情報爆発に対応するスケーラブルかつ自律的な障害解析

Autonomic, Scalable Fault Localization for the Information Explosion Era

丸山 直也 † 松岡 聡 †,††

Naoya Maruyama Satoshi Matsuoka

† 東京工業大学 †† 国立情報学研究所

Tokyo Institute of Technology National Institute of Informatics

1 はじめに

計算システムが大規模化、複雑化するにつれて、故障の迅速な原因特定が困難になってきている。大規模計算クラスタ、グリッドなどに代表されるような大規模環境では、故障の発生頻度が増加し、発生した場合の原因探索範囲が増大する。また、Linux などの標準コンポーネントの多用により、柔軟で多様なシステム構成が可能になったが、同時に予測不可能な異常な振る舞いが起こる可能性も増加した。従来の主流であった単一ベンダによる垂直統合的な構成では事前に構成を検証可能であったが、現在の主流である Linux クラスタでは種々のコンポーネントをシステム毎に独自に組み合わせて構成するため、事前の網羅的な検証は困難である。

我々は、分散システムを対象にした大規模環境における故障原因の自動絞り込み手法を提案する。本手法では、システムの正常実行時の振る舞いをあらかじめ学習し、モデルとして表現する。実際に故障が発生した場合には、その故障発生時の振る舞いとモデルを比較することで、正常とは異なる異常な振る舞いを検出する。ここで、システムの振る舞いの表現として関数呼び出しの確率を用い、通常呼ばれる関数が呼ばれなかった場合、また通常呼ばれない関数が呼ばれた場合を異常として検知する。例えば、分岐に特定の場合のみ発生するバグがある時は通常とは異なるパスに分岐するため、異なる関数呼び出しが観測される。本手法ではこのような関数呼び出し傾向の通常時と故障時の違いを、呼び出し確率をモデルとして学習することで検知する。

本稿では、提案解析手法の概要と、その有効性についての評価結果を報告する。

2 分散システムのモデル化

本モデル化は、分散システムに対する以下の仮定、観察に基づく。プロセスはメッセージを受信してそれに対応する処理を行う無限ループ（メインループ）で構成される。処理の構成はメッセージのタイプに応じて複数あり、それらが `select` システムコールなどを用いて多重化して実行される。この観察は常に成り立つものではないが、特にクラスタ環境向けの多くのシステムに共通して成立する特徴と期待できる。以上の観察に基づき、我々は以下の手法によりモデルを構築する。

1. 分散システムの構成プロセスの関数トレースを取得
2. トレースを似た振る舞いが期待できるグループへ分割
3. 全プロセスのトレースより関数の出現回数を計算し、各関数の出現確率を推定

2.1 トレースの関連した呼び出しへの分割

分散システムの各プロセスについて取得した関数トレースを、それぞれメッセージを受信しそれに応じた処理をする部分（ハンドラ）を一単位として分割する。また、プロセス初期化部や終了処理部等も別個の一単位とする。同単位を実行ユニットと呼ぶ。関数トレースはプロセス起動時からのすべての関数呼び出しを含むため、意味的に関連の少ない呼び出し同士も同じトレース中に含まれる。実行ユニットに分割することで、関連した関数のみを含んだ部分トレースを構成し、より精度の高いモデルを構築できる。

実行ユニットへの分割は、プログラムの静的解析と関数トレースより自動的に行う。これは、ループの検知、`select` システムコールの呼び出しの検知によるメインループの特定、`recv` システムコールの呼び出しの検知によるハンドラ部の特定からなる。

2.2 実行ユニット毎のモデル構築

各実行ユニットの関数 f について、その呼出し元関数 p が呼ばれた場合に f が呼ばれる条件付き出現確率を計算し、モデルとする。出現確率は、その関数 f の出現回数 c_f と呼び出し元関数 p の出現回数 c_p から、c_s/c_p と推定する。この際コールツリーを構成し、コンテキスト

センシティブな解析を行う。すなわち、モデル中の各関数について、その関数へのコールスタックの最下位関数からのすべての呼び出しを含むコールパス毎に頻度を計算する。

3 モデルを用いた故障解析

故障の原因解析は以下の手順で行う。

1. 故障発生時のトレースを取得
2. 各故障トレースをモデル構築の際と同様に実行ユニットへ分割
3. 被検証実行ユニットを、対応する正例モデルと比較し、モデルとの乖離を表す異常度を計算

異常度は被検証実行ユニットの正例モデルとの乖離を表す指標である。これは以下の条件に該当する場合に高い異常度を持つように定義する。

- 正例モデル中に対応モデルが存在しない
- 正例モデル中に頻繁に出現かつ被検証実行ユニットに非出現
- 正例モデル中に頻繁には出現せずかつ被検証実行ユニットに出現

我々は上記制約を満たすよう各被検証実行ユニットの異常度を 0 以上 1 以下の範囲で以下の通り計算する。まず対応モデルが存在しない場合は最大値である 1 を割り当てる。存在する場合はユニット中の各関数 f について、$\delta(f)$ を以下の通り計算する。

$$\delta(f) = \begin{cases} 1 - p(f) & \text{関数 } f \text{ が呼ばれた場合} \\ p(f) & \text{関数 } f \text{ が呼ばれなかった場合} \end{cases} \quad (1)$$

各関数について $\delta(f)$ を計算し、ユニット全体の異常度をその平均として計算する。

4 評価実験

提案手法の故障解析に対する有効性を評価するために MPICH2 v1.0.5p4 [1] の既知のバグに本手法を適用した。MPICH は MPI の代表的な実装のひとつであり、利用実績、可搬性などの点で優れたライブラリである。しかし、多拠点のクラスタを同時に使用した場合に高い確率でジョブを正常に実行できず、ジョブマネージャがハングアップするバグがあることが報告されている [2]。単一拠点のクラスタのみを利用した場合は発生しない。斉藤らによる人手による解析の結果、同バグの原因は Socket API の `recv` 関数の返り値をチェックせずにメッセージ受信をしたと誤って判断するためであることがわかっている。同バグは既に開発者に報告されバグであると確認されている。

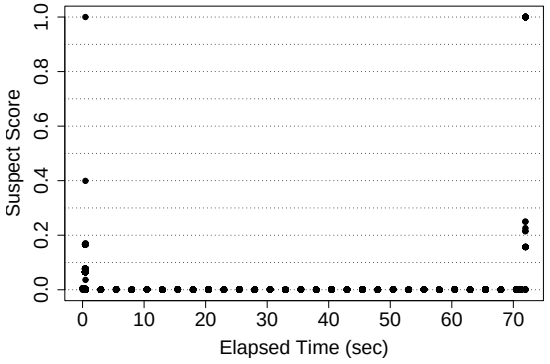

図 1: 故障発生時トレースの異常度: X 軸はジョブ投入時からの経過時間、Y 軸は各実行ユニットの異常度

InTrigger 上の 3 拠点 78 ノード上で MPICH ジョブマネージャを実行し、故障トレースを取得した。正例トレースは InTrigger 上の単一拠点のクラスタを用いて取得した。取得したトレースより異常度を計算した結果を図 1 に示す。図にある通り、実行開始直後に高い異常度を持つユニットを検出した。同ユニットについて人手によりさらに解析したところ、バグにより通信路が切断された部分に該当することがわかった。以上の結果より、本提案手法が分散システムにおける故障解析に有効であることを確認した。

5 おわりに

本稿では情報爆発時代の大規模計算システム向けの故障解析技術を提案し、有効性を示した。今後の課題として、他の分散システムへの適用、実行時解析への応用などが挙げられる。

謝辞 本研究の一部は科学研究費補助金特定領域研究 (18049028) の補助による。

参考文献

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message pass ing interface standard. *Parallel Computing*, 22(6):789–828, Sep 1996.

[2] 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中健, 高橋慧, 関谷岳史, 頓楠, 柴田剛志, 横山大作, and 田浦健次朗. Intrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境. In **情報処理学会研究報告** *HPC-111 (SWoPP 2007)*, pages 237–242, 旭川, 8 月 2007.