ（ディスク構成，CPU，ソフトウェアバージョンなど），パラメータ設定，など様々な要因が相互に依存する [2][3][4][5]．また，性能障害は，徐々に性能低下が増大する傾向にあるため，その検知が遅れがちになり，障害発生の根本原因 (root cause) の把握が難しい．

さらに，実際の障害環境では，単一のソフトウェアが単一のスレッドのみを使用して動作しているということは少なく，MySQL や Apache などをはじめとする複数のソフトウェアが多数のスレッド上で並行して動作するような環境が多く存在する．そのような多数のスレッドが並行動作するような大規模サーバ上では，一部のスレッドに大きな遅延が生じるケースが存在する [3]．多数のスレッドが複雑に噛み合って動作しているため，一部のスレッドの遅延は結果としてソフトウェアやアプリケーション全体の遅延につながっている．このような環境下では，性能障害の原因は必ずしも遅延が発生したスレッドにあるとは限らず，開発者が障害の原因を発見することはなおさら難しい．

本稿では，実行パスを比較することによって性能障害の原因関数を推定する手法を提案する．特に，大規模サーバ環境を始めとする多数のスレッドが並行動作している状況下での性能障害を対象とし，別スレッドの通常実行時のパスと比較して実行パス内で処理時間が遅い関数について焦点を当てる．性能障害が発生している場合，実行パス内の一部の関数の処理時間が著しく遅延することによって発生している可能性が高い．そこで，遅延が生じていないスレッドの実行パスと，遅延が生じているスレッドの実行パスを比較することによって，特定スレッドで発生した性能障害の原因関数を推定する．実行パスの比較対象は，共通部分で処理時間が極端にかかっている関数や，遅延時のみ実行されている関数である．スレッドの実行パスの通常時，遅延時の判別は統計学的手法の一つである管理図を用いた．

提案手法の有効性を示すため，日立から報告されてた障害事例を用いて実験を行った．障害事例再現時に記録されたログに対し提案手法を適用した．通常時の実行パスと遅延時の実行パスを比較した結果，本事例では実行パスの差分は大きな差分がないことがわかった．障害事例の原因関数に関しては，実行パスの処理時間の 90 % を占めるスケジューリングに関する関数が原因関数と推定された．推定

[1] 慶應義塾大学大学院
Keio University Graduate School

された関数は通常時の 約 90 倍の処理時間が遅延時にかかっていることがわかった．

論文の構成について説明する．2 章では，性能障害の原因関数の推定の難しさについて説明する．3 章では，実行パスの比較による原因関数の推定について説明する．4 章で実際の障害事例を用いた評価実験を行い，5 章で関連研究について紹介し，6 章で結論を述べる．

2. 原因関数の推定における問題点

性能障害が発生した時，開発者やユーザは出力されているエラーログなどを用いて障害解析を行う．しかし，Linux のログ出力コマンドである dmesg コマンドをはじめとした出力ログは障害の解析に十分な情報が得られない場合がある．その場合の解決策として，Linux に標準で搭載されているトレースツールである ftrace [6] による実行パスの解析がある．ftrace では, 関数内にトレースポイントを挿入することで，関数名，CPU 番号，呼び出し終了時までの処理時間含めた実行パスを取得できる．図 1 は実際に取得できる実行パスの例である．

実行パスを用いた原因関数の推定の問題点として，原因となる問題に様々なケースが想定されることと，正常時の実行パスが多岐にわたることが挙げられる．

CPU 番号　処理時間　　実行された関数名

```
2)                |  sys_read() {
2)     0.475 us   |    fget_light();
2)                |    vfs_read() {
2)                |      rw_verify_area() {
2)                |        security_file_permission() {
2)                |          selinux_file_permission() {
2)                |            file_has_perm() {
2)                |              inode_has_perm() {
2)                |                avc_has_perm() {
2)     0.378 us   |                  avc_has_perm_noaudit();
2)     0.282 us   |                  avc_audit();
2)     1.525 us   |                }
                  :
2)   + 11.242 us  |  }
```

図 1　ftrace で取得できる実行パス

2.1 性能障害の原因となる関数のパターン

性能障害の原因となる関数のケースの一つとして，正常時の実行パスと遅延時の実行パスの異なる関数が原因となるケースがある．これは，図 2 は，実行パスの異なる部分が原因となるケースのサンプルである．正常時には実行されない関数 (灰色の部分) が異常時には非常に遅れている (赤い部分) ため，実行パス全体の処理時間が遅れている場合である．本ケースでは，正常時の実行パスと遅延時の実行パスを比較することによって，差分にあたる関数の中に遅れている原因があると判定できる．

また別のケースとして，共通の実行パス中に極端に処理時間がかかっている関数が存在するパターンがある．図 3 は，実行パスの共通している部分の特定の関数が原因となるケースのサンプルである．正常時と異常時での実行パスはほぼ共通しているが，共通部分のある特定の関数が極端に遅れているため，実行パス全体の処理時間が遅れいてる場合である．本ケースでは，正常時の実行パスと遅延時の実行パスの共通部分を比較し，処理時間が極端に遅れている関数が原因であると判定できる．

このように，性能障害の原因となる関数のケースは複数存在し，実際の障害事例がどのケースと合致するかはわからないため，一つのケースのみを適用して原因関数を推定することは難しく，全てのケースを想定して実行パスを解析することも非常に手間がかかる．

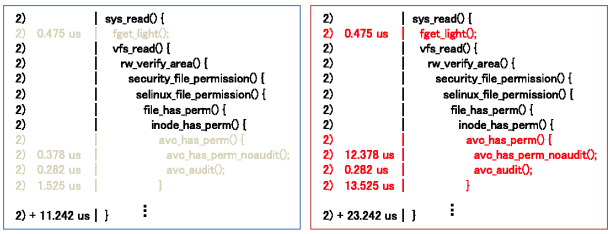

図 2　実行パスの差分が原因となる場合

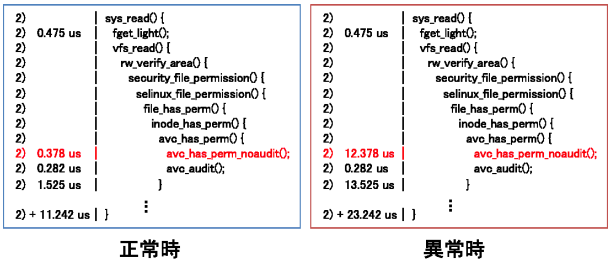

図 3　共通パス内の遅延が原因となる場合

2.2 正常時の実行パスの多様性

正常時の実行パス同士でも，パスが異なる場合がある．ftrace を用いて正常な実行パスを取得し，同一パス同士でクラスタリングした結果，406 個の実行パスが 67 種類のパターンに分類された．図 4 は，クラスタリングした結果のうち，同一パスが多い上位 20 パターンの実行パスを示している．図を見るとわかるように，1 つのパスパターンにほとんどのパスが統一されるのではなく，複数のパスパターンに分散してしまっている．どのパターンも正常時の実行パスであるため，分類されたパターン群同士の差分は直接遅延に影響しない部分である．従って，遅延時の実行パスと比較する通常実行パスパターンが非常に多くなってしまい，さらにパターン毎に原因関数の推定結果が異なってしまう．

Vol.2015-OS-135 No.12
Vol.2015-EMB-39 No.12
2015/11/24

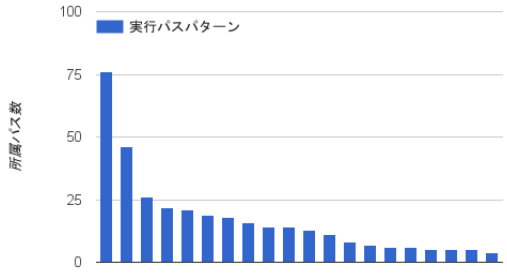

図 4 同一パスに分類される所属パス数

3. 実行パスの比較による原因関数の推定

3.1 概要

ftrace [6] によって取得した実行パスを比較することにより，性能障害の原因関数を推定する．システムの通常稼動時のパフォーマンスを基に性能障害時の実行パス群を正常時の実行パスと異常時の実行パスに分類する．さらに，正常時の実行パスをクラスタリングし，正常時の実行パスパターン群に分類する．その後，最長共通部分列を基に類似度を算出し，類似度が最も高いパスパターンと比較する．パスが完全に一致するパスパターンが見つかった場合，異常時のパスと正常時のパスパターンの各関数毎の処理時間を比較し，著しく遅くなっている関数を原因関数と推定する．完全に一致するパスパターンが見つからなかった場合，最長共通部分列部分のパスの処理時間の比較を行い，差分が関数の遅延に大きく関わっている場合は共通部分のパスの関数毎の処理時間を比較する．差分による影響が小さい場合は異常時のパスにおいて共通部分のパス以外に実行されている関数を原因関数と推定する．

3.2 正常時，異常時の判別

性能障害時の実行パス群を正常時の実行パスと異常時の実行パスに分類するにあたり，基準値の算出方法として管理図を用いる．管理図とは，統計学的に確立された異常検出手法手法の一つであり，通常は工場製品の品質管理に用いられる手法である [7]．過去の管理対象の値 (本稿では通常稼動時に取得した実行パス全体の処理時間) の分布と，現在の管理対象の値の分布を統計的に比較し，二つの分布が同じか異なるかを判定する．

管理図は過去の品質と現在の品質のズレを統計的に判定するものであるため，性能障害時の実行パスにおいて異常時の実行パスの判別が期待できる．管理図を用いて通常稼動時の実行パスの処理時間と性能障害時の実行パスの処理時間を比較することで，処理時間に現れる性能以上の兆候を経験則に頼らずシステマティックに検出できる．本稿では，といった理由のため，メディアン管理図の管理限界線を適用する．管理図の作成手順を説明する．初めに基準線を決定するために，100 個の特性値 X を測定し，測定した順に 5 個ずつ 20 の群に分類する．特性値 X は 100 個測定することが慣例となっている．また，群の大きさは管理図の数学的背景により 4 以上を選択する必要があり，中央値 Me の計算を容易にするため，通常は奇数である 5 を選択する．そして，各群について中央値 Me と 範囲 R (最大値と最小値の差) を計算する．この 20 この中央値の平均値 $\overline{Me}$ が中心線となる．さらに 20 個の範囲 R の平均値 $\overline{R}$ を計算し， 上方管理限界線を求める式

$$UCL = \overline{Me} + A_2 \cdot \overline{R} \tag{1}$$

に代入する．UCL が上方管理限界線（Upper Control Limit）である．ここで A_2 は群の大きさによって一意に定まる値であり，群の大きさが 5 の時は A_2 は 0.69 となる．このように求めた UCL を基準とし，これを超える処理時間のものを異常と判断する．具体的には，システムの通常稼動時の実行パスの処理時間を 100 個取得し，特性値としている．適用して算出された処理時間の値を基準値とし，実行パス群から通常時のパスと異常時のパスに分類する．

3.3 パス同士の類似度の算出

実行パス同士の比較を行うにあたって，類似度という指標を用いた．ここでの類似度は，比較するパスパターン同士の最長共通部分のパスを算出し，共通部分のパス長の異常時の実行パス長に占める割合を指している．

最長共通部分列とは，2 つの系列の共通な要素を取り出してできた共通部分列のうち，最も長いものを指す．今回は，異常な実行パスと，正常な実行パスのそれぞれのパスの流れを系列として最長共通部分列を求めている．図 5 は本提案に適用した最長共通部分列のイメージである．図では，異常時の実行パスが {A, B ,C, B, D, A, B} であり，正常時の実行パスが {B, D, C. B. A} である場合に最長共通部分列アルゴリズムを適用した場合を示している．図の異常時の実行パスと正常時の実行パスの部分列を取得すると，{B, D, A} と {B, D, B}, {B, C, B, A} といった部分列が取得できる．それぞれの部分列の長さは，順に 3, 3, 4 となるため，図における最長共通部分列は部分列長が 4 である {B ,C, B, A} となる．実際に最長共通部分列を実装するにあたっては，解法としてよく知られている動的計画法を用いた [8]．詳細は第 3.3.1 節にて説明をする．

求めた最長共通部分列長の異常時の実行パスにおける割合を算出して類似度とした．図のイメージの場合，類似度は約 57 % となる．これをある異常時の実行パスに対して，全ての正常時の実行パスパターンの類似度を算出し，最も類似度の高い正常時の実行パスパターンを用いて実行パス

の比較を行う．

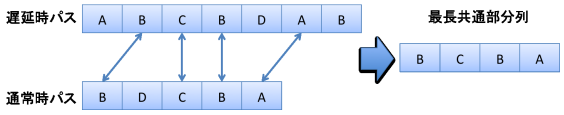

図 5 本提案に適用する最長共通部分列のイメージ

3.3.1 動的計画法を用いた最長共通部分列の解法

本稿では，正常な実行パスと異常な実行パスの類似具合を示す指標として，最長共通部分列で算出した類似度を用いた．この最長共通部分列は動的計画法を用いて解くことができる [8]．以下に解法を示す．また，解法における系列とは，正常な実行パスと異常な実行パスを示し，長さとはそれぞれの実行パスの中に含まれる関数の数を示している．

2 つの系列 a, b が存在し，それぞれの長さが n, m として，(m + 1) × (n + 1) のマトリックスである c[m, n] を生成する．このマトリックスに次の定義に合わせて値を当てはめることで共通部分列のマトリックスを生成する．

$$c[i,j] = \begin{cases} if\ i\ =\ 0\ or\ j\ =\ 0\ then\ 0 \\ if\ a[i]\ =\ b[j]\ then\ c[i\ -\ 1,\ j\ -\ 1]\ +\ 1 \\ otherwise\ max(c[i,\ j\ -\ 1],\ c[i\ -\ 1,\ j]) \end{cases}$$

以下に定義の説明をする．1 つ目の行の定義は初期化のための定義である．2 つ目の行の定義は，系列 a の i 番目と，系列 b の j 番目の値 (本稿では関数名) が一致している場合，マトリックス上の c[i - 1, j - 1] までの共通部分列の長さに，今回一致した共通部分が加算されるという定義である．例としては，図 5 の正常時の実行パスの 2 番目である D と異常時の実行パスの 5 番目である D が一致した部分である c[3, 6] の部分を見る．この場合，直前の関数である，正常時の実行パスの 1 番目である B までと，異常時の実行パスの 4 番目である B までの系列において，最長共通部分列の長さは B が一致しているため 1 である．従って，直前の最長共通部分列に今回一致した分を加えるため， c[3, 6] の値は 2 となる．3 つ目の行の定義は，系列 a の i 番目と，系列 b の j 番目の値 (本稿では関数名) が一致しなかった場合，行，列それぞれ直前の共通部分列のうち，より長い方の共通部分列が c[i, j] 地点での最長共通部分列であるという定義である．例としては，図 5 の正常時の実行パスの 5 番目である A と異常時の実行パスの 7 番目である B の部分である c[6, 8] の部分を見る．この場合，今回の比較では一致しなかったため，m, もしくは n の系列を 1 つ下げた時のマトリックスの値，つまりどちらかの系列の 1 つ前の値の場合と最長共通部分列の長さは変わらないはずである．したがってこの場合では c[6, 7] の値である 4 もしくは c[5, 8] の値である 3 のうち長い方である 4 が c[6, 8] の値として入る．

完成したマトリックスの一番右下に当たる値が最長共通部分列の長さであり，これを用いて類似度を算出する．表 1 は，図 5 について動的計画法を用いて算出したマトリックスである．マトリックスの一番右下にある数値は 4 であり，これは確かに図 5 で示した最長共通部分列の長さと一致する．

表 1 動的計画法を用いて算出したマトリックス

	n	A	B	C	B	D	A	B
m	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
B	0	0	1	2	3	3	3	3
A	0	1	1	1	1	1	4	4

3.4 原因関数の推定

異常時の実行パスと正常時の実行パスパターン群との類似度を算出して選択した組み合わせを用いて実行パスの比較を行う．本稿で述べる提案では，異常時の実行パスと正常時の実行パスパターンの類似度によって比較方法を変える．正確には，類似度が 100 % となり，異常時の実行パスが正常時の実行パスと完全に一致している場合と，そうでない場合である．類似度が 100 % の場合は実行パス全体の関数ごとの処理時間を比較することによって原因関数を推定する．異常時におけるパス内の関数の処理時間と，正常時における同関数の処理時間の統計値を比較し，上回っている関数を性能障害の原因関数と推定する．原因関数を推定するイメージは，第 2.1 節の図 3 と同様である．

それ以外の場合には，最長共通部分列内のパスの処理時間の合計が実行パス全体の処理時間に占める割合を求める．割合が大きい場合には共通部分列内の実行パスの関数ごとの処理時間の比較を行い，小さい場合には異常時の実行パスと共通部分列内の関数列の差分を求めることによって原因関数を推定する．差分を求める場合の原因関数を推定するイメージは，第 2.1 節の図 2 と同様である．

4. 実験

本章では，提案手法が実際に性能障害の解析に有効であるかの検証を行った．具体的には，1) 通常時同士の実行パスの類似具合，通常時と遅延時の実行パスの類似具合を調査する，2) 共通部分の実行パスの比較によって　実際に遅延の原因となる関数を推定できるかを検証した．検証を行う環境は，Red Hat Linux Enterprise Linux 5.4，カーネルのバージョンは Linux 2.6.34.7, メモリは 16GB 1333 MHz DDR3 でコア数が 4，プロセッサは Intel®Xeon®CPU E3-1270 @ 3.40 GHz のマシンを使用した．また，ディスクは ext3 にフォーマットした 80GB のパーティションを用いた．

4.1 障害事例

本実験では，日立の商用インフラ系サーバ内で発生した実際の性能障害の事例を対象として検証を実施した．本項では事例の説明を行っていく．

本事例は，Red Hat Enterprise Linux 5.4 で発生したパフォーマンス障害である．事例の現象としては，1 つのディスクに対して多数のプロセスが同時に read 命令を行い続けると，一部の read に 7 秒の遅延が生じるという現象である．この障害の発生条件として，次の 2 つの条件がある 1 つ目は特定のブロックデバイスに多数のプロセスが同時に同期 I/O を行い続けるという点である．2 つ目の条件はブロックデバイスの I/O スケジューラとして CFQ スケジューラを使用するというものである．

CFQ スケジューラとは Linux に標準で搭載されている I/O スケジューラである．CFQ スケジューラでは，内部に多数のリクエストキューを持ち，ディスクにアクセスするプロセス 1 つにひとつのキューが割り当てられる．また，プロセス毎に I/O の優先度をつける事ができ，各プロセスに対応するキューにはプロセスの I/O 優先度に応じた持ち時間が割り当てられている．各リクエストキューはラウンドロビン方式で順に選択され，選択されたリクエストキュー内の I/O リクエストは持ち時間が過ぎるまで次々とデバイスのキューへと送られる．優先度に応じて持ち時間を定めることによって，優先度の高いプロセスの I/O が優先的にスケジュールされるようになっている．さらに，CFQ スケジューラの設定値として，同時に発行できる I/O 要求数を設定できる．障害事例の発生時の状況では，同時発行可能な I/O 要求数は 8 であった．

4.2 実験方法

性能障害を意図的に発生させ，障害環境を構築し，同時に ftrace を動かす事で，実行パスを取得する．こうして得た実行パスを提案手法により解析する．通常時の実行パスと遅延時の実行パスの類似具合について確認する．その後，提案手法により，遅延の原因関数を推定できる事を確認する．

障害再現時の実行パスを実行パス全体の処理時間について基準値を基に通常実行パスと遅延実行パスに分類した．分類した通常時の実行パスを，さらにパスのパターンが完全に一致したパス同士で群にまとめた．まとめた通常時の実行パス群同士での類似度のマトリックスを生成し，そのうち各パス群ごとに最大の類似度を調べ，類似度の分布表を作成した．通常時の実行パス群と遅延時の実行パスについても同様の手順を用いて類似度のマトリックスを生成し，類似度の分布表を作成した．基準値は，通常時の実行パスに管理図を適用して算出した値を用いた．

次に，遅延時の実行パスから最も類似度が高い通常時の実行パス群の代表パスを 1 つ選択し，実行パス内から共通部分の実行パスの関数毎の処理時間の比較をした．さらに，特に遅延が顕著な関数に関しては遅延時の実行パスに対して，選択した通常時の実行パス群内のパス全てと比較した．

通常実行時と障害再現時の read システムコールの実行時間の累積分布グラフを図 6 に示す．グラフは縦軸が累積分布，横軸がシステムコールの実行時間である．今回取得した通常実行パスの処理時間を基に管理図を用いて算出した遅延実行パスの判定基準は 約 10000 μ s であった．この基準値は，通常実行パスでは 90% 以上を占めているが，障害再現時の実行パスでは，約 20% であり，遅延実行パスの判定基準として十分であることが確認できた．

また，本実験にて取得した性能障害時の *sys_read()* の実行パスは全部で 2675 個であり，今回の基準値を適用した結果，409 個の正常な実行パスと，2266 個の異常な実行パスに分類された．さらに，これらを完全に一致する実行パス毎に一つのパターンに分類したところ，正常な実行パスは 67 種類のパスパターンに，異常な実行パスは 100 種類のパスパターンに分類された．これらのパスパターンを用いて実験を行った．

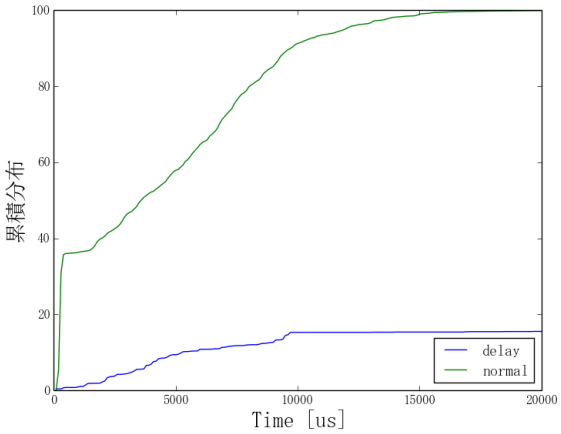

図 6 障害再現時と通常時の処理時間の累積分布グラフ

4.3 実験結果

4.3.1 実行パスの類似具合

表 2 は，通常時の実行パス群同士の類似度分布とその類似度分布に含まれる実行パス群の分布数を示している．類似度が 90 % を超える通常実行パス群の組み合わせは通常時の実行パス群全体の 90 % 以上であり，類似度の最低値も 60 % を下回らないことから，異なる通常時の実行パス群同士の差異は非常に少ないことがわかった．

また表 3 は，異常時のパスに対し，通常時のパス群の類似度を比較した結果である．算出した類似度分布に対し，該当する異常時のパスと通常時のパス群の分布数を示している．こちらも分布数を見ていくと，類似度が 90 % を超

える異常時のパスと通常時のパス群の組み合わせの分布数が，組み合わせ全体の 90 % 異常であり，類似度の最低値も 60 % 以上であり，それ以下の類似度を持つ組み合わせが存在しなかった．

以上のことから，実行パスの差分は，通常時と異常時という区分の方法ではほとんど差がなく，等しくある程度の分岐が存在することがわかった．

表 2 通常パス群同士

類似度	分布数
100-90	61
90-80	4
80-70	1
70-60	1
合計	67

表 3 通常パスと異常パス

類似度	分布数
100-90	92
90-80	6
80-70	0
70-60	2
60-50	0
50-40	0
40-30	0
30-20	0
20-10	0
10-0	0
合計	100

4.3.2 原因関数の推定

図 7 は，関数の処理時間が 3 倍以上かかっている関数群を示したグラフである．縦軸は通常時の処理時間に対する遅延時の処理時間の割合，横軸は関数名を示している．実験では，100 種類の遅延時の実行パス全てに対して，67 種類の通常時の実行パス群から最も類似度の高い実行パスを選択して原因関数を推定するが，結果は非常に似ているため，本稿では，1 つのパターンを抜粋して示している．

図 7 を見ると，他の関数が 5 倍前後の処理時間であるにもかかわらず，*schedule()* が 90 倍以上処理時間がかかっていることがわかる．したがって，本稿で取り扱った事例において，障害事例の原因となっている関数は *schedule()* と推定される．

また，図 8 は，別の実行パスのパターンにおいて，遅延時の実行パスと，類似度が最も高い通常実行パス群全てにおいて，*schedule()* の処理時間を示した図である．一番左の結果が遅延時の実行パスにおける *schedule()* の処理時間，残りの結果が通常パス群内のパス毎の scheudle の処理時間を示している．図から明らかに *schedule()* が通常実行パスと比較して遅れていることがわかった．

さらに，*schedule()* のシステムコール全体の処理時間を調査したところ，常に 90 % 以上を *schedule()* が占めていることがわかった．

schedule() は，他スレッドの実行待ちを示す関数であるため，単一のスレッドのみではこれ以上の分析を行うことは難しい．したがって，障害発生時の他スレッドの実行パスも同時に比較することを今後の課題とする．

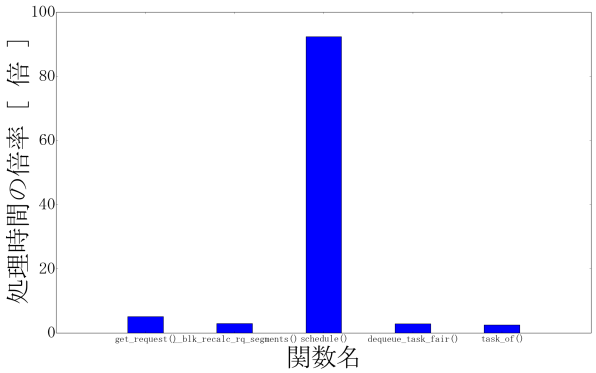

図 7 関数の処理時間の倍率

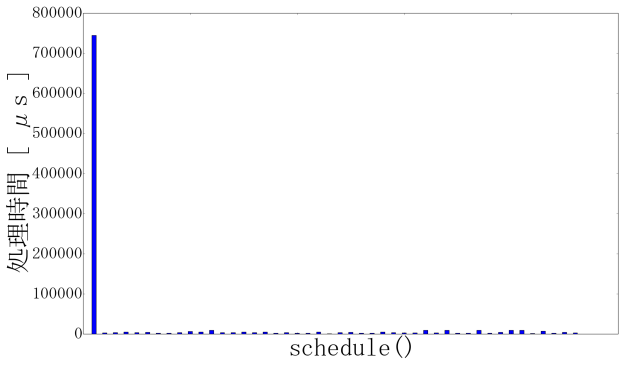

図 8 遅延時実行パスと通常時実行パス群での処理時間比較

5. 関連研究

Attriyan 達は，重み付けを行う事によって性能障害の原因を推測する X-ray というツールを開発している [9]．X-ray はエンドユーザの設定ミスを原因とする性能障害を対象としている．

Jin らは，MySQL などのアプリケーションのパッチを調査し規則性を見つけ，未知の性能障害の検知を行っている [3]．この研究では，パッチが非常に充実しているアプリケーションソフトウェアのみを対象としている．

Nistor らは，性能障害の修正時に別の障害が入らないように修正を行う CARAMEL というツールを開発している [10]．CARAMEL は，ループを抜ける部分の条件分岐のミスによる性能障害を対象としている．

David らは，コールグラフをグループに分割して，グループごとに重みを集約することで，性能障害の原因となるグループを見つける解析手法である Subsuming Method Analysis [11] を実際の商用ケースの障害解析に適用している [12]．この研究では，ソフトウェアの性能の最適化が可能な部分を検知する研究であり，性能障害の原因推定を目的としていない．

6. 終わりに

本論文では，実行パスの解析による性能障害の原因関数の推定手法を提案した．提案手法は，正常実行時の実行パスと性能障害が発生している実行パスを比較することによって，膨大な量の実行パスから，性能障害の原因関数を推定するという手法である．

本手法を提案するにあたって，日立の実用環境で発生している実際の障害事例に対して調査を行った．その結果，正常実行時と障害発生時という違いによっての実行パスの差異はほとんどないことがわかった．また，共通部分の実行パスを比較したところ，本事例ではスケジューリングに関する関数が非常に遅れていることがわかった．さらに，その関数は実行パス全体の処理時間の約 90 %を占めているため，スケジューリングに関する関数が遅延の原因であると推定できた．このことから，本手法は性能障害の原因関数の推定に使えそうであることがわかった．

今後の問題としては，スケジューリングに関する関数が原因と推定されたため，単一のスレッドの調査では十分に原因が絞り込めないということがわかった．そのため、実行されているタイミングで別スレッドで動作しているトレースデータを並行して調査することを今後の課題とする．

参考文献

[1] Networkworld. http://www.networkworld.com/news/2012/102212-amazon-ebs-263592.html.
[2] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshimi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM symposium on operating systems principles (SOSP '11)*, pages 159–172, October 2011.
[3] Guoliang Jin, Linhai Song, Ziaoming Shi, Joel Scherpelz, and Shan Lu. Understnding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation (PLDI '12)*, pages 77–88, June 2012.
[4] Domenico Cotroneo, Michael Grottke, Roberto Narella, Robert Pietrantuono, and Kishor S. Trivedi. Empirical investigation of fault triggers in open-source software. In *Proceedings of 21st ACM symposium on operating systems principles (SOSP '07)*, pages 205–220, November 2010.
[5] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performace bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pages 237–246, May 2013.
[6] ftrace. https://access.redhat.com/site/documentation/ja-JP/Red_Hat_Enterprise_Linux/6/html/Developer_Guide/ftrace.html.
[7] Shewhart control chart. JIS Z 9021.
[8] R. Rivest T. Cormen, C. Leiserson and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
[9] Mona Attariyan, Michael Chow, and Jason Finn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*, pages 307–320, October 2012.
[10] Cosmin Radoi Adrian Nistor, Po-Chun Chang and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*, pages 902–912, May 2015.
[11] J. Hosking D. Maplesden, E. Tempero and J. C. Grundy. Subsuming methods: Finding new optimisation opportunities in object-oriented software. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*, pages 175–186, January 2015.
[12] E. Tempero J. Hosking D. Maplesden, Karl von Rainbow and J. C. Grundy. Performance analysis using subsuming methods: an industrial case study. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*, pages 149–158, May 2015.