Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

動的依存性グラフを用いた障害原因解析の計算コスト削減に関する一考察

幾世知範[†1]　榎本真俊[†1]　櫨山寛章[†1]
門林雄基[†1]　山口英[†1]

仮想化技術や分散処理技術の普及に伴ってコンポーネント間の依存関係は多様化しており，障害発生時の原因解析が困難となっている．静的依存性グラフを用いて詳細な障害原因解析を行う手法として NetMedic が提案されているが，今日のサービス環境に適用するには解析に要する計算コストが高いという課題がある．そこで，本研究では計算コストの削減を目的として障害原因解析に分散処理のトレース結果に基づいた動的依存性グラフを用いることを提案する．動的依存性グラフを用いる場合には分散処理のトレースおよび障害検知の精度によって障害原因解析結果に影響が出る可能性があるが，動的依存性グラフを用いることで障害原因解析に要する計算コストを削減できることが確認できた．本稿では，分散処理のトレース結果を用いた静的依存性グラフからの動的依存性グラフの作成方法および実験と評価の結果を報告する．

Reducing Computational Costs of Root Cause Analysis using Dynamic Dependency Graph

Tomonori Ikuse,[†1] Masatoshi Enomoto,[†1]
Hiroaki Hazeyama,[†1] Youki Kadobayashi[†1]
and Suguru Yamaguchi[†1]

Along with the prevalence of virtualization and distributed processing technologies, dependencies among components of a system are becoming diversified. Therefore, it is difficult for administrators to identify the root causes when a failure occurs. Against such operational difficulty, a diagnosis method, named NetMedic, has been proposed. NetMedic can finely analyze root causes by using a static dependency graph. However, NetMedic could not be applied into actual operation due to its heavy computational cost. Thus, we extend NetMedic to employ a dynamic dependency graph instead. Through an experimentation, we verified that our extension achieved faster computation time on ranking root causes than the original algorithm of NetMedic. In this paper, we explain the way to build a dynamic dependency graph using trace data, and show the results of our evaluation.

1. はじめに

今日，省電力化を目的としたサーバ集約やクラウドサービスにおける資源の分配を実現するために仮想計算機技術が利用されている．仮想計算機技術を用いることで物理サーバの資源を仮想計算機単位に分配することができるため，従来複数の物理サーバ上で別々に運用されていたサービスを 1 台の物理サーバの上に集約して運用することが可能となっている．ただし，資源を有効活用できる反面，仮想計算機環境では同じ物理サーバ上で動作する仮想計算機間で資源競合が発生するため，サービスの性能が劣化する可能性がある[1)]．一般的に，サービスを集約する際にはモデリングを用いた設計が行われるが，モデリングはサービスの正常な動作を対象に行われるため予期しない負荷が発生した場合には資源競合が発生する可能性がある．つまり，仮想計算機環境ではサービスとしては関連していないコンポーネント同士がソフトウェアレイヤを跨いで影響し合う可能性がある．この結果，仮想計算機環境上で提供されるサービスの性能劣化が発生した場合の障害原因解析および障害対応が困難となっている．

これに加えて，このような仮想計算機環境上で運用されるサービスの特徴も障害原因解析および障害対応を困難としている．3 層構造 (web サーバ，アプリケーションサーバ，DB サーバ) を持つ Web サービスをはじめ，今日提供されているサービスでは分散処理が行われている．分散処理を行うサービスの場合，リクエストを処理する際に関連する各サーバが性能劣化の原因となる可能性がある．さらに，フェイルオーバーや負荷分散といった技術が用いられている場合には，リクエストごとにリクエストの処理を担当するコンポーネントが動的に変化する．このため，事前に把握していた依存関係だけでは正確な依存関係を捉えることができず，性能劣化発生時に原因を突き止めることが困難となっている．

このような依存関係の多様化に伴って障害発生時の障害対応が困難となり，障害原因解析を自動で行う仕組みが求められている．特に，仮想計算機環境や分散処理技術を用いている

†1 奈良先端科学技術大学院大学
Nara Institute Science of Technology

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

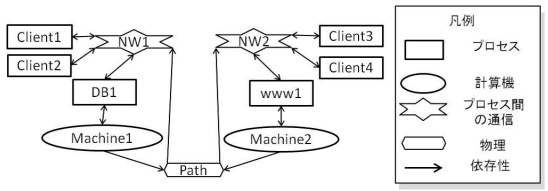

図 1 依存性グラフの例
Fig. 1 An Example of a Dependency Graph

Algorithm 1 NetMedic の処理の流れ

1: $\boldsymbol{V}, \boldsymbol{E} \leftarrow BuildDependencyGraph()$
2: $\boldsymbol{V_a} \leftarrow CalcNodeAbnormality(\boldsymbol{V}, \boldsymbol{E})$
3: $\boldsymbol{E_a} \leftarrow CalcEdgeWeight(\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{V_a})$
4: $L \leftarrow ListCandidates(\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{V_a}, \boldsymbol{E_a})$

環境ではコンポーネントの状態を正確に捉えることができる手法が必要である．コンポーネントの状態を状態ベクトルで正確に捉えた上で，依存性グラフを用いて障害原因解析をする手法として NetMedic[2] が提案されている．NetMedic はコンポーネントの状態を正確に捉えた上で解析を行うことができる反面，解析に要する計算コストが高いという問題がある．グラフ解析を行うため，依存性グラフの規模が大きくなるにつれて障害原因解析に要する時間が膨大になる．

そこで，本研究では障害原因解析に要する計算コストを削減する目的で動的依存性グラフを用いた障害原因解析を行うよう NetMedic を拡張することを提案する．本研究では，障害が発生する前に予め依存性を把握して作成したグラフを静的依存性グラフ，収取した統計情報や追跡結果など診断対象に関する情報を静的依存性グラフに反映させて動的に作成した依存性グラフを動的依存性グラフと呼ぶ．つまり，動的依存性グラフは静的依存性グラフの部分グラフとなる．本研究ではユーザからのリクエストに対するレスポンスが著しく低下した状況を障害と定義し，障害原因解析の際に分散処理の追跡結果を反映させた動的依存性グラフを用いることで計算コストの削減ができることを実験によって示す．

2. NetMedic の概要と課題

本節では最初に NetMedic の概要を説明し，その上で NetMedic の課題となっている項目を述べる．

NetMedic[2] は静的依存性グラフを用いた障害原因解析手法であり，コンポーネントの状態をベクトルで扱って詳細な解析を行うことを可能としている．コンポーネントの状態を単純化し，ベイジアンネットワークや推論グラフを用いて解析する手法[3][4] とは異なり，コンポーネントの状態をより正確に捉えることができる．

NetMedic では図 1 に示したような依存性グラフが用いられている．図 1 ではグラフ上のノードとして計算機，プロセス，物理パス，ネットワークを表現し，web サービスとデータベースサービスを提供するシステムの依存関係を表している．グラフの各ノードの状態はコンポーネントの状態を正確に捉えるために状態変数ベクトルで表現され，状態変数ベクトルを用いた異常検知およびグラフ上のエッジの重み計算が行われる．NetMedic の処理の流れをアルゴリズム 1 に示す．障害発生時には，予め作成しておいた静的依存性グラフを用いてノードの異常度計算 (アルゴリズム 1, 2 行目)，エッジの重み計算 (アルゴリズム 1, 3 行目)，障害原因解析 (アルゴリズム 1, 4 行目) を順に処理する．そして，最後に障害原因解析の結果として 1 つの障害原因候補リストを出力する．以下では各処理について処理の内容を説明する．

(1) 前提条件
静的依存性グラフは有向グラフ $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ として与えられ，ノード v_i からノード v_j 方向へのエッジは e_{ij} と表わされるものとする．$\boldsymbol{G}$ は，各ノードにおける CPU，メモリ，I/O など K 種類のパラメータの時間変動を表わす状態変数ベクトル $\boldsymbol{C}$ を保持する．ノード v_i の状態ベクトルは $\boldsymbol{C}_i$，ノード v_i の品種 k のパラメータの時間変異ベクトルは $\boldsymbol{c}_i^k$，現在から T 秒前までのある時間 t におけるノード v_i の品種 k のパラメータの値を $c_{i,t}^{k,T}$ と表わす．このとき現在の時刻は $t = 0$ とする．また，単純化のために，時刻 t は連続変数ではなく，1 秒間隔で表現する離散変数とする．

(2) 異常度の計算
異常検知とエッジの重みの計算には現在と過去 T 秒前までの状態変数ベクトル $\boldsymbol{C}^T$ を用いる．ノード v_i 異常状態変数ベクトル $\boldsymbol{a}_i^T$ は，現在から T 秒前までのノード v_i の各品種の状態変動から算出する．各パラメータの変動は正規分布に従うと仮定し，t 秒前 $(0 \leq t \leq T)$ の品種 k に関するノード v_i の異常値 $a_{i,t}^{k,T}$ は誤差関数 (erf) を用

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

いて算出した時間 t における品種 k の時刻 t における変動の誤差として次のように表現し，その中で最大の値を時間 t におけるノード v_i の代表異常値 ${a'}_{i,t}^{T}$ とする．

$$a_{i,t}^{k,T} = \mid erf(\frac{c_{i,t}^{k,T} - \mu_i^{k,T}}{\sqrt{2}\sigma_i^{k,T}}) \mid$$
$$\mu_i^{k,T} = \frac{\sum_{t=0}^{T} c_{i,t}^{k,T}}{T}$$
$$\sigma_i^{k,T} = \sqrt{\frac{\sum_{t=0}^{T}(c_{i,t}^{k,T} - \mu_i^{k,T})^2}{T-1}}$$
$${a'}_{i,t}^{T} = \mathbf{max}(a_{i,t}^{k,T} | k \in K)$$

(3) エッジの重み計算

エッジ e_{ij} の両端もしくは片側のノードが正常な状態，つまりノード v_i，ノード v_j それぞれの現在の異常値が ${a'}_{i,0}^{T} < \delta, {a'}_{j,0}^{T} < \delta$ (δ は閾値) であれば，そのエッジの重みは低く設定する．一方，エッジ e_{ij} の両端のノードが異常である場合，つまり，${a'}_{i,0}^{T} \geq \delta, {a'}_{j,0}^{T} \geq \delta$ である場合は以下の手順で計算を行う．現在から T 秒前の時間区間の中で，ノード v_i の現在の状態ベクトル $\boldsymbol{C}_{i,0}^{T}$ との類似した時間を探し出す．まず，以下の式で与えられる時間 t と 現在時間との品種 k に関する距離 $d_{i,(0,t)}^{k,T}$ を計算する．

$$d_{i,(0,t)}^{k,T} = \frac{\mid c_{i,t}^{k,T} - c_{i,t}^{k,T} \mid}{\mathbf{max}(c_{i,t}^{k,T} | 0 \leq t \leq T) - \mathbf{min}(c_{i,t}^{k,T} | 0 \leq t \leq T)}$$

次に，ヒューリスティックスなどを用いて，品種 k の異常値をノード v_i の異常値として取り入れるかどうかを決める K の要素を持つ決定変数ベクトル $\boldsymbol{y}_i$ を設定する．文献[2] では決定係数ベクトル $\boldsymbol{y}_i$ の設定方法を明記していないが，本論文で用いる品種 k に関する決定変数 y_i^k は次のように表わされる．

$$y_i^k = \begin{cases} 1 & \text{if} \quad \mathbf{max}(\mathbf{cov}(C_i^k, C_i^l) | k, l \in K, k \neq l, k < l) \geq \delta_y \\ 0 & \text{otherwise} \end{cases}$$

決定変数ベクトル $\boldsymbol{y}_i$ を用い，ノード v_i の現在の状態と t 秒前の状態の距離は次のように表現される．

$$d_{i,(0,t)}^{T} = \frac{\sum_{k=1}^{K}(\mid d_{i,(0,t)}^{k,T} \mid a_{i,0}^{k,T} y_i^k)}{\sum_{k=1}^{K} a_{i,0}^{k,T} y_i^k}$$

時間 T の区間における時刻 0 と時刻 t におけるノード v_i の重み $w_{(0,t)}^{T}(v_i)$，およびエッジ e_{ij} への重み $w_{(0,t)}^{T}(e_{ij})$ は次のように表わされる．

$$w_{(0,t)}^{T}(v_i) = \begin{cases} 1 - d_{i,(0,t)}^{T} & \text{if} \quad d_{i,(0,t)}^{T} < \delta_d \\ 0 & \text{otherwise} \end{cases}$$
$$w(e_{ij}) = \frac{\sum_{t=0}^{T}(1 - d_{i,(0,t)}^{T}) w_{(0,t)}^{T}(v_i)}{\sum_{t=0}^{T} w_{(0,t)}^{T}(v_i)}$$

この計算の結果得られるエッジの重み $w(e_{ij})$ は v_i と v_j の異常な状態に関連性があれば高い値，関係が無ければ低い値となる．言い換えると $w(e_{ij})$ はノード v_i とノード v_j との間の異常相関度である．

(4) 障害原因解析

障害原因候補リストは以下の計算式によって算出される．静的依存性グラフ G 中の全ノード $\boldsymbol{V}$ のうち，障害が検知されたノードのグループを $\boldsymbol{V_a}$ とする．$\boldsymbol{V_a}$ に含まれる任意のノード v_s から $\boldsymbol{V}$ に含まれる任意のノード v_d に対して経路探索を行い m 個の経路集合 $\boldsymbol{P_{sd}} = \{p_{sd,1}, p_{sd,2}, ..., p_{sd,m}\}$ を得る．v_s から v_d に対する異常の影響度 $I(v_s \rightarrow v_d)$ は $\boldsymbol{P_{sd}}$ に含まれる経路の中で最も重い経路の重み，つまりパス上に存在するノード間の異常相関度が高い経路における各エッジの異常相関度の相乗平均を用いる．式で表現すると次式のようになる．v_s の異常度 $S(v_s)$ は v_s から $\boldsymbol{V}$ に含まれる任意のノードへの影響度の総和で表現される．最後に障害原因候補リストのランク値として用いられるランク値 $R(v_s \rightarrow v_d)$ は $I(v_s \rightarrow v_d)$ と $S(v_s)$ との積の逆数として与えられる．v_d を障害が起きているノードとみなした場合，最もランク値の低いノードが障害原因候補となる．以上の内容を式で表現すると次式のようになる．

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

$$w(p_{sd,i}) = \left(\prod_{j=1}^{n} w(e_j) \right)^{\frac{1}{n}}, e_j \in p_{sd,i}, 1 \leq j \leq n \quad (1)$$

$$I(v_s \to v_d) = \mathbf{max}(p_{sd,i} | p_{sd,i} \in \boldsymbol{P_{sd}}) \quad (2)$$

$$S(v_s) = \sum_{v_d \in \boldsymbol{V}, d \neq s} a'^{T}_{d,0} I(v_s \to v_d) \quad (3)$$

$$R(v_s \to v_d) \propto (I(v_s \to v_d) \times S(v_s))^{-1} \quad (4)$$

上記の計算を行った結果として，NetMedic は障害原因候補のリストを提供する．このリストを受け取った管理者は障害原因候補リストの上位から順に対策を試みることになる．しかし，現状の NetMedic には以下の 2 つの課題がある．

1 つ目は，計算コストの問題である．NetMedic は障害原因解析の際に各ノードが他のノードにどれだけ影響を与えているのかを式 (3) で計算し，観測された障害への影響度を式 (4) で評価する仕組みとなっている．この際，グラフに属する全ノードに対して一番高い重みを持つ経路 p の重みを式 (1) で計算し，式 (1) の影響度の計算に利用している．この影響度の計算過程で用いる経路の重み算出に要する時間はグラフ上のエッジの数およびノードの数によって変化し，規模が大きくなればなるほど解析に要する時間は増加する．この問題が NetMedic のサービス環境への適用を妨げている．

2 つ目の課題は複数の障害原因が含まれている際に障害原因解析の切り分けができないことである．NetMedeic ではコンポーネントの状態を詳細に捉えることができているにも関わらず，障害原因候補同士の依存関係を考慮していないため，複数の障害原因が存在する場合に障害原因の切り分けを行うことができない．このため，障害原因が複数存在する場合，1 つの障害原因よりも下位に残りの障害原因が表示されるため管理者は対応の必要性に気がつくことができない．障害の再現が困難な場合や障害原因解析に時間を要する場合には一度の解析で複数の問題が含まれていることを把握できることが望ましく，課題を解決することで NetMedic はより効果的な手法になると考えられる．

このように NetMedic には大きく 2 つの課題が存在しているが，本研究では計算コストの課題に取り組む．

3. 提　　案

NetMedic の計算コストの削減を目的として，本研究では NetMedic を動的依存性グラフを用いた解析手法へ拡張する．NetMedic は静的依存性グラフを用いて解析を行なっている．大規模な計算機環境では静的依存性グラフは大量の依存関係を保持しているため，障害原因解析に要する時間的，計算資源的コストが膨大になる．静的依存性グラフの部分グラフである動的依存性グラフを作成して用いることで障害原因解析の際に必要となる計算コストを削減できると考えられる．そこで，本研究では計算コストの削減を目的として，動的依存性グラフの作成および動的依存性グラフを用いた障害原因解析を提案する．

動的依存性グラフに変換する手法としては主に 2 つの手法が考えられる．1 つ目は実際に発生した事象を静的依存性グラフに反映させる手法である．これにはメッセージログとして記録されていた情報を反映する方法や処理の流れのトレース結果を反映させる方法が該当する．もう 1 つは監視して取得した情報から依存方向を推定する手法である．この手法は限られた情報から依存性を推定する方法である．監視して取得したエラーメッセージの時間的関係性に着目して依存方向を推定する方法や監視情報を用いてパラメータ変化の相関から依存関係を推定する手法が考えられる．上記の手法のうち，本稿では分散処理のトレース結果に基づいた動的依存性グラフの作成を行う．

以降では，分散処理のトレース結果に基づいた動的依存性グラフの作成方法を説明し，作成された動的依存性グラフを用いた障害原因解析について静的依存性グラフを用いる場合との違いを監視に要するコストと解析に要する時間的コスト，および障害原因解析精度への影響の側面から評価する．

4. 分散処理のトレース結果に基づいた動的依存性グラフの生成

本節ではアルゴリズム 2 に沿って動的依存性グラフを用いた障害原因解析の概要を説明し，合わせてトレース結果を用いた動的依存性グラフの作成手法について説明する．

静的依存性グラフから動的依存性グラフを作成するために，予め静的依存性グラフを作成する (アルゴリズム 2, 1 行目)．この静的依存性グラフはシステム構成情報とサービスを構成するコンポーネント間の依存関係から作成される．システム構成情報とはどのコンポーネントの上で何のコンポーネントが動作しているかというソフトウェアスタック情報と物理リンクに関する情報である．もう一方のサービス内の依存情報では，web サーバ A と DB サーバ B の参照関係などのサービスレベルでの依存関係を保持する．つまり，静的依存性グラフは全依存関係を保持したグラフとなる．

分散処理のトレースは常時行い，以下に示す処理は障害が発生した後に行う．ユーザのリクエストに対するレスポンスが著しく低下したことを検知した場合，そのリクエストが実際

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

Algorithm 2 本研究で提案する処理の流れ

```
1: V, E ← BuildDependencyGraph()
2: Et ← GetTraceResult(Query_ID)
3:                                   ▷ トレース結果の取得
4: E ← EdgeCut(V, E, Et)
5:                          ▷ トレース結果に基づいたエッジカット
6: V_a ← CalcNodeAbnormality(V, E)
7: E_a ← CalcEdgeWeight(V, E, V_a)
8: L ← ListCandidates(V, E, V_a, E_a)
```

にどの経路を通って処理されたものなのかをトレース結果ログから復元し，リクエストが処理されたノード間を結ぶエッジの集合 $\boldsymbol{E_t}$ を得る (アルゴリズム 2, 2 行目)．$\boldsymbol{E_t}$ に基づいてリクエストの処理に関連の無かった静的依存性グラフ上のエッジをカットすることで動的依存性グラフの作成を行う (アルゴリズム 2, 3 行目)．

エッジカットを行う際のアルゴリズムに関してはアルゴリズム 3 に示した通りである．トレース結果に基づくエッジカットは依存性グラフ中のプロセスを表すノードとネットワークを表すノードを結ぶエッジに対してのみ適用する．逆方向のエッジも $\boldsymbol{E_t}$ に属さないエッジはカットする (アルゴリズム 3, 5-6 行目)．この一連の処理によって得られた動的依存性グラフを用いて NetMedic による障害原因解析を行う．

分散処理のトレースを行う研究は従来から行われておりトレース用フレームワーク機能単位でのトレース機能が実現されている[5]．また，オープンソースの分散処理トレースフレームワーク X-Trace[6] が公開されている．本研究では先行研究とは異なりトレースを行うことが主な目的ではなく，トレース結果を静的依存性グラフから動的依存性グラフへの作成に応用することが目的である．

5. トレース結果を用いた動的依存性グラフ作成の実験と評価

トレース結果に基づいてエッジカットを適用した動的依存性グラフの評価を行う．トレース結果を用いてエッジカットを行うことで解析に要する時間をどれだけ短縮できたのかを評価する．評価するに当たり NetMedic を実際に実装した．実装したソースコードの行数は NetMedic 本体が python で 701 行，そのうち提案方式であるアルゴリズム 3 の実装行数は

Algorithm 3 トレース結果に基づいたエッジカット

```
1: procedure EdgeCut(V, E, Et)
2:     for all e ∈ E do
3:         if e は Process と NbrSet の間の Edge then
4:             e_reverse ← e と逆方向の Edge
5:             if e, e_reverse ∉ Et then
6:                 e を E から取り除く
7:             end if
8:         end if
9:     end for
10: end procedure
```

表 1 評価に用いた環境
Table 1 Evaluation Environment

サーバの性能	
CPU	Intel Core 2 Quad
Memory	8GB
HDD	250GB × 2
NIC	1Gbps

VM	
CPU	1 コア
Memory	1GB
HDD	8GB
Network	Bridge

ソフトウェア	
VMM	Xen 4.0
Web	apache 2.2.3
DB	PostgreSQL
Load Balancer	Pen 0.18.0
OS kernel	2.6.31
監視ツール	iostat, netstat, top, iotop, xentop, pastmon
ベンチマークツール	bonnie++[7], curl-loader[8]

8 行，ログの csv 形式への変換には perl で 907 行，障害原因候補の解析のランク値の計算 (グラフ探索を含む計算) は C 言語で 465 行である．本節では，評価に用いた環境と実装を説明した後，評価の結果を示す．

本研究で対象とするのは仮想計算機環境上で運用されるサービスの性能劣化の問題である．ユーザからのリクエストに対する応答が著しく低下したという事象を障害と定義する．提案手法の評価を行うために図 2 の環境を表 1 に示したハードウェアおよびソフトウェア

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

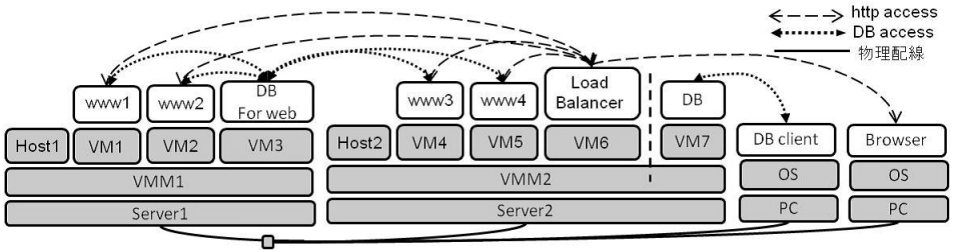

図 2 評価に用いた実験環境
Fig. 2 Experimental Topology

表 2 監視対象
Table 2 Monitoring Targets

監視対象	監視内容
process	共通項目，トレース情報 (Log)
GuestOS(VM)	共通項目
HostOS 及び VMM	共通項目, VMM から見た VM の情報 (共通項目)
ネットワーク	レスポンスタイム，トランザクション数
共通項目	cpu 使用率, memory 使用率, disk I/O, network I/O

表 3 評価のために意図的に発生させた性能劣化 (障害)
Table 3 List of injected failures

障害名	障害原因	発生させる方法
障害 1	異なるサービスでの負荷の増加	図 2 の VM7 で bonnie++を起動
障害 2	アクセス数の急激な増加	curl-loader でクライアント数を追加
障害 3	障害 1 と障害 2 がほぼ同時に発生	障害 1 と障害 2 と同じ

で構築した．実験に用いた環境では仮想マシンモニタ Xen 上で web サーバとデータベースサーバ (Web サービス用と別用途) の 2 つのサービスが集約されている．表 1 に示したベンチマークツール curl-loader でクライアントをシミュレートした．取得する情報は表 2 に示したとおりであり，表 1 に示した監視ツールを用いて取得した．トレース情報に関して，本来は，X-Trace[6] のようなフレームワークを用いてトレース可能な環境を構築するべきだが，今回はクライアントの挙動をシミュレートする web ベンチマークツール curl-loader のログ機能を利用して取得したログの解析によって代用した．

評価にあたり，発生させた障害は表 3 の通りである．表 1 に記したディスクベンチマークツールである bonnie++を図 2 の VM7 で動かすことにより擬似的に障害 1 を発生させ，web ベンチマークツールである curl-loader でシミュレートするクライアントの数を増加させることで障害 2 を発生させた．2 つの障害に関して個別に発生させた場合と合わせて発生させた場合について，トレース結果に基づいたエッジカットを行う場合と行わない場合について解析に要するコストおよび障害原因解析への影響を評価した．評価の際の診断対象となるリクエストはレスポンスが通常時 (50 ミリ秒) の 100 倍以上の値に悪化したものを手動で選出している．

表 4 障害原因候補当たりの解析コスト
Table 4 Analysis Cost on each Candidate Cause

	エッジカットあり			エッジカットなし		
	探索経路数 (万)	ランク値計算時間 (s)	ログ (MB/min)	探索経路数 (万)	ランク値計算時間 (s)	ログ (MB/min)
障害 1	1,634	1097	20.4	7,135	4,809	1.2
障害 2	1,453	959	25.3	7,320	4,770	0.8
障害 3	4,206	975	19.0	7,586	4,995	1.8

解析に要するコストの評価結果を表 4 に示す．表 4 では，トレース結果に基づいたエッジカットを行う場合と行わない場合のランク値の計算に要する時間と探索経路数，解析のために用いるログの量を合わせて表す．図 2 に示した実験環境を静的依存性グラフに変換した場合の総ノード数は 661 個であり，総エッジ数は 1,360 本である．障害原因解析を行う際，最も時間を要する計算過程はランク値の計算であった．従って，表 4 には障害原因候補 1 つ当たりの探索経路数とランク値解析時間の平均を評価結果として載せている．表 4 に示した通り，障害の種類に依存せずランク値の計算速度は実測値で約 4 倍程度向上した．その一方でトレースのためのログが発生し，トレース結果に基づいたエッジカットを行わない場合に比べて 20 倍程度のログが出力されている．

次に，障害原因解析に動的依存性グラフを用いることによる障害原因解析結果への影響について評価する．表 5 にはエッジカットを行う場合と行わない場合それぞれについて障害原因候補リストに出力された実際の障害原因の順位を示した．今回の実験環境において，負荷分散はウェブサーバ 4 台で行ったため，リクエストが通る可能性のある経路は 4 つである．エッジカットありの場合には，各経路について障害原因候補リストを作成し，障害原因の順位の最良の場合と最悪の場合，および平均を示した．障害 3 に関して，表 5 の障害原因 1 が障害 1 の障害原因，障害原因 2 が障害 2 の障害原因である．障害原因候補の中に含まれているノードの数は障害 1 と障害 2 が 28 個，障害 3 が 32 個であった．

障害 1 に関しては選んだパスに応じて障害原因解析の結果が良くなる場合も悪化する場

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

表 5 障害原因候補リスト中の障害原因の順位
Table 5 Rank of Candidate Causes

	エッジカットあり			エッジカットなし
	最良	最悪	平均	
障害 1	11	15	13	12
障害 2	1	1	1	1
障害 3(障害原因 1)	12	12	12	12
障害 3(障害原因 2)	9	9	9	4

合もあるという結果になったのに対し，障害 2 に関しては障害原因解析結果への影響は見受けられなかった．障害 3 では障害原因 1 に関して順位に変動は無かったが，障害原因 2 に関しては障害原因解析の結果に影響がでるという結果になった．

6. 考　　察

表 5 と表 4 からエッジカットを行い計算コストを削減させることは障害原因解析の精度とのトレードオフとなることが明らかとなった．表 4 に示したとおり，トレース結果を用いた動的依存性グラフを適用することで障害原因解析の際に要するコストを削減することができる．今回の実験環境における実測値ではランク値の計算速度に関して 4 倍程度の差がでる結果となった．ただし，動的依存性グラフを作成する際にエッジカットを行なった影響で障害原因解析の精度に影響を与えてしまうことも分かった．

障害原因解析の結果が悪化する可能性があることは障害 1 と障害 3 の結果から分かる．ただし，障害 1 ではカットするエッジによって障害原因解析の結果が改善される場合もあれば悪化する場合も存在した．これはカットしたエッジの先に障害原因が存在していることと，障害原因候補同士のランク値が近いことが原因であると考えられる．ランク値の計算に用いられている影響度は式 (3) で全てのノードとの影響度と異常度を用いて計算されている．エッジカットを行う場合，障害原因のノードから各ノードへの経路として本来選ばれるはずだった経路が無くなりランク値が減少する可能性がある．また障害原因候補同士は近いランク値を持つ可能性があるため，ランク値が減少した際に順位が入れ替わったものと考えられる．障害 3 では障害原因解析の結果が変わらないもくしくは悪化するという結果のみしか得られなかったが，障害 1 と同様に障害原因候補同士のランク値が近いこと，エッジカットによるランク値の減少が原因だと考えられる．さらに障害 3 では複数の障害を発生させているため高い異常値を持っているノードが多く，エッジをカットすることでそれらのノードへ到達できる最短の経路が無くなったことでランク値への影響も大きくなったと考えられる．

分散処理のトレース結果を用いて動的依存性グラフを作成することを提案したが，トレース結果に誤りがある場合も障害原因解析の結果に影響がでる可能性がある．また，障害検知の段階で誤りがあった場合についても障害原因解析の結果に影響がでる可能性がある．そのため，診断の対象となるリクエストを選択する方法についても考慮していく必要がある．

NetMedic は依存性グラフ上の各ノードの持つ状態変数の値を取得するために多くの監視ツールを用いて監視を行っている．そのため，表 4 に示したとおり，大量のログが発生している．今回は 5 秒ごとに監視ツールでパラメータの値を取得していた．障害原因解析の時間的な粒度とのトレードオフとなるが監視の間隔を長くすることでログの量を低下させることは可能である．また，サンプリングを適用することで減らすことが可能であることが知られているため，トレースに用いた量は減らすことが可能である[5]．

7. 今後の課題

本稿ではトレース情報を用いて静的依存性グラフから動的依存性グラフを作成することを試みた．実験の結果，エッジカットを行い障害原因解析に要する計算コストを削減することと障害原因解析の精度は依存関係にあることが分かった．ただし，この結果は今回用いた実験環境と起こした障害の種類に依存すると考えられるため，今後さらにパターンを増やして実験を行なっていく．また，削減される計算コストについて実測値と探索した経路数のみでしか表すことができていないため，今後，計算量として表すことにも取り組む予定である．

また，トレース機能に関しては現状の評価環境と同様な構成の環境を X-Trace[6] を適用できるように既存のデータベースとロードバランサを拡張して利用していきたいと考えている．

8. ま と め

本研究では計算コストの削減を目的として動的依存性グラフを用いた解析を行うよう NetMedic の拡張を行うことを提案した．本稿では分散処理のトレース情報を用いて静的依存性グラフから動的依存性グラフを作成する方法を示し，それを用いて解析を行うことで障害原因解析に要する計算コストを削減できることを実装と評価によって示した．動的依存性グラフを用いる場合には分散処理のトレースおよび障害検知の精度によって障害原因解析結果に影響が出る可能性があるが，実験の結果，静的依存性グラフを用いて障害原因解析を行

Vol.2011-OS-119 No.7
Vol.2011-EMB-23 No.7
2011/11/29

う場合と比べて動的依存性グラフを用いることで計算コストが削減できることが確認できた．今後は計算量など理論的な評価を行うとともに，実験環境を変えることや実験の際に発生させる障害を変更するなど，障害の特徴と実験環境の特徴を考慮した評価を行なっていく．

参 考 文 献

1) Tickoo, O., Iyer, R., Illikkal, R. and Newell, D.: Modeling virtual machine performance: challenges and approaches, *SIGMETRICS Perform. Eval. Rev.*, Vol.37, pp. 55–60 (2010).
2) Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J. and Bahl, P.: Detailed diagnosis in enterprise networks, *ACM SIGCOMM Computer Communication Review*, Vol.39, No.4, p.243 (online), DOI:10.1145/1594977.1592597 (2009).
3) Steinder, M. and Sethi, A.S.: A survey of fault localization techniques in computer networks, *Sci. Comput. Program.*, Vol.53, No.2, pp.165–194 (2004).
4) Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. and Zhang, M.: Towards highly reliable enterprise network services via inference of multi-level dependencies, *ACM SIGCOMM Computer Communication Review*, Vol.37, No.4, pp. 13–24 (2007).
5) Sigelman, B.H., Barroso, L.A., Burrows, M., Setephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Technical report, Google (2010).
6) Fonseca, R., Porter, G., Katz, R.H., Shenker, S. and Stoica, I.: X-trace: a pervasive network tracing framework, *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pp.271–284 (2007).
7) Coker, R.: Bonnie++, `http://www.coker.com.au/bonnie++/`.
8) Iakobashvili, R. and Moser, M.: curl-loader, `http://curl-loader.sourceforge.net/`.