課題番号 12-IS03

ログ解析機構を備えた並列スクリプト実行システムの研究

倉光君郎（横浜国立大学）

概要

障害時にシステムが出力するログは，障害対応を行う上で有効な手段となる．プログラマは，syslog などのライブラリを用いてログ出力を行うことになるが，障害解析に役立つログポイントを適切に入れることは難しい．我々は，言語ランタイムからシステム障害診断のためのログ情報を自動収集する手法を提案する．本研究の目的は，自動収集されたログにより，システム障害の状況の把握，さらに障害の要因特定がどの程度可能かを評価することである．本報告では，提案手法にしたがい，言語ログを実装し，予備実験を行い、フォルトインジェクションを行った結果に基づき，ログから障害の診断が可能であるかを検討した結果を示した．

1　研究の目的と意義

現在のコンピュータシステムにとって，システム障害は大きな脅威である．障害時にシステムが出力するログは，障害対応を行う上で有効な手段となる．

例えば，従来，MPI アプリケーションのチューニングや，障害解析はアプリケーションごとに個別に行われてきた．この際に，ログを収集するためのログポイントの挿入や情報の収集は，研究者やエンジニアが個別に実装して利用していたため，その都度コストがかかる問題があった．特に，システムが大規模化し，複雑化すると，こうした個人レベルの取り組みには限界が出てくると考えられ、また障害解析に役立つログポイントを適切に入れることは難しい．

こうした問題に対し我々は，言語ランタイムからシステム障害診断のためのログ情報を自動収集する手法を提案する．障害要因の特定と正しい診断を導くためには，プログラムの例外処理自動収集するログはなどにより得られる一般的な例外やエラー情報以外にも，場合によってはそのエラーの元になると考えられるシステムの状況や，関連するライブラリの呼び出し情報など，その障害に直接または間接に関わりがある様々な情報により障害の診断を行う必要がある．しかし，こうした関連した情報を統一的に収集する仕組みは通常，言語や MPI ライブラリ、オペレーティングシステム(OS)などに個別のツールとして提供されており，個々にインストールが必要であったり，ログの出力先が異なったりと，情報が統合されていないことが多く，障害時に関連する情報を収集しづらい問題がある．このことから，我々は言語ログとして，外部関数（システムコール）などの呼び出しイベントとそのエラー情報,障害要因がシステムの状態と関係する場合に必要となる OS が提供するリソース利用の状況，外部関数などの呼び出しイベント，言語ランタイムからの情報などを自動収集の対象とし，総合的に障害を診断できるものとした．

本研究の目的は，これらの自動収集されたログにより，システム障害の状況の把握，さらに障害の要因特定がどの程度可能かを評価することである．

2　当拠点公募型共同研究として実施した意義

(1) 共同研究を実施した大学名と研究体制

東京大学

(2) 共同研究分野

超大規模情報システム関連研究分野

(3) 当公募型共同研究ならではという事項など

近年、超大規模計算機を用いた超大容量のストレージおよび超大容量ネットワークなどの情報は，社会の基盤としての重要性がますます増大している。

一方、こうしたシステムでは，システムやそれ

を構成するコンポーネントが互いに複雑に依存してサービスを提供しているため、障害時の解析や、性能改善のためのチューニングが困難となる問題がある。分散アプリケーションの信頼性向上の研究者や、性能向上のための自動チューニングの研究者が参加することが可能な点が当公募型共同研究ならではの特徴である。

3　研究成果の詳細と当初計画の達成状況

(1) 研究成果の詳細について

3.1　概要

本研究は，大規模システムにおける信頼性向上，性能改善の支援のための自動収集されたログにより，システム障害の状況の把握，さらに障害の要因特定がどの程度可能かを評価するため、ログ解析の基盤技術であるログプールを提供することを目的とする．ログプールは，超大規模分散システムの要求に合わせたログの量のスケールアウトへの対応や，ログの入出力をスクリプト言語から柔軟かつ高速に行うためのインターフェイスを提供することで，ログの利用価値を高める基盤である．ログプールおよび言語支援の全体像を別紙に示した(図 1)．ログプールは，ログをプールするシステム基盤と，ログポイントの自動挿入機構を行う言語支援からなる．システム支援では，高速な分散メモリ上にいったんログをプールする仕組みにより，出力時のディスク I/O の負担を軽減し，デバッグレベルの詳細な情報を，分散共有メモリ上に記録する仕組みを持つ．また言語支援では、アプリケーションに起こりうる障害を以下の節で示すモデルによって表現し、アプリケーション中にログポイントを自動的に挿入する仕組みを構築する。これまでの期間で我々は言語支援のパートについて注力してきた。本報告では言語支援について成果を報告する。

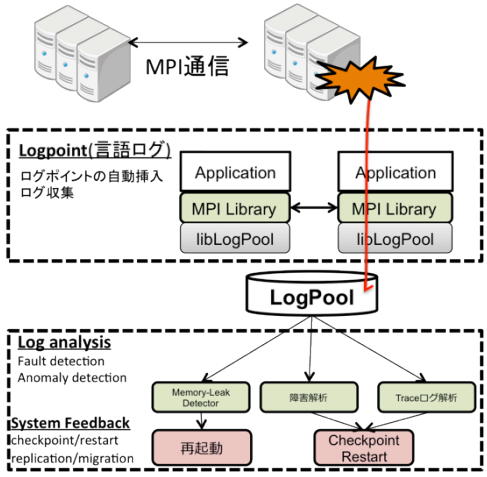

図 1 Logpool アーキテクチャ

3.2　障害発生のメカニズム

ディペンダビリティの概念と分類学は，過去 40 年にわたる対故障システムの研究や実践の集大成として，IEEE Transaction on Dependable and Secure Computing 誌の基調論文となっている．この概念と分類法は，TDSC (Taxonomy of Dependable and Secure Computing) と呼ばれ，研究者から産業界まで広く参照されている．本節では，本報告の用語を明確にするため，まず TDSC にしたがい，障害発生のメカニズムとプログラミング言語処理系におけるエラーの扱いを整頓し，フォルトモデルをおく．

3.2.1 脅威モデル

TDSC では，障害発生のメカニズムをフォルト(fault)，エラー(error)，失敗/障害 (failure) の 3 段階に分類している．フォルトは，障害要因となりうるイベント，もしくは行動である．エラーは，フォルトによって出現する正常状態から逸脱した状態である．エラーは，放置すると，失敗，つまりサービスが提供できない事態につながる．(失敗に至らない場合もあり，エラーがそのまま失敗を誘発しない．) また，ある失敗は別の失敗を引き起こすフォルトとなり，エラー伝搬が説明される(図 2)．

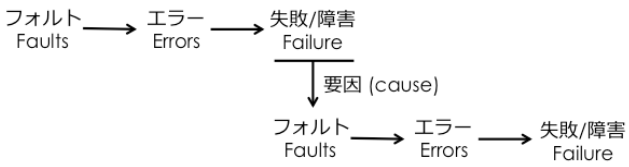

図 2 TDSC エラー伝搬モデル

エラー伝搬モデルの基礎となる TDSC の脅威モデルの特徴は，フォルト，エラー，失敗を厳密に区別することである．障害への対応手段は，フォルト予測，フォルト除去，フォルト寛容(fault tolerance)，フォルト回復の４つに分類されている．一般に，日本語では結果である障害に着目し「障害」対応と呼ぶが，TDSC ではその原因となるフォルトに着目し，フォルトへの対応を重要と考える．

3.2.2モニタリングとログ

モニタリングは，発生したイベント，もしくは状態を観測することができる．本報告では，それぞれイベントモニタリング，状態モニタリングと呼ぶ．ログは，モニタリングによって観測された値を時刻とともに記録したものである．モニタリング値を運用と同期させて処理することをオンライン処理，運用とは独立して処理することをオフライン処理と区別する．

モニタリング技術は，運用中のシステム障害を観測する手段として，広範囲に活用され，商用製品からオープンソースツールまで数多く存在する．エラーは，システムの状態であるため，適切なモニタリング技法によって観測することが可能である．また，障害発生後に，モニタリングシステムを導入しても，観測できる場合がある．たとえば，レスポンス応答が悪いときに，CPU の利用率をモニターすることができる．一方，モニタリングによって，直接，フォルトを検出することは難しい．たとえば， CPU の異常利用を発見しても，その原因となるフォルトは決定的とは言えない．フォルトの特定は根本原因解析(root cause analyses; RCA)と呼ばれる，RCA では，ログ解析や知識ベースと組み合わせた手法が用いられる．

フォルトは，イベントの一部として観測される．しかし，同じイベントであっても，のちにエラー状態やシステム障害が出現するかどうかでフォルトかどうか決まる．たとえば，ファイルを消去する操作は，のちにシステム障害になれば，それはフォルトとなる．このため，障害対応のためにログの記録を行うとき，記録するイベントを判断することの難しい一因となっている．

3.2.3プログラミングとフォルト、エラー

プログラミングでは，フォルトやエラーという用語は頻繁に使われるが，必ずしも TDSC の厳密な用例にしたがっていない．代表的な混用は，エラーは正常状態からの誤差を表す状態であるところを，読み込み「エラー」などのように，フォルトや失敗1をあらわす用語として利用している点である．本報告では，学術的な議論の正確さのため，エラーは正常からの誤差のある状態として用いる．

失敗/障害は，フォルトとして常に次のエラーに伝搬する可能性がある．たとえば，セグメンテーションフォルトは，仮想記憶の保護機構によるメモリサービスの停止であり，素直に解釈すれば，失敗である．この場合，フォルトはプログラマによるバグ2と考えられる．しかし，プログラムの異常終了の原因という視点で考えると，セグメンテーションフォルトはアクセスが許可されていないメモリ箇所へアクセス違反という「フォルト」である．このように，フォルトか失敗は，障害対応するときの視点によって異なる．

さらに，ソフトウェアバグの要因は，テスト仕様書の不備，納期不足，開発者のスキル不足，異なる開発者による拡張，テスト不足などの様々な要因が考えられる．最後は，ソフトウェア開発プロセスや組織管理，組織文化まで原因をさかのぼることができる．議論の発散を防ぐため，ディペンダビリティ分野では，対象とするフォルトを限定し，フォルトモデルを設定して，議論を進めることが一般的である．

3.2.4本報告のフォルトモデル

本報告では，対象とするプログラム実行時に発生した障害に対して，オンラインの障害対応を行うことを目指して，次のようにフォルトモデルを

1 TDSC 論文も，ヒューマンエラーなど慣用的に使われている用語は，本来，ヒューマンフォルトと呼ぶべきであると指摘しているが，慣用的な用語はそのまま利用している．

2 ソースコードの原因個所を調べることの難しさを想像すれば，根本原因解析の困難さや一般化が難しいことが理解できるだろう．

設定する．ここで,対象とするプログラムは,我々の提案する言語ランタイム上で動作するプログラムのことである．なお，言語ランタイムにはフォルトがないものと仮定する．

- ソフトウェアフォルト － 対象とするプログラムが開発段階で混入することが要因で発生するフォルト，設計ミスやバグなど．フォルト除去には開発プロセスでの修正が必要であり，オンラインでのフォルト除去は不可能
- データフォルト - ユーザの入力ミスや入力されたデータの不整合（エラー状態）を要因として発生するフォルト．入力データを修正することで，フォルトを除去できる．
- システムフォルト - 言語ランタイムが動作するコンピュータシステムが，何らかの原因で通常とは異なる動作を行い，それがもとでアプリケーションのエラー状態が引き起こされるフォルト．システム状態を正常状態（初期化，バックアップ）に戻すことで，フォルトを除去できる．
- 外部フォルト - ネットワークや外部サービスなど，外部の管理ドメイン3 のエラー状態によって発生するフォルト．フォルトを除去するためには，管理ドメインのエラー回復が必要である．

注意：近年，障害対応の説明責任が重視されるようになっている．そのため，直接，障害回復が行えなくても，外部の管理ドメインに原因があるかどうか判定できるログを残すことは有用である．

3.3　言語ランタイムの自動ロギング機構

3.3.1ログ修正の問題点

3.1 節で述べたように，ログはシステム障害の要因を特定するために有用であることは自明である．一般的にログを収集するためには，ソースコード上に，必要な情報を記録するための箇所を特定し，出力させるためのポイント（ログポイント）を指定する．

ログポイントは，システム障害の要因を特定する重要な手段となる．プログラム開発者は，障害の予兆となりえる可能性のある箇所にログポイントとして，ログ出力のプログラムを挿入する．次は，ファイルを開くのに失敗したとき，syslog に記録するためのログポイントの例である．

```
FILE *fp = fopen(filename, “a”);
if(fp == NULL)
    syslog(LOG_WARN,
     “cannot open file: %s”, filename);
```

このようなログポイントの設定は広く行われている．しかし，システム障害に関する知識がないと，適切なログポイントを選択するのが難しく，必要なデータが不足することもある．また，ログポイントの挿入し忘れやそれ自体のバグも起こりえる．また，ログポイントは，一般的に，障害が発生する前に指定する．デバッグのように必要な情報や箇所が絞り込まれていない事から，その決定は難しい．これらの要因により，障害時に，フォルトの特定に役立つような，適切なログポイントが必ずしも挿入されない問題がある．

3.3.2基本アイディア

我々の基本的なアイディアは簡単である．開発者が自分でログポイントを明示的に入れる代わりに，言語ランタイム側がシステム障害の解析に役立つ情報を収集して，自動的にログを保存する機構を追加することである．これにより，プログラムの異常終了から，なぜ異常終了したのか原因を切り分けるヒント情報が得られることを期待する．

我々は，言語ランタイムが収集するログポイントとして以下の３点を選んだ．

- リソースロギング
- イベントロギング
- エラー時のシステム診断ロギング

本節の残りは，それぞれの概要を説明する．

3.3.3リソースロギング

リソースロギングは，プログラムの実行とは非同期に，一定間隔の周期で記録を行う．我々は，

3 管理ドメインとは，ある統一的な方針に基づいて管理を行うサブシステムの集合である．ここでのサブシステムは，システムの構成要素をさす．サブシステムには，ハードウエア，ソフトウエア(OS，ミドルウエア，アプリケーション，ライブラリ)が含まれる．

過去の OS システム情報から異常検出を行った研究事例や経験事例を参考とし，次のようなパラメータを定期的にリソースロギングするものとした.

- CPU リソース．ロード･アベレージ．1 秒間で実行した命令数
- メモリ利用量，GC 発生回数
- I/O エラーのエラーカウント

注意：通常，リソースモニタリングは，閾値判定や外れ値解析などの処理をオンラインで行う．本研究は，オフライン処理で診断することを前提に，と区別な処理は行わない.

3.3.4イベントロギング

我々は，システム障害の関連するイベントとして，対象プログラムと外部システム間のやり取りのとき発生するイベントに注目する．対象プログラム内部のスタックなどの状態変化は，バグなどの障害を追跡するに便利であるが，今回は外部のみ着目する.

外部システムとのやり取りは，OS システムコールなど外部関数(Foreign Function)を通して行われる．言語ランタイム自身，これらの外部関数を用いて動作するが， FFI (Foreign Function Interface)を経由して，ユーザにライブラリとして提供される.両方の記録がもれなく必要である．そのため，言語ライブラリの FFI コール時において,イベントロギングのログポイントを設定する.

イベントロギングは，ログポイントの位置だけでなく，どのようにログデータを記録するのかも大きな論点となる．たとえば，unlink()などのファイル削除する関数は，例えばユーザのミスにより呼び出されてしまった場合，ファイルの削除に成功しても,必要なファイルを削除してしまえば,そればフォルトとなる．また，多くのログシステムが警告レベル(syslog の場合は，Emergency, Alert, Critical, Error, Warning, Notice, Info, Debug の８レベル)を用意しているが，こうした警告レベルを統一的に設定することが難しい.

我々は，外部関数の静的な属性に対するポリシーベースのログポイントの設定を提案し，導入した． 設定したポリシーは次のとおりである.

- Sampling 定期的なログポイント
- Pre-Action - 関数コールの前．実行時間がかかる可能性がある/ブロックする可能性がある
- Failed - 関数コールの失敗
- SystemChange - 外部システムに永続する影響を残す関数コール
- SecurityAudit - セキュリティ侵害に影響があり得る

3.3.5診断ロギング

診断ロギングは，外部関数コールが失敗したとき，エラーコードを記録するのと同時に，リソースモニタリングを起動し，エラー状態の診断を機能である.

我々は警告レベルの代わりに，ログポイントのエラー情報として，外部関数コールが失敗したとき，その要因として考えられるフォルトを設定するようにした． フォルトは，あまり細分化すると扱いが難しくなるため，次の４種類を導入する.

- SoftwareFault - プログラマのプログラムミスによるフォルトの可能性があるもの.
- DataFault - プログラムのユーザの入力間違いや入力されたデータに要因があるもの.
- SystemFault - 対象プログラムを動作させている OS(ファイルシステム，メモリ空間）に異状があるもの.
- ExternalFault - 対象プログラムから明示的に分離した外部システムに要因が考えられるもの

フォルトは，同じ関数コールのエラーにであっても，複数の要因が考えられる．たとえば，ファイルのオープンに失敗した場合は，一意に要因がいえない．そのため，SystemFault|DataFault のように，複数のフォルトを組み合わせることを認める.

SystemFault もしくは ExternalFault が与えられている関数コールに失敗したとき，診断プログラムとして，第 3.3.3 節のリソースロギングが起動され，エラーと同時にリソース状態も記録される.

注意：診断ロギングは，例外送出の前に実行される．本報告では，ログ解析のオンライン処理を行わないが，将来は例外に対し，SystemFault，DataFault など，ログ解析の結果を付加して，プログラマにヒント情報をフィードバックする予定である．

3.4 ロギング機構とロギング API

我々はオープンソース開発中の Konoha 言語のランタイムを拡張し，ロギングモジュールと診断モジュールを追加し，ログの自動収集機構について実装を行った．

Konoha では，モジュールを追加することで機能を拡張する仕様となっており，ロギング機構は言語ランタイムのモジュールとなっている．プログラムは，標準化されたロギング API を用いることでロギング機構をとおしてログを出力でき，ユーザはロギング機構を切り替えることで，ログの扱いを切り替えることができる．

以下では現在のシステム管理者が広く採用している標準的な syslog にデータログを出力するモジュールを実装した．

KTraceDataLog は，データログを出力するための API である．ログ出力形式への整形は，各ロギング機構が行うため，ログはキー･バリューのデータのまま受け取る．このとき，いくつかのオプションをつけることで errno の取得が可能とした．

KTraceDataLog は，第 3.3 節で述べたとおり，ポリシーベースのロギングに対応している．すべてのログポイントにて，ログポイントの開発者によるポリシーと失敗したときのフォルトを記述する．一方，警告レベルなど(Alart) は，ロギング機構側がポリシーとログポイントの情報にもとづいて調整できる．これにより，ログポイントごとのまちまちの警告レベルが出るといった不統一は避けられる．

3.5 診断モジュールと診断 API

診断モジュールは，診断 API である Diagnosis() を呼び出すことで OS システム，言語ランタイムの統計情報を得る．

OS のシステム情報は，OS ごとに実装が異なるが，Linux の場合は，/proc 以下から情報を得る．

診断モジュールの実行結果は前述の TraceDataLog を用いてログ記録される．ポリシーは，Sampling である．実際に syslog に書き出されたログを，図 2 に示した．

```
{"time(ms)": 1348591486209, "procs_running": 2,
"procs_blocked": 2, "memory_swpd(kb)": 433140,
"memory_free(kb)": 904716, "memory_buff(kb)": 67656,
"memory_cache(kb)": 470880, "swap_si(kb/s)": 35,
"swap_so(kb/s)": 176, "io_bi(block/s)": 60,
"io_bo(block/s)": 282, "system_in": 0, "system_cs":
11, "cpu_us(%)": 1, "cpu_sy(%)": 0, "cpu_id(%)": 99,
"cpu_wa(%)": 0, "[process] cpu_usage(%)": 0.0,
"[process] mem_usage(%)": 1.7307, "[process]
mem_usage(kb)": 17980, "[process] rchar(b)": 17642,
"[process] wchar(b)": 49, "[process] syscr(b)": 25,
"[process] syscw(b)": 2, "[process] read_bytes(b)": 0,
"[process] write_bytes(b)": 0, "[process]
cancelled_write_bytes(b)": 0
}
```

図 2 ログの出力結果

3.6 ログポイント

本節では 3.3 節で述べたロギング項目についてログを出現する手法について述べる。

3.6.1 リソースロギング

リソースロギングは，言語ランタイム本来の処理と並行して周期的に行われることを想定した．したがって， Mini Konoha 言語ランタイム起動時に，別スレッドを起動し，無限ループを用いて 1 秒周期で診断 API を呼び出す実装とした．Diagnosis API 従い，システム全体のリソースの使用量と，現在実行中の言語ランタイムでの使用量を測定してログとして記録するものとした．

3.6.2 イベントモニタリングにおけるログポイント

本研究では，イベントモニタリングにおいて試験的に下記の箇所にログポイントの挿入を行った．

- 言語機能拡張のためのパッケージ内での，外部ライブラリ関数呼び出し失敗時
  - POSIX 関数（fopen, fread, etc…）の失敗時
  - Libcurl で提供されている通信用の関数の失敗時

また，図 3 にイベントログを記録する処理を記述したソースコードを示した．

```
1 FILE *fp = fopen(S_text(s), mode);
2 if (fp == NULL) {
3    KTraceApi(SystemFault|ScriptFault, "System.fopen",
4       LogText("@", "fopen"),
5       LogText("path", S_text(s)),
6       LogUint("mode", mode),
7       LogText("errstr", strerror(errno))
8    );
```

図 3　外部ライブラリ関数の正否チェック例

ここでは，外部ライブラリ関数呼び出しの正否をチェックし，失敗時にログを出力する．具体的には，fopen 関数の正否をチェックし，戻り値が NULL であった場合に 3 行目にて KTraceApi という関数が呼ばれている．KTraceApi とは Mini Konoha モジュール内部の実装で定義されている関数で，key-value 形式で入力されたデータを Json 形式にフォーマッティングして syslog へ出力するために，syslog 関数をラップして実装した API である．図 3 の例ではイベントログとして，失敗した関数名，引数，及び失敗時に標準ライブラリ関数である strerror で取得できるエラー情報を記録している．

3.7　診断処理とエラー処理

診断 API の実現のために，本研究では，情報を取得し，得られた結果を syslog へ出力する Diagnosis API を実装した． Diagnosis API の内部では，/proc 以下の情報のパース，及びそれらの情報に対して簡単な統計処理を行い，出力する処理を行う．このように，Diagnosis API はリソースロギング時に 1 秒間隔で呼び出される以外に，イベントロギング時にも実行されるものとした．特に，イベントが失敗した場合にリソース状況を自動でログに記録することで，イベント失敗の原因を後にログから解析することを可能とする．Diagnosis API で出力するデータはユーザが選択することができ，必要に応じた情報を取得できるようにした．

3.8　実験

我々は，自動ロギング機構を用いて取得したログが，3 節にて定義したフォルトへの対応に役立つかどうかを議論するために，言語ランタイム上で動作するプログラムに対して，故意にフォルトを発生（フォルトインジェクション）させ，ログを取得する実験を行った．本節では，実験の概要と結果，及び実験の結果得られた知見を述べる．

3.8.1実験概要

本実験の対象プログラムとして，Web システムの運用に用いられるプログラムを想定する．また，サーバソフトウェアには Apache を使用することを想定する．我々は，次の 3 つのプログラムをフォルトインジェクションの実験対象とする．

- monitor - モニタリングプログラム（常駐）
  3 秒毎にサーバのステータスをネットワーク経由でリモートから取得する．ステータスの取得には Apache の mod_status を使用し，http の通信には libcurl を使用する．
- backup - バックアッププログラム（定期的）
  以下の動作手順に従い，サーバ上のデータをバックアップ用のディレクトリにコピーする．
  1. コピー先のバックアップ用ディレクトリの数が 4 つ以上あるとき，最も古いものを削除する
  2. 空のバックアップ用ディレクトリ (bk.0) を作成する
  3. 次のコマンドを実行する（bk.1 は前回のバックアップディレクトリ名を表す）

  rsync --delete -F --link-dest=bk.1 bk.0
- restart - 再起動プログラム（障害発生時）
  サーバソフトウェアの停止と起動を行う．Apache における apache2ctl と同等の機能を提供する．

本実験の目的は，上記のプログラムに対して，3 節で定義したフォルトを発生させた時の挙動を観測することである．まず，4 つのフォルトのうち，ソフトウェアフォルトとデータフォルトに関しては，例外処理機能をもつプログラミング言語上で，正しく例外が記述されていれば，例外で補足されるか，システムコールに関する部分のエラーであれば，言語ランタイムから出力されるログにエラーが記録されることを確認した．

ソフトウェアフォルトにおいては，エラーの原

因であるフォルトの診断は，引数の不足やプログラム名の指定のミスなど，一般的なプログラムの例外処理で扱うエラーと同等のフォルト（API が存在しない，ソケットがクローズされていない可能性がある，ファイルが存在しない，etc..）が，エラーと対応するよう設計されていれば，エラーの結果からフォルト類推がしやすい．一方，フォルトが設計ミスや仕様ミスなどの場合については，フォルトの類推は困難であった．これらは，主に既知の事実の追認であるため，本節では割愛する．

一方，フォルトが外部フォルトや，システムフォルトである場合の要因特定は，システムや外部システムの情報も同時に収集し，要因特定を行う必要がある．ここでは特にシステム，外部フォルトについて，提案手法の検証実験を行った結果を示す．以下に挿入したフォルトを示した．

- ファイルへのアクセス失敗

ファイルシステムが発生するフォルトを模擬し，プログラムのファイルへのアクセス失敗を引き起こすため，必要ファイルの削除を行う．

このフォルトはシステム内部で実行プログラム（monitor, backup, restart）以外のプログラムが発生するシステムフォルトとする．実験にはCore i7 2620M 2.7GHz 2 コア，メモリ DDR3 4GB, Ubuntu11.04, Apache2.2.22 を用いた．

3.8.2システムフォルトインジェクション

実験対象の３つのプログラムに対して，システムフォルトを発生させた場合に，自動ロギング機構によって異常値が観測された項目をまとめた．

１つめのフォルトである，ファイル削除を実行した結果を図 5-図 8 に示した．なお，図の横軸は実行開始からの経過時間を表し，図中の縦線はファイル削除のフォルトインジェクションのタイミングを表す．

図 5 では総 CPU 利用率として，cpu_us と cpu_sy の合計値を使用した．また図7および図8の io_bi, io_bo の値には，測定値の合計値を使用した．

本実験では，バックアッププログラムをループで 10 回実行するプログラムを作成し，そのプログラムに対してファイル削除（コピー元のディレクトリを rm -rf コマンドで削除する）のフォルトインジェクションを行った．

なお，バックアップ対象のディレクトリとして，100 個のディレクトリ内に 100 個のテキストファイルをもつ，計 39MB の容量のディレクトリを用意した．

図 5-8 では，ファイル削除のエラーイベント（“no such file or event”）が発生するタイミング以前と以降（縦線にて表示）にて，図 5 の CPU 利用率，図 6 のメモリ使用量の変化が観測できた．特に CPU 利用率は，backup プログラム自身の CPU 利用率(下方)と，システムの CPU 利用率(上方)の変化が同期していることから，プログラムの処理に直接関係する障害が発生したと推定できる（図 5）．また図 6 では，backup プログラムが利用するメモリ量が一貫して増大し，逆に free メモリは減っていた変化が，ファイル削除以降，変化が途絶える事から，同様にエラーイベントのタイミングで障害が発生した事が推定できる．I/O(図 7,8）では，確保された I/O のバッファは，ファイルの削除とともにすぐに解放されない事もあり，影響は十分観測できていない．

これらの情報から，エラーイベントと，発生元のプログラム(backup)，影響があったリソース情報をもとに障害要因の特定を行う．エラーイベントの内容，CPU，メモリに影響が発生していることから，backup 処理中で最も処理時間がかかり，かつメモリを利用するコマンドとして，cp (backup) コマンドを特定することができる．検証では，Cp コマンドの確認として，引数の複製元，複製先の確認を行ったところ，複製元のディレクトリが無くなっている事(フォルト)を，特定することができた．このように，収集したログの中でも，利用可能な観測情報が複数あれば，障害要因の特定を，こうした情報がない場合よりも，より行いやすくすることができたといえる．また，エラーイベントと関連づけてシステム情報を利用することで，エラーの発生とシステムの状況を総合的に判断することができ，障害対応の選択をより正しく行うことができると考えられる．

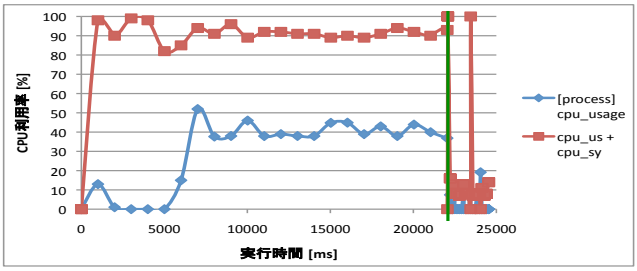

図 5　フォルト挿入時の CPU 利用率

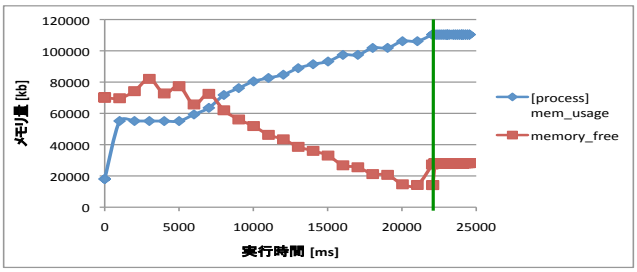

図 6　フォルト挿入時のメモリ使用量

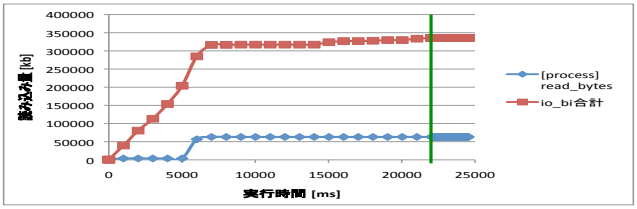

図 7　フォルト挿入時のディスクからの読み込み量

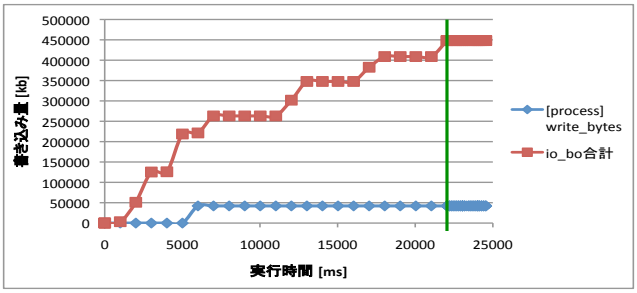

図 8　フォルト挿入時のディスクへの書き込み量

3.8.3HPC システムへの適応

我々は HPC 環境にてアプリケーションが正しく動作するというトップゴールに対し MPI アプリケーションに対するシステム障害対応について検討を行った。

我々は、JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」の一部として，システム運用の高信頼化，ディペンダビリティ化を助けるスクリプト技術 D-Script に取り組んできた.

我々は Logpool ロギング機構を組み込んだ Konoha 言語と、それを用いて記述された MPI アプリケーションに対して、D-Script 技術を用いて MPI アプリケーションが正しく動作するというディペンダビリティ要求に対して、この要求と対応つけたソフトウェア開発、特に運用スクリプトの記述を行う。

我々は、まず MPI アプリケーションに関わるステークホルダ間でディペンダビリティ要求をトップゴールとして議論を行い、下記の 5 項目をサブゴールとした Assurance Cases を記述した。

- MPI アプリケーション
- MPI ライブラリ
- ハードウエア
- ネットワーク（通信）
- I/O 不調（コマンドの反応速度）

図 9 は今回検討を行ったシステムに対する Assurance Cases の一部である。この図において我々はソフトウェア実行前におけるディペンダビリティ要求を満たすことを論じている。

一方、システム実行中に発生したディペンダビリティ要求からの逸脱，つまりシステム障害はシステム運用中に発生し、システム中に埋め込まれたロギングポイントから得られたログによって確

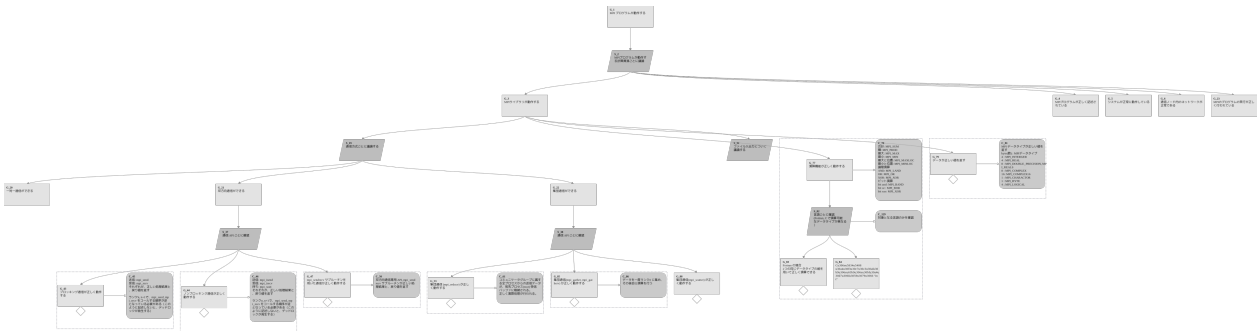

図 9　MPI アプリケーションの AssureanceCase の一部

認することができる.

(2) 当初計画の達成状況について

本提案では、研究目的を達成するため以下の３つの項目に分け、研究を行った。

・言語ログ：プログラミング言語への自動的なログ出力の統合

・言語ログを用いたシステム障害に対する要因特定の実験

・ログ解析機構の性能チューニング

我々は言語ログに注力し、自動的なログ出力を行う言語基盤の構築を行った。実験を行う段階において、実験環境全体に影響を与えるような大規模システム障害を発生させることが困難であった。フォルトインジェクションを行うこと自体の難しさのため、実際のログを用いたログ解析器は本年度に実現ができなかった。

4 今後の展望

我々は、言語支援として言語ランタイムの自動ロギング機構を構築し、フォルトインジェクションを行った結果に基づきログから障害を診断可能であるか検討を行った。今後はログ解析器を含めた言語システム基盤の構築を行う。

5 研究成果リスト

(1) 学術論文（投稿中のものは「投稿中」と明記）

1. Midori Sugaya, Hiroki Takamura, Youichi Ishiwata, Satoshi Kagami, Kimio Kuramitsu, Online Kernel Log Analysis for Robotics Application. Journal of Information Processing, 2012
2. Masahiro Ide, Kimio Kuramitsu. Just-in-time compiler for KonohaScript using LLVM. Journal of Information Processing Vol.20 No.4 1-8 (Oct. 2012) [DOI: 10.2197/ipsjjip.20.1]

(2) 国際会議プロシーディングス

1. Wakamori, T.; Ide, M.; Sugaya, M.; Kuramitsu, K., "Reconfigurable Scripting Language with Programming Risk," Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on , vol., no., pp.316,318, 27-30 Nov. 2012 doi: 10.1109/ISSREW.2012.94
2. Yoan, M.; Sugaya, M.; Kuramitsu, K., "A Study of Converting Risk to Assurance Case", Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on , vol., no., pp.313,315, 27-30 Nov. 2012 doi: 10.1109/ISSREW.2012.79

(3) 国際会議発表

1. Takuma Wakamori. Masahiro Ide, Midori Sugaya, Kimio Kuramitsu. Reconfigurable Scripting Language with Programming Risk. Workshop on Open System Dependability 2012, in conjunction with ISSRE2012, to appear, 2012.
2. Motoki Yoan, Midori Sugaya, Kimio Kuramitsu. Converting Risk to Assurance Case, Workshop on Open System Dependability 2012, in conjunction with ISSRE2012, 2012.

(4) 国内会議発表

1. 養安元気, 中田晋平, 岡本悠希, 菅谷みどり, 倉光君郎. D-Script: 分散環境における障害管理を行うスクリプト言語基盤. 先進的計算基盤システムシンポジウム SACSIS 2012
2. 菅谷 みどり, 高村 博紀, 横手 靖彦, 倉光君郎. DRE: フォルトモデルを考慮した障害回避の支援基盤の提案. 先進的計算基盤システムシンポジウム SACSIS 2012
3. 井出真広, 若松悠樹, 平岡祐太郎, 倉光君郎. 静的型付けスクリプト言語 KonohaScript の HPC 利用に向けて. 先進的計算基盤システムシンポジウム SACSIS 2012(ポスター発表)
4. 井出 真広, 志田 駿介, 倉光 君郎. LLVM を用いた静的型付きスクリプト言語 KonohaScript の Just-in-time コンパイラの設計と実装. 先進的計算基盤システムシン

ポジウム SACSIS 2012

5. 倉光君郎. Konoha = MiniKonoha + Sugar. 並列／分散／協調処理に関する『鳥取』サマー・ワークショップ (SWoPP 鳥取 2012). 2012
6. 菅谷みどり, 井出真広, 中田晋平, 養安元気, 倉光君郎. ログベースアクターを用いた統合運用管理ツール基盤. 並列／分散／協調処理に関する『鳥取』サマー・ワークショップ (SWoPP 鳥取 2012). 2012
7. 岡本 悠希, 養安 元気, 菅谷 みどり, 倉光君郎. 分散障害管理のためのアクターベースのスクリプトフレームワーク. システムソフトウェアとオペレーティング・システム (OS) Vol.2012-OS-122 No.19. 2012
8. 若森 拓馬, 中田 晋平, 菅谷 みどり, 倉光 君郎. 障害対策の事例に学ぶ障害対策支援に向けて. 日本ソフトウェア科学会第 29 回大会, 2012.
9. 中田 晋平, 菅谷 みどり, 倉光 君郎. 障害対処ワークフローに起こる二次リスクへの保証ケース記述手法. 日本ソフトウェア科学会第 29 回大会, 2012.
10. 菅谷 みどり, 山本 修一郎, 高井 利徳, 倉光 君郎. オープンシステムディペンダビリティ：新しいディペンダビリティへの挑戦. 情報処理学会ソフトウェア工学シンポジウム 2012, 2012.
11. 小野田 武朗, 菅谷 みどり, 倉光 君郎. Web 技術文書からの FFI 自動生成に関する実践. 情報処理学会ソフトウェア工学シンポジウム 2012, 2012
12. 倉光君郎. 森田直. ワークショップスクリプトによる高信頼なソフトウェアサービス構築. 情報処理学会ソフトウェア工学シンポジウム 2012, 2012
13. 井出真広, 菅谷みどり, 倉光君郎. Mini Konoha: 最小構成から目指すディペンダブルスクリプト言語の設計. 情報処理学会ソフトウェア工学シンポジウム 2012, 2012
14. 中田晋平, 菅谷みどり, 倉光君郎. D-Script: 障害対応スクリプトと回復戦略を行うスクリプトフレームワークの概要. 情報処理学会ソフトウェア工学シンポジウム 2012, 2012.
15. 菅谷 みどり, 岡本 悠希, 若森 拓馬, 倉光君郎. 言語ランタイムにおける自動的なログ収集機構. 情報処理学会 第 91 回プログラミング研究発表会, PRO91, 2012
16. 中田晋平, 菅谷みどり, 倉光君郎. 実行の失敗に備えて回復を試みる障害対応スクリプト言語モデル. 情報処理学会 第 91 回プログラミング研究発表会 PRO91. 2012
17. 岡本 悠希, 養安 元気, 菅谷 みどり, 倉光君郎. 分散環境における障害対応スクリプト実行基盤の構築. 第 10 回 ディペンダブルシステムワークショップ (DSW 2012)
18. 石井 正樹, 養安 元気, 菅谷 みどり, 倉光君郎. ソフトウェアスクリプトテストと保証ケースの統合に向けて. 第 10 回 ディペンダブルシステムワークショップ (DSW 2012)
19. 養安 元気, 岡本 悠希, 菅谷 みどり, 倉光君郎. 障害対応を迅速に行う統合運用ツールの提案. 第 10 回 ディペンダブルシステムワークショップ (DSW 2012)
20. 若森 拓馬, 菅谷 みどり, 倉光 君郎. D-Shell：ディペンダブルなシェルスクリプトの提案. 第 10 回 ディペンダブルシステムワークショップ (DSW 2012)

(5) その他（特許，プレス発表，著書等）

1. Kimio Kuramitsu. Flexible Management of Failure Response. In Chapter 7 of Mario Tokoro's Open Systems Dependability: Dependability Engineering for Evar-Changing Systems, CRC Press, 2012.